Spring Framework

Rajashekhar P

ORACLE
Certified Professional
Java SE 6 Programmer

Agenda..

- Introduction to Spring
- Overview of Spring Architecture
- Spring IOC
- Basic Example (Spring Environment setup and Bean scopes)
- AOP Overview
- Spring MVC architecture
- Integration of Spring MVC with Hibernate
- Examples..

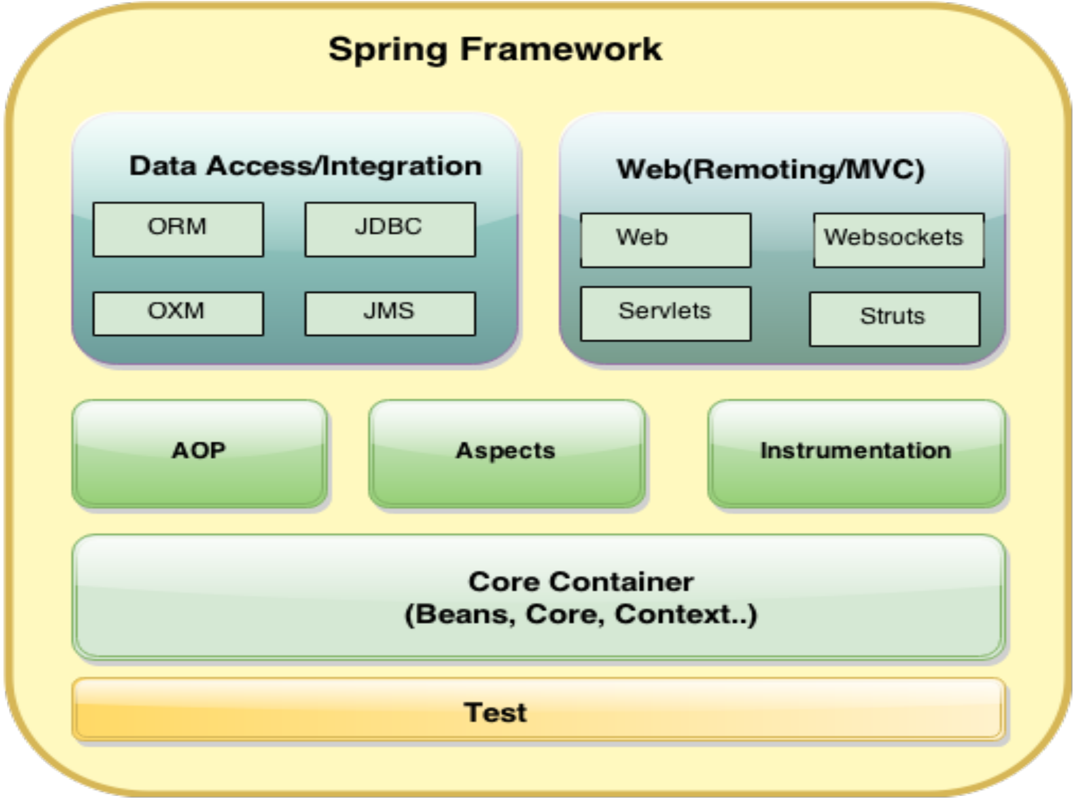
Introduction to Spring

- ➔ The **Spring Framework** is an open source application development **framework**
- ➔ Spring Provides comprehensive infrastructural support for developing enterprise Java applications very easily
- ➔ Spring is a lightweight framework. It can be thought of as a framework of frameworks because it provides support to various frameworks such as Struts, Hibernate, EJB, JSF etc.

Benefits of Using Spring Framework

- Develop enterprise-class applications using POJOs
- Spring makes use of some of the existing technologies like ORM frameworks, Logging frameworks, JEE, JDK Timers etc..
- Good web MVC framework
- Nice technical api exception handling
- Best Transaction Management
- Light weight comparing EJBs
- Provides good Testing support
- Good security and logging service etc..

