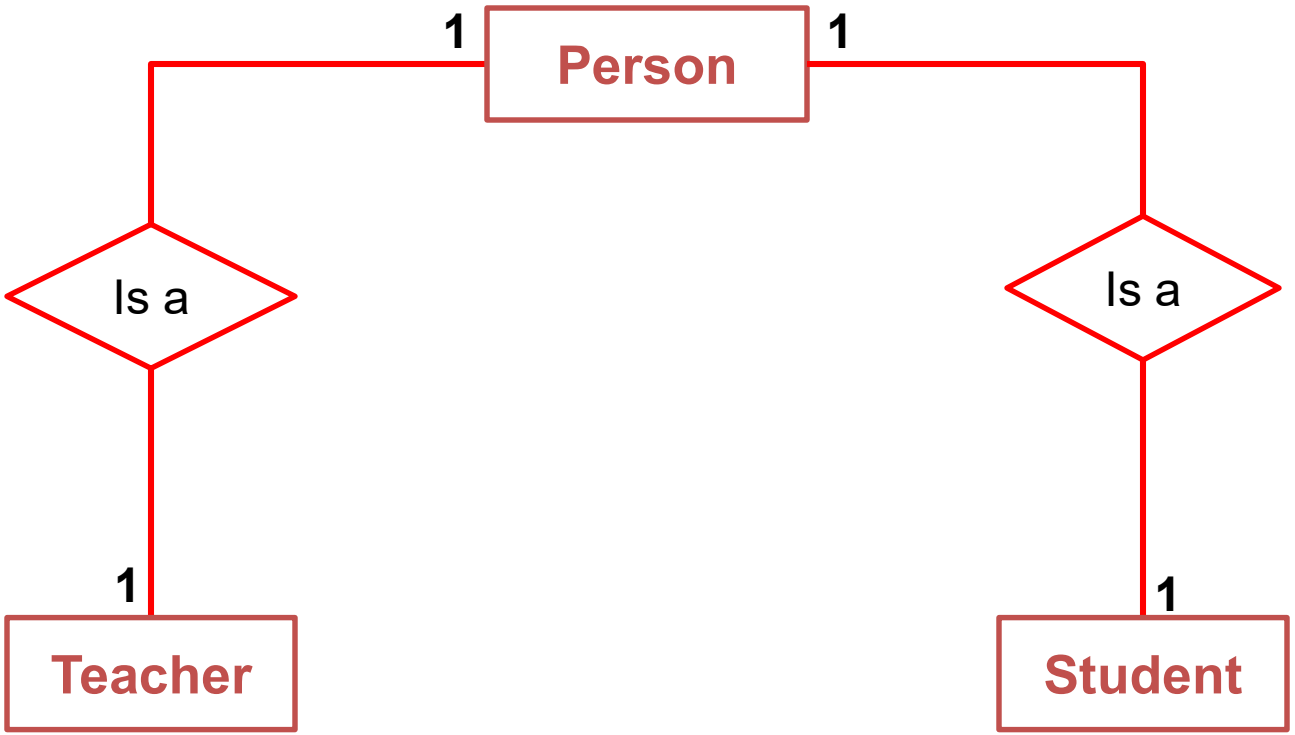
Lesson Outline

- Many-to-one (Uni-directional and Bi-directional)
- One-to-one (Uni-directional and Bi-directional)
- Many-to-many (Uni-directional and Bi-directional)
- Inheritance Mapping Strategies
- Table per concrete classes
- Table per unions subclasses
- Shared Table per subclasses
- Table per joined subclasses

DB Diagram 2 (Inheritance)

