

# Spring Framework

# Spring Framework



<http://www.springsource.com/>

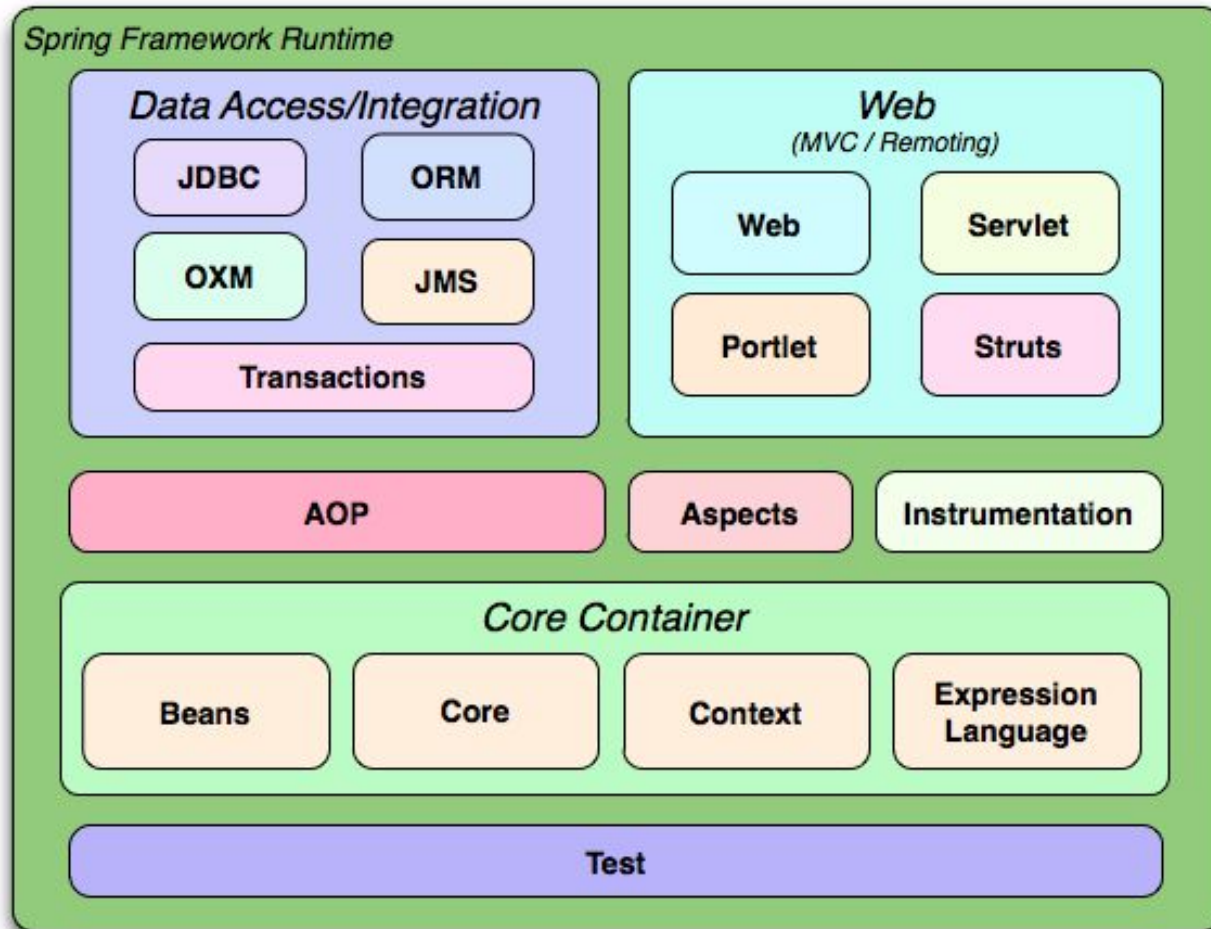
# Spring mission

- J2EE should be easier to use
- It's best to program to interfaces, rather than classes. Spring reduces the complexity cost of using interfaces to zero.
- JavaBeans offer a great way of configuring applications.
- OO design is more important than any implementation technology, such as J2EE.
- Checked exceptions are overused in Java. A framework shouldn't force you to catch exceptions you're unlikely to be able to recover from.
- Testability is essential, and a framework such as Spring should help make your code easier to test.