Overview of Spring Architecture

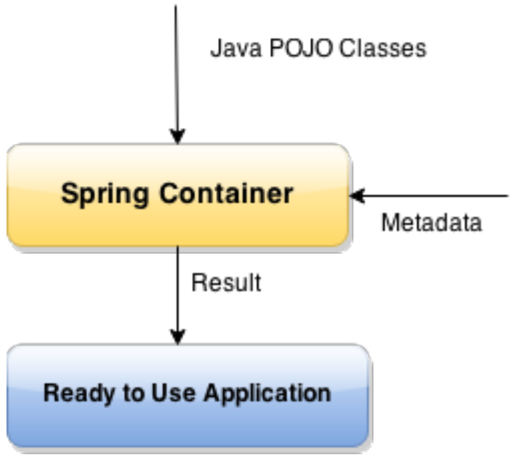


IOC Container

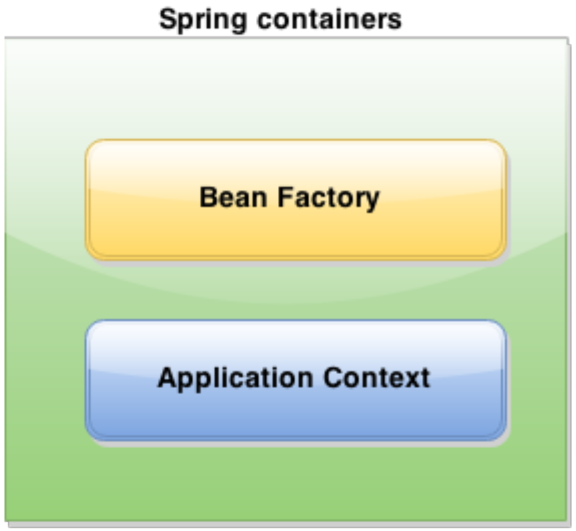
➔ The core of the Spring Framework is its **Inversion of Control** (Ioc) container.

The container will create the objects, wire them together, configure them, and manage their complete lifecycle from creation till destruction.

- ❖ To instantiate the class
- ❖ To configure the object
- ❖ To assemble the dependencies between the objects



continued..



```
1. Resource resource=new ClassPathResource
   ("applicationContext.xml");
2. BeanFactory factory=new XmlBeanFactory(resource);
```

```
ApplicationContext context =  new
ClassPathXmlApplicationContext("applicationContext.xml");
```

Dependency Injection

In Spring framework, Dependency Injection (DI) design pattern is used to define the object dependencies between each other.

Spring framework provides two ways to inject dependency

1. Setter Injection
2. Constructor Injection

Setter Injection

Constructor Injection

Bean Definition & Scopes

A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.

Basic bean scopes

- ➔ Singleton: Only once for spring container
- ➔ Prototype: New bean created with every request or reference.

Web-aware context bean scopes

- ➔ Request: New bean for servlet request
- ➔ Session: New bean for session
- ➔ Global session: New bean for global HTTP session

Possible Bean Definition

A class with no dependencies

```
<bean id="service" class="example.ServiceImpl"/>
```

Results in (via Reflection):

```
ServiceImpl service = new ServiceImpl();
```

ApplicationContext

service -> instance of ServiceImpl

Spring - Java Based Configuration

```
@Configuration
   public class HelloWorldConfig {
      @Bean
      public HelloWorld helloWorld(){
         return new HelloWorld();
      }
   }
```

```
<beans>
   <bean id="helloWorld" class="com.
  tutorialspoint.HelloWorld" />
</beans>
```

```
ApplicationContext factory=new ClassPathXmlApplicationContext("applicationContext.xml");
HelloWorld helloWorld = ctx.getBean(“helloWorld”);
```

```
ApplicationContext ctx = new AnnotationConfigApplicationContext(HelloWorldConfig.class);

HelloWorld helloWorld = ctx.getBean(HelloWorld.class);
```

Spring MVC

- ➔ Spring MVC is the web component of Spring's framework. It provides a rich functionality for building Web Applications.

