1. **Access Types**:

You can use both the types, either with field or with property types. But don't use both as the same property will be validated twice. For a property level, you have to keep the annotation on getters. Also, if you keep for the id, then the remaining all will be understood as property..and shimilar to the field level also.

2. **Mapping to a Table:**

By default, if you give a pojo with @entity, then it will search for the same classname in the database for the table name. Otherwise, you have to give the annotation as @Table after @Entity. Example: @Table(name="EMPLOYEE")

If you want to specify the schema as well, you give it as @Table(name="EMP", schema="HR")

**3. @Basic mappings:**

An optional @Basic annotation can be placed on a field or property to explicitly mark it as being

persistent. This annotation is mostly for documentation purposes and is not required for the field or

property to be persistent. Because of the annotation, we call mappings of simple types basic mappings.

@Column:

This is used, when the property name in the pojo does not match with that of the database. Here the mandatory parameter what we give is the name.

**@Basic mappings** annotation is also used in order to specify the fetch type. The fetch types will be of the form:

@Basic(fetch=FetchType.LAZY) (OR)

@Basic(fetch=FetchType.EAGER)

The directive to lazily fetch an attribute is meant only to be a hint to the persistence provider to help the application achieve better performance. The provider is not required to respect the request because the behavior of the entity is not compromised if the provider goes ahead and loads the attribute.

I see that jpa, hibernate annotations dosen't apply for normal fields. **Maybe they will be applicable to collections only..! Need to check further.**

4. BLOB and CLOB:

LOBs come in two flavors in the database: character large objects, called CLOBs, and binary large

objects, or BLOBs. As their names imply, a CLOB column holds a large character sequence, and a BLOB

column can store a large byte sequence. The Java types mapped to BLOB columns are byte[], Byte[],

and Serializable types, while char[], Character[], and String objects are mapped to CLOB columns.

The provider is responsible for making this distinction based on the type of the attribute being

mapped.

Example of BLOB column mapping:

@Entity

public class Employee {

@Id

private int id;

@Basic(fetch=FetchType.LAZY)

@Lob @Column(name="PIC")

private byte[] picture;

// ...

}

5. Enumeration Types:

In order to use the enumeration type, you can declare the properties inside the POJO as,

private <EnumType> variable;

And you can have the setter and getter methods for this. By doing so, the enum element's ordinal will be stored. Again, there could be a problem if at some point of time, we want to insert another enum element below the current element. In that case, there will be the ordinal (0, 1...) disturbance. So instead of that we can store the enum String, by giving the below annotation before the element.

@Enumerated(EnumType.String)

6. Temporal Types:

Temporal types are the set of time-based types that can be used in persistent state mappings. The list

of supported temporal types includes the three java.sql types java.sql.Date, java.sql.Time, and

java.sql.Timestamp, and it includes the two java.util types java.util.Date and java.util.Calendar.

The java.sql types are completely hassle-free. They act just like any other simple mapping type

and do not need any special consideration. The two java.util types need additional metadata,

however, to indicate which of the JDBC java.sql types to use when communicating with the JDBC

driver. This is done by annotating them with the @Temporal annotation and specifying the JDBC type as

a value of the TemporalType enumerated type. There are three enumerated values of DATE, TIME, and

TIMESTAMP to represent each of the java.sql types.

Example:

@Entity

public class Employee {

@Id

private int id;

@Temporal(TemporalType.DATE)

private Calendar dob;

@Temporal(TemporalType.DATE)

@Column(name="S\_DATE")

private Date startDate;

// ...

}

7. Mapping the Primary Key:

id mappings are generally restricted to the following types:

* **Primitive Java types:** byte, int, short, long, char
* **Wrapper classes of primitive Java types:** Byte, Integer, Short, Long, Character
* **String:** java.lang.String
* **Large numeric type:** java.math.BigInteger
* **Temporal types:** java.util.Date, java.sql.Date

8. Identifier Generation:

Applications can choose one of four different id generation strategies by specifying a strategy in

the strategy element. The value can be any one of AUTO, TABLE, SEQUENCE, or IDENTITY enumerated

values of the GenerationType enumerated type.

Automatic id generation:

If an application does not care what kind of generation is used by the provider but wants generation to

occur, it can specify a strategy of AUTO. This means that the provider will use whatever strategy it wants

to generate identifiers.

