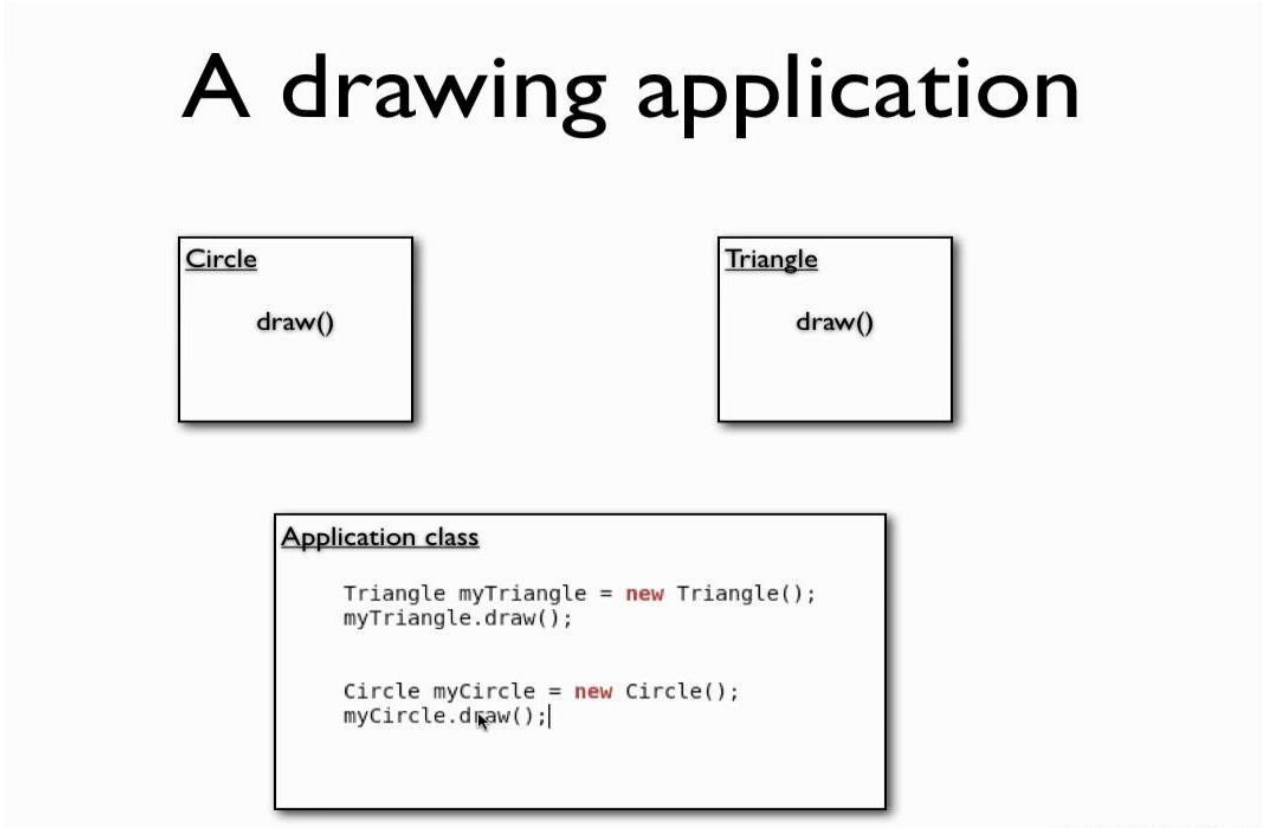
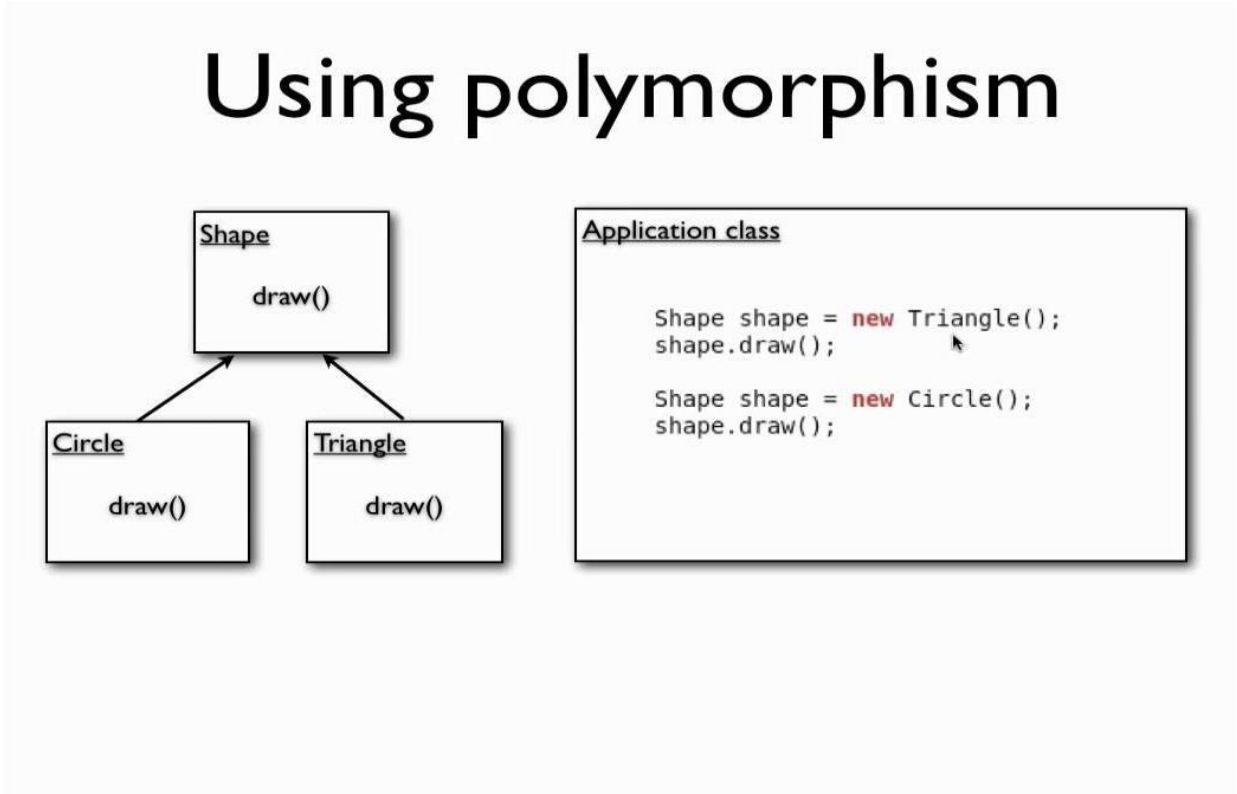
SPRING

DEPENDENCY INJECTION

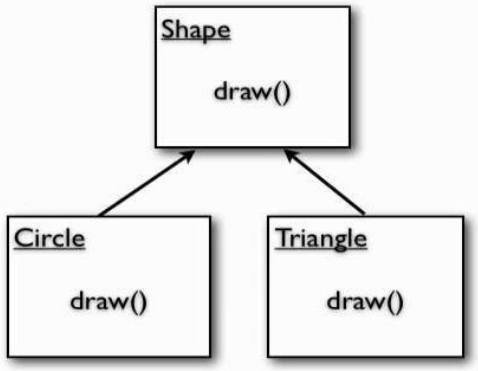


DEPENDENCY INJECTION



DEPENDENCY INJECTION

Method parameter



Application class

```
public void myDrawMethod(Shape shape) {
        shape.draw();
}
```

Somewhere else in the class

```
Shape shape = new Triangle();
myDrawMethod(shape);
```

DEPENDENCY INJECTION

Class member variable

Drawing class

Shape

draw()

Drawing Class

```
protected class Drawing {

        private Shape shape;

        public setShape(Shape shape) {
                this.shape = shape;
        }

        public drawShape() {
                this.shape.draw();
        }

}
```

Different class

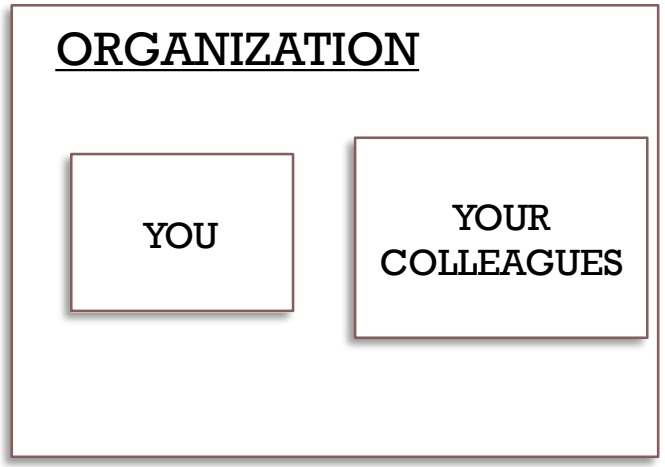
Triangle

draw()

Different class

```
Triangle myTriangle = new Triangle();
drawing.setShape(myTriangle);
drawing.drawShape();
```

DI - SCENARIO 1



Travel Planning Routine

- Decide the destination, and desired arrival date and time
- Call up the airline agency and convey the necessary information to obtain a flight booking.
- Call up the cab agency, request for a cab to be able to catch a particular flight from say your residence (the cab agency in turn might need to communicate with the airline agency to obtain the flight departure schedule, the airport, compute the distance between your residence and the airport and compute the appropriate time at which to have the cab reach your residence)
- Pickup the tickets, catch the cab and be on your way

- if your company suddenly changed the preferred agencies and their contact mechanisms, you would be subject to the following relearning scenarios
- The new agencies, and their new contact mechanisms (say the new agencies offer internet based services and the way to do the bookings is over the internet instead of over the phone)
- The typical conversational sequence through which the necessary bookings get done (Data instead of voice).
- Its not just you, but probably many of your colleagues would need to adjust themselves to the new scenario.This could lead to a substantial amount of time getting spent in the readjustment process.

