



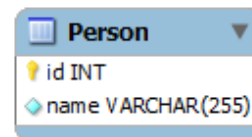
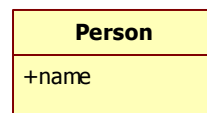
# Module 01: Entity Mapping

CS544: Enterprise Architecture



# Entity Class Mapping

- Entity classes can be thought of as the distinct business concepts in a domain driven design
- We will start with the basic Java class requirements for persisting entity classes
- After which we will go into identity mapping, mapping data types, and Hibernate access
- We will provide both XML and annotation mapping examples for each subject we cover





# Class Requirements

- Hibernate requires that entity classes have:
  - A field that can be used as identifier
  - A default constructor
  - Getter and Setter Methods for all properties

```
package intro;
```

```
public class Person {
```

```
    private long id;
```

```
    private String name;
```

```
    public Person() {}
```

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

```
    public long getId() { return id; }
```

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

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

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

```
}
```

An identifier

A default constructor

Getter and Setter Methods



# About Annotations

- Most of the mapping annotations we will use are part of the Java Persistence API (JPA)

```
@Entity(name="MY_PERSON")  
public class Person {  
    @Id  
    @Column(name="PERSON_ID")  
    private long id;  
    @Column(name="FULLNAME")  
    private String name;  
    ...  
}
```

Person class with  
JPA Annotations

- JPA is supported by many Java persistence frameworks as a standard, portable API



# Hibernate Annotations

- Additional Hibernate specific functionality can be exposed using Hibernate annotations

```
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.Id;  
  
@Entity  
public class Person {  
    @Id  
    @GeneratedValue  
    private long id;  
    @org.hibernate.annotations.AccessType("property")  
    private String name;  
  
    ...  
}
```

Full org.hibernate.annotations  
package name to clearly  
distinguish from JPA

- Best practice to use full package name for hibernate annotations to clearly distinguish from JPA



# Annotation based Mapping

```
@Entity
@Table(name="MY_PERSON")
public class Person {
    @Id
    @Column(name="PERSON_ID")
    private long id;
    @Column(name="FULLNAME")
    private String name;

    public Person() {}
    public Person(String name) { this.name = name; }

    public long getId() { return id; }
    private void setId(long id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

Map class to 'MY\_PERSON' table

Map to PERSON\_ID column

Map to FULLNAME column

Table: MY\_PERSON

PERSON_ID	FULLNAME
1	Frank Brown
2	John Smith

Defaults to 'Person' table

```
@Entity
public class Person {
    @Id
    private long id;
    private String name;
    ...
}
```

Defaults to 'id' column  
name (same as property)

No annotation needed, persisted  
to the 'name' column by default

Table: PERSON

ID	NAME
1	Frank Brown
2	John Smith



# Hibernate XML Mapping

- Person.hbm.xml
  - Usually in the same package as Person.java

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0/"
"http://hibernate.sourceforge.net/hibern
```

```
<hibernate-mapping>
  <class name="intro.Person" table="MY_PERSON">
    <id name="id" column="PERSON_ID" />
    <property name="name" column="FULLNAME" />
  </class>
</hibernate-mapping>
```

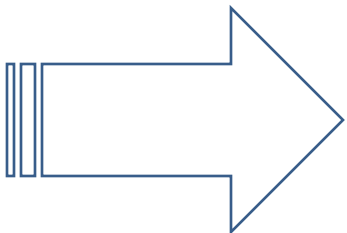
<class> tag to map Person to the 'MY\_PERSON' table

<id> tag to map the primary key

<property> tag to map normal properties

Table: MY\_PERSON

PERSON_ID	FULLNAME
1	Frank Brown
2	John Smith



```
<hibernate-mapping>
  <class name="intro.Person">
    <id name="id" />
    <property name="name" />
  </class>
</hibernate-mapping>
```

Unlike annotations, you have to specify all properties!

