Hibernate Annotations

Reference Guide

Version: 3.1 beta 9

Table of Contents

Preface

Hibernate, like all other object/relational mapping tools, requires metadata that governs the transformation of data from one representation to the other (and vice versa). In Hibernate 2.x, mapping metadata is most of the time declared in XML text files. Another option is XDoclet, utilizing Javadoc source code annotations and a preprocessor at compile time. The same kind of annotation support is now available in the standard JDK, although more powerful and better supported by tools. IntelliJ IDEA, and Eclipse for example, support autocompletion and syntax highlighting of JDK 5.0 annotations. Annotations are compiled into the bytecode and read at runtime (in Hibernate's case on startup) using reflection, so no external XML files are needed.

The EJB3 specification recognizes the interest and the success of the transparent object/relational mapping paradigm. The EJB3 specification standardizes the basic APIs and the metadata needed for any object/relational persistence mechanism. *Hibernate EntityManager* implements the programming interfaces and lifecycle rules as defined by the EJB3 persistence specification. Together with *Hibernate Annotations*, this wrapper implements a complete (and standalone) EJB3 persistence solution on top of the mature Hibernate core. You may use a combination of all three together, annotations without EJB3 programming interfaces and lifecycle, or even pure native Hibernate, depending on the business and technical needs of your project. You can at all times fall back to Hibernate native APIs, or if required, even to native JDBC and SQL.

Please note that this documentation is based on a preview release of the Hibernate Annotations that follows the public final draft of EJB 3.0/JSR-220 persistence annotations. This work is already very close to the final concepts in the new specification. Our goal is to provide a complete set of ORM annotations, including EJB3 standard annotations as well as Hibernate3 extensions for cases not covered by the specification. Eventually you will be able to create all possible mappings with annotations. See the JIRA road mapsection for more information.

The EJB3 Public final draft has change some annotations, please refer to http://www.hibernate.org/371.html as a migration guide between Hibernate Annotations 3.1beta7 and 3.1beta8.

Chapter 1. Setting up an annotations project

1.1. Requirements

- Download and unpack the Hibernate Annotations distribution from the Hibernate website.
- *This preview release requires Hibernate 3.2.0.CR2 and above. Do not use this release of Hibernate Annotations with an older version of Hibernate 3.x!*
- This release is known to work on Hibernate core 3.2.0.CR2
- Make sure you have JDK 5.0 installed. You can of course continue using XDoclet and get some of the benefits of annotation-based metadata with older JDK versions. Note that this document only describes JDK 5.0 annotations and you have to refer to the XDoclet documentation for more information.

1.2. Configuration

First, set up your classpath (after you have created a new project in your favorite IDE):

- Copy all Hibernate3 core and required 3rd party library files (see lib/README.txt in Hibernate).
- Copy `hibernate-annotations.jar` and `lib/ejb3-persistence.jar` from the Hibernate Annotations distribution to your classpath as well.
- To use the Chapter 4, *Hibernate Lucene Integration*, add the lucene jar file.

We also recommend a small wrapper class to startup Hibernate in a static initializer block, known as `HibernateUtil`. You might have seen this class in various forms in other areas of the Hibernate documentation. For Annotation support you have to enhance this helper class as follows:

```
package hello;

import org.hibernate.*;
import org.hibernate.cfg.*;
import test.*;
import test.animals.Dog;

public class HibernateUtil {

private static final SessionFactory sessionFactory;

    static {
        try {

            sessionFactory = new AnnotationConfiguration().buildSessionFactory();
        } catch (Throwable ex) {
            // Log exception!
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static Session getSession()
            throws HibernateException {
        return sessionFactory.openSession();
    }
}
```

```
```

Interesting here is the use of `AnnotationConfiguration`. The packages and annotated classes are declared in your regular XML configuration file (usually `hibernate.cfg.xml`). Here is the equivalent of the above declaration:

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

        <hibernate-configuration>
          <session-factory>
            <mapping package="test.animals"/>
            <mapping class="test.Flight"/>
            <mapping class="test.Sky"/>
            <mapping class="test.Person"/>
            <mapping class="test.animals.Dog"/>
          </session-factory>
        </hibernate-configuration>
```

Note that you can mix the hbm.xml use and the new annotation one.

Alternatively, you can define the annotated classes and packages using the programmatic API

```
            sessionFactory = new AnnotationConfiguration()
                    .addPackage("test.animals") //the fully qualified package name
                    .addAnnotatedClass(Flight.class)
                    .addAnnotatedClass(Sky.class)
                    .addAnnotatedClass(Person.class)
                    .addAnnotatedClass(Dog.class)
                    .buildSessionFactory();
```

You can also use the Hibernate Entity Manager which has it's own configuration mechanism. Please refer to this project documentation for more details.

There is no other difference in the way you use Hibernate APIs with annotations, except for this startup routine change or in the configuration file. You can use your favorite configuration method for other properties (`hibernate.properties`, `hibernate.cfg.xml`, programmatic APIs, etc). You can even mix annotated persistent classes and classic `hbm.cfg.xml` declarations with the same `SessionFactory`. You can however not declare a class several times (whether annotated or through hbm.xml). You cannot mix configuration strategies (hbm vs annotations) in a mapped entity hierarchy either.

To ease the migration process from hbm files to annotations, the configuration mechanism detects the mapping duplication between annotations and hbm files. HBM files are then prioritized over annotated metadata on a class to class basis. You can change the priority using `hibernate.mapping.precedence` property. The default is `hbm, class`, changing it to `class, hbm` will prioritize the annotated classes over hbm files when a conflict occurs.

Chapter 2. Entity Beans

2.1. Intro

This section covers EJB 3.0 entity bean annotations and Hibernate-specific extensions.

2.2. Mapping with EJB3 Annotations

EJB3 entity beans are plain POJOs. Actually they represent the exact same concept as the Hibernate persistent entities. Their mappings are defined through JDK 5.0 annotations (an XML descriptor syntax for overriding will be defined in the EJB3 specification, but it's not finalized so far). Annotations can be split in two categories, the logical mapping annotations (allowing you to describe the object model, the class associations, etc.) and the physical mapping annotations (describing the physical schema, tables, columns, indexes, etc). We will mix annotations from both categories in the following code examples.

EJB3 annotations are in the `javax.persistence.*` package. Most JDK 5 compliant IDE (like Eclipse, IntelliJ IDEA and Netbeans) can autocomplete annotation interfaces and attributes for you (even without a specific "EJB3" module, since EJB3 annotations are plain JDK 5 annotations).

For more and runnable concrete examples read the JBoss EJB 3.0 tutorial or review the Hibernate Annotations test suite. Most of the unit tests have been designed to represent a concrete example and be a inspiration source.

2.2.1. Declaring an entity bean

Every bound persistent POJO class is an entity bean and is declared using the `@Entity` annotation (at the class level):

```
@Entity
public class Flight implements Serializable {
    Long id;

    @Id
    public Long getId() { return id; }

    public void setId(Long id) { this.id = id; }
}
```

`@Entity` declares the class as an entity bean (i.e. a persistent POJO class), `@Id` declares the identifier property of this entity bean. The other mapping declarations are implicit. This configuration by exception concept is central to the new EJB3 specification and a major improvement. The class Flight is mapped to the Flight table, using the column id as its primary key column.

Depending on whether you annotate fields or methods, the access type used by Hibernate will be `field` or `property`. The EJB3 spec requires that you declare annotations on the element type that will be accessed, i.e. the getter method if you use `property` access, the field if you use `field` access. Mixing EJB3 annotations in both fields and methods should be avoided. Hibernate will guess the access type from the position of `@Id` or `@EmbeddedId`.

Defining the table

`@Table` is set at the class level; it allows you to define the table, catalog, and schema names for your entity bean mapping. If no `@Table` is defined the default values are used: the unqualified class name of the entity.

```
@Entity
@Table(name="tbl_sky")
public class Sky implements Serializable {
...
```

The `@Table` element also contains a `schema` and a `catalog` attributes, if they need to be defined. You can also define unique constraints to the table using the `@UniqueConstraint` annotation in conjunction with `@Table` (for a unique constraint bound to a single column, refer to `@Column`).

```
@Table(name="tbl_sky",
    uniqueConstraints = {@UniqueConstraint(columnNames={"month", "day"})}
)
```

A unique constraint is applied to the tuple month, day. Note that the `columnNames` array refers to the logical column names.
The logical column name is defined by the Hibernate NamingStrategy implementation. The default EJB3 naming strategy use the physical column name as the logical column name. Note that this may be different than the property name (if the column name is explicit). Unless you override the NamingStrategy, you shouldn't worry about that.

Versioning for optimistic locking

You can add optimistic locking capability to an entity bean using the `@Version` annotation:

```
@Entity
public class Flight implements Serializable {
...
    @Version
    @Column(name="OPTLOCK")
    public Integer getVersion() { ... }
}
```

The version property will be mapped to the `OPTLOCK` column, and the entity manager will use it to detect conflicting updates (preventing lost updates you might otherwise see with the last-commit-wins strategy).

The version column may be a numeric (the recommended solution) or a timestamp as per the EJB3 spec. Hibernate support any kind of type provided that you define and implement the appropriate `UserVersionType`.

2.2.2. Mapping simple properties

Declaring basic property mappings

Every non static non transient property (field or method) of an entity bean is considered persistent, unless you annotate it as `@Transient`. Not having an annotation for your property is equivalent to the appropriate `@Basic` annotation. The `@Basic` annotation allows you to declare the fetching strategy for a property:

```
public transient int counter; //transient property

private String firstname; //persistent property

@Transient
String getLengthInMeter() { ... } //transient property
```

```
String getName() {... } // persistent property

@Basic
int getLength() { ... } // persistent property

@Basic(fetch = FetchType.LAZY)
String getDetailedComment() { ... } // persistent property

@Temporal(TemporalType.TIME)
java.util.Date getDepartureTime() { ... } // persistent property

@Enumerated(STRING)
Starred getNote() { ... } //enum persisted as String in database
```

`counter`, a transient field, and `lengthInMeter`, a method annotated as `@Transient`, and will be ignored by the entity manager. `name`, `length`, and `firstname` properties are mapped persistent and eagerly fetched (the default for simple properties). The `detailedComment` property value will be lazily fetched from the database once a lazy property of the entity is accessed for the first time. Usually you don't need to lazy simple properties (not to be confused with lazy association fetching).

Note

To enable property level lazy fetching, your classes have to be instrumented: bytecode is added to the original one to enable such feature, please refer to the Hibernate reference documentation. If your classes are not instrumented, property level lazy loading is silently ignored.

The recommended alternative is to use the projection capability of EJB-QL or Criteria queries.

EJB3 support property mapping of all basic types supported by Hibernate (all basic Java types , their respective wrappers and serializable classes). Hibernate Annotations support out of the box Enum type mapping either into a ordinal column (saving the enum ordinal) or a string based column (saving the enum string representation): the persistence representation, defaulted to ordinal, can be overriden through the `@Enumerated` annotation as shown in the `note` property example.

In core Java APIs, the temporal precision is not defined. When dealing with temporal data you might want to describe the expected precision in database. Temporal data can have `DATE`, `TIME`, or `TIMESTAMP` precision (ie the actual date, only the time, or both). Use the `@Temporal` annotation to fine tune that.

`@Lob` indicates that the property should be persisted in a Blob or a Clob depending on the property type: `java.sql.Clob`, `Character[]`, `char[]` and java.lang.`String` will be persisted in a Clob. `java.sql.Blob`, `Byte[]`, `byte[]` and serializable type will be persisted in a Blob.

