JAVA PERSISTENCE API (JPA)

Tutorial

Simply Easy Learning

www.tutorialspoint.com

SIMPLY EASY LEARNING

About the Tutorial

This tutorial provides a basic understanding of how to store a copy of database objects into temporary memory using JAVA Persistence API (JPA).

Audience

This tutorial is designed for readers intend to do Java programing with Database connectivity, using Persistence API.

Prerequisites

Awareness of Java programming with JDK 1.6 or later is a prerequisite to understand this tutorial. In addition, we assume the readers are acquainted with the concepts of JDBC in Java.

Copyright & Disclaimer

© Copyright 2014 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com .

Table of Contents

1. JPA – INTRODUCTION

Any enterprise application performs database operations by storing and retrieving vast amounts of data. Despite all the available technologies for storage management, application developers normally struggle to perform database operations efficiently.

Generally, Java developers use lots of code, or use the proprietary framework to interact with the database, whereas using JPA, the burden of interacting with the database reduces significantly. It forms a bridge between object models (Java program) and relational models (database program).

Mismatch between Relational and Object Models

Relational objects are represented in a tabular format, while object models are represented in an interconnected graph of object format. While storing and retrieving an object model from a relational database, some mismatch occurs due to the following reasons:

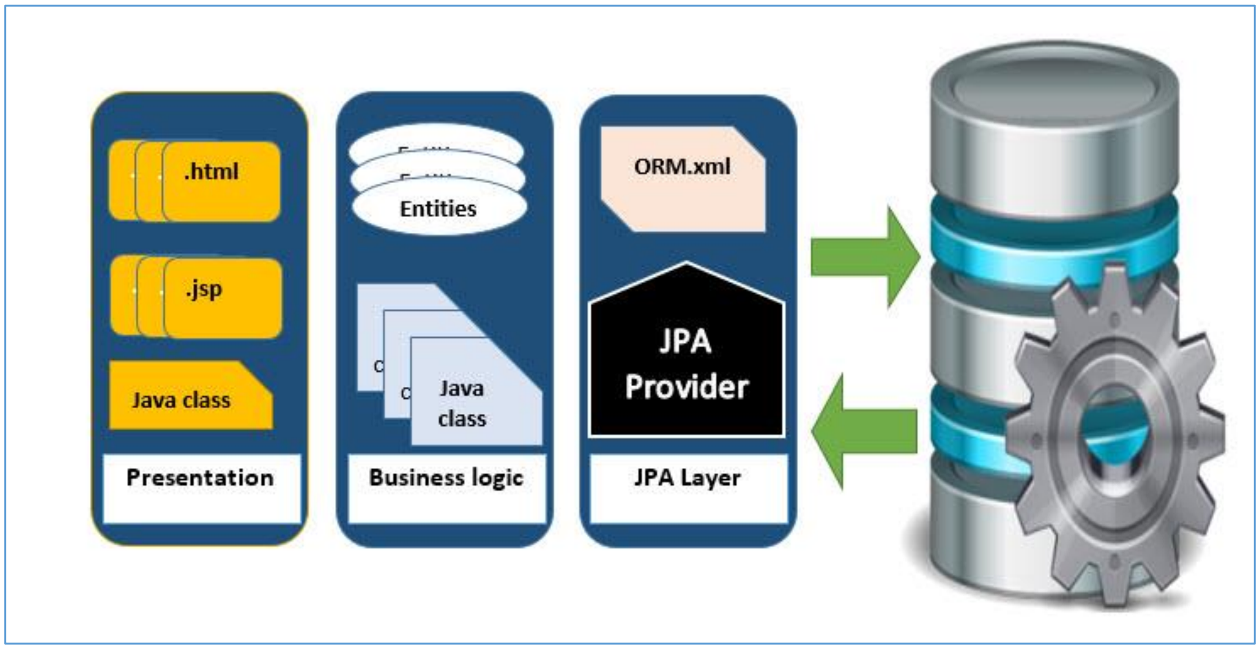
- **Granularity**: Object model has more granularity than relational model.
- **Subtypes**: Subtypes (means inheritance) are not supported by all types of relational databases.
- **Identity**: Like object model, relational model does not expose identity while writing equality.
- **Associations**: Relational models cannot determine multiple relationships while looking into an object domain model.
- **Data navigation**: Data navigation between objects in an object network is different in both models.