DI - SCENARIO 2

- You have an administration department. Whenever you needed to travel an administration department interactive telephony system simply calls you up (which in turn is hooked up to the agencies).
- Over the phone you simply state the destination, desired arrival date and time by responding to a programmed set of questions.
- The flight reservations are made for you, the cab gets scheduled for the appropriate time, and the tickets get delivered to you.

DEPENDENCY INJECTION ?

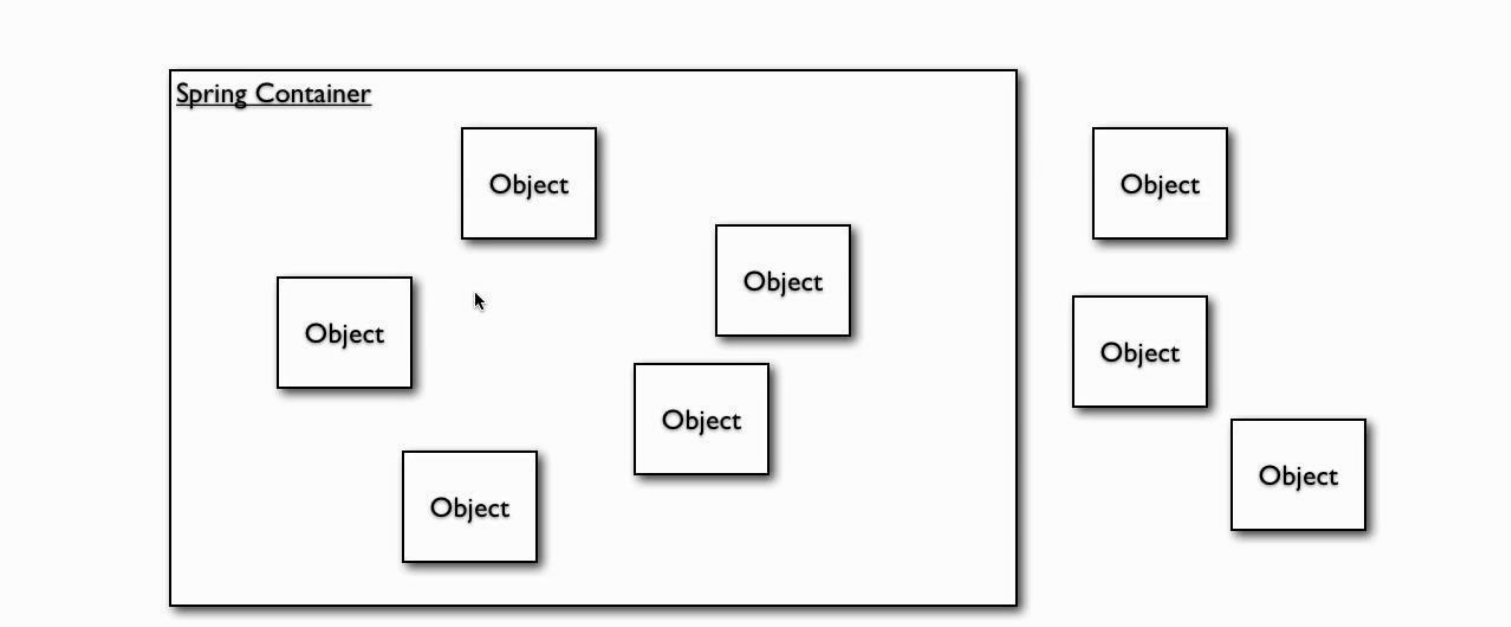
- In both the scenarios, you are the client and you are dependent upon the services provided by the agencies. However Scenario 2 has a few differences.
  You don't need to know the contact numbers / contact points of the agencies – the administration department calls you when necessary.
- You don't need to know the exact conversational sequence by which they conduct their activities (Voice / Data etc.) (though you are aware of a particular standardized conversational sequence with the administration department)
- The services you are dependent upon are provided to you in a manner that you do not need to readjust should the service providers change.

SPRING CORE

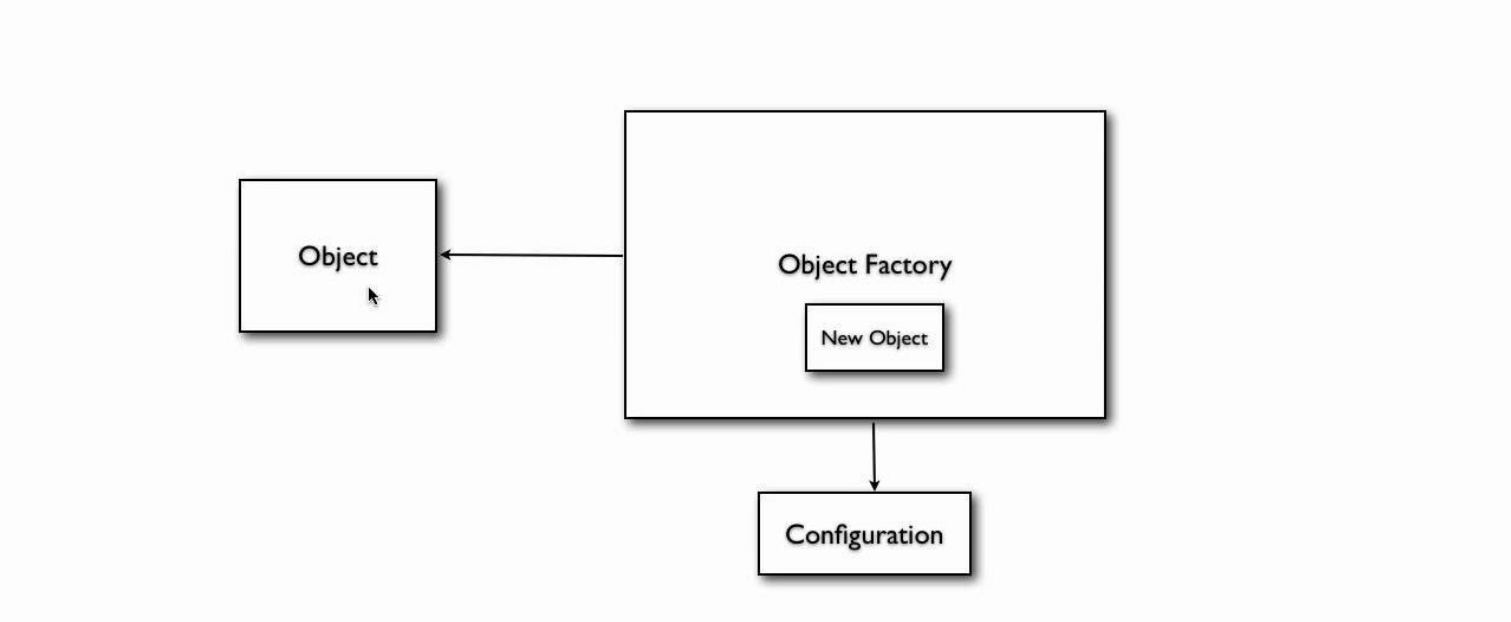
SPRING BEAN FACTORY

- A ***BeanFactory*** is like a factory class that contains a collection of beans.
- The ***BeanFactory*** holds Bean Definitions of multiple beans within itself and then instantiates the bean whenever asked for by clients.
- ***BeanFactory*** is able to create associations between collaborating objects as they are instantiated.This removes the burden of configuration from bean itself and the beans client.
- ***BeanFactory*** also takes part in the life cycle of a bean, making calls to custom initialization and destruction methods.