@Entity

public class Employee {

@Id @GeneratedValue(strategy=GenerationType.AUTO)

private int id;

// ...

}

id generation using table:

The most flexible and portable way to generate identifiers is to use a database table. Not only will it

port to different databases but it also allows for storing multiple different identifier sequences for

different entities within the same table.

An id generation table should have two columns. The first column is a string type used to identify

the particular generator sequence. It is the primary key for all the generators in the table. The second

column is an integer type that stores the actual id sequence that is being generated. The value stored

in this column is the last identifier that was allocated in the sequence. Each defined generator

represents a row in the table.

The easiest way to use a table to generate identifiers is to simply specify the generation strategy

to be TABLE in the strategy element:

@Id @GeneratedValue(strategy=GenerationType.TABLE)

private int id;

Because the generation strategy is indicated but no generator has been specified, the provider

will assume a table of its own choosing. If schema generation is used, it will be created; if not, the

default table assumed by the provider must be known and must exist in the database.

A more explicit approach would be to actually specify the table that is to be used for id storage. This

is done by defining a table generator that, contrary to what its name implies, does not actually

generate tables. Rather, it is an identifier generator that uses a table to store them. We can define one

by using a @TableGenerator annotation and then refer to it by name in the @GeneratedValue

annotation:

@TableGenerator(name="Emp\_Gen")

@Id @GeneratedValue(generator="Emp\_Gen")

private int id;

A further qualifying approach would be to specify the table details, as in the following:

@TableGenerator(name="Emp\_Gen",

table="ID\_GEN",

pkColumnName="GEN\_NAME",

valueColumnName="GEN\_VAL")

A more controlled way of doing this is:

@TableGenerator(name="Address\_Gen",

table="ID\_GEN",

pkColumnName="GEN\_NAME",

valueColumnName="GEN\_VAL",

pkColumnValue="Addr\_Gen",

initialValue=10000,

allocationSize=100)

@Id @GeneratedValue(generator="Address\_Gen")

private int id;

The script for creating a table for this:

CREATE TABLE id\_gen (

gen\_name VARCHAR(80),

gen\_val INTEGER,

CONSTRAINT pk\_id\_gen

PRIMARY KEY (gen\_name)

);

INSERT INTO id\_gen (gen\_name, gen\_val) VALUES ('Emp\_Gen', 0);

INSERT INTO id\_gen (gen\_name, gen\_val) VALUES ('Addr\_Gen', 10000);

id generation using Database Identity:

Some databases support a primary key identity column, sometimes referred to as an autonumber

column. Whenever a row is inserted into the table, the identity column will get a unique identifier

assigned to it. It can be used to generate the identifiers for objects, but once again is available only

when the underlying database supports it. Identity is often used when database sequences are not

supported by the database or because a legacy schema has already defined the table to use identity

columns. They are generally less efficient for object-relational identifier generation because they

cannot be allocated in blocks and because the identifier is not available until after commit time.

To indicate that IDENTITY generation should occur, the @GeneratedValue annotation should specify

a generation strategy of IDENTITY. This will indicate to the provider that it must re-read the inserted

row from the table after an insert has occurred. This will allow it to obtain the newly generated

identifier from the database and put it into the in-memory entity that was just persisted:

9. Relationships:

Cardinality:

It isn’t very often that a project has only a single employee working on it. We would like to be able to

capture the aspect of how many entities exist on each side of the same relationship instance. This is

called the *cardinality* of the relationship*.* Each role in a relationship will have its own cardinality,

which indicates whether there can be only one instance of the entity or many instances.

In our Employee and Department example, we might first say that one employee works in one

department, so the cardinality of both sides would be *one*. But chances are that more than one

employee works in the department, so we would make the relationship have a *many* cardinality on the

Employee or source side, meaning that many Employee instances could each point to the same

Department. The target or Department side would keep its cardinality of one. Figure 4-7 shows this

many-to-one relationship. The “many” side is marked with an asterisk (\*).