Table: PERSON

ID	NAME
1	Frank Brown
2	John Smith



Entity Class Mapping:

# MAPPING IDENTITY





# Primary key

- A primary key is
  - Unique
    - No duplicate values
  - Constant
    - Value never changes
  - Required
    - Value can never be null
  
- Primary key types:
  - Natural key
    - Has a meaning in the business domain
  - Surrogate key
    - Has no meaning in the business domain
    - Best practice





# Mapping Primary Keys

- Object / Relational mismatch
  - Hibernate requires you to specify the property that will map to the primary key
- Prefer surrogate keys
  - Natural keys often lead to a brittle schema

```
@Entity
public class Person {
    @Id
    private String name;

    ...
}
```

Name as a natural primary key for Person can give problems

```
@Entity
public class Person {
    @Id
    private long id;
    private String name;

    ...
}
```

Instead use id as a surrogate key for Person



# Generating Identity

- Generated identity values
  - Ensure identity uniqueness
- Private setId() methods
  - Ensure identity immutability

```
@Entity
public class Person {
    @Id
    @GeneratedValue
    private long id;
    private String name;

    public Person() {}
    public Person(String name) { this.name = name; }

    public long getId() { return id; }
    private void setId(long id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

Id is generated

Id can not be set by  
the application



# Generation Strategies

Hibernate	JPA	Description
native	AUTO	Selects the best strategy for your database
identity	IDENTITY	Use an identity column (MS SQL, MySQL, HSQL, ...)
sequence	SEQUENCE	Use a sequence (Oracle, PostgreSQL, SAP DB, ...)
-	TABLE	Uses a table to hold last generated values for PKs
hilo	-	Like table, uses efficient algorithm to generate values
seqhilo	-	Like regular hilo, but using a sequence
increment	-	Finds the current max value and increments by 1*
uuid.hex	-	Creates globally unique 128-bit UUIDs
guid	-	Uses MySQL and MS SQL server global uid generator
select	-	Have Hibernate select a value generated by a trigger
assigned	(implicit)	Specifies that the value is assigned by the application



# Specifying Identity Generation

## ■ @GeneratedValue

```
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private long id;
    private String name;
```

Specify the generation strategy

```
@Entity
public class Person {
    @Id
    @GeneratedValue
    private long id;
    private String name;
```

Defaults to 'AUTO' when not specified

## ■ <generator> tag

```
<hibernate-mapping>
  <class name="identity.Person">
    <id name="id">
      <generator class="native" />
    </id>
    <property name="name" />
  </class>
</hibernate-mapping>
```

Always have to specify a strategy, cannot leave it off



# Identity Column

- Identity columns are columns that can automatically generate the next unique id

```
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long id;
    private String name;
}
```

JPA identity strategy

```
<hibernate-mapping>
  <class name="identity.Person">
    <id name="id">
      <generator class="identity" />
    </id>
    <property name="name" />
  </class>
</hibernate-mapping>
```

XML identity strategy

- If your database support identity columns the native strategy will default to using them



# Sequence

- The SEQUENCE strategy will default to using the 'hibernate\_sequence' for all tables
- The AUTO strategy will do so too if sequence is the default strategy (Oracle, PostgreSQL)



```
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE)
    private long id;
    private String name;
```


Both AUTO and SEQUENCE

```
<hibernate-mapping>
  <class name="identity.Person">
    <id name="id">
      <generator class="sequence" />
    </id>
    <property name="name" />
  </class>
</hibernate-mapping>
```

Both native and sequence



# Sequences

- By default Hibernate only uses a single sequence called 'hibernate-sequence' 
- You can specify additional custom sequences



*Tip:* You can use the native strategy and still also specify a custom sequence





# Using Custom Sequences

Create Custom Sequence

```
@Entity
@SequenceGenerator(name="personSeq", sequenceName="PERSON_SEQUENCE")
public class Person_annotated_sequence {
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="personSeq")
    private long id;