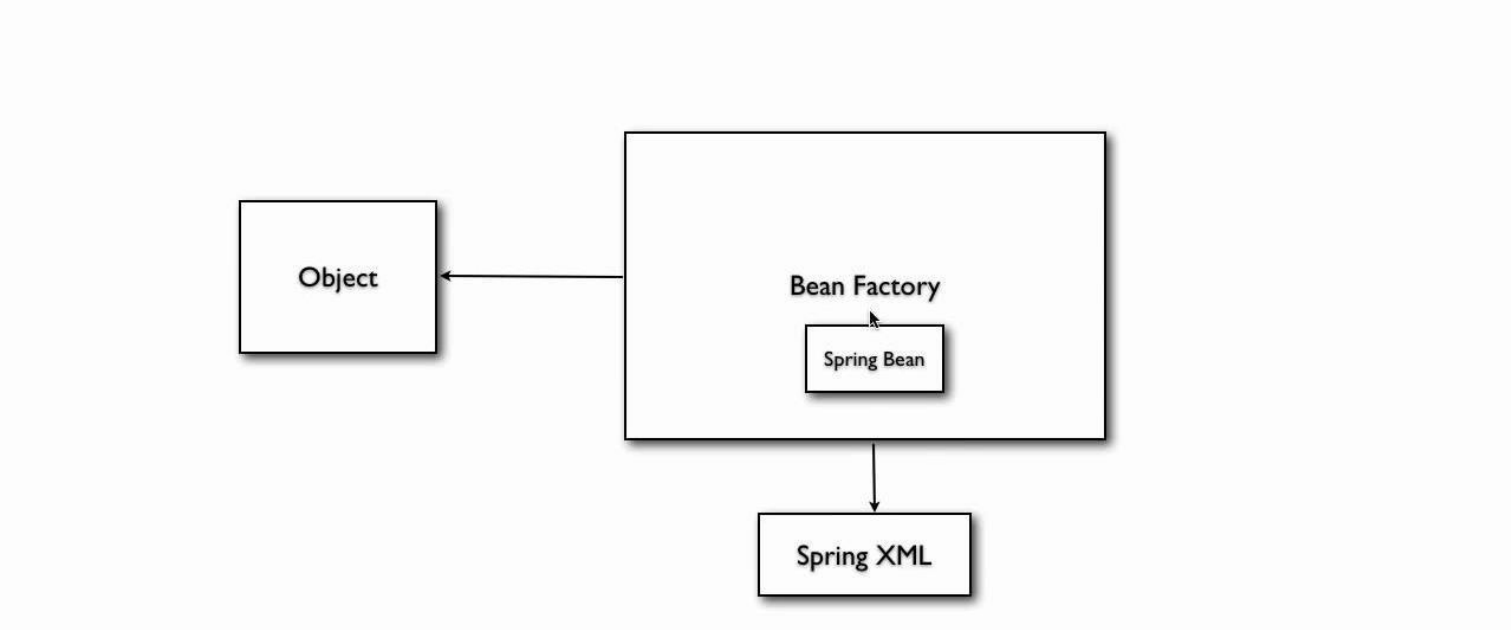
A SPRING CONTAINER



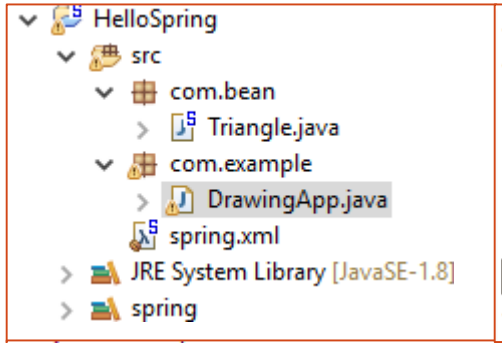
FACTORY PATTERN



SPRING BEAN FACTORY



EXAMPLE – SIMPLE BEAN




```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.springframew

    <bean id="triangle" class="com.bean.Triangle">
    </bean>

</beans>
```

```java
package com.bean;

public class Triangle {

    public void draw()
    {
        System.out.println(" Triangle drawn...");
    }
}
```

```java
public class DrawingApp {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        //Triangle myTriangle=new Triangle();
        BeanFactory factory=new XmlBeanFactory(new ClassPathResource("spring.xml"));
        Triangle myTriangle=(Triangle)factory.getBean("triangle");
        myTriangle.draw();
    }

}
```

APPLICATION CONTEXT

- **Application Context** Just like as Bean Factory's Big Brother with some additional functionality such as **AOP concept**, **event notification** and it adds more enterprise-specific functionality such as the ability to resolve **textual messages from a properties file** and the ability to **publish application events to interested event listeners**.
- This container is defined by the **org.springframework.context.ApplicationContext** interface.
- The **ApplicationContext** includes all functionality of the **BeanFactory**, it is generally recommended over the **BeanFactory**.
- **BeanFactory** can still be used for light weight applications like mobile devices or applet based applications.

EXAMPLE - APPLICATIONCONTEXT

SPRING LIFECYCLE

- The **init-method** attribute specifies a method that is to be called on the bean immediately upon instantiation
- Similarly, **destroy-method** specifies a method that is called just before a bean is removed from the container.

EXAMPLE – BEAN LIFECYCLE

INHERITANCE

- A child bean definition inherits configuration data from a parent definition.
- The child definition can override some values, or add others, as needed.
- Spring Bean definition inheritance has nothing to do with Java class inheritance but inheritance concept is same.
- You can define a parent bean definition as a template and other child beans can inherit required configuration from the parent bean.

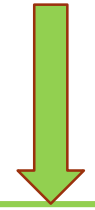
EXAMPLE – INHERITANCE WITHOUT ABSTRACT

```java
public class Triangle {
    private String type;
    private String color;
    public void draw() {
        System.out.println(this.type + " Triangle has drawn "
                + "with color "+this.color);
    }
    public String getType() {
        return type;
    }
    public void setType(String type) {
        this.type = type;
    }
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
}
```

```xml
<bean id="equilateral" class="com.bean.Triangle">
    <property name="type" value="Equilateral"></property>
</bean>
<bean id="redTriangle" parent="equilateral">
    <property name="color" value="Red"></property>
</bean>
<bean id="greenTriangle" parent="equilateral">
    <property name="color" value="Green"></property>
</bean>
```

```java
AbstractApplicationContext context=new ClassPathXmlApplicationContext("spring.xml")
Triangle myTriangle=(Triangle)context.getBean("redTriangle");
myTriangle.draw();
```

EXAMPLE – INHERITANCE WITH ABSTRACT



```java
public class Triangle {
    private String type;
    private String color;
    public void draw() {
        System.out.println(this.type + " Triangle has drawn "
                + "with color "+this.color);
    }
    public String getType() {
        return type;
    }
    public void setType(String type) {
        this.type = type;
    }
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
}
```

```xml
<bean id="equilateral" class="com.bean.Triangle" abstract="true">
    <property name="type" value="Equilateral"></property>
</bean>
<bean id="redTriangle" parent="equilateral">
    <property name="color" value="Red"></property>
</bean>
<bean id="greenTriangle" parent="equilateral">
    <property name="color" value="Green"></property>
</bean>
```

```java
AbstractApplicationContext context=new ClassPathXmlApplicationContext("spring.xml")
Triangle myTriangle=(Triangle)context.getBean("redTriangle");
myTriangle.draw();
```

BEAN SCOPE

- To force Spring to produce a new bean instance each time one is needed, you should declare the bean's scope attribute to be **prototype**
- Similar way if you want Spring to return the same bean instance each time one is needed, you should declare the bean's scope attribute to be **singleton.**
- If you don’t declare any scope attribute then it is **singleton** by default.
- There are other three scopes **(request, session and global-session)** which are only available if you use web-aware **ApplicationContext.**

SINGLETON SCOPE

```
<bean id="triangle" class="com.bean.Triangle"
      init-method="init" destroy-method="destroy">
    <property name="type" value="Equilateral"></property>
</bean>
```

```
AbstractApplicationContext context=new ClassPathXmlApplicationContext("spring.xml")
Triangle myTriangle=(Triangle)context.getBean("triangle");
myTriangle.draw();
Triangle triangle2=(Triangle)context.getBean("triangle");
triangle2.draw();
```

```
Initialize the Bean
Equilateral Triangle has drawn.
Equilateral Triangle has drawn.
```

output

PROTOTYPE SCOPE

```
<bean id="triangle" class="com.bean.Triangle"
      init-method="init" destroy-method="destroy" scope="prototype">
    <property name="type" value="Equilateral"></property>
</bean>
```

```
AbstractApplicationContext context=new ClassPathXmlApplicationContext("spring.xml")
Triangle myTriangle=(Triangle)context.getBean("triangle");
myTriangle.draw();
Triangle triangle2=(Triangle)context.getBean("triangle");
triangle2.draw();
```

```
Initialize the Bean
Equilateral Triangle has drawn.
Initialize the Bean
Equilateral Triangle has drawn.
```

output

SETTER AND CONSTRUCTOR INJECTION

- In Spring Dependency Injection (DI) design pattern is used to define the object dependencies between each other.
- There are two types in **dependency-injection**:

1. **The constructor-injection type and**
2. **The setter-injection type .**

- Historically the **setter-injection** type come from spring, whereas **constructor-injection** type are from **PicoContainer** and **Google GUICE** (***Googles lightweight dependency injection framework***).

CONSTRUCTOR V/S SETTER INJECTION

- Setter injection gets preference over constructor when both are specified
- Constructor injection can not partially initialize values
- Circular dependency can be achieved by setter injection
- Security is lesser in setter injection as it can be overridden
- Constructor injection fully ensures dependency injection but setter injection does not
- Setter injection is more readable

INJECTION – CONSTRUCTOR BASED

- The basic idea with **constructor-injection** is that the object has no defaults and instead you have a single constructor where all of the collaborators and values need to be supplied before you can instantiate the object

EXAMPLE – CONSTRUCTOR INJECTION

```java
public class Triangle {
    private String type;
    private int height;
    private Color color;
    public Triangle(String type) {
        this.type=type;
    }
    public Triangle(int height) {
        this.height=height;
    }
    public Triangle(String type,int height, Color color) {
        this.type=type;
        this.height=height;
        this.color=color;
    }
    public Color getColor() {
        return color;
    }
    public void setColor(Color color) {
        this.color = color;
    }
    public void draw()  {
        System.out.println(this.type+" Triangle has drawn of height "+this.height
                +" with color "+color.getColor());
    }
}
```

```xml
<bean id="triangle1" class="com.bean.Triangle">
    <constructor-arg name="type" value="Equilateral"></constructor-arg>
</bean>

<bean id="triangle2" class="com.bean.Triangle">
    <constructor-arg name="height" value="20" type="int"></constructor-arg>
</bean>
```

```xml
<bean id="triangle" class="com.bean.Triangle">
    <constructor-arg name="type" value="Equilateral"></constructor-arg>
    <constructor-arg name="height" value="20" type="int"></constructor-arg>
    <constructor-arg name="color">
        <bean id="color" class="com.bean.Color">
            <property name="color" value="Red"></property>
        </bean>
    </constructor-arg>
</bean>
```

DI – GETTER SETTER BASED

```java
public class Shape {
    private String color;
    private Triangle triangle;
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public Triangle getTriangle() {
        return triangle;
    }
    public void setTriangle(Triangle triangle) {
        this.triangle = triangle;
    }
}
```

```xml
<bean id="triangle" class="com.bean.Triangle">
    <property name="type" value="Equilateral"></property>
</bean>
<bean id="shape" class="com.bean.Shape">
    <property name="color" value="Red"></property>
    <property name="triangle" ref="triangle"></property>
</bean>
```

```java
ApplicationContext context=new ClassPathXmlApplicationContext("spring.xml");
Shape shape=context.getBean("shape",Shape.class);
shape.getTriangle().draw();
System.out.println("Color is : "+shape.getColor());
```

COLLECTION AND REFERENCES

Element	Description
<list>	This helps in wiring ie injecting a list of values, allowing duplicates.
<set>	This helps in wiring a set of values but without any duplicates.
<map>	This can be used to inject a collection of name-value pairs where name and value can be of any type.
<props>	This can be used to inject a collection of name-value pairs where the name and value are both Strings

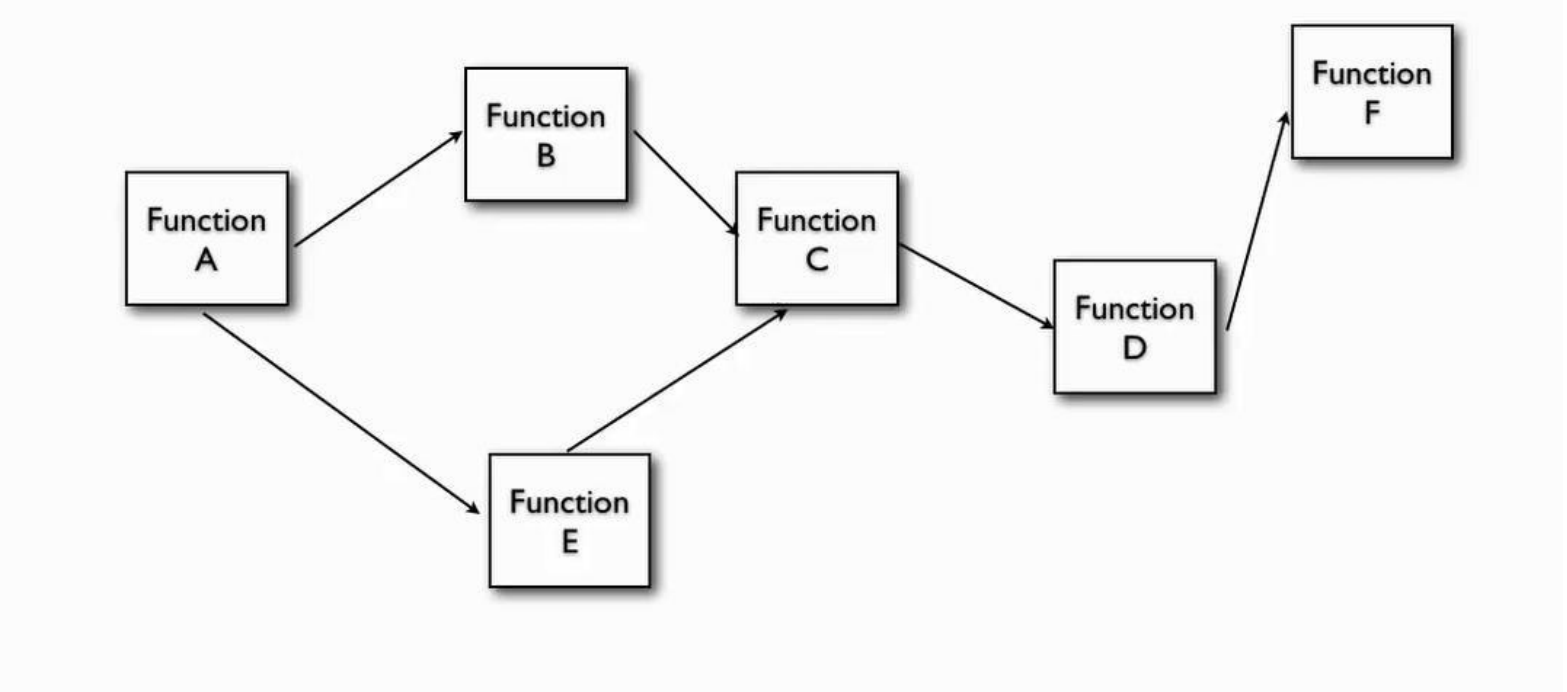
AUTO-WIRING

- **Autowiring can't be used to inject primitive and string values.**

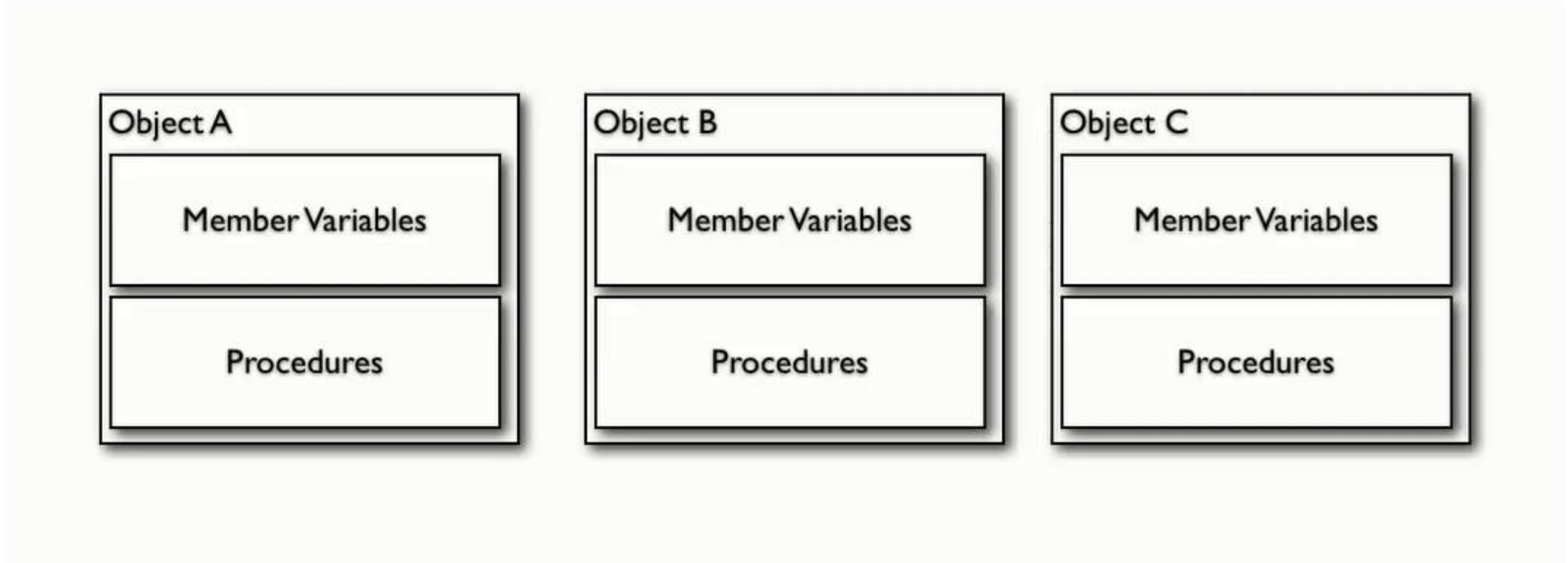
Mode	Description
No	It is the default autowiring mode. It means no autowiring bydefault.
byName	The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. It internally calls setter method.
byType	The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method.
Constructor	The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.
Autodetect	Spring first tries to wire using autowirebyconstructor, if it does not work, Spring tries to autowirebybyType. It is deprecated since Spring 3