What is JPA?

Java Persistence API is a collection of classes and methods to persistently store the vast amounts of data into a database which is provided by the Oracle Corporation.

Where to use JPA?

To reduce the burden of writing codes for relational object management, a programmer follows the 'JPA Provider' framework, which allows easy interaction with database instance. Here the required framework is taken over by JPA.



JPA History

Earlier versions of EJB, defined the persistence layer combined with the business logic layer using **javax.ejb.EntityBean** Interface.

- While introducing EJB 3.0, the persistence layer was separated and specified as JPA 1.0 (Java Persistence API). The specifications of this API were released along with the specifications of JAVA EE5 on May 11, 2006 using JSR 220.
- JPA 2.0 was released with the specifications of JAVA EE6 on December 10, 2009 as a part of Java Community Process JSR 317.
- JPA 2.1 was released with the specification of JAVA EE7 on April 22, 2013 using JSR 338.

JPA Providers

JPA is an open source API, therefore various enterprise vendors such as Oracle, Redhat, Eclipse, etc. provide new products by adding the JPA persistence flavor in them. Some of these products include:

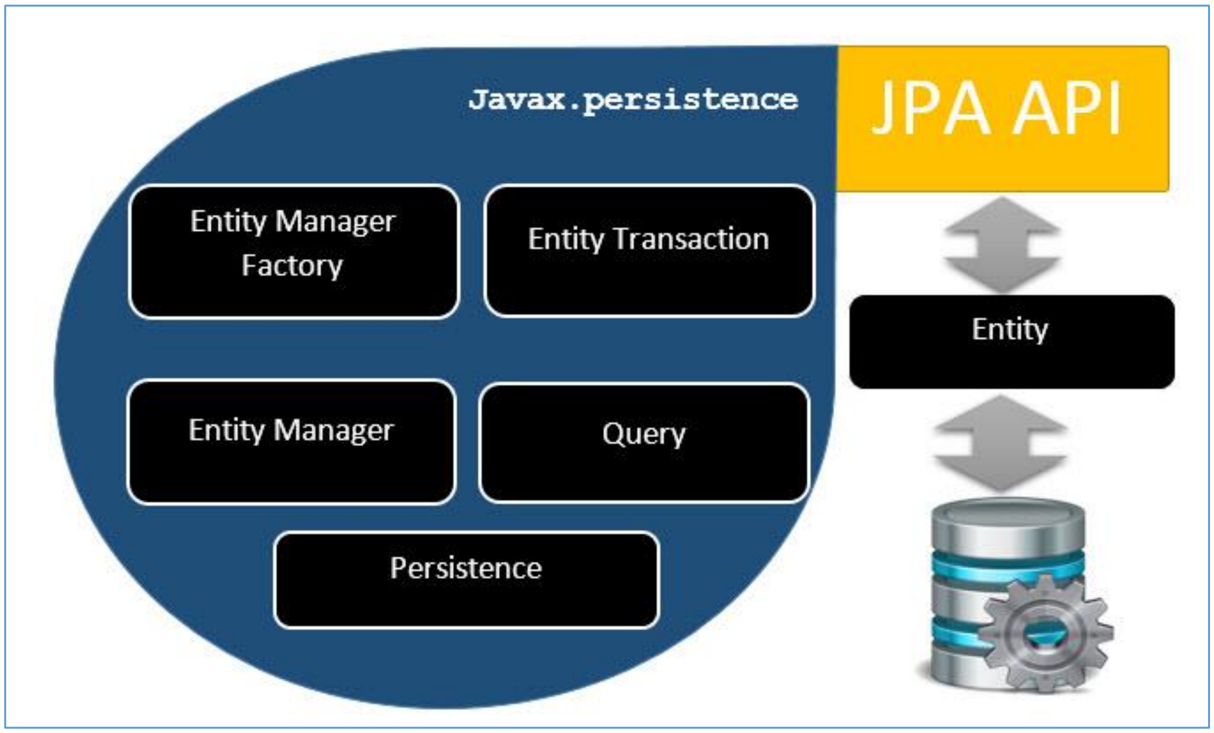
- Hibernate
- Eclipselink
- Toplink
- Spring Data JPA

2. JPA – ARCHITECTURE

Java Persistence API is a source to store business entities as relational entities. It shows how to define a Plain Oriented Java Object (POJO) as an entity and how to manage entities with relations.

