Introduction to Spring

Spring Framework (or simply *Spring)* is an application framework, open source, which aims to simplify the development of enterprise Java applications

- Spring was initially created in 2002, as a somewhat alternative approach to the Java EE platform (then calledJ2EE, today named Jakarta EE) – to achieve similar purposes but with a simpler and lighter programming model.

- Over time, Spring has evolved significantly – in terms of programming model, functionality, as well as supportfor new technologies.
- meanwhile, the Java/Java EE platform has also evolved significantly – sometimes even taking on some of Spring's innovative ideas.
- Today Spring is a modular yet cohesive framework that can support a variety of application needs – such as web applications, security and the cloud.

Strategies adopted by Spring

- Here are the main strategies adopted by Spring to simplify the development of Java applications
  - lightweight development based on POJO (Plain Old Java Object) – called in Spring "bean"
  - Weak coupling based on Dependency Injection (DI), together with extensive use of interfaces
  - declarative programming based on common expectations, configurations and conventions
  - elimination of repeated code ("boilerplate") through aspects and templates

Bean and POJO

Lightweight development based on POJO (Plain Old Java Object)

- some frameworks force you to "dirty" your application code with the use of their own specific APIs – writing classes that extend classes or implement interfaces to these APIs.
- on the contrary, Spring avoids (as far as possible) that the application code should be soiled with the use of its own APIs

- Spring's application model is based on simple objects or *POJO* (*Plain Old Java Object)* classes also called *beans*
- in theory, a POJO is a class that does not follow any specific pattern or convention or framework – in particular, it does not extend any default class, it does not implement any default interface, it does not contain any default annotation
- in practice, Spring beans slightly violate this definition – e.g., the use of get/set method conventions (JavaBean) and some specific annotations is common.

Injection of dependencies

Weak coupling based on dependency injection

- non-trivial applications are composed of multiple classes and objects, which collaborate with each other through messaggi/invocations
- the objects of the application have *dependencies* on other objects – in fact each object must know (the references to) the other objects to which it must send messages to collaborate
- a project in which objects are responsible for capturing references to the objects on which they depend (or even creating them) can be highly coupled and difficult to test
- *Dependency Injection (DI)* is a mechanism for assigning (injecting) objects with their dependencies at the time of creation – objects no longer have to deal with acquiring their dependencies directly – this reduces coupling, and also promotes testing

Common configurations and conventions

Declarative programming based on common configurations and conventions

- in Spring, many tasks (such as injecting dependencies) are based on *declarative specification* of class and object configurations – rather than writing imperative code

- configuration metadata can be specified as xml-based configurations, Java-based configurations (using annotations) and property files
- Spring also provides automatic and implicit configuration mechanisms, also based on the adoption of default values and common conventions

Templates

- Deletion of repeated code using templates
  - many technologies (and their APIs) require you to write (several times) a large amount of code, full of details, even to perform simple and common tasks – the so-called "boilerplate code" ("standard code blocks")
  - e.g., think of the code needed to execute a SQL query with JDBC and then reconstruct an object from the result of the query
  - Spring provides, for different technologies, utility classes (called *templates)* to perform the most common tasks in a simplified way, eliminating the need for "boilerplate code" and therefore reducing the amount of code to be written (and to be tested and maintained)

* Injection of addictions

Dependency Injection (DI) is a key feature of the Spring framework.

- a Spring application is typically composed of many objects, which must collaborate – therefore, these objects have *dependencies on* each other

- to organize and compose objects in a consistent application you need to manage these dependencies – e.g., one solution might be to use a Factory
- the Spring framework addresses this problem by injecting dependencies – also called "inversion of control" or *Inversion of Control (IoC),* because it is no longer the objects that have to *directly* manage their dependencies, but rather there is anyone else that manages them for them: the Spring container

Container

- In a Spring application, objects live in a Spring *container*– also called *Inversion of Control container* or *IoC container* – the notion of container is central to the Spring framework (and component technologies)
  - the container has the responsibility to create objects, configure them and link them together (by injection of dependencies) – and more generally to manage the life cycle of these objects (from creation to destruction)
  - to this end, the Spring container must be properly configured, to specify which objects it should create, and how to configure and link them to each other
  - configuration metadata can be specified either in XML or through Java annotations or property files
  - there are different types of Spring containers – including *application contexts,* with different implementations

Bean

In Spring, objects managed by the container are called *beans*

- each bean is a POJO, which is given a name
- a bean may have dependants
- there are two main mechanisms through which the injection of dependencies into a bean can take place – via the constructor (and its parameters) and through set methods