ASPECT ORIENTED PROGRAMMING

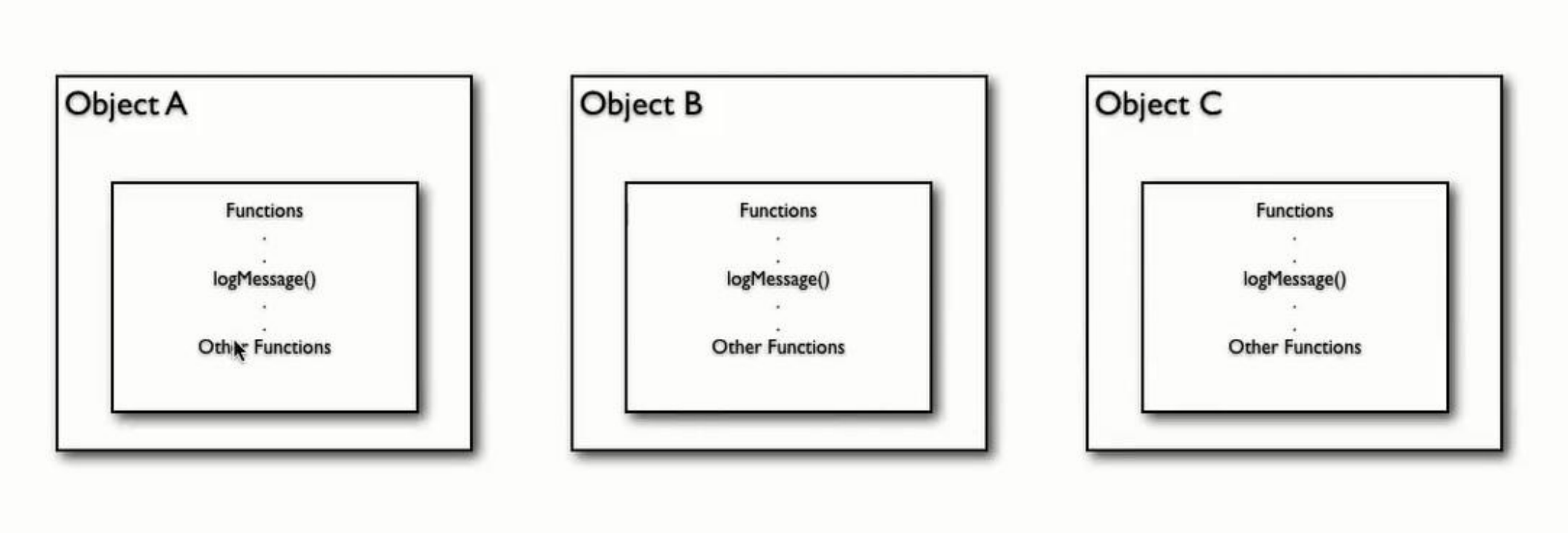
FUNCTIONAL PROGRAMMING



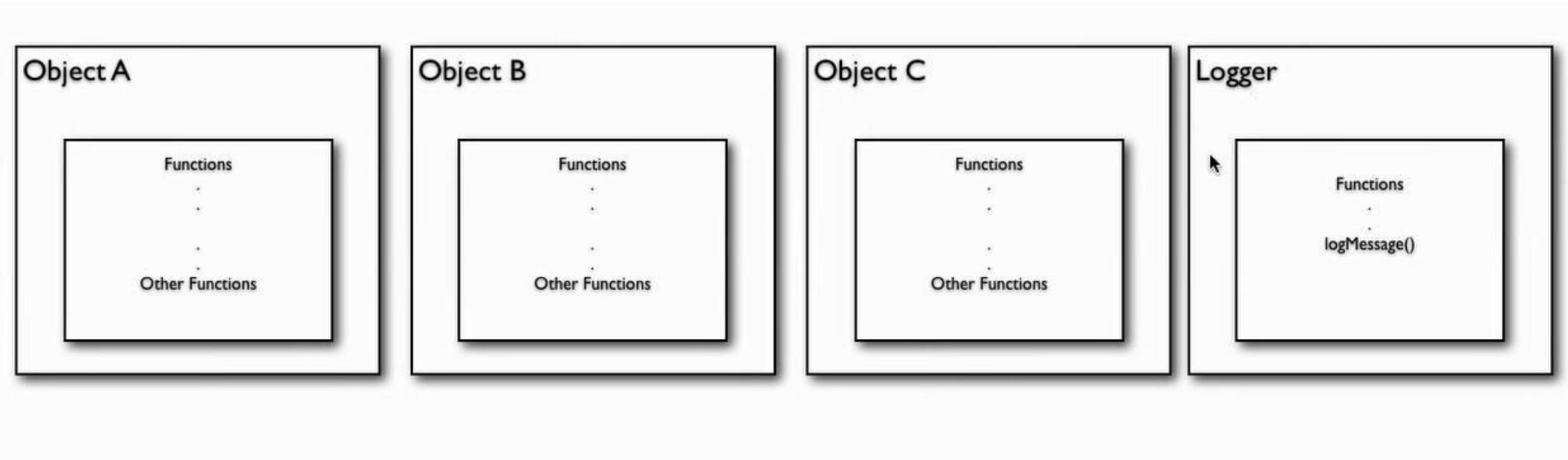
OBJECT ORIENTED PROGRAMMING



WHAT ABOUT COMMON PROCEDURES?



SEPARATE OBJECT



PROBLEMS

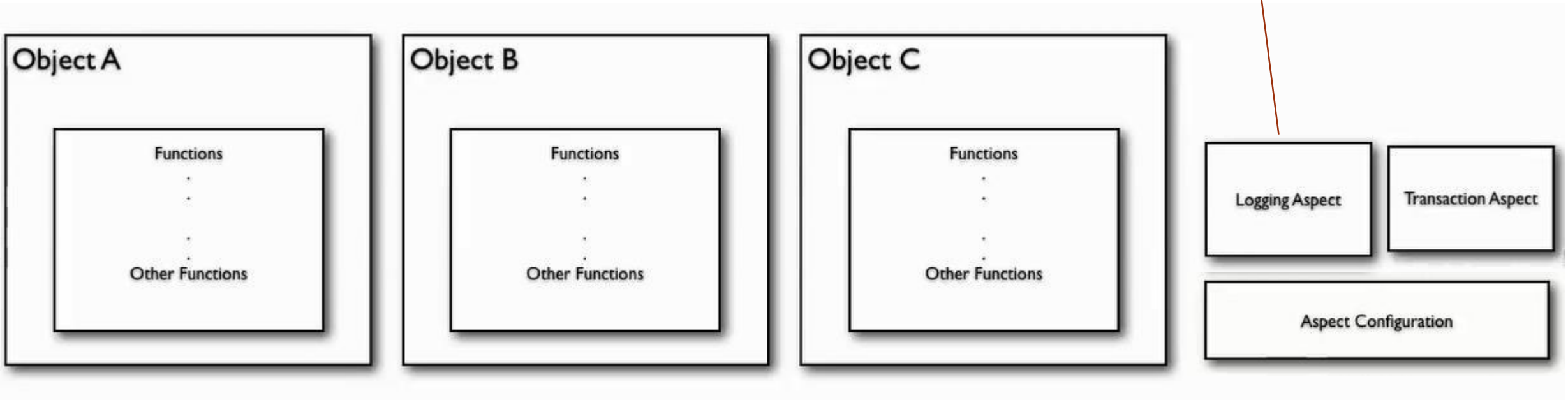
- Too many relationships to the crosscutting objects
  - Crosscutting objects that concerns different other objects in your problem domain
- Code is still required in all methods
- Cannot all be changed at once

