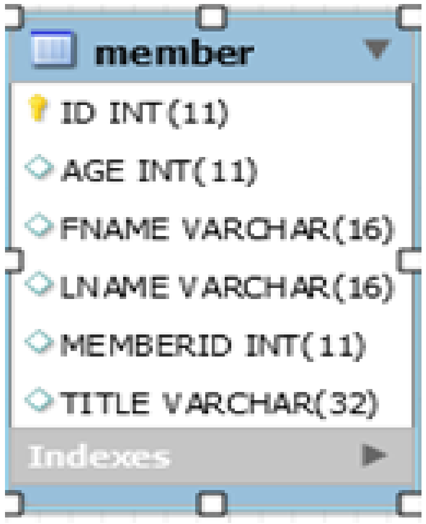
BASIC ORM MAPPING

How To Get

From Here

```
public class Member {
    private int id;
    private String firstName;
    private String lastName;
    private int age;
    private String title;
    private int memberNumber;
}
```

To Here



Basic Example

- The following example is similar to what we will use in the labs
  - Not very real world
    - not part of a big N-Tier application

- Simple console app
  - Easy to understand
  - Easy to test / practice with individual features
  - You should never write real Hibernate code like this!

```
package cs544.hibernate01.basic;

import java.util.List;

import javax.persistence.EntityManager;  import
javax.persistence.EntityManagerFactory;  import
javax.persistence.Persistence;
import javax.persistence.TypedQuery;

public class App {
        private static EntityManagerFactory emf;

        public static void main(String[] args) throws Exception {
                /* Reads persistence.xml and looks for specified unit name */
                emf = Persistence.createEntityManagerFactory("cs544.01.basic");

                EntityManager em = emf.createEntityManager();
                em.getTransaction().begin();
                Customer c = new Customer("Jack", "Welsh");
                em.persist(c);  em.getTransaction().commit();


                em.getTransaction().begin();
                TypedQuery<Customer> q = em.createQuery("from Customer", Customer.class);
                List<Customer> customers = q.getResultList();
                for (Customer c2 : customers) {
                        System.out.println(c2.getFirstName() + " " + c2.getLastName());
                }
                em.getTransaction().commit();

                emf.close();
        }
}
```

```
package cs544.hibernate01.basic;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Customer {
        @Id
        @GeneratedValue
        private Long id;
        private String firstName;
        private String lastName;

        ...

}
```

Customer Table

id	firstName	lastName
1	Jack	Welsh

```
Jack Welsh
```

Persistence.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
             http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
             version="2.1">

    <persistence-unit name="cs544.01.basic">
        <description>
            Persistence unit for Hibernate
        </description>

        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <properties>
            <property name="packagesToScan" value="cs544.hibernate01.basic" />
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/cs544?useSSL=false"/>
            <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
            <property name="javax.persistence.jdbc.user" value="root"/>
            <property name="javax.persistence.jdbc.password" value="root"/>
            <property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5Dialect" />
            <property name="hibernate.cache.provider_class" value="org.hibernate.cache.NoCacheProvider"/>
            <property name="hibernate.id.new_generator_mappings" value="false" />
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
        </properties>
  </persistence-unit>
</persistence>
```

Hibernate Framework

- Framework Just like Spring
  - Also creates your objects (IOC fancy factory)
  - Connects them together (DI)
  - Adds additional functionality (AOP/interceptor proxies)
- Unlike Spring these framework details are not always obvious – but definitely still there!
  - What Spring calls ApplicationContext
  - JPA calls: EntityManager (Hibernate used to call: Session)

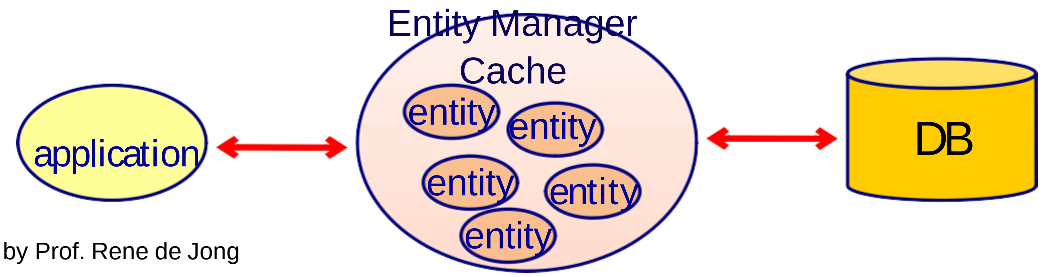
Entities

- Domain objects are called Entities
  - Hibernate manages (creates, injects, proxies) them
  - That's why it's called the EntityManager