# Spring

- Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications. Spring handles the infrastructure so you can focus on your application.
- Spring enables you to build applications from “plain old Java objects” (POJOs) and to apply enterprise services non-invasively to POJOs. This capability applies to the Java SE programming model and to full and partial Java EE.

# Spring components



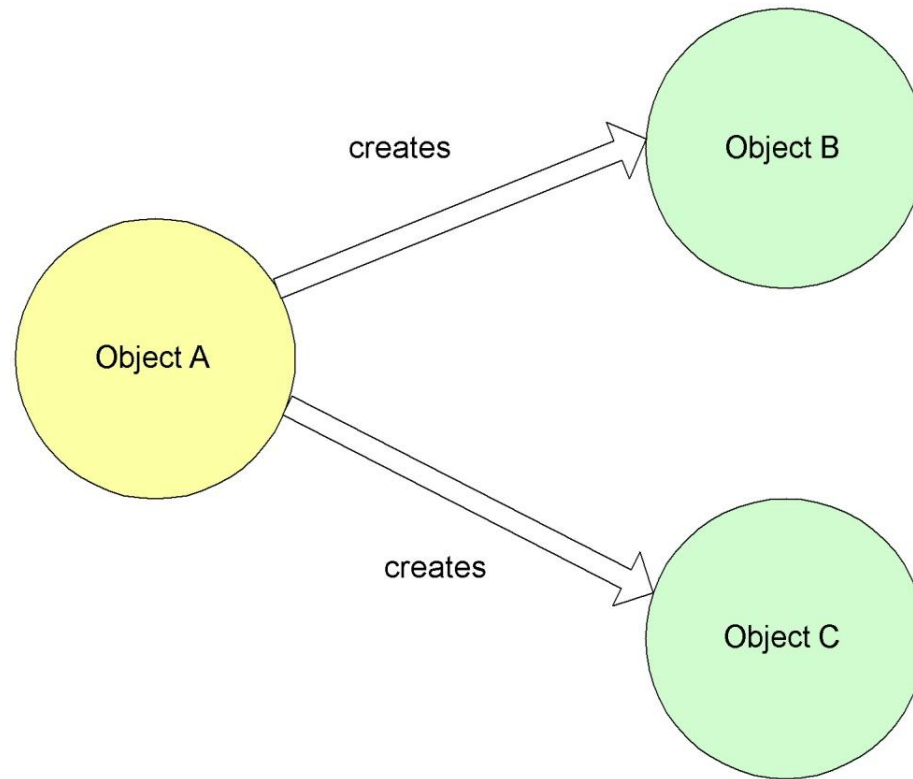
# Spring

- **Lightweight**—Spring is lightweight in terms of both size and overhead. The entire Spring framework can be distributed in a single JAR file that weighs in at just over 1 MB. And the processing overhead required by Spring is negligible. What's more, Spring is nonintrusive: objects in a Spring-enabled application typically have no dependencies on Spring specific classes.
- **Inversion of control**—Spring promotes loose coupling through a technique known as inversion of control (IoC). When IoC is applied, objects are passively given their dependencies instead of creating or looking for dependent objects for themselves.
- **Aspect-oriented** - Spring comes with rich support for aspect-oriented programming that enables cohesive development by separating application business logic from system services (such as auditing and transaction management). Application objects do what they're supposed to do—perform business logic—and nothing more. They are not responsible for (or even aware of) other system concerns, such as logging or transactional support.
- **Container** - Spring is a container in the sense that it contains and manages the life cycle and configuration of application objects. You can configure how your each of your beans should be created—either create one single instance of your bean or produce a new instance every time one is needed based on a configurable prototype—and how they should be associated with each other.
- **Framework** - Spring makes it possible to configure and compose complex applications from simpler components. In Spring, application objects are composed declaratively, typically in an XML file. Spring also provides much infrastructure functionality (transaction management, persistence framework integration, etc.), leaving the development of application logic to you.