CROSS CUTTING CONCERNS

- Logging
- Transactions
- Security

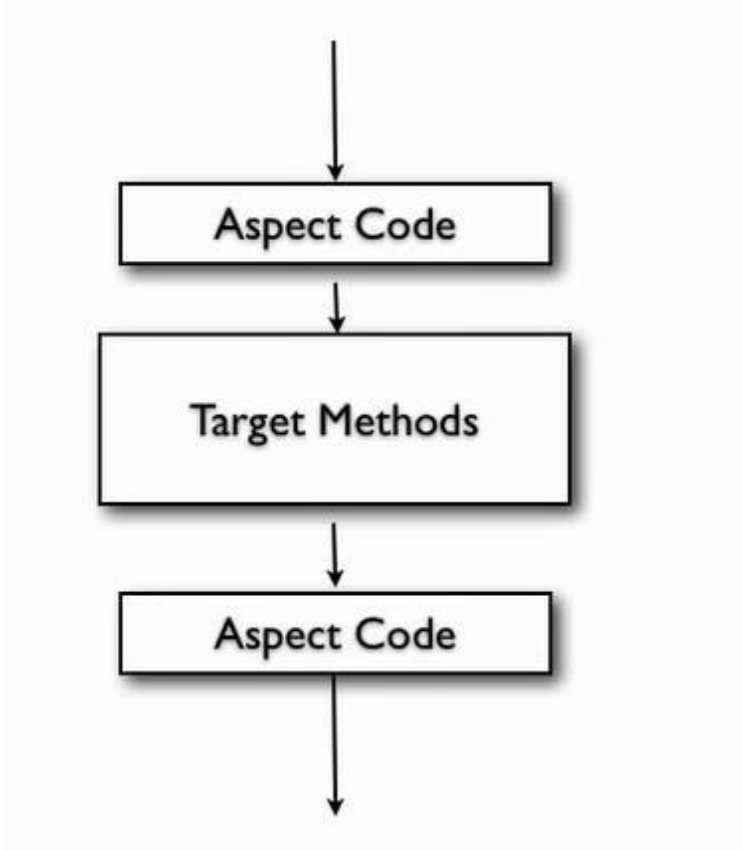
ASPECTS

- Aspect is Special class which some special privileges
- You can have as many Aspect as you want
- You don't reference Aspect class inside your object A, B and C
- Instead of creating reference we will have Aspect Configuration to define which method to call and when to call



- We can have three methods inside Object A,Two methods inside Object B and One methods inside Object C for which I want to set Aspects
- I want to invoke Aspect methods before all those method the I can configure that inside Aspect Configuration file
- We have other examples also which are similar to Aspects like we can think of Servlet filter which are invoke before Servlet in Aspect before is also working for the same
- We can also think of trigger in database which is also similar to Aspects

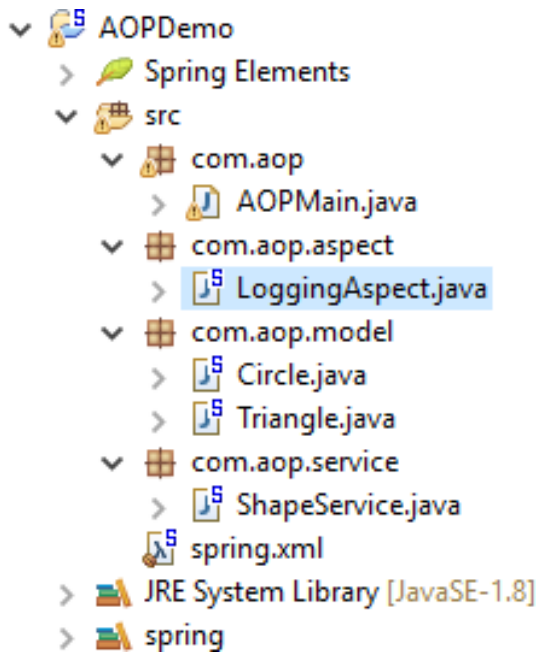
WRAPPING ASPECTS AROUND METHODS



STEPS IN AOP

- Write Aspects – we need to identify which are crosscutting concerns
- Configure where the aspects apply - for example we have logging() method then when to execute that method
- libraries required to do Aspect Oriented Programming
  - **aspectjrt.jar**
  - **aspectjweaver.jar**
  - **aopalliance.jar**
  - **Cglib.jar**
  - **Asm.jar**

FIRST ASPECT PROGRAM




```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"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-4.2.xsd">

    <!-- this indicates that we are using Aspect Oriented Programming -->
    <aop:aspectj-autoproxy/>

    <bean name="triangle" class="com.aop.model.Triangle">
        <property name="name" value="Triangle Name"/>
    </bean>
    <bean name="circle" class="com.aop.model.Circle">
        <property name="name" value="Circle Name"/>
    </bean>
    <bean name="shapeService" class="com.aop.service.ShapeService"
            autowire="byName"/>
    <bean name="loggingAspect" class="com.aop.aspect.LoggingAspect"/>
</beans>
```

FIRST ASPECT PROGRAM

```java
public class Triangle {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

```java
public class Circle {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

```java
public class ShapeService {
    private Circle circle;
    private Triangle triangle;
    public Circle getCircle() {
        return circle;
    }
    public void setCircle(Circle circle) {
        this.circle = circle;
    }
    public Triangle getTriangle() {
        return triangle;
    }
    public void setTriangle(Triangle triangle) {
        this.triangle = triangle;
    }

}
```

FIRST ASPECT PROGRAM

The Advice will executed for any matching getName() method not specific to any Circle Class or Triangle Class

Advice is standard terminology in AOP for cross cutting concerns inside Aspects

```
@Aspect
public class LoggingAspect {

    @Before("execution(public String getName())")
    public void LoggingAdvice()
    {
        System.out.println("Advice run. Get Method Called.");
    }
}

public class AOPMain {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ApplicationContext context=new ClassPathXmlApplicationContext("spring.xml");
        ShapeService shapeService=context.getBean("shapeService",ShapeService.class);
        System.out.println(shapeService.getCircle().getName());
    }
}
```

WILDCARD EXPRESSIONS

- In order to run LoggingAdvice only for Circle class getName() method we have write fully qualified class name before method name inside Aspect

```
@Before("execution(public String com.aop.model.Circle.getName())")
public void LoggingAdvice()
{
    System.out.println("Advice run. Get Method Called.");
}
```

SINGLE ADVICE APPLIED TO MULTIPLE METHODS

- If I want to apply the LoggingAdvice before all getter methods then as well as their return type may be anything,

* Is wild card character

```
@Before("execution(public * get*())")
public void LoggingAdvice()
{
    System.out.println("Advice run. Get Method Called.");
}
```

```
public class AOPMain {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ApplicationContext context=new ClassPathXmlApplicationContext("spring.xml");
        ShapeService shapeService=context.getBean("shapeService",ShapeService.class);
        System.out.println(shapeService.getCircle().getName());
    }
}
```

Here LoggingAdvice will invoke two times one is for getCircle() and second is for getName()

SOME OTHER PLACE TO USE WILDCARDS

- Here we have removed **public** modifier and also use wild card characters in side method parameters.
- **LoggingAdvice** will called irrespective to how many arguments are there

```
@Before("execution(* get*(*))")
public void LoggingAdvice()
{
    System.out.println("Advice run. Get Method Called.");
}
```

- If we want to call without argument as well as with argument get method :
- Dot dot means it can be zero arguments or it can be any number of arguments

```
@Before("execution(* get*(..))")
public void LoggingAdvice()
{
    System.out.println("Advice run. Get Method Called.");
}
```

```
@Before("execution(* com.aop.model.*get*(..))")
public void LoggingAdvice()
{
    System.out.println("Advice run. Get Method Called.");
}
```

POINTCUT

- Lot of advice method for single method then you have to repeat code ?
- Pointcut is AOP terminology for all the points in execution of code where you want advice method to cut in
- point where the method cut in is point cut

```
@Before("execution(* get*(*))")
public void LoggingAdvice()
{
    System.out.println("Advice run. Get Method Called.");
}
```

Execution(* get*(*))
Is pointcut for LoggingAdvice()

MULITPLE ADVICE FOR SINGLE METHOD

```java
@Aspect
public class LoggingAspect {

    @Before("allGetters()")
    public void LoggingAdvice()
    {
        System.out.println("Advice run. Get Method Called.");
    }

    @Before("allGetters()")
    public void secondAdvice()
    {
        System.out.println("Second Advice executed.");
    }

    @Pointcut("execution(* get*())")
    public void allGetters(){}
}
```

JDBC WITH SPRING

- Spring Data
- spring has jdbc module
- spring also provide orm module

JDBC WITHOUT SPRING

- step 1:
- model class
- step 2:
- JdbcDaoImp
- step 3:
- create method for select data
- load dirver
- connection object
- select qry
- execute the query
- etc...