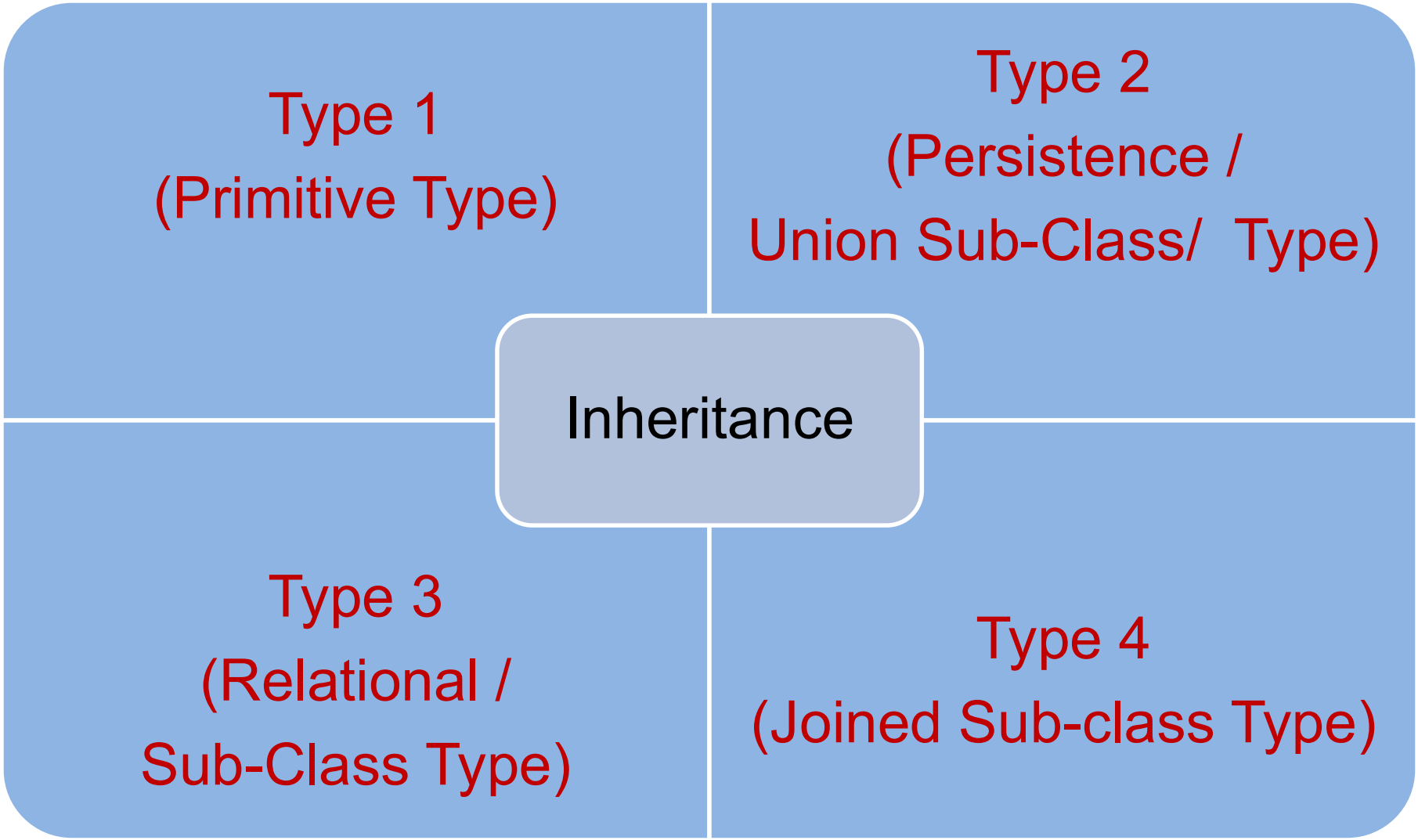


Inheritance Association

- There are 4 ways to represent the Inheritance Relation.

- Each of them are different in three areas.
  - Database.
  - Classes.
  - Relations Mapping.

Inheritance Association



Lesson Outline

- Many-to-one (Uni-directional and Bi-directional)
- One-to-one (Uni-directional and Bi-directional)
- Many-to-many (Uni-directional and Bi-directional)
- Inheritance Mapping Strategies
- Table per concrete classes
- Table per unions subclasses
- Shared Table per subclasses
- Table per joined subclasses

Type 1 (Primitive Type)

- Database: Represented in three tables.

Person (Base)		Teacher (Derived)		Student (Derived)	
id	Integer	id	Integer	id	Integer
first_name	String	first_name	String	first_name	String
last_name	String	last_name	String	last_name	String
		hire_date	String	department	String

- First Three column are repeated with the same name & type.

Type 1 (Primitive Type) (Ex.)

- Classes: Represented in three classes
  - Parent Class (Person):
    - Contains abstraction of data (id, first_name, last_name)
  - Child Class (Teacher):
    - Extends Parent & contains added properties as (hire_date)
  - Child Class (Student):
    - Extends Parent & contains added properties as (department)

Type 1 (Primitive Type) (Ex.)

- Relations Mapping: Represented in three separated classes with no relation between them.

```
<hibernate-mapping package="pojo">
    <class name="Person" table="person">
      <id name="id" column="id">
          <generator class="increment"/>
      </id>
      <property name="firstName" column="first_name">
      <property name="lastName" column="last_name">
    </class>
</hibernate-mapping>
```

Parent

Type 1 (Primitive Type) (Ex.)

```
<hibernate-mapping package="pojo">
   <class name="Student" table="student">
      <id name="id" column="id">
         <generator class="increment"/>
      </id>
      <property name="firstName" column="first_name">
      <property name="lastName" column="last_name">
      <property name="department" column="department">
   </class>
</hibernate-mapping>
```

Child(1)

Type 1 (Primitive Type) (Ex.)

```
<hibernate-mapping package="pojo">
   <class name="Teacher" table="teacher">
      <id name="id" column="id">
         <generator class="increment"/>
      </id>
      <property name="firstName" column="first_name">
      <property name="lastName" column="last_name">
      <property name="hireDate" column="hire_date"
              type="date">
   </class>
</hibernate-mapping>
```

Child(2)

Type 1 (Primitive Type) (Ex.)

- Cons:
  - Relations aren't represented in database.
  - Relations aren't represented in mapping files.
  - Column are repeated with the same name & type in each derived.

Lesson Outline

- Many-to-one (Uni-directional and Bi-directional)
- One-to-one (Uni-directional and Bi-directional)
- Many-to-many (Uni-directional and Bi-directional)
- Inheritance Mapping Strategies
- Table per concrete classes
- Table per unions subclasses
- Shared Table per subclasses
- Table per joined subclasses

Type 2 (Union Sub-Class Type)

- Database: Represented in three tables. (Same as type1)

Person (Base)		Teacher (Derived)		Student (Derived)	
id	Integer	id	Integer	id	Integer
first_name	String	first_name	String	first_name	String
last_name	String	last_name	String	last_name	String
		hire_date	String	department	String

- First Three column are repeated with the same name & type.

Type 2 (Union Sub-Class Type) (Ex.)

- Classes: Represented in three classes. (Same as type1)
  - Parent Class (Person):
    - Contains abstraction of data (id, first_name, last_name)
  - Child Class (Teacher):
    - Extends Parent & contains added properties as (hire_date)
  - Child Class (Student):
    - Extends Parent & contains added properties as (department)

Type 2 (Union Sub-Class Type) (Ex.)

```
<hibernate-mapping package="pojo">
  <class name="Person" table="person">
    <id name="id" column="id">
      <generator class="increment"/></id>
    <property name="firstName" column="first_name">
    <property name="lastName" column="last_name">
    <union-subclass name="Student" table="student">
       <property name="department" column="department">
    </union-subclass>
    <union-subclass name="Teacher" table="teacher">
       <property name="hireDate" column="hire_date"
             type="date">
    </union-subclass>
  </class>
</hibernate-mapping>
```

Type 2 (Union Sub-Class Type) (Ex.)

- Pros:
  - Relations are represented in mapping files.
- Cons:
  - Relations aren't represented in database.
  - Column are repeated with the same name & type in each derived.

Lesson Outline

- Many-to-one (Uni-directional and Bi-directional)
- One-to-one (Uni-directional and Bi-directional)
- Many-to-many (Uni-directional and Bi-directional)
- Inheritance Mapping Strategies
- Table per concrete classes
- Table per unions subclasses
- Shared Table per subclasses
- Table per joined subclasses

Type 3 (Sub-Class Type)

- Database: Represented in one table only.

Person (Base, Derived 1,Derived 2)	
id	Integer
first_name	String
last_name	String
hire_date	String
department	String
person_type	String

Type 3 (Sub-Class Type) (Ex.)

- Classes: Represented in three classes
  - Parent Class (Person):
    - Contains abstraction of data (id, first_name, last_name)
  - Child Class (Teacher):
    - Extends Parent & contains added properties as (hire_date)
  - Child Class (Student):
    - Extends Parent & contains added properties as (department)
- There is no property for person_type.

Type 3 (Sub-Class Type) (Ex.)

- Relations Mapping:
  - Represented in separated classes also derived classes.
- Discriminator:
  - It's an indicator to specify which instance is created from this tabled.
- Note: Any added properties must be enabled null.