Class Level Architecture

The following image shows the class level architecture of JPA. It shows the core classes and the interfaces of JPA.



The following table describes each of the units shown in the above architecture.

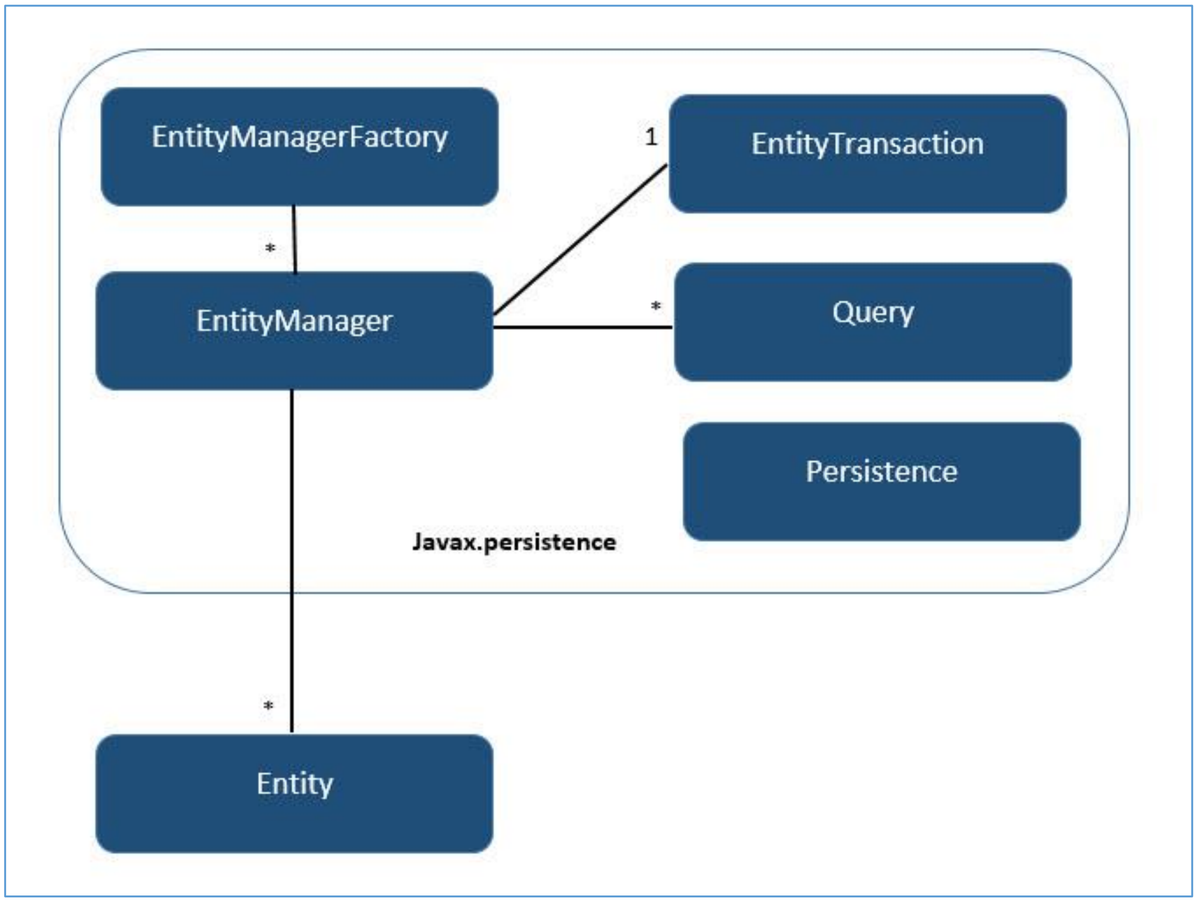
Units	Description
EntityManagerFactory	This is a factory class of EntityManager. It creates and manages multiple EntityManager instances.

EntityManager	It is an Interface. It manages the persistence operations on objects. It works like a factory for Query instance.
Entity	Entities are the persistence objects, stored as records in the database.
EntityTransaction	It has one-to-one relationship with the EntityManager. For each EntityManager, operations are maintained by the EntityTransaction class.
Persistence	This class contains static methods to obtain the EntityManagerFactory instance.
Query	This interface is implemented by each JPA vendor to obtain the relational objects that meet the criteria.