# Dependency injection

- Inversion of Control (IoC)
- “Hollywood Principle”
  - Don't call me, I'll call you
- “Container” resolves (injects) dependencies of components by setting implementation object (push)
- As opposed to component instantiating or Service Locator pattern where component locates implementation (pull)
- Martin Fowler calls Dependency Injection

# Non-IoC / Dependency Injection

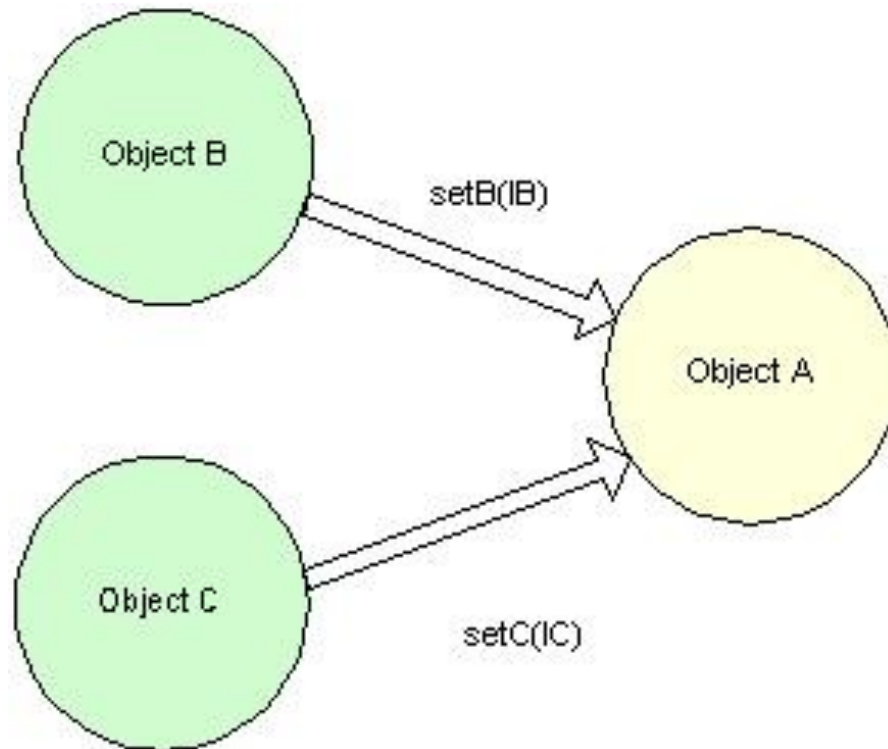




# Non-IoC Service Object

```
public class OrderServiceImpl implements IOrderService {  
    private IOrderDAO orderDAO = new OrderDaoImpl();  
  
    public Order saveOrder(Order order) throws OrderException{  
        try{  
            orderDao.saveOrder(order);  
        }catch(Exception e){  
            // handle e, rollback transaction, //cleanup, // throw e  
        }finally{  
            //Release resources and handle more exceptions  
        }  
    }  
}
```

# IoC / Dependency Injection



# IoC Service Object

```
public class OrderServiceImpl implements IOrderService {
    private IOrderDAO orderDAO ;

    public OrderServiceImpl (IOrderDAO orderDAO) {
        this.orderDAO = orderDAO;
    }

    public void setOrderDAO (IOrderDAO orderDAO) {
        this.orderDAO = orderDAO;
    }

    public Order saveOrder(Order order) throws OrderException{
        try{
            orderDao.saveOrder(order);
        }catch(Exception e){
            // handle e, rollback transaction, //cleanup, // throw e
        }finally{
            //Release resources and handle more exceptions
        }
    }
}
```