Type 3 (Sub-Class Type) (Ex.)

```
<hibernate-mapping package="pojo">
  <class name="Person" table="person">
    <id name="id" column="id">
      <generator class="increment"/></id>
    <discriminator column="person_type" type="String"/>
    <property name="firstName" column="first_name"/>
    <property name="lastName" column="last_name"/>
  </class>
  <subclass name="Student" extends="person"
        discriminator-value="Student">
    <property name="department" column="department"/>
  </subclass>
  <subclass name="Teacher" extends="person"
        discriminator-value="Teacher">
    <property name="hireDate" column="hire_date"
              type="date"/>
  </subclass>
</hibernate-mapping>
```

Type 3 (Sub-Class Type) (Ex.)

- Pros:
  - Relations are represented in mapping files.
- Cons:
  - Any added properties in derived classes must be null-enabled.
  - Relations aren't represented in database.
  - No separation between data in tables for all (base & derived).

Lesson Outline

- Many-to-one (Uni-directional and Bi-directional)
- One-to-one (Uni-directional and Bi-directional)
- Many-to-many (Uni-directional and Bi-directional)
- Inheritance Mapping Strategies
- Table per concrete classes
- Table per unions subclasses
- Shared Table per subclasses
- Table per joined subclasses

Type 4 (Joined Sub-Class Type)

- Database: Represented in three tables.

Person (Base)		Teacher (Derived)		Student (Derived)	
id	Integer	id	Integer	id	Integer
first_name	String	hire_date	String	department	String
last_name	String				

- First column is repeated with the same name & type.

Type 4 (Joined Sub-Class Type) (Ex.)

- Classes: Represented in three classes.
  - Parent Class (Person):
    - Contains abstraction of data (id, first_name, last_name)
  - Child Class (Teacher):
    - Extends Parent & contains added properties as (hire_date)
  - Child Class (Student):
    - Extends Parent & contains added properties as (department)

Type 4 (Joined Sub-Class Type) (Ex.)

```
<hibernate-mapping package="pojo">
  <class name="Person" table="person">
    <id name="id" column="id">
      <generator class="increment"/></id>
    <property name="firstName" column="first_name">
    <property name="lastName" column="last_name">
    <joined-subclass name="Student" table="student">
       <key column="id"/>
       <property name="department" column="department">
    </joined-subclass>
    <joined-subclass name="Teacher" table="teacher">
       <key column="id"/>
       <property name="hireDate" column="hire_date"
            type="date">
    </joined-subclass>
  </class></hibernate-mapping>
```

Type 4 (Joined Sub-Class Type) (Ex.)

- Pros:
  - All other Cons Are solved.
- Note:
  - It's recommended to add an attribute to parent table to specify which row represent what type of child.

Lab Assignment

- Perform these steps for each type:
  - Restore Inheritance Schema.
  - Create the mapping files to map these classes to domain models (java objects).
  - Perform CRUD operations on Person, Student, Teacher.

Lab Assignment

- Restore Bidding Schema.
- Create the mapping files to map these classes to domain models (java objects).
- Insert New Product & make bids on it and buy a product.
- Load a Product and update its values.

Hibernate
Fetching Strategies

Lesson Outline

- Lazy and Eager loading
- Fetching Strategies
  - Join Fetching Strategy
  - Select Fetching Strategy
  - Sub-Select Fetching Strategy
  - Batch Size Fetching Strategy
- The Second Level Cache
  - Read-Only Cache
  - Read-Write Cache
  - Nonstrict-Read-Write Cache

Lesson Outline

- Lazy and Eager loading
- Fetching Strategies
  - Join Fetching Strategy
  - Select Fetching Strategy
  - Sub-Select Fetching Strategy
  - Batch Size Fetching Strategy
- The Second Level Cache
  - Read-Only Cache
  - Read-Write Cache
  - Nonstrict-Read-Write Cache

Lazy & Eager Loading

- Lazy loading means that all associated entities and collections aren't initialized if you load an entity object.
- This feature is enabled by default in Hibernate.
- It can be disabled by setting the attribute lazy = "false" in the mapping file.

