Spring Framework

The Spring Framework was developed by Rod Johnson in 2003. 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, Tapestry, EJB, JSF etc. The framework, in broader sense, can be defined as a structure where we find solution of the various technical problems.

The Spring framework comprises several modules such as IOC, AOP, DAO, Context, ORM, WEB MVC.

IoC Container

The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets informations from the XML file and works accordingly. The main tasks performed by IoC container are:

- to instantiate the application class
- to configure the object
- to assemble the dependencies between the objects

There are two types of IoC containers. They are:

1. BeanFactory
2. ApplicationContext

Difference between BeanFactory and the ApplicationContext

The ApplicationContext interface is built on top of the BeanFactory interface. It adds some extra functionality than BeanFactory such as simple integration with Spring's AOP, message resource handling (for I18N), event propagation, application layer specific context (e.g. WebApplicationContext) for web application.

Using BeanFactory

The XmlBeanFactory is the implementation class for the BeanFactory interface. To use the BeanFactory, we need to create the instance of XmlBeanFactory class as given below:

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

The constructor of XmlBeanFactory class receives the Resource object so we need to pass the resource object to create the object of BeanFactory.

Using ApplicationContext

The ClssPathXmlApplicationContext class is the implementation class of ApplicationContext interface. We need to instantiate the ClassPathXmlApplicationContext class to use the ApplicationContext as given below:

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

The constructor of ClassPathXmlApplicationContext class receives string, so we can pass the name of the xml file to create the instance of ApplicationContext.

Dependency Injection in Spring

Dependency Injection (DI) is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application. Dependency Injection makes our programming code loosely coupled.

Two ways to perform Dependency Injection in Spring framework

Spring framework provides two ways to inject dependency

- By Constructor
- By Setter method

Dependency Injection by Constructor

We can inject the dependency by constructor. The **<constructor-arg>** subelement of **<bean>** is used for constructor injection.

Following will inject the dependency by a call to constructor of com.ajp.ClassName Class. The constructor having an int parameter will be invoked.

```
<bean id="e" class="com.ajp.ClassName">
<constructor-arg value="1" type="int"></constructor-arg>
</bean>
```

Following will inject the dependency by a call to constructor of com.ajp.ClassName Class. The constructor having an int and String parameter will be invoked.

```
<bean id="e" class=" com.ajp.ClassName ">
<constructor-arg value="10" type="int" ></constructor-arg>
<constructor-arg value="ABC"></constructor-arg>
</bean>
```

If the type attribute in the constructor-arg element is not specified, by default string type constructor will be invoked.

```
<bean id="e" class="com.ajp.ClassName">
<constructor-arg value="1"></constructor-arg>
</bean>
```

Dependency Injection by Setter method

Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

The **<property>** subelement of **<bean>** is used for setter injection.

```
<bean id="obj" class="com.ajp.ClassName">
<property name="id">
 <value>20</value>
</property>
</bean>
```

Spring Beans

The objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. These beans are created with the configuration metadata that you supply to the container, for example, in the form of XML <bean/> definitions.

The bean definition contains the information called **configuration metadata** which is needed for the container to know the followings:

- How to create a bean
- Bean's lifecycle details
- Bean's dependencies

All the above configuration metadata translates into a set of the following properties that make up each bean definition.

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).

Properties	Description
scope	This attribute specifies the scope of the objects created from a particular bean definition
constructor-arg	This is used to inject the dependencies using constructor
properties	This is used to inject the dependencies using setter method
lazy-initialization mode	A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.

When defining a <bean> in Spring, there is an option of declaring a scope for that bean. For example, 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.

The Spring Framework supports following five scopes, three of which are available only if you use a web-aware ApplicationContext.

Scope	Description
singleton	This scopes the bean definition to a single instance per Spring IoC container (default).
prototype	This scopes a single bean definition to have any number of object instances.
request	This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext.
session	This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.
global-session	This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

If scope is set to singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.