# Example. Printer

```
package org.lesson7.bean;

public interface IPrinter {
    void printMessage();
    void setMessage(String valueOf);
}
```

```
package org.lesson7.bean;

public class Printer implements IPrinter {

    private String message;

    public void setMessage(String message) {
        this.message = message;
    }

    public void printMessage() {
        System.out.println("Your Message : " + message);
    }
}
```

# Example. Container

```
package org.lesson7.bean;

public class Container {
    private IPrinter printer;
    private Double value;

    public IPrinter getPrinter() {
        return printer;
    }

    public void setPrinter(IPrinter printer) {
        this.printer = printer;
    }

    public void set(Double val) {
        this.value = val;
    }

    public void print() {
        printer.setMessage(String.valueOf(this.value));
        printer.printMessage();
    }
}
```

# Example. applicationContext.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="printer" class="org.lesson7.bean.Printer">
    <property name="message" value="Hello World!" />
  </bean>

  <bean id="container" class="org.lesson7.bean.Container">
    <property name="printer" ref="printer" />
  </bean>

</beans>
```

# Example. Launcher

```
public class Launcher {  
    public static void main(String[] args) {  
        ApplicationContext context = new  
        ClassPathXmlApplicationContext(  
            new String[] {"beans.xml"});  
        Printer bean = context.getBean("printer", Printer.class);  
        bean.printMessage();  
  
        Container container = context.getBean("container",  
            Container.class);  
        container.set(1234d);  
        container.print();  
  
        System.out.println(bean == container.getPrinter());  
    }  
}
```

# Example. Annotations (1)

Step 1:

```
@Service  
public class Printer implements IPrinter { ... }
```

```
@Service  
public class Container { ... }
```

Step 2.

```
@Service  
public class Container {  
  
    private IPrinter printer;  
    private Double value;  
  
    @Autowired  
    public void setPrinter(IPrinter printer) {  
        this.printer = printer;  
    }  
}
```



# Example. Annotations (2)

```
public static void main(String[] args) {  
    ApplicationContext context = new ClassPathXmlApplicationContext(  
        "beans-annot.xml");  
  
    Container container = context.getBean("container", Container.class);  
    container.set(1234d);  
    container.print();  
}
```

# Annotations

- **@Component** – common component
- **@Service** - service classes
- **@Controller** – controller classes
- **@Repository** – DAO classes

# @Required

This annotation simply indicates that the affected bean property must be populated at configuration time: either through an explicit property value in a bean definition or through autowiring. The container will throw an exception if the affected bean property has not been populated.

```
public class SimpleMovieLister {  
    private MovieFinder movieFinder;  
  
    @Required  
    public void setMovieFinder(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
}
```

# @Autowired

## 1. Field

@Autowired

```
private IPrinter printer;
```

## 2. Constructor

@Autowired

```
public Container(IPrinter printer) {  
    this.printer = printer;  
}
```

## 3. Setter

@Autowired

```
public void setPrinter(IPrinter printer) {  
    this.printer = printer;  
}
```

# @Autowired (2)

## 4. All beans of specific type

@Autowired

```
private IPrinter[] printer;
```

## 5. Well-known "resolvable dependencies"

@Autowired

```
private ApplicationContext context;
```

# @Qualifier

Since autowiring by type may lead to multiple candidates, it is often necessary to have more control over the selection process. One way to accomplish this is with Spring's **@Qualifier** annotation.