The above classes and interfaces are used for storing entities into a database as a record. They help programmers by reducing their efforts to write codes for storing data into a database so that they can concentrate on more important activities such as writing codes for mapping the classes with database tables.

JPA Class Relationships

In the above architecture, the relations between the classes and interfaces belong to the **javax.persistence** package. The following diagram shows the relationship between them.



- The relationship between EntityManagerFactory and EntityManager is one-to-many. It is a factory class to EntityManager instances.

- The relationship between EntityManager and EntityTransaction is one-to-one. For each EntityManager operation, there is an EntityTransaction instance.

- The relationship between EntityManager and Query is one-to-many. A number of queries can execute using one EntityManager instance.

- The relationship between EntityManager and Entity is one-to-many. One EntityManager instance can manage multiple Entities.

3. JPA – ORM COMPONENTS

Most contemporary applications use relational database to store data. Recently, many vendors switched to object database to reduce their burden on data maintenance. It means object database or object relational technologies are taking care of storing, retrieving, updating, and maintaining data. The core part of this object relational technology is mapping **orm.xml** files. As xml does not require compilation, we can easily make changes to multiple data sources with less administration.

Object Relational Mapping

Object Relational Mapping (ORM) briefly tells you about what is ORM and how it works. ORM is a programming ability to covert data from object type to relational type and vice versa.

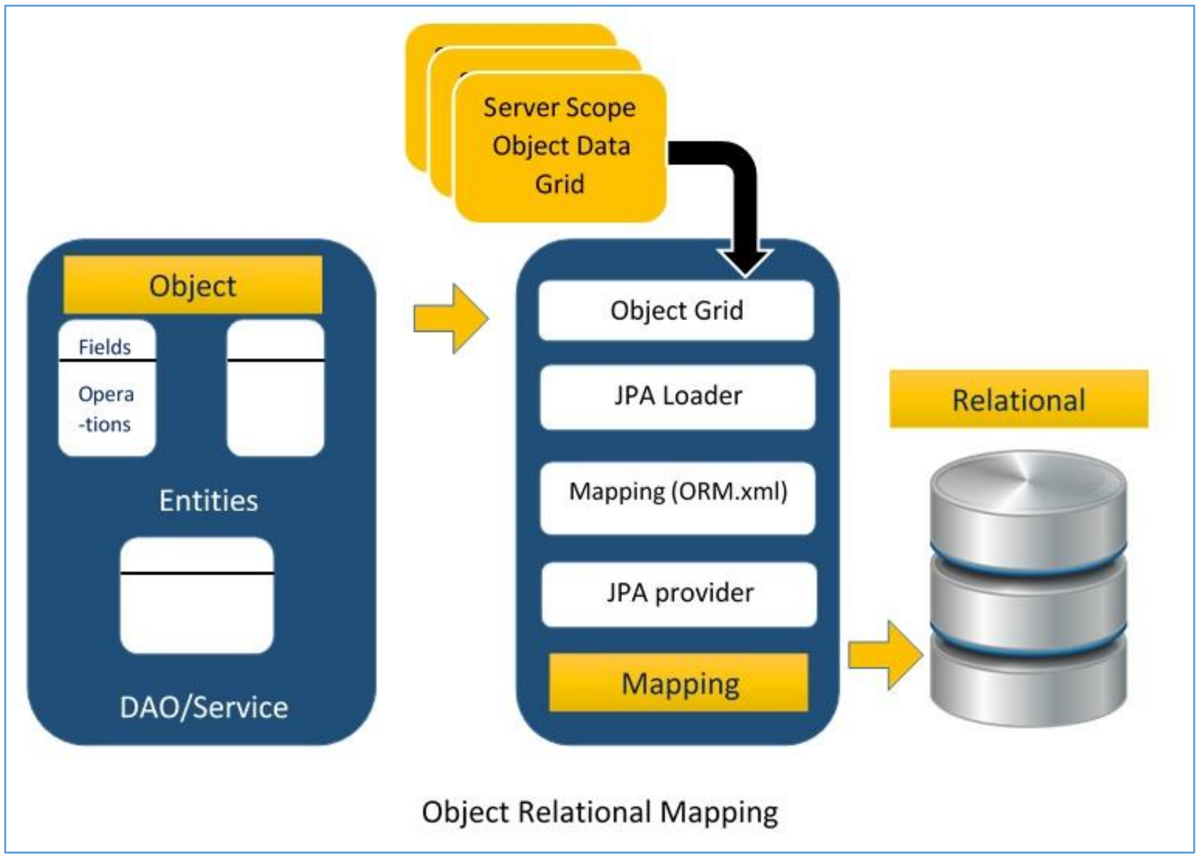
The main feature of ORM is mapping or binding an object to its data in the database. While mapping, we have to consider the data, the type of data, and its relations with self-entity or entities in any other table.