```
@Lob
public String getFullText() {
    return fullText;
}

@Lob
public byte[] getFullCode() {
    return fullCode;
}
```

If the property type implements `java.io.Serializable` and is not a basic type, and if the property is not annotated with `@Lob`, then the Hibernate `serializable` type is used.

Declaring column attributes

The column(s) used for a property mapping can be defined using the `@Column` annotation. Use it to override default values (see the EJB3 specification for more information on the defaults). You can use this annotation at the property level for properties that are:

- not annotated at all
- annotated with `@Basic`
- annotated with `@Version`
- annotated with `@Lob`
- annotated with `@Temporal`
- annotated with `@org.hibernate.annotations.CollectionOfElements` (for Hibernate only)

```
@Entity
public class Flight implements Serializable {
...
@Column(updatable = false, name = "flight_name", nullable = false, length=50)
public String getName() { ... }
```

The `name` property is mapped to the `flight_name` column, which is not nullable, has a length of 50 and is not updatable (making the property immutable).

This annotation can be applied to regular properties as well as `@Id` or `@Version` properties.

```
@Column(
    name="columnName";                                  (1)
    boolean unique() default false;                     (2)
    boolean nullable() default true;                    (3)
    boolean insertable() default true;                  (4)
    boolean updatable() default true;                   (5)
    String columnDefinition() default "";               (6)
    String table() default "";                          (7)
    int length() default 255;                           (8)
    int precision() default 0; // decimal precision     (9)
    int scale() default 0; // decimal scale
```

(1) `name` (optional): the column name (default to the property name)
(2) `unique` (optional): set a unique constraint on this column or not (default false)
(3) `nullable` (optional): set the column as nullable (default false).
(4) `insertable` (optional): whether or not the column will be part of the insert statement (default true)
(5) `updatable` (optional): whether or not the column will be part of the update statement (default true)
(6) `columnDefinition` (optional): override the sql DDL fragment for this particular column (non portable)
(7) `table` (optional): define the targeted table (default primary table)
(8) `length` (optional): column length (default 255)
(8) `precision` (optional): column decimal precision (default 0)
(10) `scale` (optional): column decimal scale if useful (default 0)

Embedded objects (aka components)

It is possible to declare an embedded component inside an entity and even override its column mapping. Component classes have to be annotated at the class level with the `@Embeddable` annotation. It is possible to override the column mapping of an embedded object for a particular entity using the `@Embedded` and

`@AttributeOverride` annotation in the associated property:

```
@Entity
public class Person implements Serializable {

    // Persistent component using defaults
    Address homeAddress;

    @Embedded
    @AttributeOverrides( {
            @AttributeOverride(name="iso2", column = @Column(name="bornIso2") ),
            @AttributeOverride(name="name", column = @Column(name="bornCountryName") )
    } )
    Country bornIn;
    ...
}
```

```
@Embeddable
public class Address implements Serializable {
    String city;
    Country nationality; //no overriding here
}
```

```
@Embeddable
public class Country implements Serializable {
    private String iso2;
    @Column(name="countryName") private String name;

    public String getIso2() { return iso2; }
    public void setIso2(String iso2) { this.iso2 = iso2; }


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

A embeddable object inherit the access type of its owning entity (note that you can override that using the Hibernate specific `@AccessType` annotations (see Hibernate Annotation Extensions).

The `Person` entity bean has two component properties, `homeAddress` and `bornIn`. `homeAddress` property has not been annotated, but Hibernate will guess that it is a persistent component by looking for the `@Embeddable` annotation in the Address class. We also override the mapping of a column name (to `bornCountryName`) with the `@Embedded` and `@AttributeOverride` annotations for each mapped attribute of `Country`. As you can see, `Country` is also a nested component of `Address`, again using auto-detection by Hibernate and EJB3 defaults. Overriding columns of embedded objects of embedded objects is currently not supported in the EJB3 spec, however, Hibernate Annotations supports it through dotted expressions.

```
    @Embedded
    @AttributeOverrides( {
            @AttributeOverride(name="city", column = @Column(name="fld_city") )
            @AttributeOverride(name="nationality.iso2", column = @Column(name="nat_Iso2") ),
            @AttributeOverride(name="nationality.name", column = @Column(name="nat_CountryName") )
            //nationality columns in homeAddress are overridden
    } )
    Address homeAddress;
```

Hibernate Annotations supports one more feature that is not explicitly supported by the EJB3 specification.

You can annotate a embedded object with the `@MappedSuperclass` annotation to make the superclass properties persistent (see `@MappedSuperclass` for more informations).

While not supported by the EJB3 specification, Hibernate Annotations allows you to use association annotations in an embeddable object (ie `@*ToOne` nor `@*ToMany`). To override the association columns you can use `@AssociationOverride`.

If you want to have the same embeddable object type twice in the same entity, the column name defaulting will not work: at least one of the columns will have to be explicit. Hibernate goes beyond the EJB3 spec and allows you to enhance the defaulting mechanism through the `NamingStrategy`. `DefaultComponentSafeNamingStrategy` is a small improvement over the default EJB3NamingStrategy that allows embedded objects to be defaulted even if used twice in the same entity.

Non-annotated property defaults

If a property is not annotated, the following rules apply:

- If the property is of a single type, it is mapped as @Basic
- Otherwise, if the type of the property is annotated as @Embeddable, it is mapped as @Embedded
- Otherwise, if the type of the property is Serializable, it is mapped as @Basic in a column holding the object in its serialized version
- Otherwise, if the type of the property is java.sql.Clob or java.sql.Blob, it is mapped as @Lob with the appropriate LobType

2.2.. Mapping identifier properties

The `@Id` annotation lets you define which property is the identifier of your entity bean. This property can be set by the application itself or be generated by Hibernate (preferred). You can define the identifier generation strategy thanks to the `@GeneratedValue` annotation:

- AUTO - either identity column, sequence or table depending on the underlying DB
- TABLE - table holding the id
- IDENTITY - identity column
- SEQUENCE - sequence

Hibernate provides more id generators than the basic EJB3 ones. Check Hibernate Annotation Extensions for more informations.

The following example shows a sequence generator using the SEQ_STORE configuration (see below)

```
@Id @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="SEQ_STORE")
public Integer getId() { ... }
```

The next example uses the identity generator:

```
@Id @GeneratedValue(strategy=GenerationType.IDENTITY)
```

```
public Long getId() { ... }
```

The AUTO generator is the preferred type for portable applications (across several DB vendors). The identifier generation configuration can be shared for several @Id mappings with the generator attribute. There are several configurations available through @SequenceGenerator and @TableGenerator. The scope of a generator can be the application or the class. Class-defined generators are not visible outside the class and can override application level generators. Application level generators are defined at package level (see package-info.java):

```
@javax.persistence.TableGenerator(
    name="EMP_GEN",
    table="GENERATOR_TABLE",
    pkColumnName = "key",
    valueColumnName = "hi"
    pkColumnValue="EMP",
    allocationSize=20
)
@javax.persistence.SequenceGenerator(
    name="SEQ_GEN",
    sequenceName="my_sequence"
)
package org.hibernate.test.metadata;
```

If package-info.java in the org.hibernate.test.metadata package is used to initialize the EJB configuration, EMP_GEN and SEQ_GEN are application level generators. EMP_GEN defines a table based id generator using the hilo algorithm with a max_lo of 20. The hi value is kept in a table "GENERATOR_TABLE". The information is kept in a row where pkColumnName "key" is equals to pkColumnValue "EMP" and column valueColumnName "hi" contains the the next high value used.

SEQ_GEN defines a sequence generator using a sequence named my_sequence. Note that this version of Hibernate Annotations does not handle initialValue and allocationSize parameters in the sequence generator.

The next example shows the definition of a sequence generator in a class scope:

```
@Entity
@javax.persistence.SequenceGenerator(
    name="SEQ_STORE",
    sequenceName="my_sequence"
)
public class Store implements Serializable {
    private Long id;

    @Id @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="SEQ_STORE")
    public Long getId() { return id; }
}
```

This class will use a sequence named my_sequence and the SEQ_STORE generator is not visible in other classes. Note that you can check the Hibernate Annotations tests in the org.hibernate.test.metadata.id package for more examples.

You can define a composite primary key through several syntaxes:

- annotate the component property as @Id and make the component class @Embeddable
- annotate the component property as @EmbeddedId

- annotate the class as @IdClass and annotate each property of the entity involved in the primary key with @Id

While quite common to the EJB2 developer, `@IdClass` is likely new for Hibernate users. The composite primary key class corresponds to multiple fields or properties of the entity class, and the names of primary key fields or properties in the primary key class and those of the entity class must match and their types must be the same. Let's look at an example:

```
@Entity
@IdClass(FootballerPk.class)
public class Footballer {
    //part of the id key
    @Id public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    //part of the id key
    @Id public String getLastname() {
        return lastname;
    }

    public void setLastname(String lastname) {
        this.lastname = lastname;
    }

    public String getClub() {
        return club;
    }

    public void setClub(String club) {
        this.club = club;
    }

    //appropriate equals() and hashCode() implementation
}

@Embeddable
public class FootballerPk implements Serializable {
    //same name and type as in Footballer
    public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    //same name and type as in Footballer
    public String getLastname() {
        return lastname;
    }

    public void setLastname(String lastname) {
        this.lastname = lastname;
    }

    //appropriate equals() and hashCode() implementation
}
```

As you may have seen, `@IdClass` points to the corresponding primary key class.