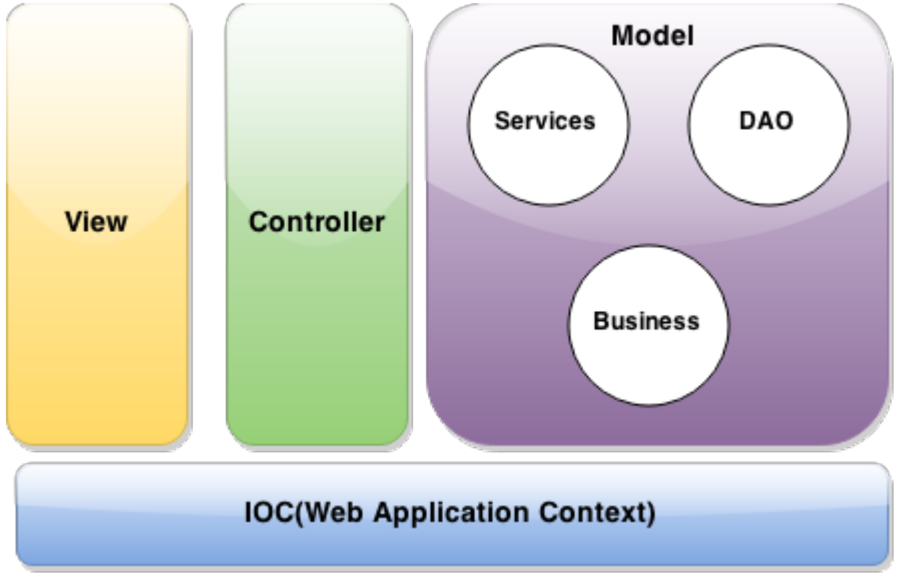
- ➔ It is a Java **Model-View-Controller (MVC)** web framework, which builds on the Spring Inversion of control(IoC) framework

- ➔ Spring MVC is designed around a DispatcherServlet that dispatches requests to handlers, with configurable handler mappings, view resolution, locale, time zone and theme resolution as well as support for uploading files.