    ...
}
```

Use Custom Sequence

```
<hibernate-mapping>
  <class name="identity.Person">
    <id name="id" >
      <generator class="sequence">
        <param name="sequence">PERSON_SEQUENCE</param>
      </generator>
    </id>
    <property name="name" />
  </class>
</hibernate-mapping>
```

Use 'sequence' strategy

Specify the custom sequence



Entity Class Mapping

# MAPPING DATA TYPES



# Data Types

- So far we haven't specified any data types
  - Hibernate automatically mapped to SQL types
- You can be very specific
  - Specify the data type, length, precision and more

```
@Entity
public class Person {
    @Id
    @GeneratedValue
    private long id;
    @Column(name="FULLNAME", length=255, nullable=false)
    private String name;
    ...
}
```

Name will be stored as:  
FULLNAME VARCHAR(255) NOT NULL



# Specify Date Type

- Sometimes you need to be specific
  - Should a `java.util.Date` be stored as date, as a time or as a timestamp in the database?

```
@Entity
public class Person {
    @Id
    @GeneratedValue
    private long id;
    private String name;
    @Temporal(TemporalType.DATE)
    private Date birthday;
    ...
}
```

Birthday will be  
stored as a Date

- If left unspecified a `java.util.Date` will be stored as a timestamp to preserve accuracy



# Annotation Types

---

- Reflection is used to find java type information
- Use @Column to specify more details
- Use @Temporal to specify how a Date should be persisted (DATE, TIME or TIMESTAMP)
- Use @Lob to indicate Large values
- Use @Transient to indicate that a property should ***not*** be persisted



# All in One Example

```
@Entity
public class Person {
    @Id
    @GeneratedValue
    private long id;
    @Column(name="FULLNAME", length=255, nullable=false)
    private String name;
    @Temporal(TemporalType.DATE)
    private Date birthday;
    @Lob
    private String biography;
    @Transient
    private String temp;

    ...
}
```

Name will be stored as:  
FULLNAME VARCHAR(255) NOT NULL

Birthday will be  
stored as a Date

Biography will be stored as CLOB  
instead of VARCHAR

Temp will not be stored in the database



# Basic Hibernate XML Types

Hibernate Type	Java Type	SQL Type
byte	byte or java.lang.Byte	TINYINT
short	short or java.lang.Short	SMALLINT
integer	int or java.lang.Integer	INTEGER
long	long or java.lang.Long	BIGINT
float	float or java.lang.Float	FLOAT
double	double or java.lang.Double	DOUBLE
big_decimal	java.math.BigDecimal	NUMERIC
boolean	boolean	BIT
yes_no	boolean	CHAR(1) – ‘Y’ or ‘N’
true_false	boolean	CHAR(1) – ‘T’ or ‘F’
string	java.lang.String	VARCHAR
character	char, java.lang.Character or java.lang.String	CHAR(1)



# Date Time & Large Values

Hibernate Type	Java Type	SQL Type
date	java.util.Date or java.sql.Date	DATE
time	java.util.Date or java.sql.Time	TIME
timestamp	java.util.Date or java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE

Hibernate Type	Java Type	SQL Type
binary	byte[] or java.lang.Byte[]	VARBINARY
text	java.lang.String	CLOB
clob	java.sql.Clob	CLOB
blob	Java.sql.Blob	BLOB
serializable	java.io.serializable	VARBINARY





# XML Type Example

```
public class Person {  
    private long id;  
    private String name;  
    private Date birthday;  
    private String biography;  
    private String temp;  
  
    ...  
}
```

Same class and properties as the previous annotation example

Now without annotations