Lazy & Eager Loading (Example)

```
<hibernate-mapping>
   <class name="dao.Seller" table="seller"
    catalog="biddingschema">
    ...
    <set name="products" table="product"
         lazy="true">
        <key>   <column name="seller_id" />   </key>
      <one-to-many class="dao.Product" />
    </set>
    ...
   </class>
</hibernate-mapping>
```

Seller.hbm.xml

Lazy & Eager Loading (Ex.)

- If the lazy attribute set to “false” when retrieving the Seller’s data

```
Seller s = (Seller)session.load(Seller.class,1);
```

- It will retrieve the list of Products as well
  - So calling the previous method will return the list of products whether in the persisted or detached mood.

Lazy & Eager Loading (Ex.)

- You can set lazy attribute to “Extra” when retrieving the Seller’s data

```
Seller s = (Seller)session.load(Seller.class,1);
```

- Individual elements of the collection are accessed from the database as needed. Hibernate tries not to fetch the whole collection into memory unless absolutely needed. It is suitable for large collections.

Lesson Outline

- Lazy and Eager loading
- Fetching Strategies
  - Join Fetching Strategy
  - Select Fetching Strategy
  - Sub-Select Fetching Strategy
  - Batch Size Fetching Strategy
- The Second Level Cache
  - Read-Only Cache
  - Read-Write Cache
  - Nonstrict-Read-Write Cache

Fetching Strategies

- Fetching:
  - It defines if and how an associated object or a collection should be loaded,
  - When the owning entity object is loaded, and when you access an associated object or collection.

Fetching Strategies (Example)

```
<hibernate-mapping>
   <class name="dao.Seller" table="seller"
    catalog="biddingschema">
    ...
    <set name="products" table="product"
         lazy="true" fetch="select">
        <key>   <column name="seller_id" />   </key>
      <one-to-many class="dao.Product" />
    </set>
    ...
   </class>
</hibernate-mapping>
```

Seller.hbm.xml

Fetching Strategies (Ex.)

- The main goal is to minimize the number of SQL statements, so that querying can be as efficient as possible.
- You do this by applying the best fetching strategy for each collection or association
- Fetching Strategies:
  - Join fetching
  - Select fetching
  - Batch fetching

Lesson Outline

- Lazy and Eager loading
- Fetching Strategies
  - Join Fetching Strategy
  - Select Fetching Strategy
  - Sub-Select Fetching Strategy
  - Batch Size Fetching Strategy
- The Second Level Cache
  - Read-Only Cache
  - Read-Write Cache
  - Nonstrict-Read-Write Cache

Fetching Strategies (Ex.)

- Select fetching:
  - a second SELECT is used to retrieve the associated entity or collection.
  - Unless you explicitly disable lazy fetching by specifying lazy="false", this second select will only be executed when you actually access the association.

```
<class name="dao.Seller" table="seller">
    <set name="products" table="product"
     lazy="true" fetch="select">
        <key column="seller_id"/>
      <one-to-many class="dao.Product" />
    </set>
</class>
```

Fetching Strategies (Ex.)

```
Seller s = (Seller)session.load(Seller.class,1);
```

- By default (the lazy attribute set to true) and if the object is detached (when the session is closed)
  - Calling **`seller.getProducts()`** will throw an Exception
- If the object is still in the persistence state, calling the previous method will hit the DB and retrieve the list of Products.

Lesson Outline

- Lazy and Eager loading
- Fetching Strategies
  - Join Fetching Strategy
  - Select Fetching Strategy
  - Sub-Select Fetching Strategy
  - Batch Size Fetching Strategy
- The Second Level Cache
  - Read-Only Cache
  - Read-Write Cache
  - Nonstrict-Read-Write Cache

Fetching Strategies (Ex.)

- Sub-Select fetching:
  - a second SELECT is used to retrieve the associated collections for all entities retrieved in a previous query.
  - Unless you explicitly disable lazy fetching by specifying lazy="false", this second select will only be executed when you access the association.

```
<class name="dao.Seller" table="seller">
    <set name="products" table="product"
     lazy="true" fetch="sub-select">
        <key column="seller_id"/>
      <one-to-many class="dao.Product" />
    </set>
</class>
```

Lesson Outline

- Lazy and Eager loading
- Fetching Strategies
  - Join Fetching Strategy
  - Select Fetching Strategy
  - Sub-Select Fetching Strategy
  - Batch Size Fetching Strategy
- The Second Level Cache
  - Read-Only Cache
  - Read-Write Cache
  - Nonstrict-Read-Write Cache

Fetching Strategies (Ex.)