The default scope is always singleton however, when you need one and only one instance of a bean, you can set the **scope** property to **singleton** in the bean configuration file

If scope is set to prototype, the Spring IoC container creates new bean instance of the object every time a request for that specific bean is made. As a rule, use the prototype scope for all state-full beans and the singleton scope for stateless beans.

To define a prototype scope, you can set the scope property to prototype in the bean configuration file.

Spring JdbcTemplate

Spring **JdbcTemplate** is a powerful mechanism to connect to the database and execute SQL queries. It internally uses JDBC api, but eliminates a lot of problems of JDBC API. It provides you methods to write the queries directly, so it saves a lot of work and time. Spring JdbcTemplate eliminates all the below mentioned problems of JDBC API.

The problems of JDBC API are as follows:

- A large amount of code before and after executing the query, such as creating connection, statement, closing resultset.
- Exception handling for the database logic.
- Handling of the transaction.

The JdbcTemplate class takes care of creation and release of resources such as creating and closing of connection object etc. So it will not lead to any problem if you forget to close the connection. It handles the exception and provides the informative exception messages by the help of excepion classes defined in the org.springframework.dao package. The CRUD operations can be performed using JdbcTemplate class.

Methods	**Functionality**
public int update(String query)	To insert, update and delete records.
public int update(String query,Object... args)	To insert, update and delete records using PreparedStatement using given arguments.
public void execute(String query)	To execute DDL query.
Public T execute(String sql, PreparedStatementCallback action)	To execute the query by using PreparedStatement callback.
public List query(String sql, RowMapper rse)	To fetch records using RowMapper.

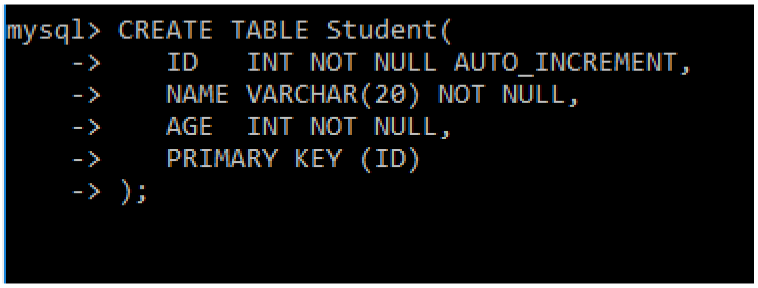
Steps to Insert, Update and Retrieve Student Details using JdbcTemplate.

Following are the steps to be followed:

1) Create table Student in the database.
2) Create the Student.java Class
3) Create StudentMapper.java Class
4) Create StudentJDBCTemplate.java Class
5) Create the Beans.xml file
6) Create the MainApp.java Class
7) Add the jars to the Classpath

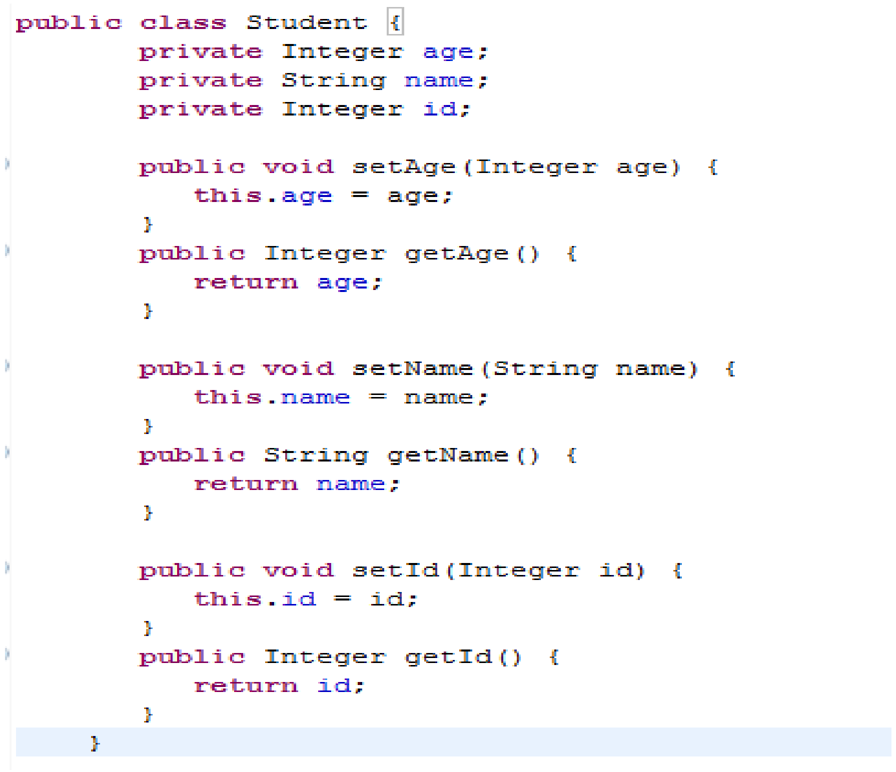
The above mentioned steps are discussed in detail along with the snapshots.

1) Create table Student in the database.



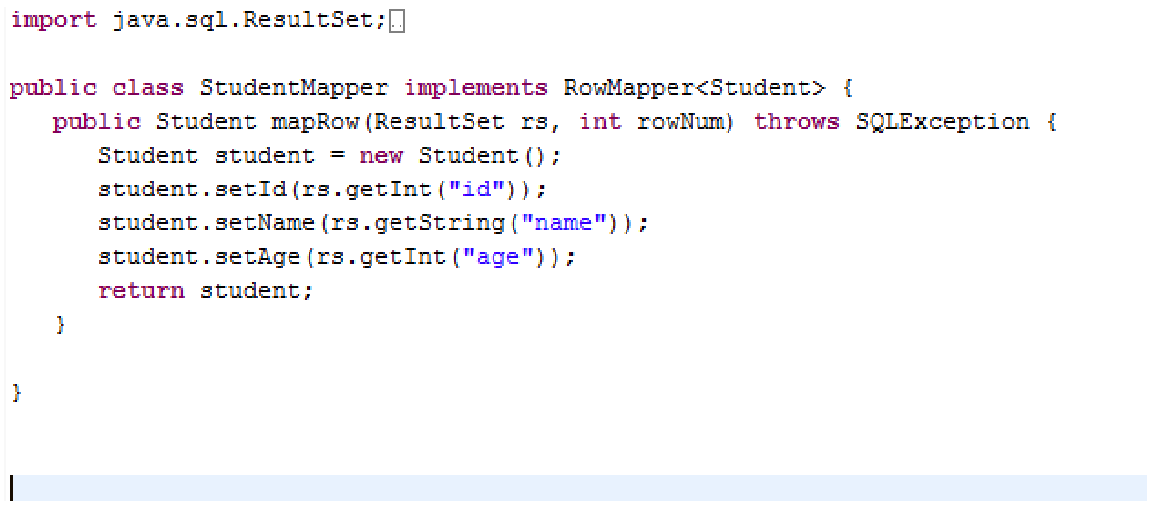
```
mysql> CREATE TABLE Student(
    ->    ID   INT NOT NULL AUTO_INCREMENT,
    ->    NAME VARCHAR(20) NOT NULL,
    ->    AGE  INT NOT NULL,
    ->    PRIMARY KEY (ID)
    -> );
```