Advanced Features

- Idiomatic persistence: It enables you to write persistence classes using object oriented classes.
- High Performance: It has many fetching techniques and helpful locking techniques.
- Reliable: It is highly stable and used by many professional programmers.

ORM Architecture

The ORM architecture looks as follows.



The above architecture explains how object data is stored into a relational database in three phases.

Phase 1

The first phase, named as the **object data phase**, contains POJO classes, service interfaces, and classes. It is the main business component layer, which has business logic operations and attributes.

For example, let us take an employee database as a schema.

- Employee POJO class contains attributes such as ID, name, salary, and designation. It also contains methods like setter and getter of those attributes.
- Employee DAO/Service classes contain service methods such as create employee, find employee, and delete employee.

Phase 2

The second phase, named as **mapping** or **persistence phase**, contains JPA provider, mapping file (ORM.xml), JPA Loader, and Object Grid.

- **JPA Provider**: It is the vendor product that contains the JPA flavor (javax.persistence). For example Eclipselink, Toplink, Hibernate, etc.
- **Mapping file**: The mapping file (ORM.xml) contains mapping configuration between the data in a POJO class and data in a relational database.
- **JPA Loader**: The JPA loader works like a cache memory. It can load the relational grid data. It works like a copy of database to interact with service classes for POJO data (attributes of POJO class).
- **Object Grid**: It is a temporary location that can store a copy of relational data, like a cache memory. All queries against the database is first effected on the data in the object grid. Only after it is committed, it affects the main database.

Phase 3

The third phase is the **relational data phase**. It contains the relational data that is logically connected to the business component. As discussed above, only when the business component commits the data, it is stored into the database physically. Until then, the modified data is stored in a cache memory as a grid format. The process of the obtaining the data is identical to that of storing the data.

The mechanism of the programmatic interaction of the above three phases is called as **object relational mapping**.

Mapping.xml

The mapping.xml file instructs the JPA vendor to map the entity classes with the database tables.

Let us take an example of Employee entity that contains four attributes. The POJO class of Employee entity named **Employee.java** is as follows:

```
public class Employee
{
	private int eid;
	private String ename;
	private double salary;
	private String deg;
```

```
    public Employee(int eid, String ename, double salary, String deg)
    {
        super( );
        this.eid = eid;
        this.ename = ename;
        this.salary = salary;
        this.deg = deg;
    }

    public Employee( )
    {
        super();
    }

    public int getEid( )
    {
        return eid;
    }
    public void setEid(int eid)
    {
        this.eid = eid;
    }

    public String getEname( )
    {
        return ename;
    }
    public void setEname(String ename)
    {
        this.ename = ename;
    }
```

```
      public double getSalary( )
      {
             return salary;
      }
      public void setSalary(double salary)
      {
             this.salary = salary;
      }

      public String getDeg( )
      {
             return deg;
      }
      public void setDeg(String deg)
      {
             this.deg = deg;
      }
}
```

The above code is the Employee entity POJO class. It contain four attributes **eid**, **ename**, **salary**, and **deg**. Consider these attributes as the table fields in a table and **eid** as the primary key of this table. Now we have to design the hibernate mapping file for it. The mapping file named **mapping.xml** is as follows:

```
<? xml version="1.0" encoding="UTF-8" ?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
                        http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
                        version="1.0">
    <description> XML Mapping file</description>
    <entity class="Employee">
        <table name="EMPLOYEETABLE"/>
        <attributes>
```

```
            <id name="eid">
                <generated-value strategy="TABLE"/>
            </id>
            <basic name="ename">
                <column name="EMP_NAME" length="100"/>
            </basic>
            <basic name="salary">
            </basic>
            <basic name="deg">
            </basic>
        </attributes>
    </entity>
</entity-mappings>
```