- Join fetching:
  - Hibernate retrieves the associated instance or collection in the same SELECT, using an OUTER JOIN.

```
<class name="dao.Seller" table="seller">
    ...
    <set name="products" table="product"
         lazy="true" fetch="join">
        <key column="seller_id"/>
      <one-to-many class="dao.Product" />
    </set>
</class>
```

Lesson Outline

- Lazy and Eager loading
- Fetching Strategies
  - Join Fetching Strategy
  - Select Fetching Strategy
  - Sub-Select Fetching Strategy
  - Batch Size Fetching Strategy
- The Second Level Cache
  - Read-Only Cache
  - Read-Write Cache
  - Nonstrict-Read-Write Cache

Fetching Strategies (Ex.)

- Batch fetching :
  - an optimization strategy for select fetching - Hibernate retrieves a batch of entity instances or collections in a single SELECT.

```
<class name="dao.Seller" table="seller">
    ...
    <set name="products" table="product"
         lazy="true" batch-size="3">
        <key column="seller_id"/>
      <one-to-many class="dao.Product" />
    </set>
</class>
```

Lesson Outline

- Lazy and Eager loading
- Fetching Strategies
  - Join Fetching Strategy
  - Select Fetching Strategy
  - Sub-Select Fetching Strategy
  - Batch Size Fetching Strategy
- The Second Level Cache
  - Read-Only Cache
  - Read-Write Cache
  - Nonstrict-Read-Write Cache

Second Level Cache

- How did hibernate will cache the persisted & detached Objects?
- We have the option to tell Hibernate which caching implementation to use by specifying the name of a class that implements org.hibernate.cache.CacheProvider using the property hibernate.cache.provider_class.
- You can also implement your own and plug it in.

Second Level Cache (Ex.)

- Cache Mappings:
  - `<cache usage="read-write"/>`

    `usage="read-write | nonstrict-read-write | read-only | transactional"`

    `region="RegionName"`

    `include="all | non-lazy" />`
  - usage**: specifies the caching strategy.
  - region: specifies the name of the second level cache region
  - include: specifies that properties of the entity mapped with lazy="true" cannot be cached when attribute-level lazy fetching is enabled

Lesson Outline

- Lazy and Eager loading
- Fetching Strategies
  - Join Fetching Strategy
  - Select Fetching Strategy
  - Sub-Select Fetching Strategy
  - Batch Size Fetching Strategy
- The Second Level Cache
  - Read-Only Cache
  - Read-Write Cache
  - Nonstrict-Read-Write Cache

Second Level Cache (Ex.)

- Read-Only cache:
  - If your application needs to read, but not modify, instances of a persistent class, a read-only cache can be used.
  - This is the simplest and optimal performing strategy. It is even safe for use in a cluster.

```
<class name="dao.Seller" table="seller"
mutable="false">

    <cache usage="read-only"/>

</class>
```

Lesson Outline

- Lazy and Eager loading
- Fetching Strategies
  - Join Fetching Strategy
  - Select Fetching Strategy
  - Sub-Select Fetching Strategy
  - Batch Size Fetching Strategy
- The Second Level Cache
  - Read-Only Cache
  - Read-Write Cache
  - Nonstrict-Read-Write Cache

Second Level Cache (Ex.)

- Read-Write cache:
  - If the application needs to update data, a read-write cache might be appropriate.
  - This cache strategy should never be used if serializable transaction isolation level is required.

```
<class name="dao.Seller" table="seller">
    <cache usage="read-write"/>
    ...
    <set name="products" table="product">
      <cache usage="read-only"/>
      ...
    </set>
</class>
```

Lesson Outline

- Lazy and Eager loading
- Fetching Strategies
  - Join Fetching Strategy
  - Select Fetching Strategy
  - Sub-Select Fetching Strategy
  - Batch Size Fetching Strategy
- The Second Level Cache
  - Read-Only Cache
  - Read-Write Cache
  - Nonstrict-Read-Write Cache

Second Level Cache (Ex.)

- nonstrict-read-write cache:
  - If the application needs to update data, a read-write cache might be appropriate.
  - This cache strategy should never be used if serializable transa
  - Cache is not locked at all.

```
<class name="dao.Seller" table="seller">
    <cache usage="nonstrict-read-write"/>
    ...
</class>
```