```
<hibernate-mapping>  
  <class name="model.Person" table="MY_PERSON">  
    <id name="id" type="long" column="PERSON_ID">  
      <generator class="native" />  
    </id>  
    <property name="name" column="FULLNAME"  
      type="string" length="255" not-null="true" />  
    <property name="birthday" column="BIRTHDAY" type="date" />  
    <property name="biography" type="text" />  
  </class>  
</hibernate-mapping>
```

Name will be stored as:  
FULLNAME VARCHAR(255) NOT NULL

Birthday will be  
stored as a Date

Biography will be stored as CLOB  
instead of VARCHAR

Temp is not listed and will not  
be stored in the database



Entity Class Mapping

# **SPECIFYING ACCESS TYPE**



# Property or Field Access

- Hibernate can access objects in two ways
  - property access gets and sets object values through getter /setter methods
  - field access gets and sets object values directly from / to the fields

```
@Entity
public class Person {
    @Id
    @GeneratedValue
    private long id;
    private String name;
    ...
}
```

JPA field access

```
@Entity
public class Person {
    private long id;
    private String name;

    public Person() {}
    public Person(String name) { this.name = name; }

    @Id
    @GeneratedValue
    public long getId() { return id; }
    private void setId(long id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

JPA property access



# Specifying Access with Annotations

---

- The JPA specification lets you set the Access Type with the location of `@Id`
  - Placing `@Id` on a field specifies field access for the entire object
    - All other mapping annotations should be on the fields
  - Placing `@Id` on a getter specifies property access for the entire object
    - All other mapping annotations should be on the getters
- Using additional annotations you can also change access for individual fields



# Access Example

```
@Entity
public class Person {
    private long id;
    private String name;
    private Date birthday;

    public Person() {}
    public Person(String name) { this.name = name; }

    @Id
    @GeneratedValue
    public long getId() { return id; }
    private void setId(long id) { this.id = id; }

    @Access(AccessType.FIELD)
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    @Temporal(TemporalType.DATE)
    public Date getBirthday() { return birthday; }
    public void setBirthday(Date birthday) { this.birthday = birthday; }
}
```

All other annotations have to be on getter methods as well

@Id on a getter sets property access for all attributes

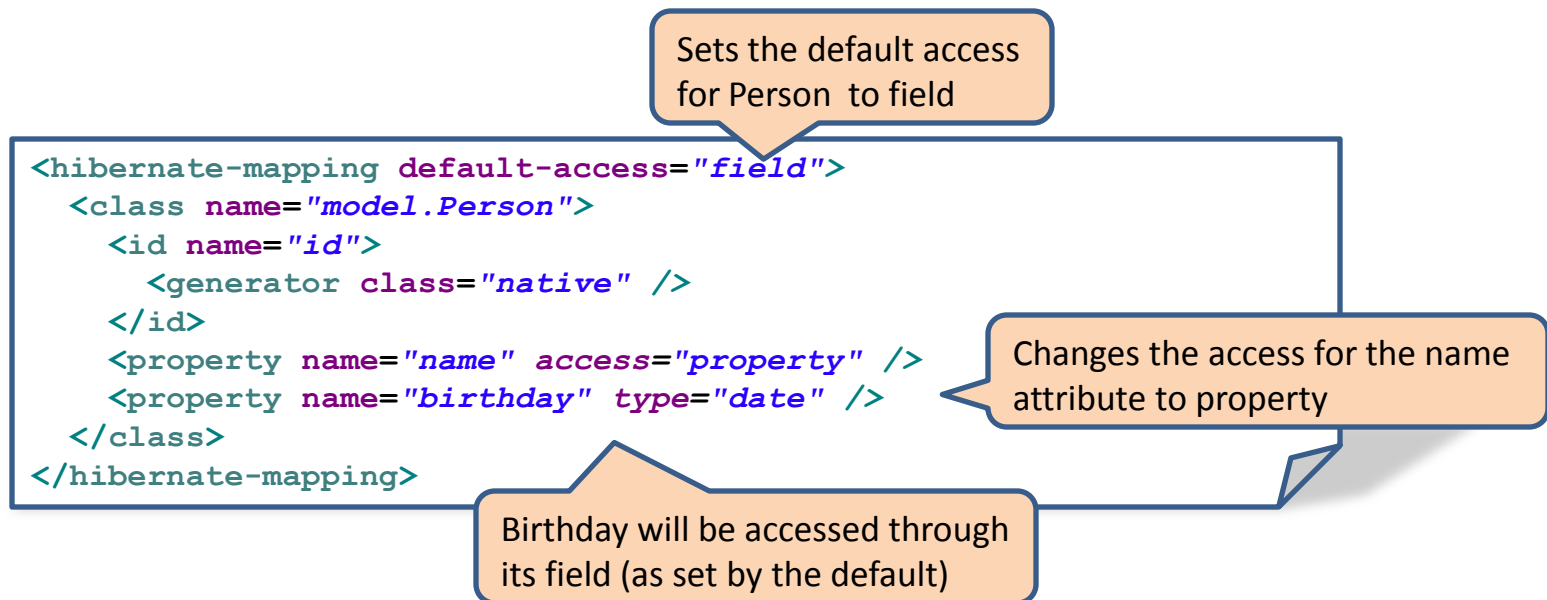
Name will be accessed through the field instead