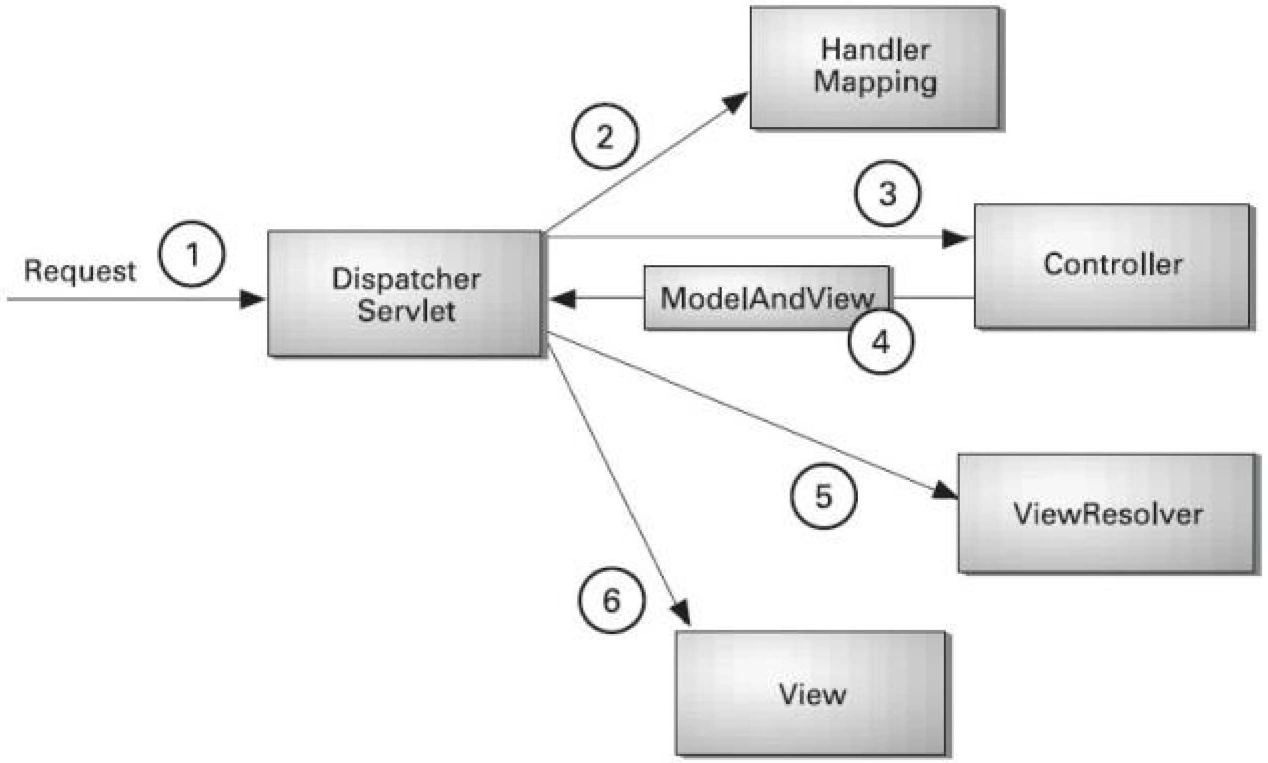
Spring MVC Supports

- Multi Actions
- Multi Form
- Form Backup
- Validation
- I18N
- Exception Handling
- View Resolvers
- Easly integrable

Spring MVC High level Architecture

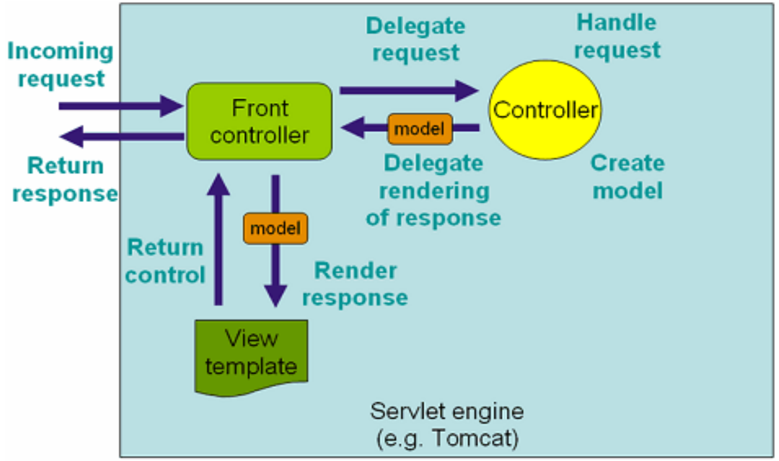


High level Spring Web MVC Flow



Request Processing Lifecycle

1. The client sends a request to web container in the form of http request.
2. This incoming request is intercepted by **Front controller** (DispatcherServlet) and it will then tries to find out appropriate **Handler Mappings**.
3. With the help of Handler Mappings, the DispatcherServlet will dispatch the request to appropriate Controller.
4. The Controller tries to process the request and returns the Model and View object in form of **ModelAndView** instance to the Front Controller.
5. The Front Controller then tries to resolve the View (which can be JSP, Freemarker, Velocity etc) by consulting the **View Resolver** object.
6. The selected view is then rendered back to client.



Configuring Spring MVC

The entry point of Spring MVC is the DispatcherServlet. DispatcherServlet is a normal servlet class which implements HttpServlet base class. Thus we need to configure it in web.xml

```
<web-app>
     <servlet>
        <servlet-name>example</servlet-name>
         <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>example</servlet-name>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>
</web-app>
```

Continued...

```
@Controller
public class HelloWorldController {

    @RequestMapping("/hello")
    public ModelAndView helloWorld(){
        String message="Hello Spring MVC how r u";
        return new ModelAndView("hellopage","message",message);
    }
}
```

```
  <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
     <property name="prefix" value="/WEB-INF/jsp/"></property>
     <property name="suffix" value=".jsp"></property>
  </bean>
```