2) Create the Student.java Class



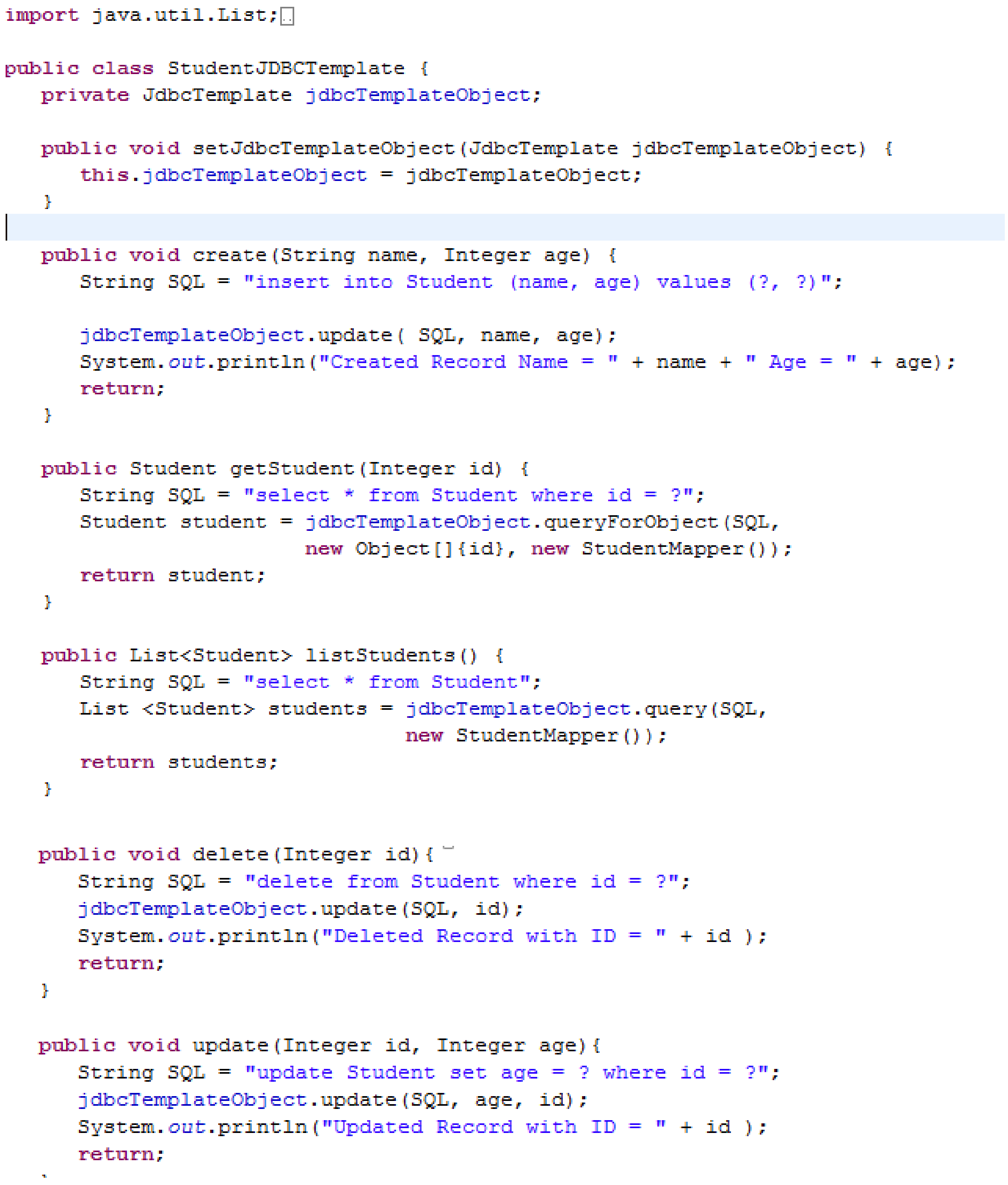
```
public class Student {
        private Integer age;
        private String name;
        private Integer id;

        public void setAge(Integer age) {
           this.age = age;
        }
        public Integer getAge() {
           return age;
        }

        public void setName(String name) {
           this.name = name;
        }
        public String getName() {
           return name;
        }

        public void setId(Integer id) {
           this.id = id;
        }
        public Integer getId() {
           return id;
        }
     }
```

3) Create StudentMapper.java Class



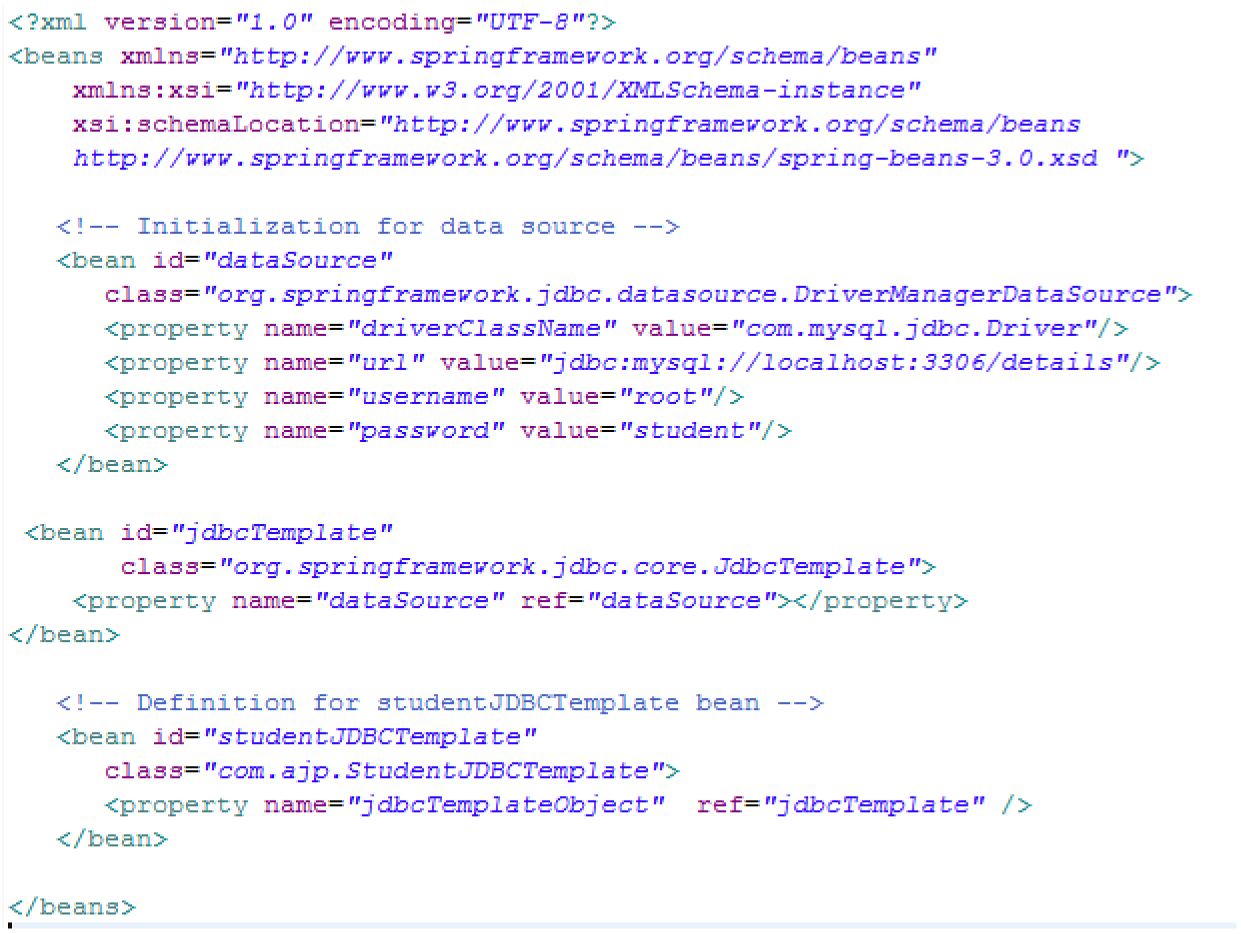
```
import java.sql.ResultSet;

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }

}
```