While not supported by the EJB3 specification, Hibernate allows you to define associations inside a composite

identifier. Simply use the regular annotations for that

```
@Entity
@AssociationOverride( name="id.channel", joinColumns = @JoinColumn(name="chan_id") )
public class TvMagazin {
    @EmbeddedId public TvMagazinPk id;
    @Temporal(TemporalType.TIME) Date time;
}

@Embeddable
public class TvMagazinPk implements Serializable {
    @ManyToOne
    public Channel channel;
    public String name;
    @ManyToOne
    public Presenter presenter;
}
```

2.2.4. Mapping inheritance

EJB3 supports the three types of inheritance:

- Table per Class Strategy: the <union-class> element in Hibernate
- Single Table per Class Hierarchy Strategy: the <subclass> element in Hibernate
- Joined Subclass Strategy: the <joined-subclass> element in Hibernate

The chosen strategy is declared at the class level of the top level entity in the hierarchy using the `@Inheritance` annotation.

Note

Annotating interfaces is currently not supported.

Table per class

This strategy has many drawbacks (esp. with polymorphic queries and associations) explained in the EJB3 spec, the Hibernate reference documentation, Hibernate in Action, and many other places. Hibernate work around most of them implementing this strategy using `SQL UNION` queries. It is commonly used for the top level of an inheritance hierarchy:

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Flight implements Serializable {
```

This strategy support one to many associations provided that they are bidirectional. This strategy does not support the `IDENTITY` generator strategy: the id has to be shared across several tables. Consequently, when using this strategy, you should not use `AUTO` nor `IDENTITY`.

Single table per class hierarchy

All properties of all super- and subclasses are mapped into the same table, instances are distinguished by a special discriminator column:

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name="planetype",
    discriminatorType=DiscriminatorType.STRING
)
@DiscriminatorValue("Plane")
public class Plane { ... }

@Entity
@DiscriminatorValue("A320")
public class A320 extends Plane { ... }
```

`Plane` is the superclass, it defines the inheritance strategy `InheritanceType.SINGLE_TABLE`. It also defines the discriminator column through the `@DiscriminatorColumn` annotation, a discriminator column can also define the discriminator type. Finally, the `@DiscriminatorValue` annotation defines the value used to differentiate a class in the hierarchy. All of these attributes have sensible default values. The default name of the discriminator column is `DTYPE`. The default discriminator value is the entity name (as defined in `@Entity.name`) for DiscriminatorType.STRING. `A320` is a subclass; you only have to define discriminator value if you don't want to use the default value. The strategy and the discriminator type are implicit.

`@Inheritance` and `@DiscriminatorColumn` should only be defined at the top of the entity hierarchy.

Joined subclasses

The `@PrimaryKeyJoinColumn` and `@PrimaryKeyJoinColumns` annotations define the primary key(s) of the joined subclass table:

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Boat implements Serializable { ... }

@Entity
public class Ferry extends Boat { ... }

@Entity
@PrimaryKeyJoinColumn(name="BOAT_ID")
public class AmericaCupClass  extends Boat { ... }
```

All of the above entities use the `JOINED` strategy, the `Ferry` table is joined with the `Boat` table using the same primary key names. The `AmericaCupClass` table is joined with `Boat` using the join condition `Boat.id = AmericaCupClass.BOAT_ID`.

Inherit properties from superclasses

This is sometimes useful to share common properties through a technical or a business superclass without including it as a regular mapped entity (ie no specific table for this entity). For that purpose you can map them as `@MappedSuperclass`.

```
@MappedSuperclass
public class BaseEntity {
    @Basic
    @Temporal(TemporalType.TIMESTAMP)
    public Date getLastUpdate() { ... }
    public String getLastUpdater() { ... }
    ...
}
```

```
@Entity class Order extends BaseEntity {
    @Id public Integer getId() { ... }
    ...
}
```

In database, this hierarchy will be represented as an `Order` table having the `id`, `lastUpdate` and `lastUpdater` columns. The embedded superclass property mappings are copied into their entity subclasses. Remember that the embeddable superclass is not the root of the hierarchy though.

Note

Properties from superclasses not mapped as `@MappedSuperclass` are ignored.

Note

The access type (field or methods), is inherited from the root entity, unless you use the Hibernate annotation `@AccessType`

Note

The same notion can be applied to `@Embeddable` objects to persist properties from their superclasses. You also need to use `@MappedSuperclass` to do that (this should not be considered as a standard EJB3 feature though)

Note

It is allowed to mark a class as `@MappedSuperclass` in the middle of the mapped inheritance hierarchy.

Note

Any class in the hierarchy non annotated with `@MappedSuperclass` nor `@Entity` will be ignored.

You can override columns defined in entity superclasses at the root entity level using the `@AttributeOverride` annotation.

```
@MappedSuperclass
public class FlyingObject implements Serializable {

    public int getAltitude() {
        return altitude;
    }

    @Transient
    public int getMetricAltitude() {
        return metricAltitude;
    }

    @ManyToOne
    public PropulsionType getPropulsion() {
        return metricAltitude;
    }
    ...
}

@Entity
@AttributeOverride( name="altitude", column = @Column(name="fld_altitude") )
@AssociationOverride( name="propulsion", joinColumns = @JoinColumn(name="fld_propulsion_fk") )
public class Plane extends FlyingObject {
    ...
}
```

The `altitude` property will be persisted in an `fld_altitude` column of table `Plane` and the propulsion association will be materialized in a `fld_propulsion_fk` foreign key column.

You can define `@AttributeOverride`(s) and `@AssociationOverride`(s) on `@Entity` classes, `@MappedSuperclass` classes and properties pointing to an `@Embeddable` object.

2.2.5. Mapping entity bean associations/relationships

One-to-one

You can associate entity beans through a one-to-one relationship using `@OneToOne`. There are two cases for one-to-one associations: either the associated entities share the same primary keys values or a foreign key is held by one of the entities (note that this FK column in the database should be constrained unique to simulate one-to-one multiplicity).

First, we map a real one-to-one association using shared primary keys:

```
@Entity
public class Body {
    @Id
    public Long getId() { return id; }

    @OneToOne(cascade = CascadeType.ALL)
    @PrimaryKeyJoinColumn
    public Heart getHeart() {
        return heart;
    }
    ...
}
```

```
@Entity
public class Heart {
    @Id
    public Long getId() { ...}
}
```

The one to one is marked as true by using the `@PrimaryKeyJoinColumn` annotation.

In the following example, the associated entities are linked through a foreign key column:

```
@Entity
public class Customer implements Serializable {
    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name="passport_fk")
    public Passport getPassport() {
        ...
    }

@Entity
public class Passport implements Serializable {
    @OneToOne(mappedBy = "passport")
    public Customer getOwner() {
    ...
}
```

A `Customer` is linked to a `Passport`, with a foreign key column named `passport_fk` in the `Customer` table. The join column is declared with the `@JoinColumn` annotation which looks like the `@Column` annotation. It has one more parameters named `referencedColumnName`. This parameter declares the column in the targeted entity that will be used to the join. Note that when using `referencedColumnName` to a non primary key column, the associated class has to be `Serializable`. Also note that the `referencedColumnName` to a non primary key column has to be mapped to a property having a single column (other cases might not work).

The association may be bidirectional. In a bidirectional relationship, one of the sides (and only one) has to be the owner: the owner is responsible for the association column(s) update. To declare a side as *not* responsible for the relationship, the attribute `mappedBy` is used. `mappedBy` refers to the property name of the association on the owner side. In our case, this is `passport`. As you can see, you don't have to (must not) declare the join column since it has already been declared on the owners side.

If no `@JoinColumn` is declared on the owner side, the defaults apply. A join column(s) will be created in the owner table and its name will be the concatenation of the name of the relationship in the owner side, `_` (underscore), and the name of the primary key column(s) in the owned side. In this example `passport_id` because the property name is `passport` and the column id of `Passport` is `id`.

Many-to-one

Many-to-one associations are declared at the property level with the annotation `@ManyToOne`:

```
@Entity()
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE} )
    @JoinColumn(name="COMP_ID")
    public Company getCompany() {
        return company;
    }
    ...
}
```

The `@JoinColumn` attribute is optional, the default value(s) is like in one to one, the concatenation of the name of the relationship in the owner side, `_` (underscore), and the name of the primary key column in the owned side. In this example `company_id` because the property name is `company` and the column id of Company is `id`.

`@ManyToOne` has a parameter named `targetEntity` which describes the target entity name. You usually don't need this parameter since the default value (the type of the property that stores the association) is good in almost all cases. However this is useful when you want to use interfaces as the return type instead of the regular entity.