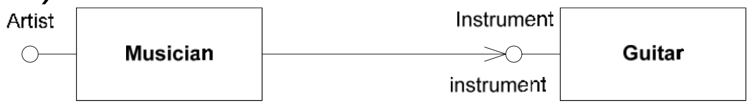
- in both cases it is the container that will invoke the constructor and/or the set methods
- in fact, it is also possible to inject dependencies directly into the fields (instance variables) of the bean

Bean

- In Spring, objects managed by the container are called *beans*
  - in Spring, a "bean" is a different notion from both an object and a class
  - intuitively, a "bean" is a "type of object"
  - the definition of a bean requires a class – but there can be multiple beans defined by the same class
  - there may also be multiple instances of the same bean

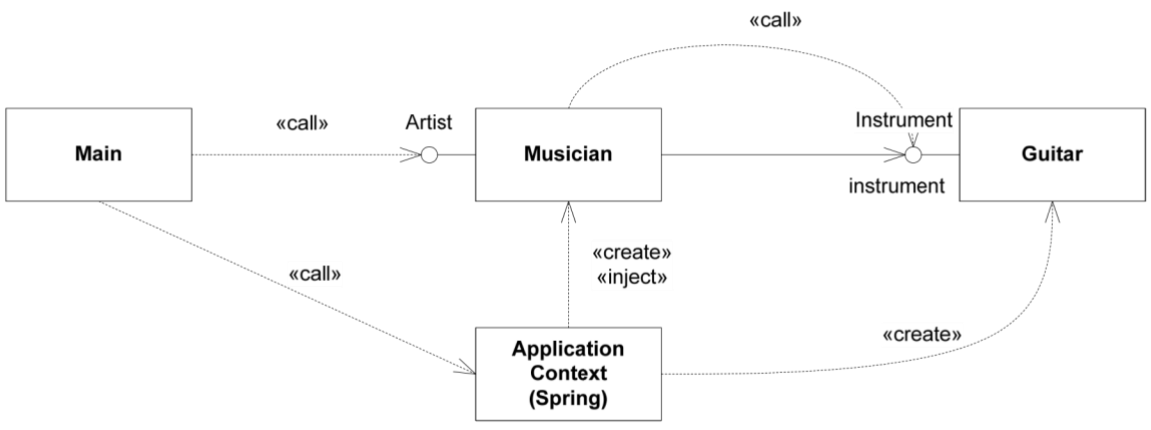
- XML-based configuration

Suppose you need to make an application with a bean **Musician** (who is an **Artist)** connected to a bean **Guitar** (which is a **Instrument**)



- the configuration of these beans (including their connection) can be specified using an xml configuration file
- access to these beans can be done through an application context – in the example, of type **ClassPathXmlApplicationContext,**which reads the configuration from an XML file

Esempio



XML Configuration

- Here is the XML configuration file show-beans.xml
  - describes the beans of the application – with their names and types (classes) – as well as the dependencies between them

name (id) of a bean

bean class

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

  <Bean id="hendrix" class="asw.spring.show.Musician">
    <constructor-arg value="Jimi"/>
    <constructor-arg ref="stratocaster"/>
  </bean>

  <Bean id="stratocaster" class="asw.spring.show.Guitar">
    <property name="sound" value="Ta Ta taa"/>
  </bean>

</beans>
```

in bean

a dependency to be satisfied

reference to another bean (by name)

The Main class

The **Main** class needs a reference to the bean **hendrix** package

```
asw.spring.show;

import org.springframework.context.ApplicationContext; import
org.springframework.context.support.
                ClassPathXmlApplicationContext;

public class Main {
        public static void main(String[] args) {
            ApplicationContext context = new
                ClassPathXmlApplicationContext("show-beans.xml");
                 Artist artist = (Artist) context.getBean("hendrix");
            System.out.println( artist.perform() );
        }
    }
```

- running this application will display the string I'm Jimi: Ta ta taa

- Dependency resolution process

- Spring container handles dependency resolution as follows
  - the application context is created and initialized with configuration metadata
  - the dependencies of each bean consist of the constructor arguments and its proprieties (which can be assigned by set methods)
  - these dependencies will be provided to a bean at the time of its creation – which, depending on the scope (discussed later), can occur at the time of initialization of the container or on request
  - when the container needs to create a bean, the container determines, for each constructor argument and for each property, the reference to another bean in the container (ref) or the value (value, a literal or constant) to be assigned/injected

Scope in the creation of beans

Each type of bean is characterized by a *scope,* which defines how many beans to create and when (for that type of bean) – the scope can be specified using the XML scope attribute

- *singleton* (this is the default) – exactly one instance of the bean per container – the bean is usually created whenthe container isinitialized there
- *prototype* – one instance for each request for a bean of that type
- *request* – in a web application, one instance for each HTTP request
- *session* – in a web application, one instance for each HTTP session
- *application* – in a web application, an instance for the entire web application

Discussion

- In XML-based configuration
  - the interfaces and classes for beans are actually POJO
  - configuration metadata is encoded in XML
  - the client of a bean can access the bean through the application context, based on the name (or type) of the bean of interest

- Parentheses: annotations

In Java (as in other programming languages) *annotations* are a syntactic element used to annotate (i.e., label) code elements – such as classes, interfaces, variables, or methods

- for example, **@Test** (of JUnit), **@Bean,** or **@Configuration** (of Spring)
- the compiler reports the annotations in the bytecode as metadata

– but the compiler does not further interpret the annotations

- annotations are taken into account by appropriate development tools (e.g., JUnit) and/or the execution environment (e.g., the Spring framework), which can act accordingly – but are ignored by other tools/environments
- in the following will be exemplified some annotations of the Spring framework

- Java-based configuration

□ The configuration metadata of an application can also be specified using a Java-based configuration, using appropriate annotations

- a Java-based configuration requires a configuration class, which must be noted **@Configuration**, which contains a method for each bean (i.e., type of bean)
- the method for a bean must be noted **@Bean** – the method must create the bean and link it with other beans (whose references must be obtained by invoking the corresponding methods) – it should be noted that then the container will decide when to invoke these methods
- bean access can be done through an application context of type **AnnotationConfigApplicationContext** – which gets the configuration from the configuration class
- in the example (which is equivalent to the first one), the interfaces and classes for the beans are defined as before

The @Bean annotation

- The annotation **@Bean** defines a bean
  - is similar to an XML<bean/> element
  - the name of the bean implicitly corresponds to the name of the method – otherwise it can be defined explicitly with **@Bean(name="hendrix")**
  - the scope of a bean can be specified with the annotation **@Scope**

Discussion

- In Java-based configuration
  - interfaces and classes for beans are still actually POJO
  - configuration metadata is encoded in Java, in the form of annotations
  - the client of a bean can still access the bean through the application context, based on the name (or type) of the bean of interest

Comparison

XML-based and Java-based configuration are two different ways to describe the configuration metadata needed in a Spring application

- the spring container is completely independent of how configuration metadata is described
- which of the two configuration modes is preferable?

- Components and autowiring

- Spring also simplifies the specification of configuration metadata – i.e. the specification of beans and their dependencies and relationships – based on an inspection of application code and automatic configuration mechanisms
  (implicit)
  - component definition and automatic scanning simplifies bean specification
  - autowiring simplifies the specification of connections between beans
  - these are two separate mechanisms, but they are often used together
  - the main advantage of these mechanisms is the simplification of the configuration metadata specification
  - the disadvantage is that these mechanisms have some limitations (not particularly serious) because, in some cases, it is difficult to specify a set of desired beans or a set of desired links between the beans

- in general the answer is "depends" – because each mode has its advantages and disadvantages
- each developer can choose between XML-based and Java-based configuration based on their preferences
- in any case, the developer must specify in detail all the configuration information
- it is also possible to mix the two modes – the container first applies the Java-based configuration and then the XML one (which could overwrite the effects of the first)

Components

A *component* (in the sense of Spring) is a class annotated with the annotation **@Component**

- in Spring, a "component" is intuitively a bean that can be automatically identified and configured
- in addition to **@Component,** Spring defines other annotations, for specific types of components

- e.g., **@Controller, @Repository** and **@Service**
- in addition, the **annotation @ComponentScan** in the Java configuration class enables automatic scanning of components (which are then considered beans)
- in the following, component and bean are considered synonyms

Autowiring

- The automatic connection between components (autowiring)is mainly based on the annotation **@Autowired**
  - the annotation **@Autowired** specifies that the arguments of a constructor or set method must be automatically identified by the container – usually this is done based on the type or name of each argument
  - the annotation **@Autowired** can also be used in the declaration of a field(that is, an instance variable, even private) of a component, to indicate that the container must also assign a value to the field
  - if the type of a topic or field is an interface and there are multiple components that imply that interface, there may be an ambiguity – which can be resolved using the annotation **@Primary** on the component of interest, or by using qualifiers (which however go beyond the scope of this introduction)

Autowiring

With autowiring, it is often useful to be able to specify values to be used in the initialization of components – this can be done through configuration files and annotation **@Value**

- the annotation **@Value** can be used to specify the value (letterale or constant) to be used for the argument of a constructor or set method

- you can also use the **annotation @Value** in the declaration of a field (i.e. an instance variable, even private) of a component
- a common form is **@Value("${property.name}")** where **property.name** is the name of a property specified in a property file (it is a textual config file)
- in this case the java configuration class must use the **annotation @PropertySource** to specify what the property configuration file is

Example

- Let's take our example to show the use of components and autowiring
  - the definition of the **Artist** and **Instrument** interfaces is as before