The above script is used for mapping the entity class with the database table. In this file,

- <entity-mappings> tag defines the schema definition to allow entity tags into the xml file.
- <description> tag provides a description of the application.
- <entity> tag defines the entity class which you want to convert into a table in a database. Attribute class defines the POJO entity class name.
- <table> tag defines the table name. If you want to have identical names for both the class as well as the table, then this tag is not necessary.
- <attributes> tag defines the attributes (fields in a table).
- <id> tag defines the primary key of the table. The <generated-value> tag defines how to assign the primary key value such as Automatic, Manual, or Taken from Sequence.
- <basic> tag is used for defining the remaining attributes of the table.
- <column-name> tag is used to set user-defined field names in the table.

Annotations

Generally xml files are used to configure specific components, or mapping two different specifications of components. In our case, we have to maintain xml files separately in a framework. That means while writing a mapping xml file, we need to compare the POJO class attributes with entity tags in the mapping.xml file.

Here is the solution. In the class definition, we can write the configuration part using annotations. Annotations are used for classes, properties, and methods. Annotations start with '**@**' symbol. Annotations are declared prior to a class, property, or method. All annotations of JPA are defined in the **javax.persistence** package.

The list of annotations used in our examples are given below.

Annotation	Description
@Entity	Declares the class as an entity or a table.
@Table	Declares table name.
@Basic	Specifies non-constraint fields explicitly.
@Embedded	Specifies the properties of class or an entity whose value is an instance of an embeddable class.
@Id	Specifies the property, use for identity (primary key of a table) of the class.
@GeneratedValue	Specifies how the identity attribute can be initialized such as automatic, manual, or value taken from a sequence table.
@Transient	Specifies the property that is not persistent, i.e., the value is never stored in the database.
@Column	Specifies the column attribute for the persistence property.
@SequenceGenerator	Specifies the value for the property that is specified in the @GeneratedValue annotation. It creates a sequence.

@TableGenerator	Specifies the value generator for the property specified in the @GeneratedValue annotation. It creates a table for value generation.
@AccessType	This type of annotation is used to set the access type. If you set @AccessType(FIELD), then access occurs Field wise. If you set @AccessType(PROPERTY), then access occurs Property wise.
@JoinColumn	Specifies an entity association or entity collection. This is used in many- to-one and one-to-many associations.
@UniqueConstraint	Specifies the fields and the unique constraints for the primary or the secondary table.
@ColumnResult	References the name of a column in the SQL query using select clause.
@ManyToMany	Defines a many-to-many relationship between the join Tables.
@ManyToOne	Defines a many-to-one relationship between the join Tables.
@OneToMany	Defines a one-to-many relationship between the join Tables.
@OneToOne	Defines a one-to-one relationship between the join Tables.
@NamedQueries	Specifies a list of named queries.
@NamedQuery	Specifies a Query using static name.

Java Bean Standard

The Java class encapsulates the instance values and their behaviors into a single unit called object. Java Bean is a temporary storage and reusable component or an object. It is a serializable class which has a default constructor and getter and setter methods to initialize the instance attributes individually.

Bean Conventions

- Bean contains its default constructor or a file that contains a serialized instance. Therefore, a bean can instantiate another bean.
- The properties of a bean can be segregated into Boolean properties or non-Boolean properties.
- Non-Boolean property contains **getter** and **setter** methods.
- Boolean property contains **setter** and **is** method.
- **Getter** method of any property should start with small lettered **get** (Java method convention) and continued with a field name that starts with a capital letter. For example, the field name is **salary**, therefore the getter method of this field is **getSalary ()**.
- **Setter** method of any property should start with small lettered **set** (Java method convention), continued with a field name that starts with a capital letter and the **argument value** to set to field. For example, the field name is **salary**, therefore the setter method of this field is **setSalary (double sal)**.
- For Boolean property, the **is** method is used to check if it is true or false. For example, for the Boolean property **empty**, the **is** method of this field is **isEmpty ()**.

End of ebook preview
If you liked what you saw…
Buy it from our store @ **https://store.tutorialspoint.com**