```
@Entity()
public class Flight implements Serializable {
    @ManyToOne( cascade = {CascadeType.PERSIST, CascadeType.MERGE}, targetEntity=CompanyImpl.class )
    @JoinColumn(name="COMP_ID")
    public Company getCompany() {
        return company;
    }
    ...
}

public interface Company {
    ...
```

Collections

Overview

You can map `Collection`, `List` (ie ordered lists, not indexed lists), `Map` and `Set`. The EJB3 specification describes how to map an ordered list (ie a list ordered at load time) using `@javax.persistence.OrderBy` annotation: this annotation takes into parameter a list of comma separated (target entity) properties to order the collection by (eg `firstname asc, age desc`), if the string is empty, the collection will be ordered by id. `@OrderBy` currently works only on collections having no association table. For true indexed collections, please refer to the Hibernate Annotation Extensions. EJB3 allows you to map Maps using as a key one of the target entity property using `@MapKey(name="myProperty")` (myProperty is a property name in the target entity). When using `@MapKey` (without property name), the target entity primary key is used. The map key uses the same column as the property pointed out: there is no additional column defined to hold the map key, and it does make sense since the map key actually represent a target ptoperty. Be aware that once loaded, the key is no longer kept in sync with the property, in other words, if you change the property value, the key will not change automatically in your Java model (Map support the way Hibernate 3 does is currently not supported in this release). Many people confuse `<map>` capabilities and `@MapKey` ones. These are two different features. `@MapKey` still has some limitations, please check the forum or the JIRA tracking system for more informations.

Hibernate has several notions of collections.

Table 2.1. Collections semantics

Semantic	java representation	annotations
Bag semantic	java.util.List, java.util.Collection	@org.hibernate.annotations.CollectionOfElements, @OneToMany, @ManyToMany
List semantic	java.util.List	@org.hibernate.annotations.CollectionOfElements, @OneToMany, @ManyToMany + @OrderBy, @org.hibernate.annotations.IndexColumn
Set semantic	java.util.Set	@org.hibernate.annotations.CollectionOfElements, @OneToMany, @ManyToMany
Map semantic	java.util.Map	@org.hibernate.annotations.CollectionOfElements, @OneToMany, @ManyToMany + @MapKey

So specifically, java.util.List collections wo @OrderBy nor @org.hibernate.annotations.IndexColumn are going to be considered as bags.

Collection of primitive, core type or embedded objects is not supported by the EJB3 specification. Hibernate Annotations allows them however (see Hibernate Annotation Extensions).

```
@Entity public class City {
    @OneToMany(mappedBy="city")
    @OrderBy("streetName")
    public List<Street> getStreets() {
        return streets;
    }
...
}
```

```
@Entity public class Street {
    public String getStreetName() {
        return streetName;
    }

    @ManyToOne
    public City getCity() {
        return city;
    }
    ...
}


@Entity
public class Software {
    @OneToMany(mappedBy="software")
    @MapKey(name="codeName")
    public Map<String, Version> getVersions() {
        return versions;
    }
...
}

@Entity
@Table(name="tbl_version")
public class Version {
    public String getCodeName() {...}

    @ManyToOne
    public Software getSoftware() { ... }
...
}
```

So `City` has a collection of `Streets` that are ordered by `streetName` (of `Street`) when the collection is loaded. `Software` has a map of `Versions` which key is the `Version` `codeName`.

Unless the collection is a generic, you will have to define `targetEntity`. This is a annotation attribute that take the target entity class as a value.

One-to-many

One-to-many associations are declared at the property level with the annotation `@OneToMany`. One to many associations may be bidirectional.

Bidirectional

Since many to one are (almost) always the owner side of a bidirectional relationship in the EJB3 spec, the one to many association is annotated by `@OneToMany( mappedBy=... )`

```
@Entity
public class Troop {
    @OneToMany(mappedBy="troop")
    public Set<Soldier> getSoldiers() {
    ...
}

@Entity
public class Soldier {
    @ManyToOne
    @JoinColumn(name="troop_fk")
    public Troop getTroop() {
    ...
}
```

`Troop` has a bidirectional one to many relationship with `Soldier` through the `troop` property. You don't have to

(must not) define any physical mapping in the `mappedBy` side.

To map a bidirectional one to many, with the one-to-many side as the owning side, you have to remove the `mappedBy` element and set the many to one `@JoinColumn` as insertable and updatable to false. This solution is obviously not optimized from the number of needed statements.

```
@Entity
public class Troop {
    @OneToMany
    @JoinColumn(name="troop_fk") //we need to duplicate the physical information
    public Set<Soldier> getSoldiers() {
    ...
}

@Entity
public class Soldier {
    @ManyToOne
    @JoinColumn(name="troop_fk", insertable=false, updatable=false)
    public Troop getTroop() {
    ...
}
```

Unidirectional

A unidirectional one to many using a foreign key column in the owned entity is not that common and not really recommended. We strongly advise you to use a join table for this kind of association (as explained in the next section). This kind of association is described through a `@JoinColumn`

```
@Entity
public class Customer implements Serializable {
    @OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
    @JoinColumn(name="CUST_ID")
    public Set<Ticket> getTickets() {
    ...
}

@Entity
public class Ticket implements Serializable {
    ... //no bidir
}
```

`Customer` describes a unidirectional relationship with `Ticket` using the join column `CUST_ID`.

Unidirectional with join table

A unidirectional one to many with join table is much preferred. This association is described through an `@JoinTable`.

```
@Entity
public class Trainer {
    @OneToMany
    @JoinTable(
            name="TrainedMonkeys",
            joinColumns = { @JoinColumn( name="trainer_id") },
            inverseJoinColumns = @JoinColumn( name="monkey_id")
    )
    public Set<Monkey> getTrainedMonkeys() {
    ...
}

@Entity
public class Monkey {
    ... //no bidir
```

```
}
```

`Trainer` describes a unidirectional relationship with `Monkey` using the join table `TrainedMonkeys`, with a foreign key `trainer_id` to `Trainer` (`joinColumns`) and a foreign key `monkey_id` to `Monkey` (`inversejoinColumns`).

Defaults

Without describing any physical mapping, a unidirectional one to many with join table is used. The table name is the concatenation of the owner table name, _, and the other side table name. The foreign key name(s) referencing the owner table is the concatenation of the owner table, _, and the owner primary key column(s) name. The foreign key name(s) referencing the other side is the concatenation of the owner property name, _, and the other side primary key column(s) name. A unique constraint is added to the foreign key referencing the other side table to reflect the one to many.

```
@Entity
public class Trainer {
    @OneToMany
    public Set<Tiger> getTrainedTigers() {
    ...
}

@Entity
public class Tiger {
    ... //no bidir
}
```

`Trainer` describes a unidirectional relationship with `Tiger` using the join table `Trainer_Tiger`, with a foreign key `trainer_id` to `Trainer` (table name, _, trainer id) and a foreign key `trainedTigers_id` to `Monkey` (property name, _, Tiger primary column).

Many-to-many

Definition

A many-to-many association is defined logically using the `@ManyToMany` annotation. You also have to describe the association table and the join conditions using the `@JoinTable` annotation. If the association is bidirectional, one side has to be the owner and one side has to be the inverse end (ie. it will be ignored when updating the relationship values in the association table):

```
@Entity
public class Employer implements Serializable {
    @ManyToMany(
        targetEntity=org.hibernate.test.metadata.manytomany.Employee.class,
        cascade={CascadeType.PERSIST, CascadeType.MERGE}
    )
    @JoinTable(
        name="EMPLOYER_EMPLOYEE",
        joinColumns={@JoinColumn(name="EMPER_ID")},
        inverseJoinColumns={@JoinColumn(name="EMPEE_ID")}
    )
    public Collection getEmployees() {
        return employees;
    }
    ...
}
```

```
@Entity
public class Employee implements Serializable {
    @ManyToMany(
        cascade={CascadeType.PERSIST, CascadeType.MERGE},
        mappedBy="employees"
        targetEntity=Employer.class
    )
    public Collection getEmployers() {
        return employers;
    }
}
```

We've already shown the many declarations and the detailed attributes for associations. We'll go deeper in the `@JoinTable` description, it defines a `name`, an array of join columns (an array in annotation is defined using { A, B, C }), and an array of inverse join columns. The latter ones are the columns of the association table which refer to the `Employee` primary key (the "other side").

As seen previously, the other side don't have to (must not) describe the physical mapping: a simple `mappedBy` argument containing the owner side property name bind the two.

Default values

As any other annotations, most values are guessed in a many to many relationship. Without describing any physical mapping in a unidirectional many to many the following rules applied. The table name is the concatenation of the owner table name, _ and the other side table name. The foreign key name(s) referencing the owner table is the concatenation of the owner table name, _ and the owner primary key column(s). The foreign key name(s) referencing the other side is the concatenation of the owner property name, _, and the other side primary key column(s). These are the same rules used for a unidirectional one to many relationship.

```
@Entity
public class Store {
    @ManyToMany(cascade = CascadeType.PERSIST)
    public Set<City> getImplantedIn() {
        ...
    }
}

@Entity
public class City {
    ... //no bidirectional relationship
}
```

A `Store_Table` is used as the join table. The `Store_id` column is a foreign key to the `Store` table. The `implantedIn_id` column is a foreign key to the `City` table.

Without describing any physical mapping in a bidirectional many to many the following rules applied. The table name is the concatenation of the owner table name, _ and the other side table name. The foreign key name(s) referencing the owner table is the concatenation of the other side property name, _, and the owner primary key column(s). The foreign key name(s) referencing the other side is the concatenation of the owner property name, _, and the other side primary key column(s). These are the same rules used for a unidirectional one to many relationship.

```
@Entity
public class Store {
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    public Set<Customer> getCustomers() {
```

```
        ...
    }
}

@Entity
public class Customer {
    @ManyToMany(mappedBy="customers")
    public Set<Store> getStores() {
        ...
    }
}
```

A `Store_Customer` is used as the join table. The `stores_id` column is a foreign key to the `Store` table. The `customers_id` column is a foreign key to the `City` table.

Transitive persistence with cascading

You probably have noticed the `cascade` attribute taking an array of `CascadeType` as a value. The cascade concept in EJB3 is very is similar to the transitive persistence and cascading of operations in Hibernate, but with slightly different semantics and cascading types:

- CascadeType.PERSIST: cascades the persist (create) operation to associated entities persist() is called or if the entity is managed
- CascadeType.MERGE: cascades the merge operation to associated entities if merge() is called or if the entity is managed
- CascadeType.REMOVE: cascades the remove operation to associated entities if delete() is called
- CascadeType.REFRESH: cascades the refresh operation to associated entities if refresh() is called
- CascadeType.ALL: all of the above

Please refer to the chapter 6.3 of the EJB3 specification for more information on cascading and create/merge semantics.

Association fetching

You have the ability to either eagerly or lazily fetch associated entities. The `fetch` parameter can be set to `FetchType.LAZY` or `FetchType.EAGER`. `EAGER` will try to use an outer join select to retrieve the associated object, while `LAZY` is the default and will only trigger a select when the associated object is accessed for the first time. EJBQL also has a `fetch` keyword that allows you to override laziness when doing a particular query. This is very useful to improve performance and is decided on a use case to use case basis.

2.2.6. Mapping composite primary and foreign keys

Composite primary keys use a embedded class as the primary key representation, so you'd use the `@Id` and `@Embeddable` annotations. Alternatively, you can use the `@EmbeddedId` annotation. Note that the dependent class has to be serializable and implements `equals()`/`hashCode()`. You can also use `@IdClass` as described in Mapping identifier properties.