Accessing Data

Most enterprise applications access data stored

in a relational database

– To carry out business functions

– To enforce business rules

Spring Data Access

Spring makes data access easier to do

effectively

– Manages resources for you

– Provides API helpers

– Supports all major data access technologies

- JDBC
- Hibernate
- JPA
- JDO
- iBatis

Spring - JDBC

- ➔ The Spring Framework takes care of all the low-level details that can make JDBC such a tedious API to develop with.
- ➔ Spring JDBC provides several approaches and correspondingly different classes to interface with the database.
- ➔ **JdbcTemplate Class** executes SQL queries, update statements and stored procedure calls, performs iteration over ResultSets and extraction of returned parameter values.

Configuring Data Source

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
   <property name="url" value="jdbc:mysql://localhost:3306/dbname"/>
   <property name="username" value="root"/>
   <property name="password" value="password"/>
</bean>
```

Spring with ORM

Spring provides API to easily integrate Spring with ORM frameworks such as Hibernate, JPA (Java Persistence API), JDO(Java Data Objects), Oracle Toplink.

Benefits of using the Spring Framework to create your ORM DAOs include:

- Less coding is required
- Integrated transaction management
- exception handling
- Easy to test

Spring Hibernate Example.. pseudo code

```
<bean id="sessionFactory"
        class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="annotatedClasses">
                <list>
                <value>org.cdac.domain.Employee</value>
                </list>
       </property>
        <property name="hibernateProperties">
                 <props>
                          <prop     key="hibernate.dialect">org.hibernate.dialect.
 MySQL5Dialect</prop>
                          <prop key="hibernate.show_sql">${hibernate.show_sql}</prop>
                 </props>
        </property>
    </bean>
```

Continued..

```
@Autowired
 SessionFactory sessionFactory;