```
@Autowired
@Qualifier("main")
private MovieCatalog movieCatalog;
```

```
@Autowired
public void prepare(@Qualifier("main") MovieCatalog movieCatalog,
                   CustomerPreferenceDao customerPreferenceDao) {
    this.movieCatalog = movieCatalog;
    this.customerPreferenceDao = customerPreferenceDao;
}
```

# JSR-250 Annotations

Spring also provides support for Java EE 5 Common Annotations (JSR-250). The supported annotations are:

- `@Resource`
- `@PostConstruct`
- `@PreDestroy`

```
@Resource(name = "dataSource")  
public void createTemplate(DataSource dataSource) {  
    this.jdbcTemplate = new SimpleJdbcTemplate(dataSource);  
}
```

```
public class CachingMovieLister {  
    @PostConstruct  
    public void populateMovieCache() {... }  
  
    @PreDestroy  
    public void clearMovieCache() {... }  
}
```

# <bean>

Properties	Description
class	This attribute is mandatory and specify the bean class to be used to create the bean.
name	This attribute specifies the bean identifier uniquely. In XML-based configuration metadata, you use the id and/or name attributes to specify the bean identifier(s).
scope (@Scope)	This attribute specifies the scope of the objects created from a particular bean definition
constructor-arg	This is used to inject the dependencies
property	This is used to inject the dependencies
autowiring	This is used to inject the dependencies
lazy-init (@Lazy)	A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.
Init-method (@PostConstruct)	A callback to be called just after all necessary properties on the bean have been set by the container
destroy-method (@PreDestroy)	A callback to be used when the container containing the bean is destroyed



# Bean scopes

Scope	Description
<b>singleton</b>	Scopes a single bean definition to a single object instance per Spring IoC container. <b>(Default)</b>
<b>prototype</b>	Scopes a single bean definition to any number of object instances.
<b>request</b>	Scopes a single bean definition to the lifecycle of a single HTTP request; that is each and every HTTP request will have its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.
<b>session</b>	Scopes a single bean definition to the lifecycle of a HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.
<b>global session</b>	Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext.

# Spring AOP

Spring AOP (**Aspect-oriented programming**) framework is used to modularize cross-cutting concerns in aspects.

Put it simple, it's just an interceptor to intercept some processes, for example, when a method is execute, Spring AOP can hijack the executing method, and add extra functionality before or after the method execution.

# Spring AOP

In Spring AOP, comes with three very technical terms – **Advice**, **Pointcut**, **Advisor**:

- **Advice** – Indicate the action to take either before or after the method execution.
- **Pointcut** – Indicate which method should be intercept, by method name or regular expression pattern.
- **Advisor** – Group 'Advice' and 'Pointcut' into a single unit, and pass it to a proxy factory object.

# Spring AOP

In Spring AOP, 5 type of advices are supported :

- **Before advice** – Run before the method execution
- **After returning advice** – Run after the method returns a result
- **After throwing advice** – Run after the method throws an exception
- **After (finally) advice** – Run after normal or exceptional return
- **Around advice** – Run around the method execution, combine all three advices above.

# Spring AOP

- Beans.xml: `<aop:aspectj-autoproxy />`

```
@Component
@Aspect
public class TraceLogger {
    @Pointcut("execution(* org.lesson7.bean.*.print(..)")
    public void log() {
    }

    @Around("log()")
    public Object profile(ProceedingJoinPoint pjp) throws Throwable {
        MethodSignature signature = (MethodSignature) pjp.getSignature();
        Method method = signature.getMethod();
        long start = System.currentTimeMillis();
        System.out.println("Going to call the method: " + method.getName());

        Object output = pjp.proceed();

        System.out.println("Method execution completed.");
        long elapsedTime = System.currentTimeMillis() - start;
        System.out.println("Method execution time: " + elapsedTime
            + " milliseconds.");

        return output;
    }
}
```

# Spring AOP

```
@Before("execution(* org.lesson7.bean.Container.*(..))")  
public void logBefore(JoinPoint joinPoint) {  
    System.out.println("logBefore() is running!");  
}
```

```
@Before("log()")  
public void logBefore_2(JoinPoint joinPoint) {  
    System.out.println("logBefore_2() is running!");  
}  
}
```