```
@Entity
public class RegionalArticle implements Serializable {

    @Id
```

```
    public RegionalArticlePk getPk() { ... }
}

@Embeddable
public class RegionalArticlePk implements Serializable { ... }
```

or alternatively

```
@Entity
public class RegionalArticle implements Serializable {

    @EmbeddedId
    public RegionalArticlePk getPk() { ... }
}

public class RegionalArticlePk implements Serializable { ... }
```

`@Embeddable` inherit the access type of its owning entity unless the Hibernate specific annotation `@AccessType` is used. Composite foreign keys (if not using the default sensitive values) are defined on associations using the `@JoinColumns` element, which is basically an array of `@JoinColumn`. It is considered a good practice to express `referencedColumnNames` explicitly. Otherwise, Hibernate will suppose that you use the same order of columns as in the primary key declaration.

```
@Entity
public class Parent implements Serializable {
    @Id
    public ParentPk id;
    public int age;

    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumns ({
        @JoinColumn(name="parentCivility", referencedColumnName = "isMale"),
        @JoinColumn(name="parentLastName", referencedColumnName = "lastName"),
        @JoinColumn(name="parentFirstName", referencedColumnName = "firstName")
    })
    public Set<Child> children; //unidirectional
    ...
}
```

```
@Entity
public class Child implements Serializable {
    @Id @GeneratedValue
    public Integer id;

    @ManyToOne
    @JoinColumns ({
        @JoinColumn(name="parentCivility", referencedColumnName = "isMale"),
        @JoinColumn(name="parentLastName", referencedColumnName = "lastName"),
        @JoinColumn(name="parentFirstName", referencedColumnName = "firstName")
    })
    public Parent parent; //unidirectional
}
```

```
@Embeddable
public class ParentPk implements Serializable {
    String firstName;
    String lastName;
    ...
```

```
}
```

Note the explicit usage of the `referencedColumnName`.

2.2.7. Mapping secondary tables

You can map a single entity bean to several tables using the `@SecondaryTable` or `@SecondaryTables` class level annotations. To express that a column is in a particular table, use the `table` parameter of `@Column` or `@JoinColumn`.

```
@Entity
@Table(name="MainCat")
@SecondaryTables({
    @SecondaryTable(name="Cat1", pkJoinColumns={
        @PrimaryKeyJoinColumn(name="cat_id", referencedColumnName="id")
    ),
    @SecondaryTable(name="Cat2", uniqueConstraints={@UniqueConstraint(columnNames={"storyPart2"})})
})
public class Cat implements Serializable {

    private Integer id;
    private String name;
    private String storyPart1;
    private String storyPart2;

    @Id @GeneratedValue
    public Integer getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    @Column(table="Cat1")
    public String getStoryPart1() {
        return storyPart1;
    }

    @Column(table="Cat2")
    public String getStoryPart2() {
        return storyPart2;
    }
```

In this example, `name` will be in `MainCat`. `storyPart1` will be in `Cat1` and `storyPart2` will be in `Cat2`. `Cat1` will be joined to `MainCat` using the `cat_id` as a foreign key, and `Cat2` using `id` (ie the same column name, the `MainCat` id column has). Plus a unique constraint on `storyPart2` has been set.

Check out the JBoss EJB 3 tutorial or the Hibernate Annotations unit test suite for more examples.

2.3. Mapping Queries

2.3.1. Mapping EJBQL/HQL queries

You can map EJBQL/HQL queries using annotations. `@NamedQuery` and `@NamedQueries` can be defined at the class or at the package level. However their definitions are global to the session factory/entity manager factory scope. A named query is defined by its name and the actual query string.

```
javax.persistence.NamedQueries(
    @javax.persistence.NamedQuery(name="plane.getAll", query="select p from Plane p")
)
package org.hibernate.test.annotations.query;

...

@Entity
@NamedQuery(name="night.moreRecentThan", query="select n from Night n where n.date >= :date")
public class Night {
    ...
}

public class MyDao {
    doStuff() {
        Query q = s.getNamedQuery("night.moreRecentThan");
        q.setDate( "date", aMonthAgo );
        List results = q.list();
        ...
    }
    ...
}
```

You can also provide some hints to a query through an array of `QueryHint` through a `hints` attribute.

The availabe Hibernate hints are

Table 2.2. Query hints

hint	description
org.hibernate.cacheable	Whether the query should interact with the second level cache (defualt to false)
org.hibernate.cacheRegion	Cache region name (default used otherwise)
org.hibernate.timeout	Query timeout
org.hibernate.fetchSize	resultset fetch size
org.hibernate.flushMode	Flush mode used for this query
org.hibernate.cacheMode	Cache mode used for this query
org.hibernate.readOnly	Entities loaded by this query should be in read only mode or not (default to false)
org.hibernate.comment	Query comment added to the generated SQL

2.3.2. Mapping native queries

You can also map a native query (ie a plain SQL query). To achieve that, you need to describe the SQL result-set structure using `@SqlResultSetMapping` (or `@SqlResultSetMappings` if you plan to define several resulset mappings). Like `@NamedQuery`, a `@SqlResultSetMapping` can be defined at both package level or class level. However its scope is global to the application. As we will see, a `resultSetMapping` parameter is defined the `@NamedNativeQuery`, it represents the name of a defined `@SqlResultSetMapping`. The resultset mapping declares the entities retrieved by this native query. Each field of the entity is bound to an SQL alias (or column

name). All fields of the entity including the ones of subclasses and the foreign key columns of related entities have to be present in the SQL query. Field definitions are optional provided that they map to the same column name as the one declared on the class property.

```
@NamedNativeQuery(name="night&area", query="select night.id nid, night.night_duration, "
    + " night.night_date, area.id aid, night.area_id, area.name "
    + "from Night night, Area area where night.area_id = area.id", resultSetMapping="joinMapping")
@SqlResultSetMapping(name="joinMapping", entities={
    @EntityResult(entityClass=org.hibernate.test.annotations.query.Night.class, fields = {
        @FieldResult(name="id", column="nid"),
        @FieldResult(name="duration", column="night_duration"),
        @FieldResult(name="date", column="night_date"),
        @FieldResult(name="area", column="area_id"),
        discriminatorColumn="disc"
    }),
    @EntityResult(entityClass=org.hibernate.test.annotations.query.Area.class, fields = {
        @FieldResult(name="id", column="aid"),
        @FieldResult(name="name", column="name")
    })
    }
)
```

In the above example, the `night&area` named query use the `joinMapping` result set mapping. This mapping returns 2 entities, `Night` and `Area`, each property is declared and associated to a column name, actually the column name retrieved by the query. Let's now see an implicit declaration of the property / column.

```
@Entity
@SqlResultSetMapping(name="implicit", entities=@EntityResult(entityClass=org.hibernate.test.annotation
@NamedNativeQuery(name="implicitSample", query="select * from SpaceShip", resultSetMapping="implicit"
public class SpaceShip {
    private String name;
    private String model;
    private double speed;

    @Id
    public String getName() {
        return name;
    }

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

    @Column(name="model_txt")
    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public double getSpeed() {
        return speed;
    }

    public void setSpeed(double speed) {
        this.speed = speed;
    }
}
```

In this example, we only describe the entity member of the result set mapping. The property / column mappings is done using the entity mapping values. In this case the `model` property is bound to the `model_txt` column. If the association to a related entity involve a composite primary key, a `@FieldResult` element should be used for each foreign key column. The `@FieldResult` name is composed of the property name for the relationship, fol-

lowed by a dot ("."), followed by the name or the field or property of the primary key.

```
@Entity
@SqlResultSetMapping(name="compositekey",
        entities=@EntityResult(entityClass=org.hibernate.test.annotations.query.SpaceShip.class,
            fields = {
                    @FieldResult(name="name", column = "name"),
                    @FieldResult(name="model", column = "model"),
                    @FieldResult(name="speed", column = "speed"),
                    @FieldResult(name="captain.firstname", column = "firstn"),
                    @FieldResult(name="captain.lastname", column = "lastn"),
                    @FieldResult(name="dimensions.length", column = "length"),
                    @FieldResult(name="dimensions.width", column = "width")
                    }),
        columns = { @ColumnResult(name = "surface"),
                    @ColumnResult(name = "volume") } )

@NamedNativeQuery(name="compositekey",
    query="select name, model, speed, lname as lastn, fname as firstn, length, width, length * width a
    resultSetMapping="compositekey")
} )
public class SpaceShip {
    private String name;
    private String model;
    private double speed;
    private Captain captain;
    private Dimensions dimensions;

    @Id
    public String getName() {
        return name;
    }

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

    @ManyToOne(fetch= FetchType.LAZY)
    @JoinColumns( {
            @JoinColumn(name="fname", referencedColumnName = "firstname"),
            @JoinColumn(name="lname", referencedColumnName = "lastname")
            } )
    public Captain getCaptain() {
        return captain;
    }

    public void setCaptain(Captain captain) {
        this.captain = captain;
    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public double getSpeed() {
        return speed;
    }

    public void setSpeed(double speed) {
        this.speed = speed;
    }

    public Dimensions getDimensions() {
        return dimensions;
    }

    public void setDimensions(Dimensions dimensions) {
        this.dimensions = dimensions;
```

```
    }
}

@Entity
@IdClass(Identity.class)
public class Captain implements Serializable {
    private String firstname;
    private String lastname;

    @Id
    public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    @Id
    public String getLastname() {
        return lastname;
    }

    public void setLastname(String lastname) {
        this.lastname = lastname;
    }
}
```

Note

If you look at the dimension property, you'll see that Hibernate supports the dotted notation for embedded objects (you can even have nested embedded objects). EJB3 implementations do not have to support this feature, we do :-)

If you retrieve a single entity and if you use the default mapping, you can use the `resultClass` attribute instead of `resultSetMapping`:

```
@NamedNativeQuery(name="implicitSample", query="select * from SpaceShip",
    resultClass=SpaceShip.class)
public class SpaceShip {
```

In some of your native queries, you'll have to return scalar values, for example when building report queries. You can map them in the `@SqlResultsetMapping` through `@ColumnResult`. You actually can even mix, entities and scalar returns in the same native query (this is probably not that common though).