 @Override
 @Transactional
  public int insertRow(Employee employee) {
       Session session = sessionFactory.openSession();
       Transaction tx = session.beginTransaction();
       session.saveOrUpdate(employee);
       tx.commit();
      return (Integer) id;
 }
```

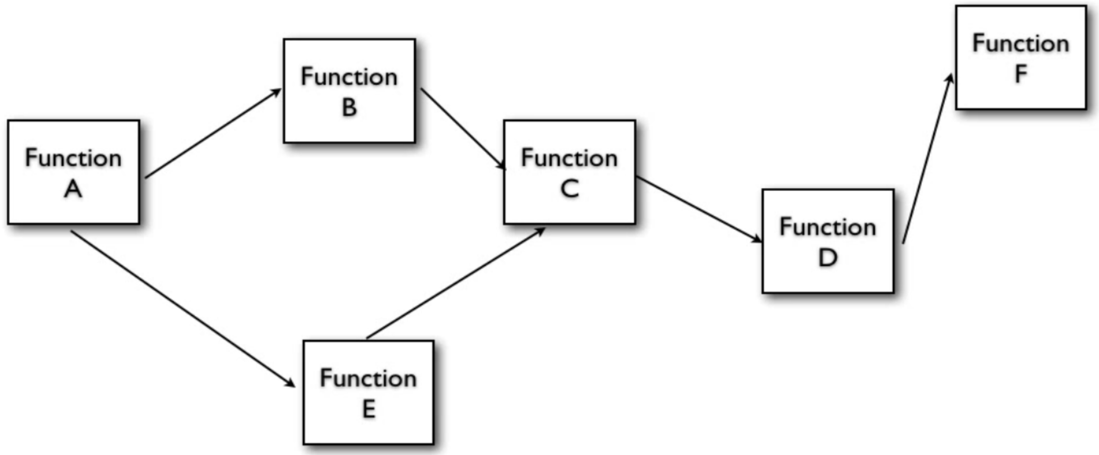
Aspect Oriented Programming (AOP)

Aspect Oriented Programming (AOP) complements OOPs in the sense that it also provides modularity. But the key unit of modularity is aspect than class.

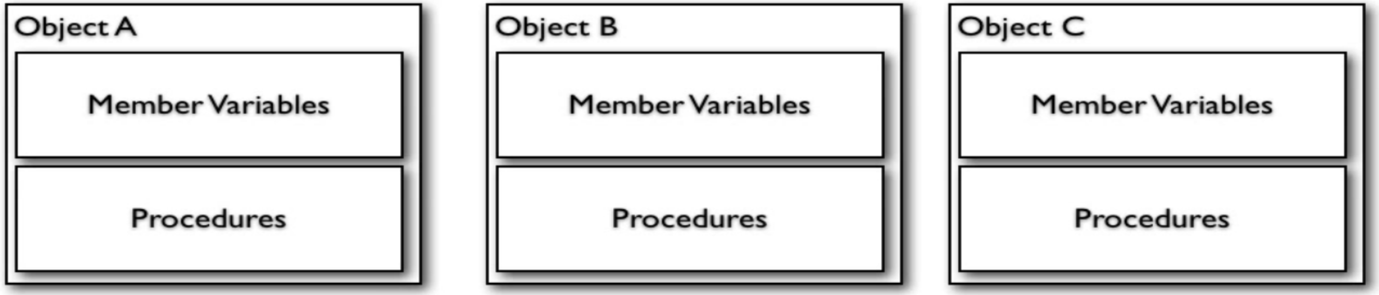
AOP breaks the program logic into distinct parts (called concerns). It is used to increase modularity by **cross-cutting concerns**.

A **cross-cutting concern** is a concern that can affect the whole application and should be centralized in one location in code as possible, such as transaction management, authentication, logging, security etc.

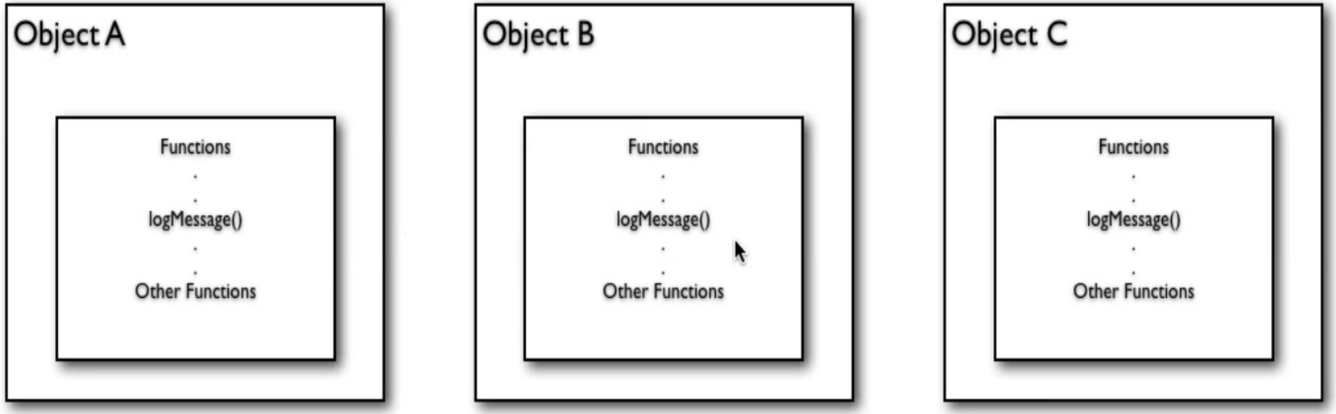
Functional Programming



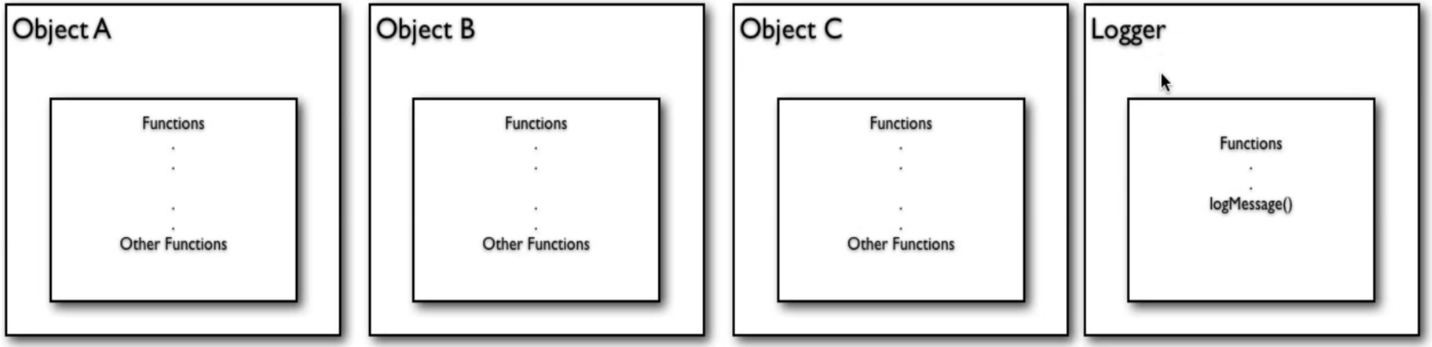
Object Oriented Programming



What about common procedures?



Separate object



Problems

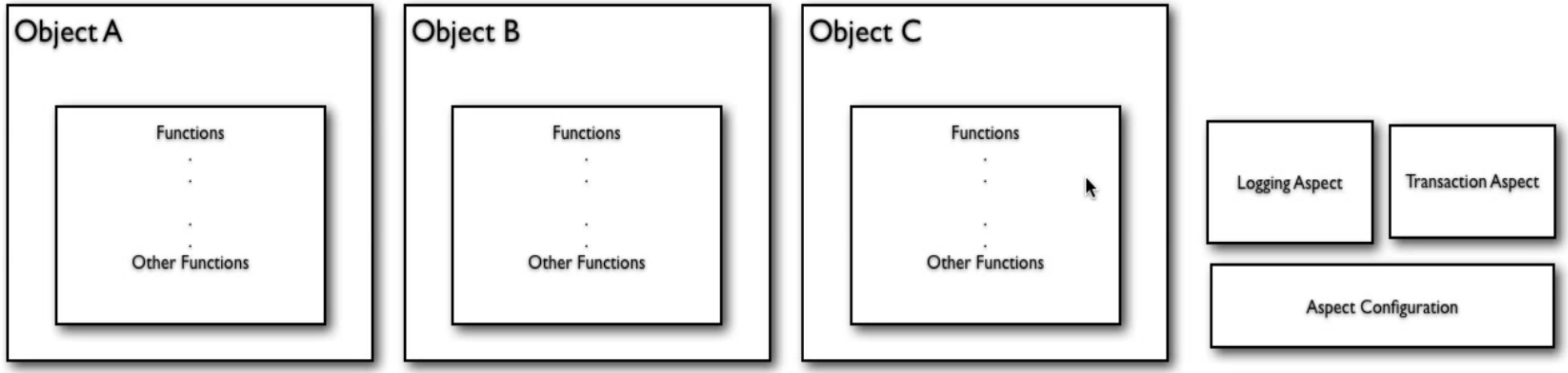
- To many relationships
- Code is still required in all methods
- We can not change all at once

Cross cutting concerns

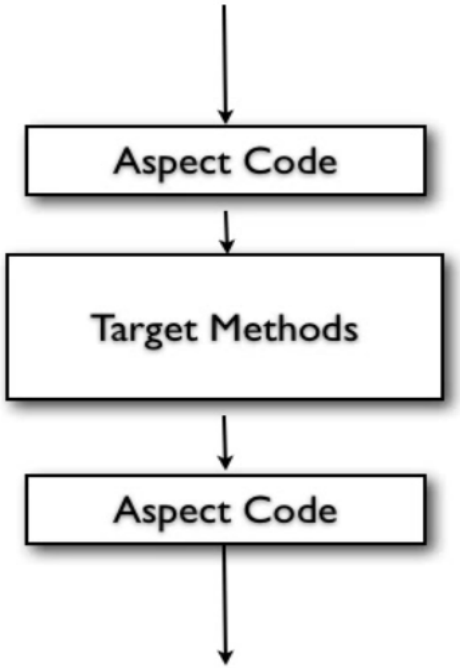
- Logging
- Transactions
- Security

solutions is aspect oriented programming..

Aspects


Object A
Functions
Other Functions
Object B
Functions
Other Functions
Object C
Functions
Other Functions
Logging Aspect
Transaction Aspect
Aspect Configuration

Aspects around the methods



Why use AOP?

It provides the pluggable way to dynamically add the additional concern before, after or around the actual logic. Suppose there are 10 methods in a class as given below:

```
class A {
public void m1(){...}
public void m2(){...}
public void m3(){...}
public void m4(){...}
public void m5(){...}
public void n1(){...}
public void n2(){...}
public void p1(){...}
public void p2(){...}
public void p3(){...}
}
```

There are 5 methods that starts from m, 2 methods that starts from n and 3 methods that starts from p.

Understanding Scenario I have to maintain log and send notification after calling methods that starts from m.

Problem without AOP We can call methods (that maintains log and sends notification) from the methods starting with m. In such scenario, we need to write the code in all the 5 methods.

But, if client says in future, I don't have to send notification, you need to change all the methods. It leads to the maintenance problem.

Solution with AOP We don't have to call methods from the method. Now we can define the additional concern like maintaining log, sending notification etc. in the method of a class. Its entry is given in the xml file.

In future, if client says to remove the notifier functionality, we need to change only in the xml file. So, maintenance is easy in AOP.

Where use AOP?

AOP is mostly used in following cases:

- ➔ To provide declarative enterprise services such as declarative transaction management.
- ➔ It allows users to implement custom aspects.

AOP Concepts and Terminology

AOP concepts and terminologies are as follows:

Join point

Join point is any point in your program such as method execution, exception handling, field access etc. Spring supports only method execution join point.

Advice

Advice represents an action taken by an aspect at a particular join point. There are different types of advices:

Before Advice: it executes before a join point.

After Returning Advice: it executes after a joint point completes normally.

After Throwing Advice: it executes if method exits by throwing an exception.

After (finally) Advice: it executes after a join point regardless of join point exit whether normally or exceptional return.

Around Advice: It executes before and after a join point.

Pointcut	It is an expression language of AOP that matches join points.
Introduction	It means introduction of additional method and fields for a type. It allows you to introduce new interface to any advised object.
Target Object	It is the object i.e. being advised by one or more aspects. It is also known as proxied object in spring because Spring AOP is implemented using runtime proxies.
Aspect	It is a class that contains advices, joinpoints etc.

Interceptor

It is an aspect that contains only one advice.

AOP Proxy

It is used to implement aspect contracts, created by AOP framework. It will be a JDK dynamic proxy or CGLIB proxy in spring framework.

Weaving

It is the process of linking aspect with other application types or objects to create an advised object. Weaving can be done at compile time, load time or runtime. Spring AOP performs weaving at runtime.

Example...

```
<bean id="doBeforeMethodBean" class="org.cdac.main.aop.DoBeforeMethod" />
```

```
<bean id="simpleServiceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="simpleServiceBean" />
        <property name="interceptorNames">
                <list>
                        <value>doBeforeMethodBean</value>
                </list>
        </property>
  </bean>
```

Continued..

```
public class DoBeforeMethod implements MethodBeforeAdvice
{
    public void before(Method method, Object[] args, Object target)
           throws Throwable {
          System.out.println("****SPRING AOP**** DoBeforeMethod : Executing before method!");
    }
}
```

References

- http://spring.io
- http://www.springbyexample.org/

THANK YOU

For further Queries reach me at rajashekharp@cdac.in