```
@Entity(name = "MEMBER")
public class Member {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="ID")
    private long id;

    @Column(name="FNAME", length = 16)
    private String firstName;
    @Column(name="LNAME", length = 16)
    private String lastName;
    @Column(name="AGE")
    private int age;
}
```

LifeTime

- An EntityManager usually exists for the short time span of a (web) request
- During this time it keeps a cache of all the Objects it has retrieved from the DB
  - Ask for same object many times → one DB access
  - Objects usually don't go stale (EM exists briefly)



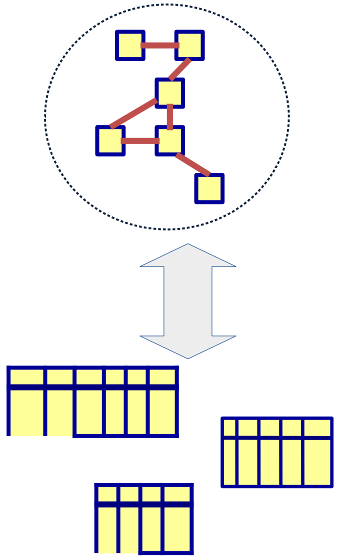
Model by Prof. Rene de Jong

EntityManagerFactory

- To create a new EntityManager for every request Hibernate provides a factory
  - Created once on startup (singleton)
  - Reads all the mappings
  - Thread safe methods

Entity

- An entity is a Domain Class
  - The most basic part of the domain
- Classes map to Tables in the DB
  - There is almost no mis-match here
  - They just need to specify an ID field



Entity Class

- JPA Requires that entity classes have:
  - A field to use as ID
  - A default constructor
  - Getters and setters for all properties

Java Bean

```
@Entity(name = "MEMBER")
public class Member {

@Id
@GeneratedValue(strategy=GenerationTy
pe.AUTO)
   private long id;

   @Column(name="FNAME", length = 16)
   private String firstName;


   public Member(){}

   public long getId() {
      return id;
   }
   public void setId(long id) {
      this.id = id;
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName(String
firstName) {
      this.firstName = firstName;
   }

 }
```

@Entity

- @Entity specifies that a class is an entity.
  - By default the class name = entity name
  - This entity name is used in Queries

“from MEMBER”

- @Entity(name=“OtherName”)
  - Gives the entity a different name
  - Also causes the table name to change
    - When generating tables from annotations

Optional @Table

- @Table(name=”OtherName”)
  - Changes table name that entity is mapped to
  - Without changing the name of the Entity

- @Table also has options for:
  - Mapping to a different schema (db)
  - Specifying unique constraints (if generating schema)
  - Specifying indexes (if generating schema)

Mapping Primary Keys

- Object / Relational mismatch
  - JPA requires you to specify the property that will map to the primary key (best non-primitive)
- Prefer surrogate keys
  - Natural keys often lead to a brittle schema

```
@Entity(name = "user")
public class UserCredentials {

    @Id
    private String username;
    ..
}
```

Natural key "name" can give problems

```
@Entity(name = "MEMBER")
public class Member {

    @Id
    private long id;
    …
}
```

Instead use "id" as surrogate key

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

Generating Identity

- The DB can generate surrogate key values
  - Using @GeneratedValue
  - Ensuring identity uniqueness
  - No meaning in business anyway

```
@Entity(name = "MEMBER")
public class Member {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="ID")
    private long id;
}
```

Generation Strategies

- On optional strategy argument
  - Hibernate will guess the best strategy based on the database if strategy is not specified
- Strategy options are:

Value	Description
AUTO (or not specified)	Selects the best strategy for your database
IDENTITY	Use an identity column (MS SQL, MySQL, HSQL, …)
SEQUENCE	Use a sequence (Oracle, PostgreSQL, SAPDB, …)
TABLE	Uses a table to hold last generated values for PKs
(no annotation)	Specifies that the value is assigned by the application

Identity Column

- Identity columns automatically generate the next ID value
  - