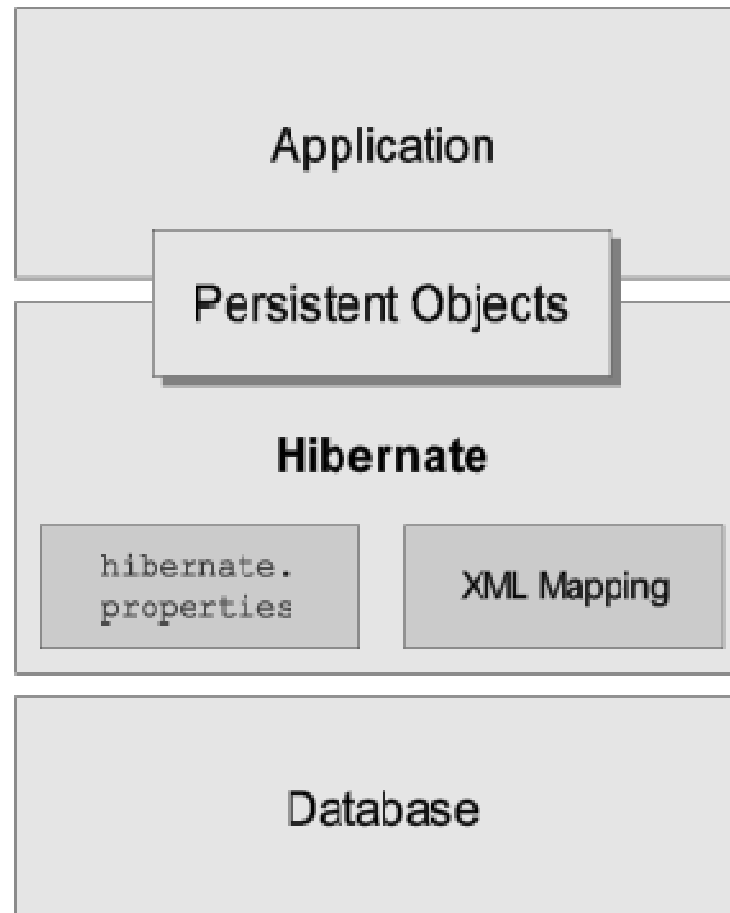


Remember the Structure



Hibernate Model

- Hibernate Core offers native API's & object/relational mapping with XML metadata.
- Hibernate Annotations offers JDK 5.0 code annotations as a replacement or in addition to XML metadata.
- Hibernate EntityManager involves standard JPA for Java SE and Java EE.

JPA (Java Persistent API)



- JPA is a part of EJB 3.0 specification
- JPA is a POJO API for object/relational mapping that supports the use both of Java metadata annotations and/or XML metadata

What is Annotation ?

- **Annotation** is a specific construction in java 5 for adding additional information to Java source code.
- **Annotations** are embedded in class files generated by compiler & can be used by other frameworks.


Annotation Using Syntax



```
@AnnotationName (element1 = "value1",  
    element2 = "value2")
```

```
@AnnotationName ("value")
```

Can be used for

- 
- ☐ classes
 - ☐ methods
 - ☐ variables
 - ☐ parameters
 - ☐ packages
 - ☐ annotations

Hibernate Annotations

- Hibernate works best if these classes follow the Plain Old Java Object (POJO) / JavaBean programming model.
 - ▣ However, none of these rules are hard requirements.
 - ▣ Indeed, Hibernate assumes very little about the nature of your persistent objects.
- You can express a domain model in other ways (i.e., using trees of `java.util.Map` instances).
- Historically applications using Hibernate would have used its proprietary XML mapping file format for this purpose.
- With the coming of JPA, most of this information is now defined in a way that is portable across ORM/JPA providers using annotations (and/or standardized XML format).
- This lecture will focus on JPA mapping where possible.
- For Hibernate mapping features not supported by JPA we will prefer Hibernate extension annotations.

Extension Annotations



- Contained in `org.hibernate.annotations` package
- Examples:
 - `@org.hibernate.annotations.Entity`
 - `@org.hibernate.annotations.Table`
 - `@BatchSize`
 - `@Where`
 - `@Check`

Annotated Java class

```
@Entity //Declares this an entity bean
@Table(name="people") //Maps the bean to SQL table "people"
class Person {
    @Id //Map this to the primary key column.
    @GeneratedValue(strategy=GenerationType.AUTO) //DB generates new PKs
    private int id;

    @Column(length = 32) //Truncate column values to 32 characters.
    private String name;

    public int getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

Basic Annotations



@Entity

Declares this an entity bean

@Id

Identity

@EmbeddedId

@GeneratedValue

@Table

Database Schema Attributes

@Column

@OneToOne

Relationship mappings

@ManyToOne

@OneToMany

& etc.

hibernate.cfg.xml

```
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
    <property name="hibernate.connection.password">secret</property>
    <property name="hibernate.connection.url">jdbc:mysql://host:3306/dbname</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

    <mapping class="com.me.pojo.Person"/>
  </session-factory>
</hibernate-configuration>
```

Hibernate Class Entities

- Class attributes
 - ▣ Hibernate uses reflection to populate
 - ▣ Can be private or whatever
- Class requirements
 - ▣ Default constructor (private or whatever)
 - “However, package or public visibility is required for runtime proxy generation and efficient data retrieval without bytecode instrumentation.”
- JavaBean pattern common
 - ▣ Not required though but easier
- 3 methods of serialization definition
 - ▣ Following slides

Hibernate Annotation Mappings

- Annotations in code
 - ▣ Beginning of class
 - ▣ Indicate class is Entity
 - Class doesn't have to implement `java.lang.Serializable`
 - ▣ Define database table
 - ▣ Define which attributes to map to columns
 - Supports auto-increment IDs too
 - Can dictate value restrictions (not null, etc)
 - Can dictate value storage type
- Existed before JPA standard (later slides)
- Doesn't require a separate `hbm.xml` mapping file
 - ▣ But is tied to code

Example 1. Hibernate Annotation

```
@Entity
@Table(name = "EVENTS")
public class Event {
    @Id
    @GeneratedValue(generator="increment"
    @GenericGenerator(name="increment", strategy = "increment")
    private long id;

    @Column(name = "EVENT_DATE")
    Date eventdate;
    .....

    public long getId() {
        return id;
    }
    public Date getDate() {
        return date;
    }
    .....
}
```

JPA Annotation

- Became standard
 - ▣ Came after Hibernate annotations
- Works almost like Hibernate annotations
 - ▣ Requires “META-INF/persistence.xml” file
 - Defines data source configuration
 - HibernatePersistence provider
 - Auto-detects any annotated classes
 - Auto-detects any hbm.xml class mapping files
 - (later slides)
 - Allows explicit class loading for mapping
- Annotation syntax
 - ▣ Same as Hibernate
 - ▣ Hibernate has a few extensions (see docs)

Example 2. A simple table and domain model

```
create table Contact (  
    id integer not null,  
    first varchar(255),  
    last varchar(255),  
    middle varchar(255),  
    notes varchar(255),  
    starred boolean not null,  
    website varchar(255),  
    primary key (id)  
)
```



```
@Entity(name = "Contact")
public static class Contact {

    @Id
    private Integer id;

    private Name name;

    private String notes;

    private URL website;

    private boolean starred;

    //Getters and setters are omitted for brevity
}

@Embeddable
public class Name {

    private String first;

    private String middle;

    private String last;

    // getters and setters omitted
}
```

Hibernate categorizes types into two groups:

- 
- Value types
 - Entity types

Value types

- A value type is a piece of data that does not define its own lifecycle.
- It is, in effect, owned by an entity, which defines its lifecycle.
- These state fields or JavaBean properties are termed persistent attributes.
- The persistent attributes of the Contact class are value types.
- Value types are further classified into three sub-categories:
 - ▣ Basic types
 - ▣ Embeddable types
 - ▣ Collection types

Entity types



- Entities, by nature of their unique identifier, exist independently of other objects whereas values do not.
- Entities are domain model classes which correlate to rows in a database table, using a unique identifier.
- Because of the requirement for a unique identifier, entities exist independently and define their own lifecycle.
- The Contact class itself would be an example of an entity.

Naming strategies

- Part of the mapping of an object model to the relational database is mapping names from the object model to the corresponding database names.
- Hibernate looks at this as 2-stage process:
 - ▣ The first stage is determining a proper logical name from the domain model mapping. A logical name can be either explicitly specified by the user (e.g., using @Column or @Table) or it can be implicitly determined by Hibernate through an ImplicitNamingStrategy contract.
 - ▣ Second is the resolving of this logical name to a physical name which is defined by the PhysicalNamingStrategy contract.

The @Basic annotation

- Strictly speaking, a basic type is denoted by the `javax.persistence.Basic` annotation.
- Generally speaking, the `@Basic` annotation can be ignored, as it is assumed by default.
- Both of the following examples are ultimately the same.

Example 3. @Basic declared explicitly

```
@Entity(name = "Product")
public class Product {
    @Id
    @Basic
    private Integer id;

    @Basic
    private String sku;

    @Basic
    private String name;

    @Basic
    private String description;
}
```

Example 4. @Basic being implicitly implied

```
@Entity(name = "Product")
public class Product {

    @Id
    private Integer id;

    private String sku;

    private String name;

    private String description;
}
```


The JPA specification strictly limits the Java types that can be marked as basic to the following listing:

- Java primitive types (`boolean` , `int` , etc)
- wrappers for the primitive types (`java.lang.Boolean` , `java.lang.Integer` , etc)
- `java.lang.String`
- `java.math.BigInteger`
- `java.math.BigDecimal`
- `java.util.Date`
- `java.util.Calendar`
- `java.sql.Date`
- `java.sql.Time`
- `java.sql.Timestamp`
- `byte[]` or `Byte[]`
- `char[]` or `Character[]`
- `enums`
- any other type that implements `Serializable` (JPA's "support" for `Serializable` types is to directly serialize their state to the database).

The @Basic annotation defines 2 attributes

□ optional - boolean (defaults to true)

- ▣ Defines whether this attribute allows nulls. JPA defines this as "a hint", which essentially means that its effect is specifically required.
- ▣ As long as the type is not primitive, Hibernate takes this to mean that the underlying column should be NULLABLE.

□ fetch - FetchType (defaults to EAGER)

- ▣ Defines whether this attribute should be fetched eagerly or lazily.
- ▣ JPA says that EAGER is a requirement to the provider (Hibernate) that the value should be fetched when the owner is fetched, while LAZY is merely a hint that the value is fetched when the attribute is accessed.
- ▣ Hibernate ignores this setting for basic types unless you are using bytecode enhancement.

The @Column annotation

- JPA defines rules for implicitly determining the name of tables and columns.
- For basic type attributes, the implicit naming rule is that the column name is the same as the attribute name.
- If that implicit naming rule does not meet your requirements, you can explicitly tell Hibernate (and other providers) the column name to use.

Example 5. Explicit column naming

```
@Entity(name = "Product")
public class Product {
    @Id
    private Integer id;
    private String sku;
    private String name;
    @Column( name = "NOTES" )
    private String description;
}
```

Here we use `@Column` to explicitly map the description attribute to the NOTES column, as opposed to the implicit column name description.

The `@Column` annotation defines other mapping information as well.
See its Javadocs for details.

BasicTypeRegistry

- We said before that a Hibernate type is not a Java type, nor an SQL type, but that it understands both and performs the marshalling between them.
- But looking at the basic type mappings from the previous examples, how did Hibernate know to use
 - ▣ `org.hibernate.type.StringType` for mapping for `java.lang.String` attributes, or
 - ▣ `org.hibernate.type.IntegerType` for mapping `java.lang.Integer` attributes
- The answer lies in a service inside Hibernate called the `org.hibernate.type.BasicTypeRegistry`, which essentially maintains a map of `org.hibernate.type.BasicType` (an `org.hibernate.type.Type` specialization) instances keyed by a name.
- Also, we can explicitly tell Hibernate which `BasicType` to use for a particular attribute.

Explicit BasicTypes



- Sometimes you want a particular attribute to be handled differently.
- Occasionally Hibernate will implicitly pick a BasicType that you do not want (and for some reason you do not want to adjust the BasicTypeRegistry).
- In these cases, you must explicitly tell Hibernate the BasicType to use, via the [org.hibernate.annotations.Type](#) annotation.

Example 6. Using @org.hibernate.annotations.Type

```
@Entity(name = "Product")
public class Product {
    @Id
    private Integer id;


    private String sku;

    @org.hibernate.annotations.Type(type="nstring")
    private String name;

    @org.hibernate.annotations.Type(type="materialized_nclob")
    private String description;
}
```

This tells Hibernate to store the Strings as nationalized data. Additionally, the description is to be handled as a LOB.

The `org.hibernate.annotations.Type#type` attribute can name any of the following:

- 
- Fully qualified name of any `org.hibernate.type.Type` implementation
 - Any key registered with `BasicTypeRegistry`
 - The name of any known type definitions

Custom BasicTypes

- Hibernate makes it relatively easy for developers to create their own basic type mappings type.
- For example, you might want to persist properties of type `java.util.BigInteger` to `VARCHAR` columns, or support completely new types.
- There are two approaches to developing a custom type:
 - ▣ implementing a `BasicType` and registering it
 - ▣ implementing a `UserType` which doesn't require type registration
- As a means of illustrating the different approaches, let's consider a use case where we need to support a `java.util.BitSet` mapping that's stored as a `VARCHAR`.

Implementing a BasicType

- The first approach is to directly implement the BasicType interface.
- Because the BasicType interface has a lot of methods to implement, if the value is stored in a single database column, it's much more convenient to extend the AbstractStandardBasicType or the AbstractSingleColumnStandardBasicType Hibernate classes.
- First, we need to extend the AbstractSingleColumnStandardBasicType as shown in the next slide.

Example 7. Custom BasicType implementation

```
public class BitSetType
    extends AbstractSingleColumnStandardBasicType<BitSet>
    implements DiscriminatorType<BitSet> {

    public static final BitSetType INSTANCE = new BitSetType();

    public BitSetType() {
        super( VarcharTypeDescriptor.INSTANCE, BitSetTypeDescriptor.INSTANCE );
    }

    @Override
    public BitSet stringToObject(String xml) throws Exception {
        return fromString( xml );
    }

    @Override
    public String objectToSQLString(BitSet value, Dialect dialect) throws Exception {
        return toString( value );
    }

    @Override
    public String getName() {
        return "bitset";
    }
}
```

Example 8. Custom AbstractTypeDescriptor implementation

```
public class BitSetTypeDescriptor extends AbstractTypeDescriptor<BitSet> {

    private static final String DELIMITER = ",";

    public static final BitSetTypeDescriptor INSTANCE = new BitSetTypeDescriptor();

    public BitSetTypeDescriptor() {
        super( BitSet.class );
    }

    @Override
    public String toString(BitSet value) {
        StringBuilder builder = new StringBuilder();
        for ( long token : value.toLongArray() ) {
            if ( builder.length() > 0 ) {
                builder.append( DELIMITER );
            }
            builder.append( Long.toString( token, 2 ) );
        }
        return builder.toString();
    }

    @Override
    public BitSet fromString(String string) {
        if ( string == null || string.isEmpty() ) {
            return null;
        }
        String[] tokens = string.split( DELIMITER );
        long[] values = new long[tokens.length];

        for ( int i = 0; i < tokens.length; i++ ) {
            values[i] = Long.valueOf( tokens[i], 2 );
        }
        return BitSet.valueOf( values );
    }

    @SuppressWarnings("unchecked")
    public <X> X unwrap(BitSet value, Class<X> type, WrapperOptions options) {
        if ( value == null ) {
            return null;
        }
        if ( BitSet.class.isAssignableFrom( type ) ) {
            return (X) value;
        }
        if ( String.class.isAssignableFrom( type ) ) {
            return (X) toString( value );
        }
        throw unknownUnwrap( type );
    }

    public <X> BitSet wrap(X value, WrapperOptions options) {
        if ( value == null ) {
            return null;
        }
        if ( String.class.isInstance( value ) ) {
            return fromString( (String) value );
        }
        if ( BitSet.class.isInstance( value ) ) {
            return (BitSet) value;
        }
        throw unknownWrap( value.getClass() );
    }
}
```

Example 9. Register a Custom BasicType implementation

- The BasicType must be registered, and this can be done at bootstrapping time:

```
configuration.registerTypeContributor( (typeContributions, serviceRegistry) -> {  
    typeContributions.contributeType( BitSetType.INSTANCE );  
} );
```

or using the MetadataBuilder

```
ServiceRegistry standardRegistry =  
    new StandardServiceRegistryBuilder().build();  
  
MetadataSources sources = new MetadataSources( standardRegistry );  
  
MetadataBuilder metadataBuilder = sources.getMetadataBuilder();  
  
metadataBuilder.applyBasicType( BitSetType.INSTANCE );
```

Example 10. Custom BasicType mapping

- With the new BitSetType being registered as bitset, the entity mapping looks like this:

```
@Entity(name = "Product")
public static class Product {

    @Id
    private Integer id;

    @Type( type = "bitset" )
    private BitSet bitSet;

    public Integer getId() {
        return id;
    }

    //Getters and setters are omitted for brevity
}
```

Example 11. Using @TypeDef to register a custom Type

- Alternatively, you can use the @TypeDef and skip the registration phase:

```
@Entity(name = "Product")
@TypeDef(
    name = "bitset",
    defaultForType = BitSet.class,
    typeClass = BitSetType.class
)
public static class Product {

    @Id
    private Integer id;

    private BitSet bitSet;

    //Getters and setters are omitted for brevity
}
```

Implementing a UserType

```
public class BitSetUserType implements UserType {

    public static final BitSetUserType INSTANCE = new BitSetUserType();

    private static final Logger log = Logger.getLogger( BitSetUserType.class );

    @Override
    public int[] sqlTypes() {
        return new int[] {StringType.INSTANCE.sqlType()};
    }

    @Override
    public Class returnedClass() {
        return BitSet.class;
    }

    @Override
    public boolean equals(Object x, Object y)
        throws HibernateException {
        return Objects.equals( x, y );
    }

    @Override
    public int hashCode(Object x)
        throws HibernateException {
        return Objects.hashCode( x );
    }
}
```

Part of the implementation is shown due to space.

Example 12. Custom UserType mapping

- The entity mapping looks as follows:

```
@Entity(name = "Product")
public static class Product {

    @Id
    private Integer id;

    @Type( type = "bitset" )
    private BitSet bitSet;

    //Constructors, getters, and setters are omitted for brevity
}
```

Register a Custom UserType implementation

In previous example, the UserType is registered under the bitset name, and this is done like this

```
configuration.registerTypeContributor( (typeContributions, serviceRegistry) -> {  
    typeContributions.contributeType( BitSetUserType.INSTANCE, "bitset");  
} );
```

or using the MetadataBuilder

```
ServiceRegistry standardRegistry =  
    new StandardServiceRegistryBuilder().build();  
  
MetadataSources sources = new MetadataSources( standardRegistry );  
  
MetadataBuilder metadataBuilder = sources.getMetadataBuilder();  
  
metadataBuilder.applyBasicType( BitSetUserType.INSTANCE, "bitset" );
```

Like BasicType, you can also register the UserType using a simple name.

- Without registering a name, the UserType mapping requires the fully qualified class name:

```
@Type( type = "org.hibernate.userguide.mapping.basic.BitSetUserType" )
```

Mapping enums

- Hibernate supports the mapping of Java enums as basic value types in a number of different ways.
- **@Enumerated**
- The original JPA-compliant way to map enums was via the @Enumerated or @MapKeyEnumerated for map keys annotations, working on the principle that the enum values are stored according to one of 2 strategies indicated by `javax.persistence.EnumType`:
- ORDINAL
 - ▣ stored according to the enum value's ordinal position within the enum class, as indicated by `java.lang.Enum#ordinal`
- STRING
 - ▣ stored according to the enum value's name, as indicated by `java.lang.Enum#name`

Example 13. PhoneType enumeration

```
public enum PhoneType {  
    LAND_LINE,  
    MOBILE;  
}
```

In the ORDINAL example, the `phone_type` column is defined as a (nullable) INTEGER type and would hold:

NULL

For null values

0

For the `LAND_LINE` enum

1

For the `MOBILE` enum

Example 14. @Enumerated(ORDINAL) example

```
@Entity(name = "Phone")
public static class Phone {

    @Id
    private Long id;

    @Column(name = "phone_number")
    private String number;

    @Enumerated(EnumType.ORDINAL)
    @Column(name = "phone_type")
    private PhoneType type;

    //Getters and setters are omitted for brevity

}
```

Example 15. @Enumerated(String) example

- In the STRING example, the phone_type column is defined as a (nullable) VARCHAR type and would hold:
 - NULL For null values
 - LAND_LINE For the LAND_LINE enum
 - MOBILE For the MOBILE enum

```
@Entity(name = "Phone")
public static class Phone {

    @Id
    private Long id;

    @Column(name = "phone_number")
    private String number;

    @Enumerated(EnumType.STRING)
    @Column(name = "phone_type")
    private PhoneType type;

    //Getters and setters are omitted for brevity
}
```

Mapping Date/Time Values

Hibernate allows various Java Date/Time classes to be mapped as persistent domain model entity properties. The SQL standard defines three Date/Time types:

DATE

Represents a calendar date by storing years, months and days. The JDBC equivalent is `java.sql.Date`

TIME

Represents the time of a day and it stores hours, minutes and seconds. The JDBC equivalent is `java.sql.Time`

TIMESTAMP

It stores both a DATE and a TIME plus nanoseconds. The JDBC equivalent is `java.sql.Timestamp`



To avoid dependencies on the `java.sql` package, it's common to use the `java.util` or `java.time` Date/Time classes instead of the `java.sql.Timestamp` and `java.sql.Time` ones.

While the `java.sql` classes define a direct association to the SQL Date/Time data types, the `java.util` or `java.time` properties need to explicitly mark the SQL type correlation with the `@Temporal` annotation. This way, a `java.util.Date` or a `java.util.Calendar` can be mapped to either an SQL `DATE`, `TIME` or `TIMESTAMP` type.

Example 16. java.util.Date mapped as DATE

```
@Entity(name = "DateEvent")
public static class DateEvent {


    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "`timestamp`")
    @Temporal(TemporalType.DATE)
    private Date timestamp;

    //Getters and setters are omitted for brevity

}
```

Example 17. java.util.Date mapped as TIME



```
@Column(name = "`timestamp`")  
@Temporal(TemporalType.TIME)  
private Date timestamp;
```

Mapping Java 8 Date/Time Values

Java 8 came with a new Date/Time API, offering support for instant dates, intervals, local and zoned Date/Time immutable instances, bundled in the `java.time` package.

The mapping between the standard SQL Date/Time types and the supported Java 8 Date/Time class types looks as follows;

DATE

`java.time.LocalDate`

TIME

`java.time.LocalTime`, `java.time.OffsetTime`

TIMESTAMP

`java.time.Instant`, `java.time.LocalDateTime`, `java.time.OffsetDateTime` and `java.time.ZonedDateTime`



Because the mapping between the Java 8 Date/Time classes and the SQL types is implicit, there is not need to specify the `@Temporal` annotation.

Setting it on the `java.time` classes throws the following exception:

```
org.hibernate.AnnotationException: @Temporal should only be set on a java.util.Date or
java.util.Calendar property
```

@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)
- You should be aware that the @Formula annotation takes a native SQL clause which may affect database portability.

Example 18. @Formula mapping usage

```
@Entity(name = "Account")
public static class Account {

    @Id
    private Long id;

    private Double credit;

    private Double rate;

    @Formula(value = "credit * rate")
    private Double interest;

    //Getters and setters omitted for brevity

}
```

Embeddable types

- Historically Hibernate called these components.
- JPA calls them embeddables.
- Either way, the concept is the same: a composition of values.
- For example, we might have a Publisher class that is a composition of name and country, or a Location class that is a composition of country and city.

```
@Embeddable
public static class Publisher {

    private String name;

    private Location location;

    public Publisher(String name, Location location) {
        this.name = name;
        this.location = location;
    }

    private Publisher() {}

    //Getters and setters are omitted for brevity
}

@Embeddable
public static class Location {

    private String country;

    private String city;

    public Location(String country, String city) {
        this.country = country;
        this.city = city;
    }

    private Location() {}

    //Getters and setters are omitted for brevity
}
```

Embeddable Type



- An embeddable type is another form of a value type, and its lifecycle is bound to a parent entity type, therefore inheriting the attribute access from its parent.
- Embeddable types can be made up of basic values as well as associations, with the caveat that, when used as collection elements, they cannot define collections themselves.

Most often, embeddable types are used to group multiple basic type mappings and reuse them across several entities.

```
@Entity(name = "Book")
public static class Book {

    @Id
    @GeneratedValue
    private Long id;

    private String title;

    private String author;

    private Publisher publisher;

    //Getters and setters are omitted for brevity
}

@Embeddable
public static class Publisher {

    @Column(name = "publisher_name")
    private String name;

    @Column(name = "publisher_country")
    private String country;

    //Getters and setters, equals and hashCode methods omitted for brevity
}
```

Entity types

- The entity type describes the mapping between the actual persistable domain model object and a database table row.
- The entity class must be annotated with the `javax.persistence.Entity` annotation (or be denoted as such in XML mapping).
- The entity class must have a public or protected no-argument constructor. It may define additional constructors as well.
- The entity class must be a top-level class.
- An enum or interface may not be designated as an entity.
- The entity class must not be final. No methods or persistent instance variables of the entity class may be final.
- If an entity instance is to be used remotely as a detached object, the entity class must implement the `Serializable` interface.
- Both abstract and concrete classes can be entities. Entities may extend non-entity classes as well as entity classes, and non-entity classes may extend entity classes.
- The persistent state of an entity is represented by instance variables, which may correspond to JavaBean-style properties.

Mapping the entity

- The main piece in mapping the entity is the `javax.persistence.Entity` annotation.
- The `@Entity` annotation defines just the `name` attribute which is used to give a specific entity name for use in JPQL queries.
- By default, if the `name` attribute of the `@Entity` annotation is missing, the unqualified name of the entity class itself will be used as the entity name.
- Because the entity name is given by the unqualified name of the class, Hibernate does not allow registering multiple entities with the same name even if the entity classes reside in different packages.
- Without imposing this restriction, Hibernate would not know which entity class is referenced in a JPQL query if the unqualified entity name is associated with more than one entity classes.

Example 19. @Entity mapping with an implicit name

- In the following example, the entity name (e.g. Book) is given by the unqualified name of the entity class name.

```
@Entity
public class Book {

    @Id
    private Long id;

    private String title;

    private String author;

    //Getters and setters are omitted for brevity
}
```

Example 20. @Entity mapping with an explicit name

- The entity name can also be set explicitly as illustrated by the following example.

```
@Entity(name = "Book")
public static class Book {

    @Id
    private Long id;

    private String title;

    private String author;

    //Getters and setters are omitted for brevity
}
```

Example 21. Simple @Entity with @Table

- An entity models a database table.
- The identifier uniquely identifies each row in that table.
- By default, the name of the table is assumed to be the same as the name of the entity.
- To explicitly give the name of the table or to specify other information about the table, we would use the `javax.persistence.Table` annotation.

```
@Entity(name = "Book")
@Table(
    catalog = "public",
    schema = "store",
    name = "book"
)
public static class Book {

    @Id
    private Long id;

    private String title;

    private String author;

    //Getters and setters are omitted for brevity
}
```

Example 22. Library entity mapping

- Consider we have a Library parent entity which contains a java.util.Set of Book entities:

```
@Entity(name = "Library")
public static class Library {

    @Id
    private Long id;

    private String name;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "book_id")
    private Set<Book> books = new HashSet<>();

    //Getters and setters are omitted for brevity
}
```

Natural Id

- Hibernate 4 has brought lots of improvements and `@NaturalId` is one of such nice improvements.
- As you know `@Id` annotation is used as meta data for specifying the primary key of an entity.
- But sometimes, entity is usually used in DAO layer code with id which is not primary key but its logical or natural id.
- In such cases, `@NaturalId` annotation will prove good replacement of named queries in hibernate.

For example, in any application there can be an employee entity

- In this case, primary key will definitely be “employee id” but in cases such as login by email, user will provide email and password.
- In this case, in stead of writing named query, you can directly use @NaturalId annotation on “email” field.
- When you get an entity by its natural id then
 - ▣ First primary key of entity is found by executing where clause of natural id
 - ▣ This primary key is used fetch the information of entity

Example 23. EmployeeEntity with @NaturalId

```
@Entity
@Table(name = "Employee", uniqueConstraints = {
    @UniqueConstraint(columnNames = "ID"),
    @UniqueConstraint(columnNames = "EMAIL") })
public class EmployeeEntity implements Serializable {

    private static final long serialVersionUID = -1798070786993154676L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID", unique = true, nullable = false)
    private Integer employeeId;

    //Use the natural id annotation here
    @NaturalId (mutable = false)
    @Column(name = "EMAIL", unique = true, nullable = false, length = 100)
    private String email;

    @Column(name = "FIRST_NAME", unique = false, nullable = false, length = 100)
    private String firstName;

    @Column(name = "LAST_NAME", unique = false, nullable = false, length = 100)
    private String lastName;

    //Setters and Getters

}
```

How to use this in code:

```
public class TestHibernate
{
    public static void main(String[] args)
    {
        Session session = HibernateUtil.getSessionFactory().openSession();
        session.beginTransaction();

        //Add new Employee object
        EmployeeEntity emp = new EmployeeEntity();
        emp.setEmail("demo-user@mail.com");
        emp.setFirstName("demo");
        emp.setLastName("user");
        //Save entity
        session.save(emp);

        EmployeeEntity empGet = (EmployeeEntity) session.bySimpleNaturalId( EmployeeEntity.class ).load( "demo-user@mail.com" );

        System.out.println(empGet.getFirstName());
        System.out.println(empGet.getLastName());

        session.getTransaction().commit();
        HibernateUtil.shutdown();
    }
}
```

Example 24. Field-based access

- When using field-based access, adding other entity-level methods is much more flexible because Hibernate won't consider those part of the persistence state.
- To exclude a field from being part of the entity persistent state, the field must be marked with the `@Transient` annotation.

```
@Entity(name = "Book")
public static class Book {

    @Id
    private Long id;

    private String title;

    private String author;

    //Getters and setters are omitted for brevity
}
```

Example 25. Property-based access

- When using property-based access, Hibernate uses the accessors for both reading and writing the entity state.
- Every other method that will be added to the entity (e.g. helper methods for synchronizing both ends of a bidirectional one-to-many association) will have to be marked with the `@Transient` annotation.

```
@Entity(name = "Book")
public static class Book {

    private Long id;

    private String title;

    private String author;

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

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

Example 26. Overriding the default access strategy

- The default access strategy mechanism can be overridden with the JPA `@Access` annotation.
- In the following example, the `@Version` attribute is accessed by its field and not by its getter, like the rest of entity attributes.

```
@Entity(name = "Book")
public static class Book {

    private Long id;

    private String title;

    private String author;

    @Access( AccessType.FIELD )
    @Version
    private int version;

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

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

Example 27. Embeddable types and access strategy

- Because embeddables are managed by their owning entities, the access strategy is therefore inherited from the entity too.
- This applies to both simple embeddable types as well as for collection of embeddables.
- The embeddable types can overrule the default implicit access strategy (inherited from the owning entity).
- In the following example, the embeddable uses property-based access, no matter what access strategy the owning entity is choosing:

```
@Embeddable
@Access( AccessType.PROPERTY )
public static class Author {

    private String firstName;

    private String lastName;

    public Author() {
    }
}
```

Example 28. Entity including a single embeddable type

- The owning entity can use field-based access while the embeddable uses property-based access as it has chosen explicitly:

```
@Entity(name = "Book")
public static class Book {

    @Id
    private Long id;

    private String title;

    @Embedded
    private Author author;

    //Getters and setters are omitted for brevity
}
```


Identifiers

- Identifiers model the primary key of an entity.
- They are used to uniquely identify each specific entity.
- Hibernate and JPA both make the following assumptions about the corresponding database column(s):
 - ▣ UNIQUE
 - ▣ NOT NULL
 - ▣ IMMUTABLE
- The values, once inserted, can never be changed.
- This is more a general guide, than a hard-fast rule as opinions vary.
- JPA defines the behavior of changing the value of the identifier attribute to be undefined; Hibernate simply does not support that.
- In cases where the values for the PK you have chosen will be updated, Hibernate recommends mapping the mutable value as a natural id, and use a surrogate id for the PK.

Simple identifiers

- Simple identifiers map to a single basic attribute, and are denoted using the `javax.persistence.Id` annotation.
- According to JPA only the following types to be used as identifier attribute types:
 - ▣ any Java primitive type
 - ▣ any primitive wrapper type
 - ▣ `java.lang.String`
 - ▣ `java.util.Date` (`TemporalType#DATE`)
 - ▣ `java.sql.Date`
 - ▣ `java.math.BigDecimal`
 - ▣ `java.math.BigInteger`
- Any types used for identifier attributes beyond this list will not be portable.

Example 29. Assigned entity identifier

- Values for simple identifiers can be assigned, as we have seen in the previous examples.
- The expectation for assigned identifier values is that the application assigns (sets them on the entity attribute) prior to calling save/persist.

```
@Entity(name = "Book")
public static class Book {

    @Id
    private Long id;

    private String title;

    private String author;

    //Getters and setters are omitted for brevity
}
```

Example 30. Generated identifier

- Values for simple identifiers can be generated.
- To denote that an identifier attribute is generated, it is annotated with `javax.persistence.GeneratedValue`.

```
@Entity(name = "Book")
public static class Book {

    @Id
    @GeneratedValue
    private Long id;

    private String title;

    private String author;

    //Getters and setters are omitted for brevity
}
```

Composite identifiers

- Composite identifiers correspond to one or more persistent attributes.
- Here are the rules governing composite identifiers, as defined by the JPA specification:
 - ▣ The composite identifier must be represented by a "primary key class".
 - ▣ The primary key class may be defined using the `javax.persistence.EmbeddedId` annotation `@EmbeddedId`, or defined using the `javax.persistence.IdClass` annotation with `@IdClass`.
 - ▣ The primary key class must be public and must have a public no-arg constructor.
 - ▣ The primary key class must be serializable.
 - ▣ The primary key class must define `equals` and `hashCode` methods, consistent with equality for the underlying database types to which the primary key is mapped.

Example 31. Basic @EmbeddedId

```
@Entity(name = "SystemUser")
public static class SystemUser {

    @EmbeddedId
    private PK pk;

    private String name;

    //Getters and setters are omitted for brevity
}

@Embeddable
public static class PK implements Serializable {

    private String subsystem;

    private String username;

    public PK(String subsystem, String username) {
        this.subsystem = subsystem;
        this.username = username;
    }

    private PK() {
    }
}
```

Example 32. Basic @IdClass

```
@Entity(name = "SystemUser")
@IdClass( PK.class )
public static class SystemUser {

    @Id
    private String subsystem;

    @Id
    private String username;

    private String name;

    public PK getId() {
        return new PK(
            subsystem,
            username
        );
    }

    public void setId(PK id) {
        this.subsystem = id.getSubsystem();
        this.username = id.getUsername();
    }

    //Getters and setters are omitted for brevity
}

public static class PK implements Serializable {

    private String subsystem;

    private String username;

    public PK(String subsystem, String username) {
        this.subsystem = subsystem;
    }
}
```

Generated identifier values

- Hibernate supports identifier value generation across a number of different types.
- Remember that JPA portably defines identifier value generation just for integer types.
- ID value generation is indicated with the `javax.persistence.GeneratedValue` annotation.
- The most important piece of information here is the specified `javax.persistence.GenerationType` which indicates how values will be generated.
- **AUTO (the default)**
 - ▣ Indicates that Hibernate should choose an appropriate generation strategy.
- **IDENTITY**
 - ▣ Indicates that database IDENTITY columns will be used for PK value generation.
- **SEQUENCE**
 - ▣ Indicates that database sequence should be used for obtaining primary key values.
- **TABLE**
 - ▣ Indicates that a database table should be used for obtaining primary key values.

Example 33. Named sequence

- For implementing database sequence-based identifier value generation Hibernate makes use of its `org.hibernate.id.enhanced.SequenceStyleGenerator` id generator.
- The preferred (and portable) way to configure this generator is using the JPA-defined `javax.persistence.SequenceGenerator` annotation.
- The simplest form is to simply request sequence generation; Hibernate will use a single, implicitly-named sequence (`hibernate_sequence`) for all such unnamed definitions.

```
@Entity(name = "Product")
public static class Product {

    @Id
    @GeneratedValue(
        strategy = GenerationType.SEQUENCE
    )
    private Long id;

    @Column(name = "product_name")
    private String name;

    //Getters and setters are omitted for brevity

}
```

Example 34. Named sequence

- Using `javax.persistence.SequenceGenerator`, you can specify a specific database sequence name.

```
@Entity(name = "Product")
public static class Product {

    @Id
    @GeneratedValue(
        strategy = GenerationType.SEQUENCE,
        generator = "sequence-generator"
    )
    @SequenceGenerator(
        name = "sequence-generator",
        sequenceName = "product_sequence"
    )
    private Long id;

    @Column(name = "product_name")
    private String name;

    //Getters and setters are omitted for brevity

}
```

Example 35. Unnamed table generator

- Hibernate achieves table-based identifier generation based on its `org.hibernate.id.enhanced`.
- TableGenerator which defines a table capable of holding multiple named value segments for any number of entities.
- The basic idea is that a given table-generator table (`hibernate_sequences` for example) can hold multiple segments of identifier generation values.

```
@Entity(name = "Product")
public static class Product {

    @Id
    @GeneratedValue(
        strategy = GenerationType.TABLE
    )
    private Long id;

    @Column(name = "product_name")
    private String name;

    //Getters and setters are omitted for brevity
}
```

Example 36. Using UUID generation

- ❑ Hibernate supports UUID identifier value generation.
- ❑ This is supported through its `org.hibernate.id.UUIDGenerator` id generator.
- ❑ `UUIDGenerator` supports pluggable strategies for exactly how the UUID is generated.
- ❑ These strategies are defined by the `org.hibernate.id.UUIDGenerationStrategy` contract.
- ❑ The default strategy is a version 4 (random) strategy according to IETF RFC 4122.
- ❑ Hibernate does ship with an alternative strategy which is a RFC 4122 version 1 (time-based) strategy (using IP address rather than mac address)

```
@Entity(name = "Book")
public static class Book {

    @Id
    @GeneratedValue
    private UUID id;

    private String title;

    private String author;

    //Getters and setters are omitted for brevity
```

Example 37. Implicitly using the random UUID strategy

- To specify an alternative generation strategy, we'd have to define some configuration via `@GeneratedValue`. Here we choose the RFC 4122 version 1 compliant strategy named `org.hibernate.id.uuid.CustomVersionOneStrategy`.

```
@Entity(name = "Book")
public static class Book {

    @Id
    @GeneratedValue( generator = "custom-uuid" )
    @GenericGenerator(
        name = "custom-uuid",
        strategy = "org.hibernate.id.UUIDGenerator",
        parameters = {
            @Parameter(
                name = "uuid_gen_strategy_class",
                value = "org.hibernate.id.uuid.CustomVersionOneStrategy"
            )
        }
    )
    private UUID id;

    private String title;

    private String author;

    //Getters and setters are omitted for brevity
```

Activat

Example 38. @RowId entity mapping

- If you annotate a given entity with the @RowId annotation and the underlying database supports fetching a record by ROWID (e.g. Oracle), then Hibernate can use the ROWID pseudo-column for CRUD operations.

```
@Entity(name = "Product")
@RowId("ROWID")
public static class Product {

    @Id
    private Long id;

    @Column(name = "`name`")
    private String name;

    @Column(name = "`number`")
    private String number;

    //Getters and setters are omitted for brevity
}
```

Example 39. @ManyToOne Association

```
@Entity(name = "Person")
public static class Person {

    @Id
    @GeneratedValue
    private Long id;

    //Getters and setters are omitted for brevity
}

@Entity(name = "Phone")
public static class Phone {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "`number`")
    private String number;

    @ManyToOne
    @JoinColumn(name = "person_id",
                foreignKey = @ForeignKey(name = "PERSON_ID_FK"))
    private Person person;

    //Getters and setters are omitted for brevity
}
```

Example 40. Unidirectional @OneToMany Association

```
@Entity(name = "Person")
public static class Person {

    @Id
    @GeneratedValue
    private Long id;

    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Phone> phones = new ArrayList<>();

    //Getters and setters are omitted for brevity
}

@Entity(name = "Phone")
public static class Phone {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "`number`")
    private String number;

    //Getters and setters are omitted for brevity
}
```


Bidirectional @OneToMany

- The bidirectional @OneToMany association also requires a @ManyToOne association on the child side.
- Although the Domain Model exposes two sides to navigate this association, behind the scenes, the relational database has only one foreign key for this relationship.
- Every bidirectional association must have one owning side only (the child side), the other one being referred to as the inverse (or the mappedBy) side.

Example 41. @OneToMany association mappedBy the @ManyToOne side

```
@Entity(name = "Person")
public static class Person {

    @Id
    @GeneratedValue
    private Long id;

    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Phone> phones = new ArrayList<>();

    //Getters and setters are omitted for brevity

    public void addPhone(Phone phone) {
        phones.add( phone );
        phone.setPerson( this );
    }

    public void removePhone(Phone phone) {
        phones.remove( phone );
        phone.setPerson( null );
    }
}

@Entity(name = "Phone")
public static class Phone {

    @Id
    @GeneratedValue
    private Long id;

    @NaturalId
    @Column(name = "`number`", unique = true)
    private String number;

    @ManyToOne
    private Person person;
```

Both Sides to in-sync in bidirectional association

- Whenever a bidirectional association is formed, the application developer must make sure both sides are in-sync at all times.
- The `addPhone()` and `removePhone()` are utility methods that synchronize both ends whenever a child element is added or removed.

@OneToOne

- The @OneToOne association can either be unidirectional or bidirectional.
- A unidirectional association follows the relational database foreign key semantics, the client-side owning the relationship.
- A bidirectional association features a mappedBy @OneToOne parent side too.

Example 42. Unidirectional @OneToOne

```
@Entity(name = "Phone")
public static class Phone {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "`number`")
    private String number;

    @OneToOne
    @JoinColumn(name = "details_id")
    private PhoneDetails details;

    //Getters and setters are omitted for brevity
}

@Entity(name = "PhoneDetails")
public static class PhoneDetails {

    @Id
    @GeneratedValue
    private Long id;

    private String provider;

    private String technology;

    //Getters and setters are omitted for brevity
}
```

Example 43. Bidirectional @OneToOne

```
@Entity(name = "Phone")
public static class Phone {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "`number`")
    private String number;

    @OneToOne(
        mappedBy = "phone",
        cascade = CascadeType.ALL,
        orphanRemoval = true,
        fetch = FetchType.LAZY
    )
    private PhoneDetails details;

    //Getters and setters are omitted
}
```

```
@Entity(name = "PhoneDetails")
public static class PhoneDetails {

    @Id
    @GeneratedValue
    private Long id;

    private String provider;

    private String technology;

    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "phone_id")
    private Phone phone;

    //Getters and setters are omitted
}
```

Example 44. Bidirectional @OneToOne lazy parent-side association

```
@Entity(name = "Phone")
public static class Phone {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "`number`")
    private String number;

    @OneToOne(
        mappedBy = "phone",
        cascade = CascadeType.ALL,
        orphanRemoval = true,
        fetch = FetchType.LAZY
    )
    @LazyToOne( LazyToOneOption.NO_PROXY )
    private PhoneDetails details;

    //Getters and setters are omitted
}
```

```
@Entity(name = "PhoneDetails")
public static class PhoneDetails {

    @Id
    @GeneratedValue
    private Long id;

    private String provider;

    private String technology;

    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "phone_id")
    private Phone phone;

    //Getters and setters are omitted
}
```

Example 45. Unidirectional @ManyToMany

```
@Entity(name = "Person")
public static class Person {

    @Id
    @GeneratedValue
    private Long id;

    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    private List<Address> addresses = new ArrayList<>();

    //Getters and setters are omitted for brevity
}

@Entity(name = "Address")
public static class Address {

    @Id
    @GeneratedValue
    private Long id;

    private String street;

    @Column(name = "`number`")
    private String number;

    //Getters and setters are omitted for brevity
}
```


Example 46. Bidirectional @ManyToMany

```
@Entity(name = "Person")
public static class Person {

    @Id
    @GeneratedValue
    private Long id;

    @NaturalId
    private String registrationNumber;

    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    private List<Address> addresses = new ArrayList<>();

    //Getters and setters are omitted for brevity
}

@Entity(name = "Address")
public static class Address {

    @Id
    @GeneratedValue
    private Long id;

    private String street;

    @Column(name = "`number`")
    private String number;

    private String postalCode;

    @ManyToMany(mappedBy = "addresses")
    private List<Person> owners = new ArrayList<>();

    //Getters and setters are omitted for brevity
}
```

Splitting @ManyToMany association into two bidirectional @OneToMany

- If a bidirectional @OneToMany association performs better when removing or changing the order of child elements, the @ManyToMany relationship cannot benefit from such an optimization because the foreign key side is not in control.
- To overcome this limitation, the link table must be directly exposed and the @ManyToMany association split into two bidirectional @OneToMany relationships.

Example 47. Bidirectional many-to-many with link entity

```
@Entity(name = "Person")
public static class Person implements Serializable {

    @Id
    @GeneratedValue
    private Long id;

    @NaturalId
    private String registrationNumber;

    @OneToMany(
        mappedBy = "person",
        cascade = CascadeType.ALL,
        orphanRemoval = true
    )
    private List<PersonAddress> addresses = new ArrayList<>();

    //Getters and setters are omitted for brevity
}

@Entity(name = "PersonAddress")
public static class PersonAddress implements Serializable {

    @Id
    @ManyToOne
    private Person person;

    @Id
    @ManyToOne
    private Address address;

    //Getters and setters are omitted for brevity
}
```

@NotFound association mapping

- When dealing with associations which are not enforced by a Foreign Key, it's possible to bump into inconsistencies if the child record cannot reference a parent entity.
- By default, Hibernate will complain whenever a child association references a non-existing parent record.
- However, you can configure this behavior so that Hibernate can ignore such an Exception and simply assign null as a parent object referenced.
- To ignore non-existing parent entity references, even though not really recommended, it's possible to use the annotation `org.hibernate.annotation.NotFound` with a value of `org.hibernate.annotations.NotFoundAction.IGNORE`.

Example 48. @NotFound mapping example

```
@Entity
@Table( name = "Person" )
public static class Person {

    @Id
    private Long id;

    private String name;

    private String cityName;

    @ManyToOne
    @NotFound ( action = NotFoundAction.IGNORE )
    @JoinColumn(
        name = "cityName",
        referencedColumnName = "name",
        insertable = false,
        updatable = false
    )
    private City city;

    //Getters and setters are omitted for brevity
}
```

```
@Entity
@Table( name = "City" )
public static class City implements Serializable {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

    //Getters and setters are omitted for brevity
}
```

@Any mapping

- ❑ The @Any mapping is useful to emulate a unidirectional @ManyToOne association when there can be multiple target entities.
- ❑ Because the @Any mapping defines a polymorphic association to classes from multiple tables, this association type requires the FK column which provides the associated parent identifier and a metadata information for the associated entity type.
- ❑ This is not the usual way of mapping polymorphic associations and you should use this only in special cases (e.g. audit logs, user session data, etc).
- ❑ The @Any annotation describes the column holding the metadata information.
- ❑ To link the value of the metadata information and an actual entity type, the @AnyDef and @AnyDefs annotations are used.
- ❑ The metaType attribute allows the application to specify a custom type that maps DB column values to persistent classes that have ID properties of the type specified by idType.
- ❑ You must specify the mapping from values of the metaType to class names.

Example 49. Property class hierarchy

```
@Entity
@Table(name="integer_property")
public class IntegerProperty implements Property<Integer> {
```

```
    @Id
    private Long id;
```

```
    @Column(name = "`name`")
    private String name;
```

```
    @Column(name = "`value`")
    private Integer value;
```

```
    @Override
    public String getName() {
        return name;
    }
```

```
    @Override
    public Integer getValue() {
        return value;
    }
```

```
    //Getters and setters omitted for brevity
```

```
@Entity
@Table(name="string_property")
public class StringProperty implements Property<String> {
```

```
    @Id
    private Long id;
```

```
    @Column(name = "`name`")
    private String name;
```

```
    @Column(name = "`value`")
    private String value;
```

```
    @Override
    public String getName() {
        return name;
    }
```

```
    @Override
    public String getValue() {
        return value;
    }
```

```
    //Getters and setters omitted for brevity
```

Example 50. @Any mapping usage

- A PropertyHolder can reference any such property, and, because each Property belongs to a separate table, the @Any annotation is, therefore, required.

```
@Entity
@Table( name = "property_holder" )
public class PropertyHolder {

    @Id
    private Long id;

    @Any(
        metaDef = "PropertyMetaDef",
        metaColumn = @Column( name = "property_type" )
    )
    @JoinColumn( name = "property_id" )
    private Property property;

    //Getters and setters are omitted for brevity
}
```


@ManyToMany mapping

- While the @Any mapping is useful to emulate a @ManyToOne association when there can be multiple target entities, to emulate a @OneToMany association, the @ManyToMany annotation must be used.
- In the following example, the PropertyRepository entity has a collection of Property entities.
- The repository_properties link table holds the associations between PropertyRepository and Property entities.

Example 51. @ManyToMany mapping usage

```
@Entity
@Table( name = "property_repository" )
public class PropertyRepository {

    @Id
    private Long id;

    @ManyToMany(
        metaDef = "PropertyMetaDef",
        metaColumn = @Column( name = "property_type" )
    )
    @Cascade( { org.hibernate.annotations.CascadeType.ALL } )
    @JoinTable(name = "repository_properties",
        joinColumns = @JoinColumn(name = "repository_id"),
        inverseJoinColumns = @JoinColumn(name = "property_id")
    )
    private List<Property<?>> properties = new ArrayList<>( );

    //Getters and setters are omitted for brevity
}
```

Example 52. @JoinFormula mapping usage

The @JoinFormula annotation is used to customize the join between a child Foreign Key and a parent row Primary Key.

```
@Entity(name = "User")
@Table(name = "users")
public static class User {

    @Id
    private Long id;

    private String firstName;

    private String lastName;

    private String phoneNumber;

    @ManyToOne
    @JoinFormula( "REGEXP_REPLACE(phoneNumber, '\\+(\\d+)-.*', '\\1')::int" )
    private Country country;

    //Getters and setters omitted for brevity
}

@Entity(name = "Country")
@Table(name = "countries")
public static class Country {

    @Id
    private Integer id;

    private String name;

    //Getters and setters, equals and hashCode methods omitted for brevity
}
```

Example 53. @JoinColumnOrFormula mapping usage

The @JoinColumnOrFormula annotation is used to customize the join between a child FK and a parent row PK when we need to take into consideration a column value as well as a @JoinFormula.

```
@Entity(name = "User")
@Table(name = "users")
public static class User {

    @Id
    private Long id;

    private String firstName;

    private String lastName;

    private String language;

    @ManyToOne
    @JoinColumnOrFormula( column =
        @JoinColumn(
            name = "language",
            referencedColumnName = "primaryLanguage",
            insertable = false,
            updatable = false
        )
    )
    @JoinColumnOrFormula( formula =
        @JoinFormula(
            value = "true",
            referencedColumnName = "is_default"
        )
    )
    private Country country;

    //Getters and setters omitted for brevity
}
```

```
@Entity(name = "Country")
@Table(name = "countries")
public static class Country implements Serializable {

    @Id
    private Integer id;

    private String name;

    private String primaryLanguage;

    @Column(name = "is_default")
    private boolean _default;

    //Getters and setters, equals and hashCode
}
```

Collections

- Collections can contain almost any other Hibernate type, including basic types, custom types, embeddables, and references to other entities.
- The owner of the collection is always an entity, even if the collection is defined by an embeddable type.
- Collections form one/many-to-many associations between types so there can be:
 - ▣ value type collections
 - ▣ embeddable type collections
 - ▣ entity collections
- Hibernate uses its own collection implementations which are enriched with lazy-loading, caching or state change detection semantics.
- For this reason, persistent collections must be declared as an interface type.
- The actual interface might be `java.util.Collection`, `java.util.List`, `java.util.Set`, `java.util.Map`, `java.util.SortedSet`, `java.util.SortedMap` or even other object types (meaning you will have to write an implementation of `org.hibernate.usertype.UserCollectionType`).

Example 54. Hibernate uses its own collection implementations

- As the following example demonstrates, it's important to use the interface type and not the collection implementation, as declared in the entity mapping.

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    @ElementCollection
    private List<String> phones = new ArrayList<>();

    //Getters and setters are omitted for brevity

}

Person person = entityManager.find( Person.class, 1L );
//Throws java.lang.ClassCastException: org.hibernate.collection.internal.PersistentBag cannot
//be cast to java.util.ArrayList
ArrayList<String> phones = (ArrayList<String>) person.getPhones();
```

Collections as a value type

- Value and embeddable type collections have a similar behavior to basic types since they are automatically persisted when referenced by a persistent object and deleted when unreferenced.
- If a collection is passed from one persistent object to another, its elements might be moved from one table to another.
- Two entities cannot share a reference to the same collection instance.
- Collection-valued properties do not support null value semantics because Hibernate does not distinguish between a null collection reference and an empty collection.
- Collections of value type include basic and embeddable types.
- Collections cannot be nested, and, when used in collections, embeddable types are not allowed to define other collections.
- For collections of value types, JPA 2.0 defines the `@ElementCollection` annotation.
- The lifecycle of the value-type collection is entirely controlled by its owning entity.

Example 55. Embeddable type collections

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    @ElementCollection
    private List<Phone> phones = new ArrayList<>();

    //Getters and setters are omitted for brevity
}

@Embeddable
public static class Phone {

    private String type;

    @Column(name = "`number`")
    private String number;

    //Getters and setters are omitted for brevity
}
```


Collections of entities

- If value type collections can only form a one-to-many association between an owner entity and multiple basic or embeddable types, entity collections can represent both @OneToMany and @ManyToMany associations.
- From a relational DB perspective, associations are defined by the FK side (the child-side).
- With value type collections, only the entity can control the association (the parent-side), but for a collection of entities, both sides are managed by the persistence context.
- For this reason, entity collections can be devised into two main categories:
 - ▣ Unidirectional
 - very similar to value type collections since only the parent side controls this relationship.
 - ▣ bidirectional associations.
 - more tricky since, even if sides need to be in-sync at all times, only one side is responsible for managing the association.
- A bidirectional association has an owning side and an inverse (mappedBy) side.

Example 56. Unidirectional bag

- The unidirectional bag is mapped using a single `@OneToMany` annotation on the parent side of the association. Behind the scenes, Hibernate requires an association table to manage the parent-child relationship, as we can see in the following example:
- The cascading mechanism allows you to propagate an entity state transition from a parent entity to its children.

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    @OneToMany(cascade = CascadeType.ALL)
    private List<Phone> phones = new ArrayList<>();

    //Getters and setters are omitted for brevity
}
```

```
@Entity(name = "Phone")
public static class Phone {

    @Id
    private Long id;

    private String type;

    @Column(name = "`number`")
    private String number;

    //Getters and setters
}
```

Example 57. Bidirectional bag

The @ManyToOne side is the owning side of the bidirectional bag association, while the @OneToMany is the inverse side, being marked with the `mappedBy` attribute.

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL)
    private List<Phone> phones = new ArrayList<>();

    //Getters and setters are omitted
```

```
@Entity(name = "Phone")
public static class Phone {

    @Id
    private Long id;

    private String type;

    @Column(name = "`number`", unique = true)
    @NaturalId
    private String number;

    @ManyToOne
    private Person person;

    //Getters and setters are omitted for brevity
```

Ordered Lists

- Although they use the List interface on the Java side, bags don't retain element order.
- To preserve the collection element order, there are two possibilities:
 - ▣ **@OrderBy**
 - the collection is ordered upon retrieval using a child entity property
 - ▣ **@OrderColumn**
 - the collection uses a dedicated order column in the collection link table

Example 58. Unidirectional @OrderBy list

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    @OneToMany(cascade = CascadeType.ALL)
    @OrderBy("number")
    private List<Phone> phones = new ArrayList<>();

    //Getters and setters are omitted for brevity
}

@Entity(name = "Phone")
public static class Phone {

    @Id
    private Long id;

    private String type;

    @Column(name = "`number`")
    private String number;

    //Getters and setters are omitted for brevity
}
```

The @OrderBy annotation can take multiple entity properties, and each property can take an ordering direction too (e.g. @OrderBy("name ASC, type DESC")).

If no property is specified (e.g. @OrderBy), the primary key of the child entity table is used for ordering.

Example 59. Unidirectional @OrderColumn list

- This time, the link table takes the order_id column and uses it to materialize the collection element order.

```
@OneToMany(cascade = CascadeType.ALL)
@OrderColumn(name = "order_id")
private List<Phone> phones = new ArrayList<>();
```

```
CREATE TABLE Person_Phone (
    Person_id BIGINT NOT NULL ,
    phones_id BIGINT NOT NULL ,
    order_id INTEGER NOT NULL ,
    PRIMARY KEY ( Person_id, order_id )
)
```

Example 60. Bidirectional @OrderBy list

```
@OneToMany(mappedBy = "person", cascade = CascadeType.ALL)
@OrderBy("number")
private List<Phone> phones = new ArrayList<>();
```

Example 61. Bidirectional @OrderColumn list

```
@OneToMany(mappedBy = "person", cascade = CascadeType.ALL)
@OrderColumn(name = "order_id")
private List<Phone> phones = new ArrayList<>();
```

```
CREATE TABLE Phone (
    id BIGINT NOT NULL ,
    number VARCHAR(255) ,
    type VARCHAR(255) ,
    person_id BIGINT ,
    order_id INTEGER ,
    PRIMARY KEY ( id )
)
```


Example 62. Customizing ORDER BY SQL clause

While the JPA `@OrderBy` annotation allows you to specify the entity attributes used for sorting when fetching the current annotated collection, the Hibernate specific `@OrderBy` annotation is used to specify a SQL clause instead.

In the following example, the `@OrderBy` annotation uses the `CHAR_LENGTH` SQL function to order the `Article` entities by the number of characters of the `name` attribute.

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    private String name;

    @OneToMany(
        mappedBy = "person",
        cascade = CascadeType.ALL
    )
    @org.hibernate.annotations.OrderBy(
        clause = "CHAR_LENGTH(name) DESC"
    )
    private List<Article> articles = new ArrayList<>();

    //Getters and setters are omitted for brevity
}
```

```
@Entity(name = "Article")
public static class Article {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

    private String content;

    @ManyToOne(fetch = FetchType.LAZY)
    private Person person;

    //Getters and setters are omitted
}
```

Example 63. Unidirectional set

Sets are collections that don't allow duplicate entries.

Hibernate supports both the unordered Set and the natural-ordering SortedSet.

The unidirectional set uses a link table to hold the parent-child associations and the entity mapping looks as follows:

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    @OneToMany(cascade = CascadeType.ALL)
    private Set<Phone> phones = new HashSet<>();

    //Getters and setters are omitted for brevity
}

@Entity(name = "Phone")
public static class Phone {

    @Id
    private Long id;

    private String type;

    @NaturalId
    @Column(name = "`number`")
    private String number;

    //Getters and setters are omitted for brevity
}
```

Example 64. Bidirectional sets

Just like bidirectional bags, the bidirectional set doesn't use a link table, and the child table has a foreign key referencing the parent table primary key. The lifecycle is just like with bidirectional bags except for the duplicates which are filtered out.

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL)
    private Set<Phone> phones = new HashSet<>();

    //Getters and setters are omitted for brevity
```

```
@Entity(name = "Phone")
public static class Phone {

    @Id
    private Long id;

    private String type;

    @Column(name = "`number`", unique = true)
    @NaturalId
    private String number;

    @ManyToOne
    private Person person;

    //Getters and setters are omitted for brevity
```

Example 65. Unidirectional natural sorted set

- For sorted sets, the entity mapping must use the SortedSet interface instead.
- According to the SortedSet contract, all elements must implement the Comparable interface and therefore provide the sorting logic.

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    @OneToMany(cascade = CascadeType.ALL)
    @SortNatural
    private SortedSet<Phone> phones = new TreeSet<>();

    //Getters and setters are omitted for brevity
}
```

```
@Entity(name = "Phone")
public static class Phone implements Comparable<Phone> {

    @Id
    private Long id;

    private String type;

    @NaturalId
    @Column(name = "`number`")
    private String number;

    //Getters and setters are omitted for brevity
}
```

Example 66. Unidirectional custom comparator sorted set

To provide a custom sorting logic, Hibernate also provides a `@SortComparator` annotation:

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    @OneToMany(cascade = CascadeType.ALL)
    @SortComparator(ReverseComparator.class)
    private SortedSet<Phone> phones = new TreeSet<>();

    //Getters and setters are omitted for brevity
}
```

```
@Entity(name = "Phone")
public static class Phone implements Comparable<Phone> {

    @Id
    private Long id;

    private String type;

    @NaturalId
    @Column(name = "`number`")
    private String number;

    //Getters and setters are omitted
}
```

```
public static class ReverseComparator implements Comparator<Phone> {

    @Override
    public int compare(Phone o1, Phone o2) {
        return o2.compareTo( o1 );
    }
}
```

Example 67. Bidirectional natural sorted set

- @SortNatural and @SortComparator work the same for bidirectional sorted sets too

```
@OneToMany(mappedBy = "person", cascade = CascadeType.ALL)
```

```
@SortNatural
```

```
private SortedSet<Phone> phones = new TreeSet<>();
```

```
@SortComparator(ReverseComparator.class)
```

```
private SortedSet<Phone> phones = new TreeSet<>();
```

Maps

- Map is a ternary association because it requires a parent entity, a map key, and a value.
- An entity can either be a map key or a map value, depending on the mapping.
- Hibernate allows using the following map keys:
 - **MapKeyColumn**
 - for value type maps, the map key is a column in the link table that defines the grouping logic
 - **MapKey**
 - the map key is either the PK or another property of the entity stored as a map entry value
 - **MapKeyEnumerated**
 - the map key is an Enum of the target child entity
 - **MapKeyTemporal**
 - the map key is a Date or a Calendar of the target child entity
 - **MapKeyJoinColumn**
 - the map key is an entity mapped as an association in the child entity stored as a map entry key
- Value type maps
 - A map of value type must use the **@ElementCollection** annotation, just like value type lists, bags or sets.

Example 68. Value type map with an entity as a map key

```
public enum PhoneType {  
    LAND_LINE,  
    MOBILE  
}
```

```
@Entity(name = "Person")  
public static class Person {
```

```
    @Id  
    private Long id;
```

```
    @Temporal(TemporalType.TIMESTAMP)  
    @ElementCollection  
    @CollectionTable(name = "phone_register")  
    @Column(name = "since")  
    private Map<Phone, Date> phoneRegister = new HashMap<>();
```

```
    //Getters and setters are omitted for brevity
```

```
@Embeddable
```

```
public static class Phone {
```

```
    private PhoneType type;
```

```
    @Column(name = "`number`")  
    private String number;
```

```
    //Getters and setters are omitted
```


Example 69. @MapKeyType mapping example

```
@Entity
@Table(name = "person")
public static class Person {

    @Id
    private Long id;

    @ElementCollection
    @CollectionTable(
        name = "call_register",
        joinColumns = @JoinColumn(name = "person_id")
    )
    @MapKeyType(
        @Type(
            type = "org.hibernate.userguide.collections.type.TimestampEpochType"
        )
    )
    @MapKeyColumn( name = "call_timestamp_epoch" )
    @Column(name = "phone_number")
    private Map<Date, Integer> callRegister = new HashMap<>();

    //Getters and setters
}
```

```
public class TimestampEpochType
    extends AbstractSingleColumnStandardBasicType<Date>
    implements VersionType<Date>, LiteralType<Date> {

    public static final TimestampEpochType INSTANCE = new TimestampEpochType();

    public TimestampEpochType() {
        super(
            BigIntTypeDescriptor.INSTANCE,
            JdbcTimestampTypeDescriptor.INSTANCE
        );
    }
}
```

Example 70. @MapKeyClass mapping example

If you want to use the PhoneNumber interface as a java.util.Map key, then you need to supply the @MapKeyClass annotation as well.

```
@Entity
@Table(name = "person")
public static class Person {

    @Id
    private Long id;

    @ElementCollection
    @CollectionTable(
        name = "call_register",
        joinColumns = @JoinColumn(name = "person_id")
    )
    @MapKeyColumn( name = "call_timestamp_epoch" )
    @MapKeyClass( MobilePhone.class )
    @Column(name = "call_register")
    private Map<PhoneNumber, Integer> callRegister = new HashMap<>();

    //Getters and setters are omitted for brevity
}
```

Example 71. Unidirectional Map

A unidirectional map exposes a parent-child association from the parent-side only.

The @MapKey annotation is used to define the entity attribute used as a key of the java.util.Map in question.

```
public enum PhoneType {
    LAND_LINE,
    MOBILE
}

@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    @JoinTable(
        name = "phone_register",
        joinColumns = @JoinColumn(name = "phone_id"),
        inverseJoinColumns = @JoinColumn(name = "person_id"))
    @MapKey(name = "since")
    @MapKeyTemporal(TemporalType.TIMESTAMP)
    private Map<Date, Phone> phoneRegister = new HashMap<>();

    //Getters and setters are omitted for brevity
}
```

```
@Entity(name = "Phone")
public static class Phone {

    @Id
    @GeneratedValue
    private Long id;

    private PhoneType type;

    @Column(name = "`number`")
    private String number;

    private Date since;

    //Getters and setters are omitted
}
```

Example 72. Bidirectional Map

Like most bidirectional associations, this relationship is owned by the child-side while the parent is the inverse side and can propagate its own state transitions to the child entities.

`@MapKeyEnumerated` was used so that the Phone enumeration becomes the map key.

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL, orphanRemoval = true)
    @MapKey(name = "type")
    @MapKeyEnumerated
    private Map<PhoneType, Phone> phoneRegister = new HashMap<>();

    //Getters and setters are omitted for brevity
}
```

```
@Entity(name = "Phone")
public static class Phone {

    @Id
    @GeneratedValue
    private Long id;

    private PhoneType type;

    @Column(name = "`number`")
    private String number;

    private Date since;

    @ManyToOne
    private Person person;

    //Getters and setters
}
```

Example 73. Comma delimited collection

Previously, we marked the collection attribute as either `ElementCollection`, `OneToMany` or `ManyToMany`. Collections not marked as such require a custom Hibernate Type and the collection elements must be stored in a single database column.

```
@Entity(name = "Person")
public static class Person {

    @Id
    private Long id;

    @Type(type = "comma_delimited_strings")
    private List<String> phones = new ArrayList<>();

    public List<String> getPhones() {
        return phones;
    }
}

public class CommaDelimitedStringsJavaTypeDescriptor extends AbstractTypeDescriptor<List> {

    public static final String DELIMITER = ",";

    public CommaDelimitedStringsJavaTypeDescriptor() {
        super(
            List.class,
            new MutableMutabilityPlan<List>() {
                @Override
                protected List deepCopyNotNull(List value) {
                    return new ArrayList( value );
                }
            }
        );
    }
}
```

Natural Ids

- Natural ids represent domain model unique identifiers that have a meaning in the real world too.
- Even if a natural id does not make a good primary key (surrogate keys being usually preferred), it's still useful to tell Hibernate about it.
- As we will see later, Hibernate provides a dedicated, efficient API for loading an entity by its natural id much like it offers for loading by its identifier (PK).
- Natural ids are defined in terms of one or more persistent attributes.

Example 74. Natural id using single basic attribute

```
@Entity(name = "Book")
public static class Book {

    @Id
    private Long id;

    private String title;

    private String author;

    @NaturalId
    private String isbn;

    //Getters and setters are omitted for brevity
}
```

Example 75. Natural id using single embedded attribute

```
@Entity(name = "Book")
public static class Book {

    @Id
    private Long id;

    private String title;

    private String author;

    @NaturalId
    @Embedded
    private Isbn isbn;

    //Getters and setters
```

```
@Embeddable
public static class Isbn implements Serializable {

    private String isbn10;

    private String isbn13;

    //Getters and setters are omitted for brevity
```


Example 76. Natural id using multiple persistent attributes

```
@Entity(name = "Book")
public static class Book {

    @Id
    private Long id;

    private String title;

    private String author;

    @NaturalId
    private String productNumber;

    @NaturalId
    @ManyToOne(fetch = FetchType.LAZY)
    private Publisher publisher;

    //Getters and setters are omitted
```

```
@Entity(name = "Publisher")
public static class Publisher implements Serializable {

    @Id
    private Long id;

    private String name;

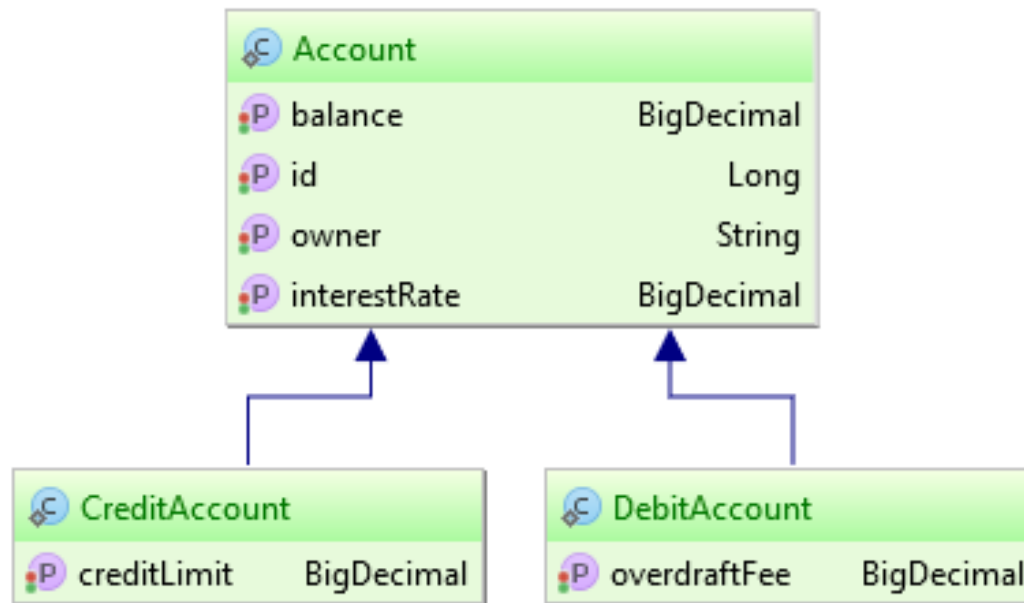
    //Getters and setters are omitted for brevity
```

Inheritance

- Although relational database systems don't provide support for inheritance, Hibernate provides several strategies to leverage this object-oriented trait onto domain model entities:
- **MappedSuperclass**
 - ▣ Inheritance is implemented in the domain model only without reflecting it in the database schema.
- **Single table**
 - ▣ The domain model class hierarchy is materialized into a single table which contains entities belonging to different class types.
- **Joined table**
 - ▣ The base class and all the subclasses have their own database tables and fetching a subclass entity requires a join with the parent table as well.
- **Table per class**
 - ▣ Each subclass has its own table containing both the subclass and the base class properties.

MappedSuperclass

- In the following domain model class hierarchy, a DebitAccount and a CreditAccount share the same Account base class.
- When using MappedSuperclass, the inheritance is visible in the domain model only, and each database table contains both the base class and the subclass properties.



Example 77. @MappedSuperclass inheritance

@MappedSuperclass

```
public static class Account {  
  
    @Id  
    private Long id;  
  
    private String owner;  
  
    private BigDecimal balance;  
  
    private BigDecimal interestRate;  
  
    //Getters and setters are omitted for brevity  
}  
  
@Entity(name = "DebitAccount")  
public static class DebitAccount extends Account {  
  
    private BigDecimal overdraftFee;  
  
    //Getters and setters are omitted for brevity  
}  
  
@Entity(name = "CreditAccount")  
public static class CreditAccount extends Account {  
  
    private BigDecimal creditLimit;  
  
    //Getters and setters are omitted for brevity  
}
```

Example 78. Single Table Inheritance

The single table inheritance strategy maps all subclasses to only one database table.

Each subclass in a hierarchy must define a unique discriminator value, which is used to differentiate between rows belonging to separate subclass types.

If this is not specified, the DTYPE column, which has the name of the entity as a value, is used as a discriminator, storing the associated subclass name.

To customize the discriminator column, we can use the `@DiscriminatorColumn` annotation

```
@Entity(name = "Account")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public static class Account {

    @Id
    private Long id;

    private String owner;

    private BigDecimal balance;

    private BigDecimal interestRate;

    //Getters and setters are omitted

}
```

```
@Entity(name = "DebitAccount")
public static class DebitAccount extends Account {

    private BigDecimal overdraftFee;

    //Getters and setters are omitted for brevity

}

@Entity(name = "CreditAccount")
public static class CreditAccount extends Account {

    private BigDecimal creditLimit;

    //Getters and setters are omitted for brevity

}
```

Example 79. Example Joined table

Each subclass can also be mapped to its own table.

This is also called table-per-subclass mapping strategy.

An inherited state is retrieved by joining with the table of the superclass.

```
@Entity(name = "Account")
@Inheritance(strategy = InheritanceType.JOINED)
public static class Account {

    @Id
    private Long id;

    private String owner;

    private BigDecimal balance;

    private BigDecimal interestRate;

    //Getters and setters are omitted
}
```

```
@Entity(name = "DebitAccount")
public static class DebitAccount extends Account {

    private BigDecimal overdraftFee;

    //Getters and setters are omitted for brevity
}

@Entity(name = "CreditAccount")
public static class CreditAccount extends Account {

    private BigDecimal creditLimit;

    //Getters and setters are omitted for brevity
}
```

Example 80. Join Table with @PrimaryKeyJoinColumn

```
@Entity(name = "Account")
@Inheritance(strategy = InheritanceType.JOINED)
public static class Account {

    @Id
    private Long id;

    private String owner;

    private BigDecimal balance;

    private BigDecimal interestRate;

    //Getters and setters are omitted for brevity
}

@Entity(name = "DebitAccount")
@PrimaryKeyJoinColumn(name = "account_id")
public static class DebitAccount extends Account {

    private BigDecimal overdraftFee;

    //Getters and setters are omitted for brevity
}

@Entity(name = "CreditAccount")
@PrimaryKeyJoinColumn(name = "account_id")
public static class CreditAccount extends Account {

    private BigDecimal creditLimit;

    //Getters and setters are omitted for brevity
}
```

Example 81. Table per class

A third option is to map only the concrete classes of an inheritance hierarchy to tables.

This is called the table-per-concrete-class strategy.

Each table defines all persistent states of the class, including the inherited state.

In Hibernate, it is not necessary to explicitly map such inheritance hierarchies.

You can map each class as a separate entity root.

However, if you wish to use polymorphic associations (e.g. an association to the superclass of your hierarchy), you need to use the union subclass mapping.

```
@Entity(name = "Account")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public static class Account {

    @Id
    private Long id;

    private String owner;

    private BigDecimal balance;

    private BigDecimal interestRate;

    //Getters and setters are omitted
```

```
@Entity(name = "DebitAccount")
public static class DebitAccount extends Account {

    private BigDecimal overdraftFee;

    //Getters and setters are omitted for brevity
}

@Entity(name = "CreditAccount")
public static class CreditAccount extends Account {

    private BigDecimal creditLimit;

    //Getters and setters are omitted for brevity
}
```


Immutability

- Immutability can be specified for both entities and collections.
- If a specific entity is immutable, it is good practice to mark it with the `@Immutable` annotation.

```
@Entity(name = "Event")
@Immutable
public static class Event {

    @Id
    private Long id;

    private Date createdOn;

    private String message;

    //Getters and setters are omitted for brevity
}
```