```
@SqlResultSetMapping(name="scalar", columns=@ColumnResult(name="dimension"))
@NamedNativeQuery(name="scalar", query="select length*width as dimension from SpaceShip", resultSetMap
```

An other query hint specific to native queries has been introduced: `org.hibernate.callable` which can be true or false depending on whether the query is a stored procedure or not.

2.4. Hibernate Annotation Extensions

Hibernate 3.1 offers a variety of additional annotations that you can mix/match with your EJB 3 entities. They have been designed as a natural extension of EJB3 annotations.

To empower the EJB3 capabilities, hibernate provides specific annotations that match hibernate features. The `org.hibernate.annotations` package contains all these annotations extensions.

2.4.1. Entity

You can fine tune some of the actions done by Hibernate on entities beyond what the EJB3 spec offers.

`@org.hibernate.annotations.Entity` adds additional metadata that may be needed beyond what is defined in the standard `@Entity`

- mutable: whether this entity is mutable or not
- dynamicInsert: allow dynamic SQL for inserts
- dynamicUpdate: allow dynamic SQL for updates
- selectBeforeUpdate: Specifies that Hibernate should never perform an SQL UPDATE unless it is certain that an object is actually modified.
- polymorphism: whether the entity polymorphism is of PolymorphismType.IMPLICIT (default) or PolymorphismType.EXPLICIT
- persister: allow the overriding of the default persister implementation
- optimisticLock: optimistic locking strategy (OptimisticLockType.VERSION, OptimisticLockType.NONE, OptimisticLockType.DIRTY or OptimisticLockType.ALL)

Note

@javax.persistence.Entity is still mandatory, @org.hibernate.annotations.Entity is not a replacement.

Here are some additional Hibernate annotation extensions

`@org.hibernate.annotations.BatchSize` allows you to define the batch size when fetching instances of this entity (eg. `@BatchSize(size=4)`). When loading a given entity, Hibernate will then load all the uninitialized entities of the same type in the persistence context up to the batch size.

`@org.hibernate.annotations.Proxy` defines the laziness attributes of the entity. lazy (default to true) define whether the class is lazy or not. proxyClassName is the interface used to generate the proxy (default is the class itself).

`@org.hibernate.annotations.Where` defines an optional SQL WHERE clause used when instances of this class is retrieved.

`@org.hibernate.annotations.Check` defines an optional check constraints defined in the DDL statetement.

`@OnDelete(action=OnDeleteAction.CASCADE)` on joined subclasses: use a SQL cascade delete on deletion instead of the regular Hibernate mechanism.

`@Table(appliesTo="tableName", indexes = { @Index(name="index1", columnNames={"column1", "column2"} ) } )` creates the defined indexes on the columns of table `tableName`. This can be applied on the primary table or any secondary table. The `@Tables` annotation allows your to apply indexes on different tables. This annotation is expected where `@javax.persistence.Table` or `@javax.persistence.SecondaryTable(s)` occurs.

Note

`@org.hibernate.annotations.Table` is a complement, not a replacement to `@javax.persistence.Table`. Especially, if you want to change the default name of a table, you must use `@javax.persistence.Table`, not `@org.hibernate.annotations.Table`.

```
@Entity
@BatchSize(size=5)
@org.hibernate.annotations.Entity(
        selectBeforeUpdate = true,
        dynamicInsert = true, dynamicUpdate = true,
        optimisticLock = OptimisticLockType.ALL,
        polymorphism = PolymorphismType.EXPLICIT)
@Where(clause="1=1")
@org.hibernate.annotations.Table(name="Forest", indexes = { @Index(name="idx", columnNames = { "name",
public class Forest { ... }
```

```
@Entity
@Inheritance(
    strategy=InheritanceType.JOINED
)
public class Vegetable { ... }

@Entity
@OnDelete(action=OnDeleteAction.CASCADE)
public class Carrot extends Vegetable { ... }
```

2.4.2. Identifier

`@org.hibernate.annotations.GenericGenerator` allows you to define an Hibernate specific id generator.

```
@Id @GeneratedValue(generator="system-uuid")
@GenericGenerator(name="system-uuid", strategy = "uuid")
public String getId() {

@Id @GeneratedValue(generator="hibseq")
@GenericGenerator(name="hibseq", strategy = "seqhilo",
    parameters = {
        @Parameter(name="max_lo", value = "5"),
        @Parameter(name="sequence", value="heybabyhey")
    }
)
public Integer getId() {
```

`strategy` is the short name of an Hibernate3 generator strategy or the fully qualified class name of an `IdentifierGenerator` implementation. You can add some parameters through the `parameters` attribute

2.4.3. Property

Access type

The access type is guessed from the position of `@Id` or `@EmbeddedId` in the entity hierarchy. Sub-entities, embedded objects and mapped superclass inherit the access type from the root entity.

In Hibernate, you can override the access type to:

- use a custom access type strategy
- fine tune the access type at the class level or at the property level

An @AccessType annotation has been introduced to support this behavior. You can define the access type on

- an entity
- a superclass
- an embeddable object
- a property

The access type is overriden for the annotated element, if overriden on a class, all the properties of the given class inherit the access type. For root entities, the access type is considered to be the default one for the whole hierarchy (overridable at class or property level).

If the access type is marked as "property", the getters are scanned for annotations, if the access type is marked as "field", the fields are scanned for annotations. Otherwise the elements marked with @Id or @embeddedId are scanned.

You can override an access type for a property, but the element to annotate will not be influenced: for example an entity having access type `field`, can annotate a field with `@AccessType("property")`, the access type will then be property for this attribute, the the annotations still have to be carried on the field.

If a superclass or an embeddable object is not annotated, the root entity access type is used (even if an access type has been define on an intermediate superclass or embeddable object). The russian doll principle does not apply.

```
@Entity
public class Person implements Serializable {
    @Id  @GeneratedValue //access type field
    Integer id;

    @Embedded
    @AttributeOverrides({
    @AttributeOverride(name = "iso2", column = @Column(name = "bornIso2")),
    @AttributeOverride(name = "name", column = @Column(name = "bornCountryName"))
            })
    Country bornIn;
}

@Embeddable
@AccessType("property") //override access type for all properties in Country
public class Country implements Serializable {
    private String iso2;
    private String name;

    public String getIso2() {
        return iso2;
    }

    public void setIso2(String iso2) {
        this.iso2 = iso2;
    }

    @Column(name = "countryName")
    public String getName() {
        return name;
    }

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

Formula

Sometimes, you want the Database to do some computation for you rather than in the JVM, you might also create some kind of virtual column. You can use a SQL fragment (aka formula) instead of mapping a property into a column. This kind of property is read only (its value is calculated by your formula fragment).

```
@Formula("obj_length * obj_height * obj_width")
public long getObjectVolume()
```

The SQL fragment can be as complex as you want avec even include subselects.

Type

`@org.hibernate.annotations.Type` overrides the default hibernate type used: this is generally not necessary since the type is correctly inferred by Hibernate. Please refer to the Hibernate reference guide for more informations on the Hibernate types.

`@org.hibernate.annotations.TypeDef` and `@org.hibernate.annotations.TypeDefs` allows you to declare type definitions. These annotations are placed at the class or package level. Note that these definitions will be global for the session factory (even at the class level) and that type definition has to be defined before any usage.

```
@TypeDefs(
    {
    @TypeDef(
        name="caster",
        typeClass = CasterStringType.class,
        parameters = {
            @Parameter(name="cast", value="lower")
        }
    )
    }
)
package org.hibernate.test.annotations.entity;

...
public class Forest {
    @Type(type="caster")
    public String getSmallText() {
    ...
}
```

When using composite user type, you will have to express column definitions. The `@Columns` has been introduced for that purpose.

```
@Type(type="org.hibernate.test.annotations.entity.MonetaryAmountUserType")
@Columns(columns = {
    @Column(name="r_amount"),
    @Column(name="r_currency")
})
public MonetaryAmount getAmount() {
    return amount;
}


public class MonetaryAmount implements Serializable {
    private BigDecimal amount;
    private Currency currency;
    ...
}
```

Index

You can define an index on a particular column using the `@Index` annotation on a one column property, the columnNames attribute will then be ignored

```
@Column(secondaryTable="Cat1")
@Index(name="story1index")
public String getStoryPart1() {
    return storyPart1;
}
```

2.4.4. Inheritance

SINGLE_TABLE is a very powerful strategy but sometimes, and especially for legacy systems, you cannot add an additional discriminator column. For that purpose Hibernate has introduced the notion of discriminator formula: `@DiscriminatorFormula` is a replacement of `@DiscriminatorColumn` and use a SQL fragment as a formula for discriminator resolution (no need to have a dedicated column).

```
@Entity
@DiscriminatorForumla("case when forest_type is null then 0 else forest_type end")
public class Forest { ... }
```

2.4.5. Single Association related annotations

By default, when Hibernate cannot resolve the association because the expected associated element is not in database (wrong id on the association column), an exception is raised by Hibernate. This might be inconvenient for lecacy and badly maintained schemas. You can ask Hibernate to ignore such elements instead of raising an exception using the `@NotFound` annotation. This annotation can be used on a `@OneToOne` (with FK), `@ManyToOne`, `@OneToMany` or `@ManyToMany` association.

```
@Entity
public class Child {
    ...
    @ManyToOne
    @NotFound(action=NotFoundAction.IGNORE)
    public Parent getParent() { ... }
    ...
}
```

Sometimes you want to delegate to your database the deletion of cascade when a given entity is deleted.

```
@Entity
public class Child {
    ...
    @ManyToOne
    @OnDelete(action=OnDeleteAction.CASCADE)
    public Parent getParent() { ... }
    ...
}
```

In this case Hibernate generates a cascade delete constraint at the database level.