- adding spring and data source configuration
- - add spring library
- - add spring xml
- <context:annotation-config/>
- <context:component-scan base-package="org.tops.demo"/>
- this tags for annotations
- @Component => to identify the bean the class as bean
- -> connecting to database is not going to change so this

- spring provides some support class for this connection
- jdbc inside jdbc library/ datasource / drivermanagerdatasource.class
- this class we declare inside spring.xml
- <bean id="datasource"
  class="org.springframework.jdbc.datasource.DriverManagerDataSource">
- <property name="driverClassName" value="mysql.jdbc.DriverManager"/>
- <property name="url" value="jdbc:mysql://localhost:3306/db"/>
- <property name="username" value="root"/>
- <property name="password" value=""/>
- </bean>

- - we can use that inside our DAO class
- @Autowired
- private DataSource datasource;
- getter and setter
- - we don't need to specity driver, username, password, etc.
- we directly get Connection from that dataSource object
- conn=dataSource.getConnection();

JDBCTEMPLATE

- Query is changed from method to method
- executeQuery and get ResultSet then looping thorough it then close resultSet and preparedset will be same
- Jdbctemplate is class provided in spring which have method beforeQueryExecution() and afterQueryExecution()
- JdbcTemplate inside core package

QUERY RETURNING INT VALUE

```
class JdbcDaoImpl{
private JdbcTemplate jdbcTemplate=new JdbcTemplate();
getter and setter
        public int getCircleCount()
        {
                String sql="select count(*) from Circle";
                jdbcTemplate.setDataSource(getDataSource());
                return jdbcTemplate.queryForInt(sql);
        }
}
```

- instead of crating jdbcTemplate and assign datasource to it
- we can also create bean for jdbctemplate
- second way is inside setter method of datasource

```
@Autowired
public void setDataSource(DataSource dataSource)
{
this.jdbcTemplate=new JdbcTemplate(dataSource);
}
```

QUERY RETURNING STRING

- Other datatypes from jdbctemplate

```
public String getCircleName(int circleId){
        String sql="select name from Circle where id=?";
        return jdbcTemplate.queryForObject(sql,new
Object[]{circleId},String.class);
}
```

IMPLEMENTING ROWMAPPER

- if query return Circle object data
- jdbc does not know custom class like circle
- we need to write code for mapping also
- RowMapper is class which provide call back method fro mapping

```
public Circle getCircleForId(int id){

        String sql="select * from circle where id=?";

         return jdbcTemplate.queryForObject( sql, new Object[]{circleId},new CircleMapper())

}
```

- make innerclass inside dao class

```
private static final class CircleMapper implements RowMapper<Circle>{

mapRow(ResultSet resultSet, int position)

{

Circle circle=new Circle();

circle.setId(resultSet.getInt("id"));

circle.setNAme(resultSet.getString("Name"));

return circle;

}

}
```

QUERY RETURNING LIST OF OBJECTS

```
public List<Circle> getAllCircles()
{
        String sql="select * from Circle";
        return jbbcTemplate.query(sql,new CircleMapper());
}
```

WRITE OPERATIONS WITH JDBCTEMPLATE

```
public void insertCircle(Circle circle)
{
String sql="insert into Circle(id,name) values (?,?)";
jdbcTemplate.update(sql,new Object[]{circle.getId(),circle.getName()});
}
```

- **update method run insert, update, delete and also stored procedure**

RUN DDL STATEMENTS

- **execute method to run DDL statements**

```
public void createTriangleTable(){
        String sql="create table triangle(id integer,name varchar(50)";
        jdbcTemplate.execute(sql);
}
```

NAMEDPARAMJDBCTEMPLATE

- jdbcTemplate limitation it can only have ? parameters
- jdbc.core.namedparam package provide NamedParamJdbcTemplate class which can be used for named parameter

```
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;
setDataSource(DataSource dataSource){
        this.namedParameterJdbcTemplate=new NamedParameterJdbcTemplate(dataSource);
}
public void insertCircle(Circle circle){
        String sql="insert into Circle (id, name) values (:id, :name)";
        SqlParameterSource namedParameters=new MapSqlParameterSource("id",circle.getId())
                                                .addValue("name",circle.getName());
        namedParameterJdbcTemplate.update(sql,namedParameter);
}
```

SIMPLEJDBCTEMPLATE

- instead of using both JdbcTemplate or Named ParamterJdbcTemplate we hava
- SimpleJdbcTemplate does not contain all method of JdbcTemplate and NamedParameterJdbcTemplate but it contains most used methods
- So it is recommanded to use this class

private SimpleJdbcTemplate simpleJdbcTemplate;

DAO SUPPORT CLASSES

- if you have 10 DAO classes then all those classed needs some common implementations like creating DataSource instance and its setter methods, and so on.
- being programmer you have to think like lazy
- so let's be more lazy and reduce more code
- one common parent Dao class we can have generic DAO class which all DAO class can extend
- Spring has support class that does setter method of dataSource,it provide generic DAO class

- there are three different DAO classes
- JdbcDaoSupport for JdbcTemplate
- NamedParameterDaoSupport for NamedParameterJdbcTemplate
- SimpleDaoSupport for SimpleJdbcTemplate;

SIMPLEJDBCDAO SUPPORT CLASS

- SimpleJdbcDaoSupport class has SimpleJdbcTemplate as datamember so need to create that object
- getSimpleJdbcTemplate() method return SimpleJdbcTemplate instance

```
class SimpleJdbcDaoImpl extends SimpleJdbcDaoSupport  {
        public int getCircleCount()  {
                String sql="select count(*) from Circle";
                return this.getSimpleJdbcTemplate().queryForInt(sql);
        }
}
```

DAO BEAN INSIDE SPRING.XML

- but how DaoSupport class know where is database

spring.xml

- <bean id="simpleJdbcDaoImpl" classs="SimpleJdbcDaoImpl">
- <property name="dataSource" ref="dataSource"/>
- </bean>

SPRING WITH HIBERNATE

- Add hibernate library
- session factory is expensive bean it must be singleton bean
- just like jdbc support class spring also have sessionfactory support class
- when using hibernate with spring we dont need configuration file
- those configuration parameter in spring.xml file itself we can have
- @Repository :- Indicates that an annotated class is a "Repository", originally defined by Domain-Driven Design (Evans, 2003) as "**a mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects**".

```
@Repository
class HibernateDaoImpl {
        @Autowired
        private SessionFactory sessionFactory;
        getter and setter
        public int getCircleCount(){
                String hql="select count(*) from Circle";
                Session session=getSessionFactory().openSession();
                Query query = session.createQuery(hql);
                return query.uniqueResult();
        }
}
```

- spring.xml

```
<bean class="sessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean"
>
<property name="dataSource" ref="dataSource"/>
<property name="packagesToScan" value="org.tops.model"/>
<property name="hibernateProperties">
<props>
<prop key="dialect" value="org.hibernate.dialect.MySQL"/>
</props>
</property>
</bean>
```

FIRST SPRING MVC PROJECT

1. Modify web.xml file
2. Create spring-dispatcher-servlet.xml
3. Create the HelloController.java class (A Controller)
4. Create the HelloPage.jsp file (A view)

WEB.XML

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://java.sun.com/xml/ns/javaee"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
  <display-name>FirstSpringMVCProject</display-name>

  <servlet>
    <servlet-name>spring-dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>spring-dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

Front Controller

all url request to web application should be map to spring mvc framework for further processing

SPRING-DISPATCHER-SERVLET.XML

```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.xsd">

<bean id="HandlerMapping"
    class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
</bean>

<bean name="/welcome.html"
        class="com.tops.controller.HelloController">
</bean>

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

which HandlerMapping to use whose job is to scan the whole URL and tell FrontController which data generated class to use

Controller or Data generator class

Which ViewResolver to use whose job is inform our application exact location of views

HOMECONTROLLER.JAVA

```
public class HelloController extends AbstractController{

    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        // TODO Auto-generated method stub
        ModelAndView modelAndView=new ModelAndView("HelloPage");
        modelAndView.addObject("welcomeMessage","Hi user, welcome to the first "
                + "Spring MVC Application");
        return modelAndView;
    }

}
```

HELLOPAGE.JSP

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
    <h1>${welcomeMessage}</h1>
</body>
</html>
```

ANNOTATION BASED CONTROLLER CLASS

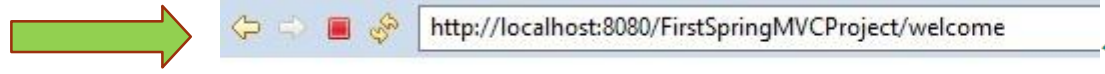
- There are two ways of writing controller class in your application
  1. Without annotation
  2. With annotation

```java
@Controller
public class HelloController{

    @RequestMapping("/welcome")
    public ModelAndView helloworld()
    {
        ModelAndView modelAndView=new ModelAndView("HelloPage");
        modelAndView.addObject("welcomeMessage","Hi user, welcome to the first "
                + "Spring MVC Application");
        return modelAndView;
    }

}
```