```
package asw.spring.show;                              package asw.spring.show;

 /* An artist. */      /* A musical instrument. */ public interface Artist { public
                                  interface Instrument {

    /* Artist's performance. */      /* Plays the instrument. */ public String perform();
         public String  play();

}                                                     }
```

The Guitar component

Instrument's Guitar implementation is noted with **@Component** and requires the use of **@Value**

```
package asw.spring.show;

import org.springframework.stereotype.Component; import
org.springframework.beans.factory.annotation.Value;
```

```
@Component(value="stratocaster") public class
Guitar implements Instrument { private String
sound; public Guitar() { }
      @Value("${show.stratocaster.sound}") public void setSound(String
      sound) { this.sound = sound;  } public String play() { return sound;  }
}
```

The Musician component

- **The Musician** implementation also requires the use of **@Autowired**

```
package asw.spring.show;

import org.springframework.stereotype.Component; import
org.springframework.beans.factory.annotation.Autowired; import
org.springframework.beans.factory.annotation.Value;

@Component(value="hendrix") public class
Musician implements Artist {

      private String name; private
      Instrument instrument;

      @Autowired
        public Musician(@Value("${show.hendrix.name}")  String name,
                                  Instrument instrument) {
            this.name = name;
            this.instrument = instrument;
      } public String perform() { "I'm " + name + ": " + instrument.play(); }
}
```

Configuration class

The new **ShowConfig** configuration class (now the important information is in the annotations) package asw.spring.show.config; import asw.spring.show.*;

```
import org.springframework.context.annotation.Configuration; import
org.springframework.context.annotation.ComponentScan; import
org.springframework.context.annotation.PropertySource;
```

```
/* Spring configuration for the Show application. */
@Configuration
@ComponentScan("asw.spring.show")
@PropertySource("classpath:config.properties") public
class ShowConfig {

}
```

 The config.properties file

```
show.hendrix.name=Jimi show.stratocaster.sound=Ta
ta taa
```

The Main class

 The **Main** class is as before

```
package asw.spring.show; import

asw.spring.show.config.ShowConfig;

import org.springframework.context.ApplicationContext; import
org.springframework.context.annotation.
                AnnotationConfigApplicationContext;

/* Application that gets and starts the client. */ public class
Main {

    /* Gets and starts a Client object. */ public static void
    main(String[] args) {
        ApplicationContext context = new
            AnnotationConfigApplicationContext(ShowConfig.class);
             Artist artist = (Artist) context.getBean("hendrix");
        System.out.println( artist.perform() );
    }

}
```

Discussion

In component-based and autowiring configuration

- classes for components are no more strictly than POJO
- components do not have to lay out or implement any default type of Spring APIs, but still require the use of Spring annotations

- configuration metadata is encoded in Java, in the form of annotations (but could also be encoded in XML)
- using only Java annotations, components and autowiring, there is a direct correspondence between Java classes, components and beans (questor constraint does not exist in explicit configurations)
- in the example, it is not possible to have multiple beans from the same classes (such as two musicians playing different instruments)
- the client of a bean can still access the bean through the application context, based onthe name (or type) of the bean of interest

* Dependency management

- Another important aspect of complex applications is *dependencies between modules*
  - attention, they are a different notion from the dependencies between objects or components that we have talked about so far

- In many cases, a software project uses reusable functionality (in the form of libraries) and is divided into several parts, to compose a modular project
  - each part of a project or library constitutes a *module*
  - each module may have *depends on* other modules
  - *dependency management* is a technique for declaring, resolving, and using dependencies required by a software project, in an automated manner
  - the dependencies of a software project are usually managed through build automation tools – such as *Gradle* or *Maven*

Dependency Management

In particular, applications based on the Spring framework depend on the presence of certain libraries (jar files, in specific versions) in the application's classpath – for compilation, execution and/or testing

- each of these libraries or resources is an application *dependency* – not to be confused with bean and component dependencies

- e.g., the Spring applications shown so far depend on the org.springframework:spring-context library (which deals with dependency injection) – and also on org.springframework:spring-test for testing (which for simplicity were not shown)
- a simple Spring web application often requires a dozen dependencies or more

* Discussion

- Spring is a modular application framework for the development of enterprise Java applications
  - this handout described the injection of dependencies, which is a key feature of Spring, implemented by the core modules of the framework
  - Spring is a framework composed of about twenty modules – Spring's modules concern, among other things,
  - web application development (Spring Web MVC)
  - access to data(for example with JDBC or via ORM) – Spring Data
  - messaging and integration (Spring Integration)
  - further support and simplification in the development of Spring-based applications – Spring Boot
  - support for the development of distributed applications for the cloud – Spring Cloud