4) Create StudentJDBCTemplate.java Class



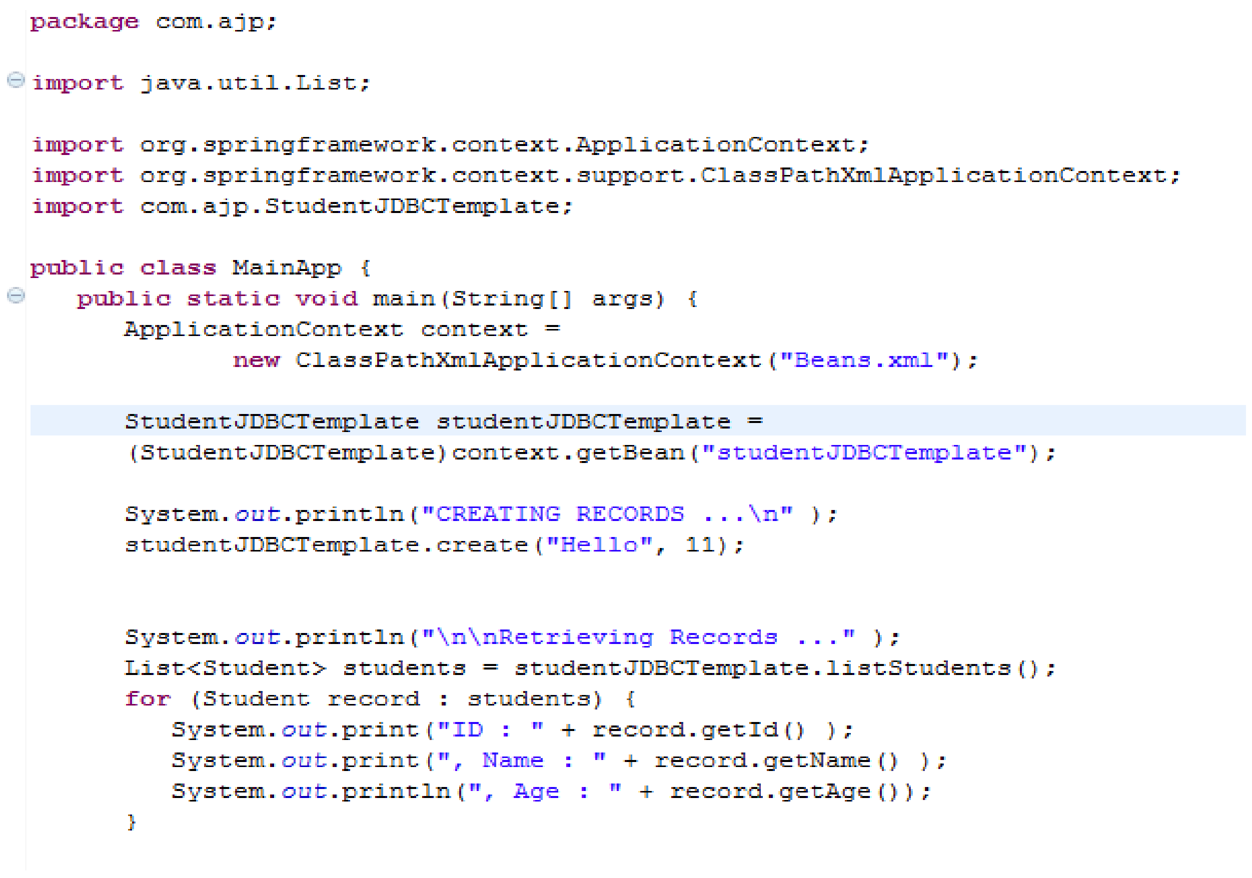
```
import java.util.List;

public class StudentJDBCTemplate {
   private JdbcTemplate jdbcTemplateObject;

   public void setJdbcTemplateObject(JdbcTemplate jdbcTemplateObject) {
      this.jdbcTemplateObject = jdbcTemplateObject;
   }

   public void create(String name, Integer age) {
      String SQL = "insert into Student (name, age) values (?, ?)";

      jdbcTemplateObject.update( SQL, name, age);
      System.out.println("Created Record Name = " + name + " Age = " + age);
      return;
   }

   public Student getStudent(Integer id) {
      String SQL = "select * from Student where id = ?";
      Student student = jdbcTemplateObject.queryForObject(SQL,
                        new Object[]{id}, new StudentMapper());
      return student;
   }

   public List<Student> listStudents() {
      String SQL = "select * from Student";
      List <Student> students = jdbcTemplateObject.query(SQL,
                                new StudentMapper());
      return students;
   }

   public void delete(Integer id){
      String SQL = "delete from Student where id = ?";
      jdbcTemplateObject.update(SQL, id);
      System.out.println("Deleted Record with ID = " + id );
      return;
   }

   public void update(Integer id, Integer age){
      String SQL = "update Student set age = ? where id = ?";
      jdbcTemplateObject.update(SQL, age, id);
      System.out.println("Updated Record with ID = " + id );
      return;
```