Spring-dispatcher-servlet.xml

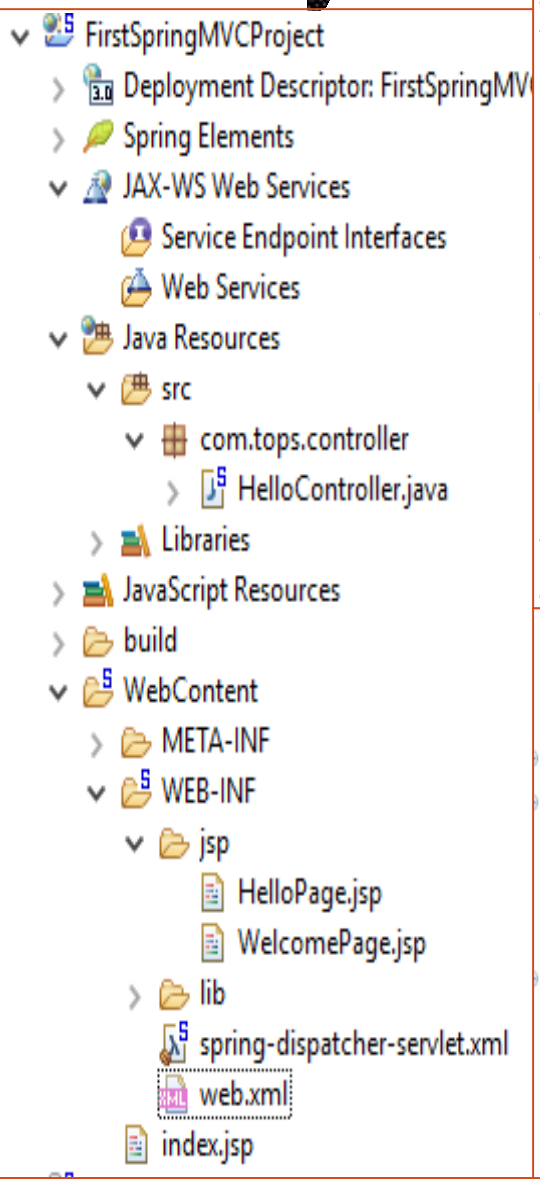
```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.2.xsd">

<context:component-scan base-package="com.tops.controller" />

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

</beans>
```

Output



Hi user, welcome to the first Spring MVC Application

INDEX.JSP




```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"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.2.xsd">

<context:component-scan base-package="com.tops.controller" />

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

</beans>
```

```jsp
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://w
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
    <a href="hello">Hello</a>  
    <a href="welcome">Welcome</a>
</body>
</html>
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://java.sun.com/xml/ns/javaee"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
  <display-name>FirstSpringMVCProject</display-name>
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
  <servlet>
    <servlet-name>spring-dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>spring-dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

```java
@Controller
public class HelloController{

    @RequestMapping("/welcome")
    public ModelAndView welcome()
    {
        ModelAndView modelAndView=new ModelAndView("WelcomePage");
        modelAndView.addObject("welcomeMessage","Hi user, welcome to the first "
                + "Spring MVC Application");
        return modelAndView;
    }
    @RequestMapping("/hello")
    public ModelAndView hello()
    {
        ModelAndView modelAndView=new ModelAndView("HelloPage");
        modelAndView.addObject("msg","Hi user, This is Hello Page");
        return modelAndView;
    }
}
```

SPRING PAGE REDIRECTION

```
@Controller
public class HelloController{

    @RequestMapping(value="/",method=RequestMethod.GET)
    public String index()
    {
        return "index";
    }

    @RequestMapping("/hello")
    public String hello()
    {
        return "redirect:welcome";
    }

    @RequestMapping("/welcome")
    public ModelAndView welcome()
    {
        ModelAndView modelAndView=new ModelAndView("WelcomePage");
        modelAndView.addObject("welcomeMessage","Hi user, welcome to the first "
                + "Spring MVC Application");
        return modelAndView;
    }
}
```

No need to declare welcome-file-list in web.xml file

Page Redirection

STATIC PAGE REDIRECTION

SPRING MVC WEBFORMS

First way is ModelAndViewtechnique to return the object of ModelAndViewby the passing view name, commander name, and its object in constructor.

```jsp
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"
<title>Insert title here</title>
</head>
<body>
    <a href="hello">Hello</a>  
    <a href="welcome">Welcome</a>  
    <a href="contact">Contact</a>  <br><br>

    <form action="form" method="post">
        Name : <input type="text" name="name"/><br>
        Mobile : <input type="text" name="mobile"><br>
        <input type="submit" value="Add"/>
    </form>

</body>
</html>
```

```java
@RequestMapping(value="/form", method=RequestMethod.POST)
public ModelAndView callForm(HttpServletRequest request,
        HttpServletResponse response){
    String name=request.getParameter("name");
    String mobile=request.getParameter("mobile");
    ModelMap modelMap=new ModelMap();
    modelMap.addAttribute("name", name);
    modelMap.addAttribute("mobile",mobile);
    ModelAndView modelAndView=new ModelAndView("show", "model", modelMap);
    return modelAndView;
}
```

```jsp
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://ww
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
    <h1>Welcome, ${model.name}</h1>
    <h2>Your mobile number is : ${model.mobile}</h2>
</body>
</html>
```

SPRING MVC WEBFORMS

Second way to pass multiple primitive and object values by using the Model interface object by setting the attributes and return the view name as string parameter.

```jsp
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://ww
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
    <a href="hello">Hello</a>  
    <a href="welcome">Welcome</a>  
    <a href="contact">Contact</a>  <br><br>

    <form action="form1" method="post">
        Name : <input type="text" name="name"/><br>
        Mobile : <input type="text" name="mobile"><br>
        <input type="submit" value="Add"/>
    </form>

</body>
</html>
```

```java
@RequestMapping(value="/form1", method=RequestMethod.POST)
public String multiParam(HttpServletRequest request,
        HttpServletResponse response, ModelMap modelMap){
    String name=request.getParameter("name");
    String mobile=request.getParameter("mobile");
    modelMap.addAttribute("name", name);
    modelMap.addAttribute("mobile",mobile);
    return "show";
}
```

```jsp
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://ww
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
    <h1>Welcome, ${name}</h1>
    <h2>Your mobile number is : ${mobile}</h2>
</body>
</html>
```

SPRING FORMS TAG VALIDATION

- Data binding is useful for allowing user input to be dynamically bound to the domain model of an application
- Spring provides the so-called **DataBinder** to do exactly that
- The **Validator** and the DataBinder make up the validation package, which is primarily used in but not limited to the MVC framework
- Using this API, you annotate domain model properties with declarative validation constraints and the runtime enforces them.
- There are a number of built-in constraints you can take advantage of.
- You may also define your own custom constraints

SPRINGCONTROLLER.JAVA

```java
@Controller
public class SpringController {

    @RequestMapping(value="/",method = RequestMethod.GET)
    public String showValidatinForm(Map<String,User> model) {
        User user = new User();

        model.put("userform", user);

        return "UserForm";
    }

    @RequestMapping(value="/user",method = RequestMethod.POST)
    public String postUserForm(@Valid @ModelAttribute("userform") User user,
            BindingResult result, Map<String,User> model) {
        if (result.hasErrors()) {
            return "UserForm";
        }
        // Add the saved validationForm to the model
        model.put("userForm", user);
        return "registrationSuccess";
    }

}
```

SPRING-DISPATCHER-SERVLET.XML

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:component-scan base-package="com.tops.controller" />
<mvc:annotation-driven />

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

USER.JAVA

```java
public class User {
    @NotEmpty(message="User Name must not be blank")
    @Size(min = 1, max = 20,message="User Name must between 1 to 20 characters.")
    private String userName;

    @NotNull(message="Age must not be blank")
    @NumberFormat(style = Style.NUMBER)
    @Range(min=1, max=110, message="Age must between 1 to 110")
    private Integer age;

    @NotEmpty(message = "Password must not be blank.")
    @Size(min = 1, max = 10, message = "Password must between 1 to 10 Characters.")
    private String password;

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getUserName() {
        return userName;
    }

    public void setAge(Integer age) {
        this.age = age;
```

USERFORM.JSP

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@taglib prefix="f" uri="http://www.springframework.org/tags/form"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>

    <f:form modelAttribute="userform" method="post" action="user" commandName="userform">
    <div>
        <font color="red">
        <f:errors path="userName"/><br>
        <f:errors path="password"/><br>
        <f:errors path="age"/><br>
        </font>
    </div><br><br>
    Username : <f:input path="userName"/><br><br>
    Password : <f:input path="password"/><br><br>
    Age : <f:input path="age"/><br><br>
    <input type="submit" value="Submit"/>
    </f:form>
</body>
</html>
```

REGISTRATIONSUCCESS.JSP

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@taglib prefix="core" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/T
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
    <h1>Registration Success.</h1>
    <h2>Welcome, ${userForm.userName}</h2>
</body>
</html>
```

SESSION MANAGEMENT

CRUD OPERATION WITH JDBC