Birthday is still accessed through getters / setters



# Setting Access with XML

- Use the “default-access” attribute on the <hibernate-mapping> tag to set the default
- Use the “access” attribute on the <property> tag to change access for individual fields





# Access / Encapsulation

- Property access hides implementation details
  - Maintains OO Principle of Encapsulation
- Hibernate dirty checking compares by value
  - With property access your implementation can differ from your relational mapping without disturbing Hibernate



# Encapsulation Example XML

```
public class Person {  
    private long id;  
    private String firstname;  
    private String lastname;
```

3 field implementation  
details are encapsulated

```
    public Person() {}
```

Mapped to 2 column table

```
    public long getId() { return id; }  
    public void setId(long id) { this.id = id; }
```

Table: PERSON

ID	NAME
1	Frank Brown
2	John Smith

Firstname and lastname are  
combined in getter

```
    public String getName() { return firstname + " " + lastname; }  
    public void setName(String name) {  
        StringTokenizer st = new StringTokenizer(name);  
        firstname = st.nextToken();  
        lastname = st.nextToken();  
    }
```

Firstname and lastname are  
separated in setter

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

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

```
}
```

```
<hibernate-mapping default-access="property">  
    <class name="model.Person" >  
        <id name="id">  
            <generator class="native"/>  
        </id>  
        <property name="name" />  
    </class>  
</hibernate-mapping>
```

Property access

maps the 2 fields:  
id and name





# Annotations Encapsulation Example

@Entity

```
public class Person {  
    private long id;  
    private String firstname;  
    private String lastname;
```

Three class properties  
mapped to two columns

Table: PERSON

ID	NAME
1	Frank Brown
2	John Smith

@Id

Property access

@GeneratedValue

```
public long getId() { return id; }  
public void setId(long id) { this.id = id; }
```

@Column

```
public String getName() { return firstname + " " + lastname; }  
public void setName(String name) { /* see code prev slide */; }
```

@Transient

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

@Transient firstname  
and lastname

@Transient

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

```
}
```



Entity Class Mapping

# **WRAPPING UP**



# Mapping Tips

- You can specify a package attribute on the `<hibernate-mapping>` XML element
  - No longer need fully qualified class names
  - Useful when mapping associations

Class name Order  
instead of tips.Order

Package specified for this mapping

```
<hibernate-mapping package="cs544">
  <class name="Order" table="`order`">
    <id name="id" column="`id`"/>
    <many-to-one name="customer" class="Person" />
  </class>
</hibernate-mapping>
```

Person instead of cs544.Person



# Active Learning

---

- What do the following annotations do:  
@Temporal @Transient
  
  
  
  
  
  
  
  
  
  
- How does property access work, how does field access work?



# Module Summary

---

- In this module we covered entity class mapping including:
  - Mapping an entity classes to tables
  - Mapping properties to columns
  - Mapping Identity and setting Identity Generation
  - Mapping Data types
  - Property / Field Access
- These are the tools you use to map any class