***Figure 4-7.*** *Unidirectional many-to-one relationship*

*Employee \*-------->1 department (diagram needs to be modified)*

In our Employee and Project example, we have a bidirectional relationship, or two relationship

directions. If an employee can work on multiple projects, and a project can have multiple employees

working on it, then we would end up with cardinalities of “many” on the sources and targets of both

directions. Figure 4-8 shows the UML diagram of this relationship.

***Figure 4-8.*** *Bidirectional many-to-many relationship*

*Employee\*<-------->\*Project*

Like basic mappings, relationship mappings can be applied to either fields or properties of

the entity.

Types of Relationships:

Many-to-One

One-to-One

One-to-Many

Many-to-Many

Many-to-One

In our cardinality discussion of the Employee and Department relationship (shown in Figure 4-7), we

first thought of an employee working in a department, so we just assumed that it was a one-to-one

relationship. However, when we realized that more than one employee works in the same department,

we changed it to a many-to-one relationship mapping. It turns out that many-to-one is the most

common mapping and is the one that is normally used when creating an association to an entity.

The below figure shows a many-to-one relationship between Employee and Department. Employee is the

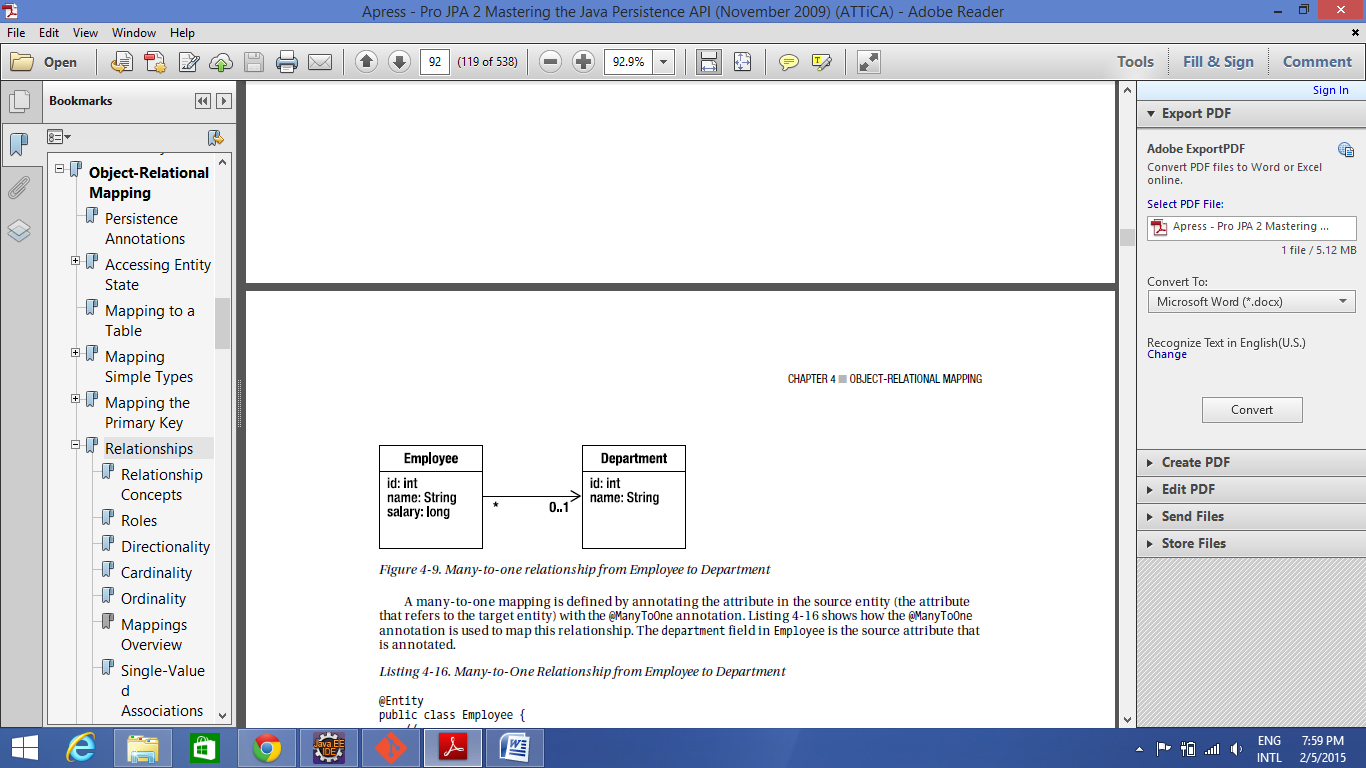
“many” side and the source of the relationship, and Department is the “one” side and the target. Once

again, because the arrow points in only one direction, from Employee to Department, the relationship is

unidirectional. Note that in UML, the source class has an implicit attribute of the target class type if it

can be navigated to. For example, Employee has an attribute called department that will contain a

reference to a single Department instance.