2.4.6. Collection related annotations

Parameter annotations

It is possible to set

- the batch size for collections using @BatchSize
- the where clause, using @Where
- the check clause, using @Check
- the SQL order by clause, using @OrderBy
- the delete cascade strategy through @OnDelete(action=OnDeleteAction.CASCADE)

You can also declare a sort comparator. Use the `@Sort` annotation. Expressing the comparator type you want between unsorted, natural or custom comparator. If you want to use your own comparator implementation, you'll also have to express the implementation class using the `comparator` attribute.

```
    @OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
    @JoinColumn(name="CUST_ID")
    @Sort(type = SortType.COMPARATOR, comparator = TicketComparator.class)
    @Where(clause="1=1")
    @OnDelete(action=OnDeleteAction.CASCADE)
    public SortedSet<Ticket> getTickets() {
        return tickets;
    }
```

Please refer to the previous descriptions of these annotations for more informations.

Extra collection types

Beyond EJB3, Hibernate Annotations supports true `List` and `Array`. Map your collection the same way as usual and add the `@IndexColumn`. This annotation allows you to describe the column that will hold the index. You can also declare the index value in DB that represent the first element (aka as base index). The usual value is `0` or `1`.

```
@OneToMany(cascade = CascadeType.ALL)
@IndexColumn(name = "drawer_position", base=1)
public List<Drawer> getDrawers() {
    return drawers;
}
```

Note

If you forgot to set `@IndexColumn`, the bag semantic is applied

Hibernate Annotations also supports collections of core types (Integer, String, Enums, ...), collections of embeddable objects and even arrays of primitive types. This is known as collection of elements.

A collection of elements as to be annotated as `@CollectionOfElements` (as a replacement of `@OneToMany`) To define the collection table, the `@JoinTable` annotation is used on the association property, `joinColumns` defines the join columns between the entity primary table and the collection table (inverseJoincolumn is useless and should be left empty). For collection of core types or array of primitive types, you can override the element column definition using a `@Column` on the association property. You can also override the columns of a collection of embeddable object using `@AttributeOverride`.

```
@Entity
public class Boy {
    private Integer id;
    private Set<String> nickNames = new HashSet<String>();
```

```
    private int[] favoriteNumbers;
    private Set<Toy> favoriteToys = new HashSet<Toy>();
    private Set<Character> characters = new HashSet<Character>();

    @Id @GeneratedValue
    public Integer getId() {
        return id;
    }

    @CollectionOfElements
    public Set<String> getNickNames() {
        return nickNames;
    }

    @CollectionOfElements
    @JoinTable(
            table=@Table(name="BoyFavoriteNumbers"),
            joinColumns = @JoinColumn(name="BoyId")
    )
    @Column(name="favoriteNumber", nullable=false)
    @IndexColumn(name="nbr_index")
    public int[] getFavoriteNumbers() {
        return favoriteNumbers;
    }

    @CollectionOfElements
    @AttributeOverride( name="serial", column=@Column(name="serial_nbr") )
    public Set<Toy> getFavoriteToys() {
        return favoriteToys;
    }

    @CollectionOfElements
    public Set<Character> getCharacters() {
        return characters;
    }
    ...
}

public enum Character {
    GENTLE,
    NORMAL,
    AGGRESSIVE,
    ATTENTIVE,
    VIOLENT,
    CRAFTY
}

@Embeddable
public class Toy {
    public String name;
    public String serial;

    public String getName() {
        return name;
    }

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

    public String getSerial() {
        return serial;
    }

    public void setSerial(String serial) {
        this.serial = serial;
    }

    public boolean equals(Object o) {
        if ( this == o ) return true;
        if ( o == null || getClass() != o.getClass() ) return false;
```

```
        final Toy toy = (Toy) o;

        if ( !name.equals( toy.name ) ) return false;
        if ( !serial.equals( toy.serial ) ) return false;

        return true;
    }

    public int hashCode() {
        int result;
        result = name.hashCode();
        result = 29 * result + serial.hashCode();
        return result;
    }
}
```

Note

Previous versions of Hibernate Annotations used the `@OneToMany` to mark a collection of elements. Due to semantic inconsistencies, we've introduced the annotation `@CollectionOfElements`. Marking collections of elements the old way still work but is considered deprecated and is going to be unsupported in future releases

2.4.7. Cache

In order to optimize your database accesses, you can activave the so called second level cache of Hibernate. This cache is configurable on a per entity and per collection basis.

`@org.hibernate.annotations.Cache` defines the caching strategy and region of a given second level cache. This annotation can be applied on the root entity (not the sub entities), and on the collections.

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Forest { ... }
```

```
    @OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
    @JoinColumn(name="CUST_ID")
    @Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
    public SortedSet<Ticket> getTickets() {
        return tickets;
    }
```

```
@Cache(
    CacheConcurrencyStrategy usage();                (1)
    String region() default "";                      (2)
    String include() default "all";                  (3)
)
```

(1) usage: the given cache concurrency strategy (NONE, READ_ONLY, NONSTRICT_READ_WRITE, READ_WRITE, TRANSACTIONAL)

(2) region (optional): the cache region (default to the fqcn of the class or the fq role name of the collection)

(3) `include` (optional): all to include all properties, non-lazy to only include non lazy properties (default all).

2.4.8. Filters

Hibernate has the notion of data filter that can be applied at runtime on a given session. Those filters has to be defined first.

`@org.hibernate.annotations.FilterDef` or `@FilterDefs` define filter definition(s) used by filter(s) using the same name. A filter definition has a name() and an array of parameters(). A `@ParamDef` has a name and a type. You can also define a defaultCondition() parameter for a given `@filterDef` to set the default condition to use when none are defined in the `@Filter`. A `@FilterDef`(s) can be defined at the class or package level.

We now need to define the SQL filter clause applied to either the entity load or the collection load. `@Filter` is used and placed either on the entity or the collection element

```
@Entity
@FilterDef(name="minLength", parameters={ @ParamDef( name="minLength", type="integer" ) } )
@Filters( {
    @Filter(name="betweenLength", condition=":minLength <= length and :maxLength >= length"),
    @Filter(name="minLength", condition=":minLength <= length")
} )
public class Forest { ... }
```

2.4.9. Queries

Since Hibernate has more features on named queries than the one defined in the EJB3 specification, `@org.hibernate.annotations.NamedQuery`, `@org.hibernate.annotations.NamedQueries`, `@org.hibernate.annotations.NamedNativeQuery` and `@org.hibernate.annotations.NamedNativeQueries` have been introduced. They add some attributes to the standard version and can be used as a replacement:

- flushMode: define the query flush mode (Always, Auto, Commit or Never)
- cacheable: whether the query should be cached or not
- cacheRegion: cache region used if the query is cached
- fetchSize: JDBC statement fetch size for this query
- timeout: query time out
- callable: for native queries only, to be set to true for stored procedures
- comment: if comments are activated, the comment seen when the query is sent to the database.
- cacheMode: Cache interaction mode (get, ignore, normal, put or refresh)
- readOnly: whether or not the elements retrievent from the query are in read only mode.

Note, that the EJB3 public final draft has introduced the notion of `@QueryHint`, which is probably a better way to define those hints.

Chapter 3. Hibernate Validator

Annotations are a very convenient and elegant way to specify invariant constraints for a domain model. You can, for example, express that a property should never be null, that the account balance should be strictly positive, etc. These domain model constraints are declared in the bean itself by annotating its properties. A validator can then read them and check for constraint violations. The validation mechanism can be executed in different layers in your application without having to duplicate any of these rules (presentation layer, data access layer). Hibernate Validator has been designed for that purpose.

Hibernate Validator works at two levels. First, it is able to check in-memory instances of a class for constraint violations. Second, it can apply the constraints to the Hibernate metamodel and incorporate them into the generated database schema.

Each constraint annotation is associated to a validator implementation responsible for checking the constraint on the entity instance. A validator can also (optionally) apply the constraint to the Hibernate metamodel, allowing Hibernate to generate DDL that expresses the constraint. With the appropriate event listener, you can execute the checking operation on inserts and updates done by Hibernate. Hibernate Validator is not limited to use with Hibernate. You can easily use it anywhere in your application.

When checking instances at runtime, Hibernate Validator returns information about constraint violations in an array of `InvalidValues`. Among other information, the `InvalidValue` contains an error description message that can embed the parameter values bundle with the annotation (eg. length limit), and message strings that may be externalized to a `ResourceBundle`.

3.1. Constraints

3.1.1. What is a constraint?

A constraint is represented by an annotation. A constraint usually has some attributes used to parameterize the constraints limits. The constraint apply to the annotated element.

3.1.2. Built in constraints

Hibernate Validator comes with some built-in constraints, which covers most basic data checks. As we'll see later, you're not limited to them, you can in a minute write your own constraints.

Table 3.1. Built-in constraints

Annotation	Apply on	Runtime checking	Hibernate Metadata impact
@Length(min=, max=)	property (String)	check if the string length match the range	Column length will be set to max
@Max(value=)	property (numeric or string representation of a numeric)	check if the value is less than or equals to max	Add a check constraint on the column
@Min(value=)	property (numeric or string representation of a numeric)	check if the value is more than or equals to min	Add a check constraint on the column

Annotation	Apply on	Runtime checking	Hibernate Metadata impact
@NotNull	property	check if the value is not null	Column(s) are not null
@Past	property (date or calendar)	check if the date is in the past	Add a check constraint on the column
@Future	property (date or calendar)	check if the date is in the future	none
@Pattern(regex="regexp", flag=)	property (string)	check if the property match the regular expression given a match flag (see `java.util.regex.Pattern`)	none
@Range(min=, max=)	property (numeric or string representation of a numeric)	check if the value is between min and max (included)	Add a check constraint on the column
@Size(min=, max=)	property (array, collection, map)	check if the element size is between min and max (included)	none
@AssertFalse	property	check that the method evaluates to false (useful for constraints expressed in code rather than annotations)	none
@AssertTrue	property	check that the method evaluates to true (useful for constraints expressed in code rather than annotations)	none
@Valid	property (object)	perform validation recursively on the associated object. If the object is a Collection or an array, the elements are validated recursively. If the object is a Map, the value elements are validated recursively.	none
@Email	property (String)	check whether the string is conform to the email address specification	none

3.1.3. Error messages

Hibernate Validator comes with a default set of error messages translated in a few languages (if yours is not part of it, please sent us a patch). You can override those messages by creating a `ValidatorMessages.properties` or (`ValidatorMessages_loc.properties`) out of `org.hibernate.validator.resources.DefaultValidatorMessages.properties` and change the appropriate keys. You can even add your own additional set of messages while writing your validator annotations.

Alternatively you can provide a `ResourceBundle` while checking programmatically the validation rules on a bean.

3.1.4. Writing your own constraints

Extending the set of built-in constraints is extremely easy. Any constraint consists of two pieces: the constraint *descriptor* (the annotation) and the constraint *validator* (the implementation class). Here is a simple user-defined descriptor:

```
@ValidatorClass(CapitalizedValidator.class)
@Target(METHOD)
@Retention(RUNTIME)
@Documented
public @interface Capitalized {
    CapitalizeType type() default Capitalize.FIRST;
    String message() default "has incorrect capitalization";
}
```

`type` is a parameter describing how the property should to be capitalized. This is a user parameter fully dependant on the annotation business.

`message` is the default string used to describe the constraint violation and is mandatory. You can hard code the string or you can externalize part/all of it through the Java ResourceBundle mechanism. Parameters values are going to be injected inside the message when the `{parameter}` string is found (in our example `Capitalization is not {type}` would generate `Capitalization is not FIRST`), externalizing the whole string in `ValidatorMessages.properties` is considered good practice. See Error messages.

```
@ValidatorClass(CapitalizedValidator.class)
@Target(METHOD)
@Retention(RUNTIME)
@Documented
public @interface Capitalized {
    CapitalizeType type() default Capitalize.FIRST;
    String message() default "{validator.capitalized}";
}

...
#in ValidatorMessages.properties
validator.capitalized=Capitalization is not {type}
```

As you can see the {} notation is recursive.

To link a descriptor to its validator implementation, we use the `@ValidatorClass` meta-annotation. The validator class parameter must name a class which implements `Validator<ConstraintAnnotation>`.

We now have to implement the validator (ie. the rule checking implementation). A validation implementation can check the value of the a property (by implementing `PropertyConstraint`) and/or can modify the hibernate mapping metadata to express the constraint at the database level (by implementing `PersistentClassConstraint`).

```
public class CapitalizedValidator
        implements Validator<Capitalized>, PropertyConstraint {
```

```
    private CapitalizeType type;

    //part of the Validator<Annotation> contract,
    //allows to get and use the annotation values
    public void initialize(Capitalized parameters) {
        type = parameters.type();
    }

    //part of the property constraint contract
    public boolean isValid(Object value) {
        if (value==null) return true;
        if ( !(value instanceof String) ) return false;
        String string = (String) value;
        if (type == CapitalizeType.ALL) {
            return string.equals( string.toUpperCase() );
        }
        else {
            String first = string.substring(0,1);
            return first.equals( first.toUpperCase();
        }
    }
}
```

The `isValid()` method should return false if the constraint has been violated. For more examples, refer to the built-in validator implementations.

We only have seen property level validation, but you can write a Bean level validation annotation. Instead of receiving the return instance of a property, the bean itself will be passed to the validator. To activate the validation checking, just annotated the bean itself instead. A small sample can be found in the unit test suite.

3.1.5. Annotating your domain model

Since you are already familiar with annotations now, the syntax should be very familiar.

```
public class Address {
    private String line1;
    private String line2;
    private String zip;
    private String state;
    private String country;
    private long id;

    // a not null string of 20 characters maximum
    @Length(max=20)
    @NotNull
    public String getCountry() {
        return country;
    }

    // a non null string
    @NotNull
    public String getLine1() {
        return line1;
    }

    //no constraint
    public String getLine2() {
        return line2;
    }

    // a not null string of 3 characters maximum
    @Length(max=3) @NotNull
    public String getState() {
        return state;
    }
```

```
    // a not null numeric string of 5 characters maximum
    // if the string is longer, the message will
    //be searched in the resource bundle at key 'long'
    @Length(max=5, message="{long}")
    @Pattern(regex="[0-9]+")
    @NotNull
    public String getZip() {
        return zip;
    }

    // should always be true
    @AssertTrue
    public boolean isValid() {
        return true;
    }

    // a numeric between 1 and 2000
    @Id @Min(1)
    @Range(max=2000)
    public long getId() {
        return id;
    }
}
```

While the example only shows public property validation, you can also annotate fields of any kind of visibility.

```
@MyBeanConstraint(max=45)
public class Dog {
    @AssertTrue private boolean isMale;
    @NotNull protected String getName() { ... };
    ...
}
```

You can also annotate interfaces. Hibernate Validator will check all superclasses and interfaces extended or implemented by a given bean to read the appropriate validator annotations.

```
public interface Named {
    @NotNull String getName();
    ...
}

public class Dog implements Named {

    @AssertTrue private boolean isMale;

    public String getName() { ... };

}
```

The name property will be checked for nullity when the Dog bean is validated.

3.2. Using the Validator framework

Hibernate Validator is intended to be used to implement multi-layered data validation, where we express constraints in one place (the annotated domain model) and apply them at various different layers of the application.

3.2.1. Database schema-level validation

Out of the box, Hibernate Annotations will translate the constraints you have defined for your entities into mapping metadata. For example, if a property of your entity is annotated `@NotNull`, its columns will be declared as `not null` in the DDL schema generated by Hibernate.

3.2.2. Hibernate event-based validation

Hibernate Validator has two built-in Hibernate event listeners. Whenever a `PreInsertEvent` or `PreUpdateEvent` occurs, the listeners will verify all constraints of the entity instance and throw an exception if any constraint is violated. Basically, objects will be checked before any inserts and before any updates made by Hibernate. This is the most convenient and the easiest way to activate the validation process. On constraint violation, the event will raise a runtime `InvalidStateException` which contains an array of `InvalidValues` describing each failure.

```
<hibernate-configuration>
    ...
    <event type="pre-update">
        <listener
          class="org.hibernate.validator.event.ValidatePreUpdateEventListener"/>
    </event>
    <event type="pre-insert">
        <listener
          class="org.hibernate.validator.event.ValidatePreInsertEventListener"/>
    </event>
</hibernate-configuration>
```

Note

When using Hibernate Entity Manager, the Validation framework is activated out of the box. If the beans are not annotated with validation annotations, there is no performance cost.

3.2.3. Application-level validation

Hibernate Validator can be applied anywhere in your application code.

```
ClassValidator personValidator = new ClassValidator( Person.class );
ClassValidator addressValidator = new ClassValidator( Address.class, ResourceBundle.getBundle("message

InvalidValue[] validationMessages = addressValidator.getInvalidValues(address);
```

The first two lines prepare the Hibernate Validator for class checking. The first one relies upon the error messages embedded in Hibernate Validator (see Error messages), the second one uses a resource bundle for these messages. It is considered a good practice to execute these lines once and cache the validator instances.

The third line actually validates the `Address` instance and returns an array of `InvalidValues`. Your application logic will then be able to react to the failure.

You can also check a particular property instead of the whole bean. This might be useful for property per property user interaction

```
ClassValidator addressValidator = new ClassValidator( Address.class, ResourceBundle.getBundle("message

//only get city property invalid values
InvalidValue[] validationMessages = addressValidator.getInvalidValues(address, "city");

//only get potential city property invalid values
InvalidValue[] validationMessages = addressValidator.getPotentialInvalidValues("city", "Paris")
```

3.2.4. Validation informations

As a validation information carrier, hibernate provide an array of `InvalidValue`. Each `InvalidValue` has a

buch of methods describing the individual issues.

`getBeanClass()` retrieves the failing bean type

`getBean()`retrieves the failing instance (if any ie not when using `getPotentianInvalidValues()`)

`getValue()` retrieves the failing value

`getMessage()` retrieves the proper internationalized error message

`getRootBean()` retrieves the root bean instance generating the issue (useful in conjunction with `@Valid`), is null if getPotentianInvalidValues() is used.

`getPropertyPath()` retrieves the dotted path of the failing property starting from the root bean

Chapter 4. Hibernate Lucene Integration

Lucene is a high-performance Java search engine library available from the Apache Software Foundation. Hibernate Annotations includes a package of annotations that allows you to mark any domain model object as indexable and have Hibernate maintain a Lucene index of any instances persisted via Hibernate.

4.1. Using Lucene to index your entities

4.1.1. Annotating your domain model

First, we must declare a persistent class as `@Indexed`:

```
@Entity
@Indexed(index="indexes/essays")
public class Essay {
    ...
}
```

The `index` attribute tells Hibernate where the Lucene index is located (a directory on your file system). If you wish to define a base directory for all lucene indexes, you can use the `hibernate.lucene.index_dir` property in your configuration file.

Lucene indexes contain four kinds of fields: *keyword* fields, *text* fields, *unstored* fields and *unindexed* fields. Hibernate Annotations provides annotations to mark a property of an entity as one of the first three kinds of indexed fields.

```
@Entity
@Indexed(index="indexes/essays")
public class Essay {
    ...

    @Id
    @Keyword(id=true)
    public Long getId() { return id; }

    @Text(name="Abstract")
    public String getSummary() { return summary; }

    @Lob
    @Unstored
    public String getText() { return text; }

}
```

These annotations define an index with three fields: `Id`, `Abstract` and `Text`.

Note: you *must* specify `@Keyword(id=true)` on the identifier property of your entity class.

The analyzer class used to index the elements is configurable through the `hibernate.lucene.analyzer` property. If none defined, `org.apache.lucene.analysis.standard.StandardAnalyzer` is used as the default.

4.1.2. Enabling automatic indexing

Finally, we enable the `LuceneEventListener` for the three Hibernate events that occur after changes are committed to the database.

```
<hibernate-configuration>
    ...
    <event type="post-commit-update"
        <listener
            class="org.hibernate.lucene.event.LuceneEventListener"/>
    </event>
    <event type="post-commit-insert"
        <listener
            class="org.hibernate.lucene.event.LuceneEventListener"/>
    </event>
    <event type="post-commit-delete"
        <listener
            class="org.hibernate.lucene.event.LuceneEventListener"/>
    </event>
</hibernate-configuration>
```