5) Create the Beans.xml file

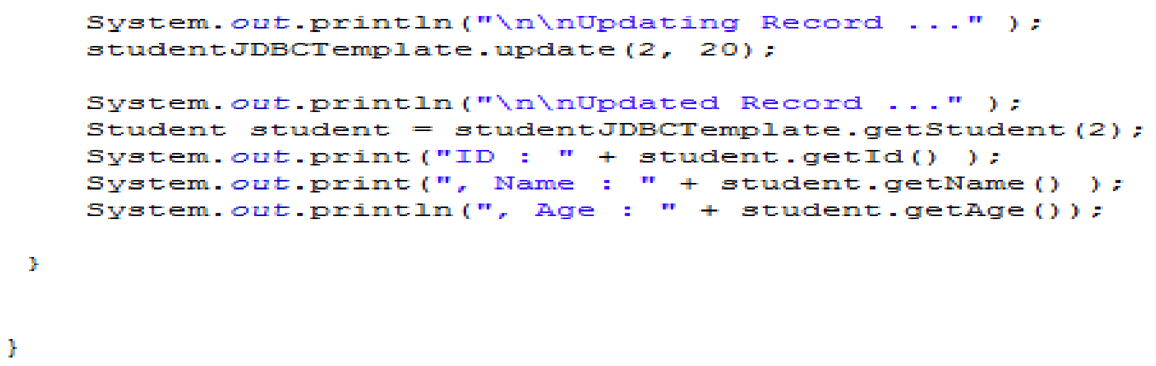


```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">

   <!-- Initialization for 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/details"/>
      <property name="username" value="root"/>
      <property name="password" value="student"/>
   </bean>

 <bean id="jdbcTemplate"
       class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"></property>
</bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.ajp.StudentJDBCTemplate">
      <property name="jdbcTemplateObject"  ref="jdbcTemplate" />
   </bean>

</beans>
```

6) Create the MainApp.java Class



```java
package com.ajp;

import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.ajp.StudentJDBCTemplate;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context =
             new ClassPathXmlApplicationContext("Beans.xml");

      StudentJDBCTemplate studentJDBCTemplate =
      (StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

      System.out.println("CREATING RECORDS ...\n" );
      studentJDBCTemplate.create("Hello", 11);


      System.out.println("\n\nRetrieving Records ..." );
      List<Student> students = studentJDBCTemplate.listStudents();
      for (Student record : students) {
         System.out.print("ID : " + record.getId() );
         System.out.print(", Name : " + record.getName() );
         System.out.println(", Age : " + record.getAge());
      }
```



```java
    System.out.println("\n\nUpdating Record ..." );
    studentJDBCTemplate.update(2, 20);

    System.out.println("\n\nUpdated Record ..." );
    Student student = studentJDBCTemplate.getStudent(2);
    System.out.print("ID : " + student.getId() );
    System.out.print(", Name : " + student.getName() );
    System.out.println(", Age : " + student.getAge());

 }


}
```