The multiplicity value next to the Department class of 0..1 means that when an instance of a Employee exists, it can either have one instance of a Department associated with it or no Departments associated with it (i.e., maybe a plane has not yet been assigned).

A many-to-one mapping is defined by annotating the attribute in the source entity (the attribute

that refers to the target entity) with the @ManyToOne annotation. Listing 4-16 shows how the @ManyToOne

annotation is used to map this relationship. The department field in Employee is the source attribute that

is annotated.

Example:

@Entity

public class Employee {

// ...

@ManyToOne

private Department department;

// ...

}

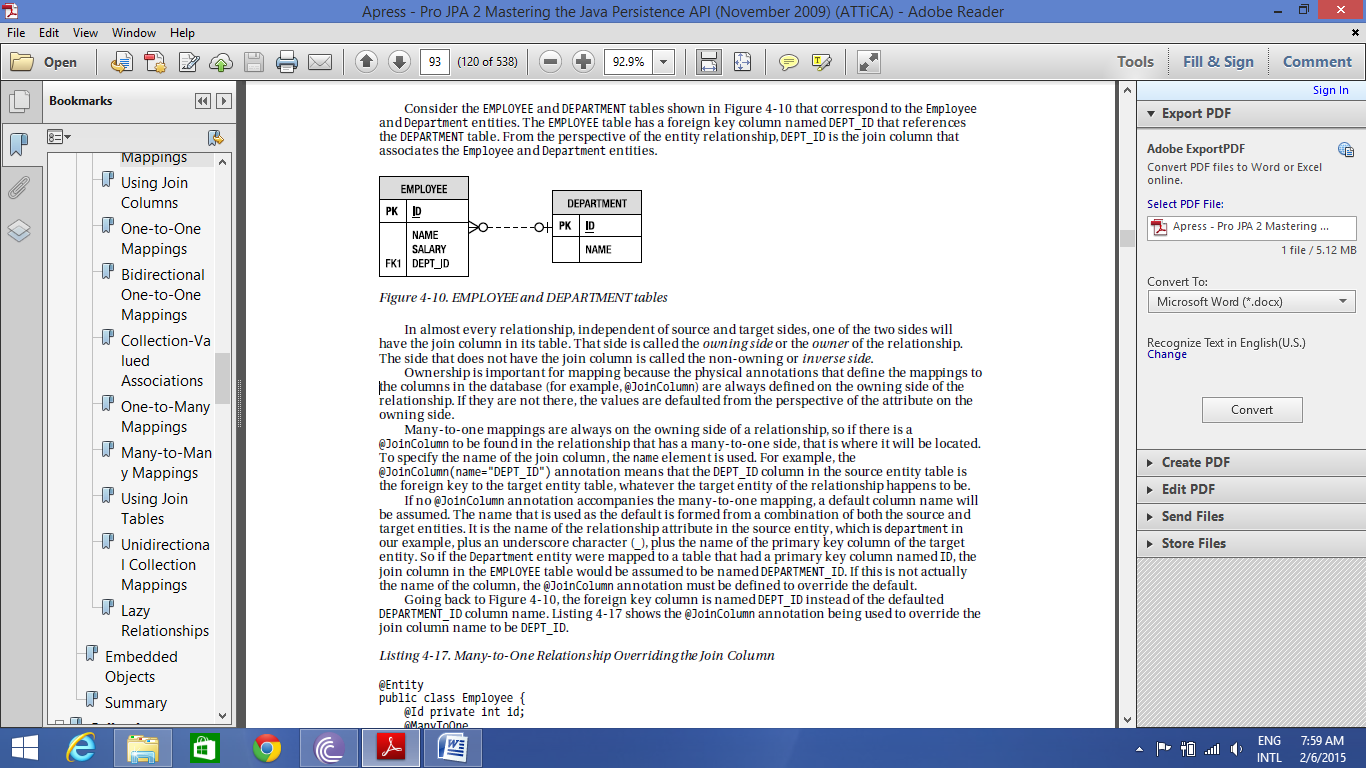
Using Join Columns (required when using @many-to-one above. if you don't give it, as default one will be taken)

In the database, a relationship mapping means that one table has a reference to another table. The

database term for a column that refers to a key (usually the primary key) in another table is a *foreign*

*key* column. In JPA, we call them *join columns*, and the @JoinColumn annotation is the primary

annotation used to configure these types of columns.



In almost every relationship, independent of source and target sides, one of the two sides will

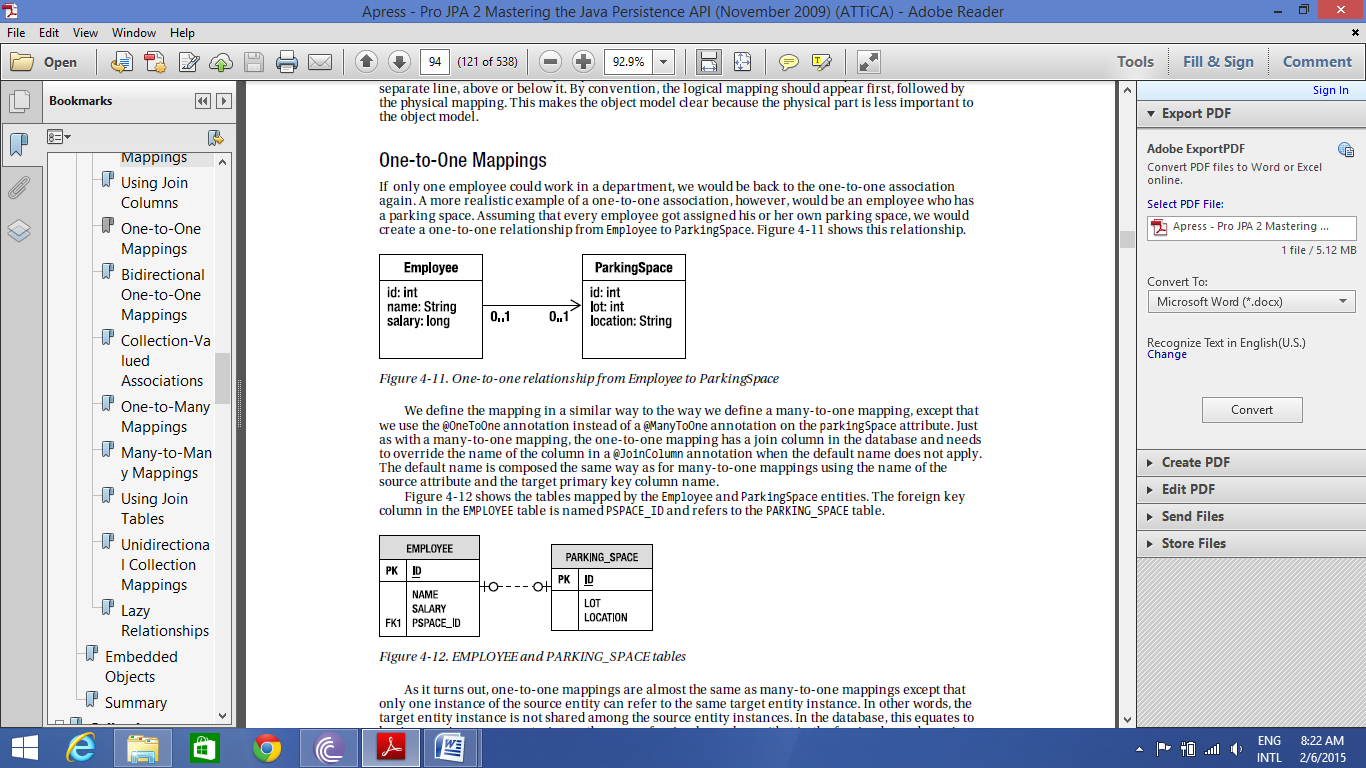
have the join column in its table. That side is called the *owning side* or the *owner* of the relationship.

The side that does not have the join column is called the non-owning or *inverse side*.

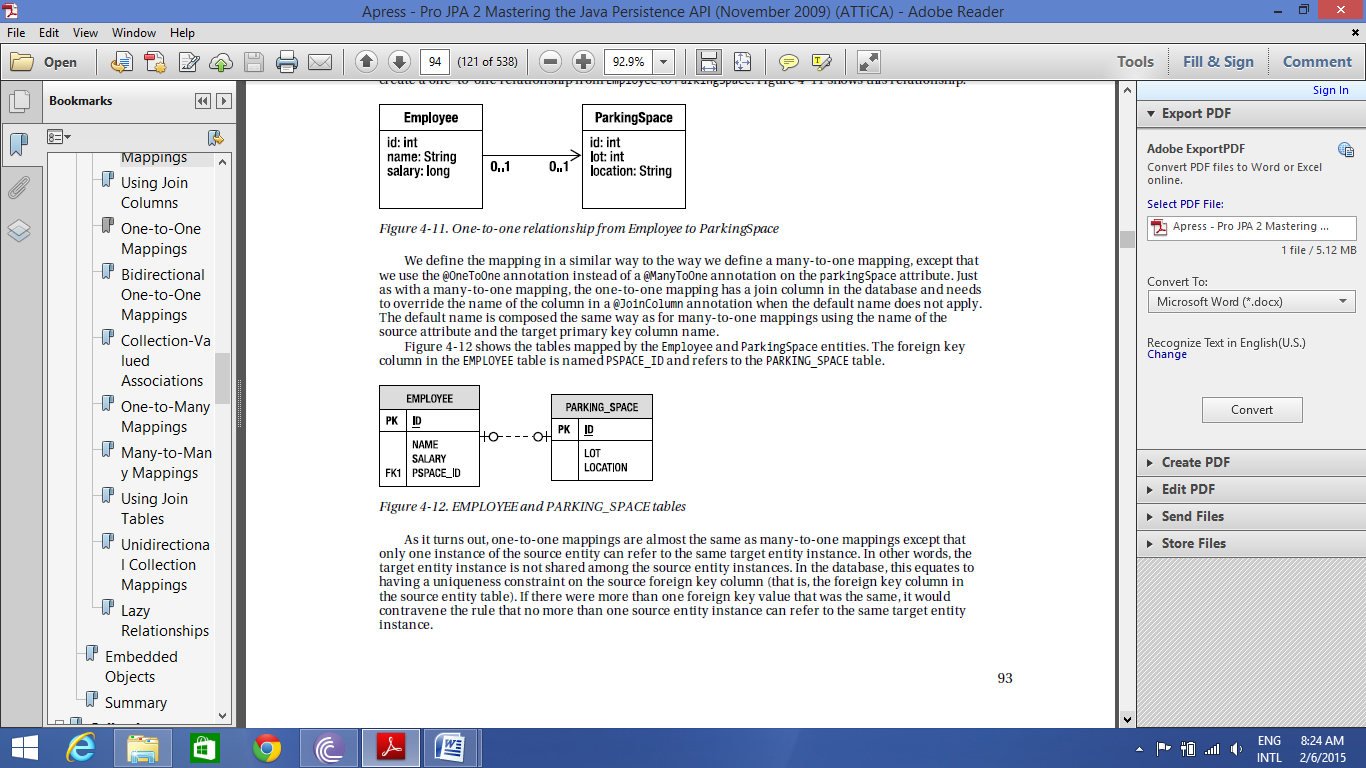
If no join column is given, then the default join column will be of the form <owning side table name>\_ID. In order to override it, we give the above @joinColumn annotation with name attribute in it.

One-to-One Mappings:

An example of this mapping is an employee who has a parking space. Assuming that every employee got assigned his or her own parking space, we would create a one-to-one relationship from Employee to ParkingSpace



The foreign key column in the EMPLOYEE table is named PSPACE\_ID and refers to the PARKING\_SPACE table.



In terms of code, nothing will change as compared to many-to-one. Only @many-to-one will be changed to @one-to-one.

public class Employee {

@Id private int id;

private String name;

@OneToOne

@JoinColumn(name="PSPACE\_ID")

private ParkingSpace parkingSpace;

// ...

}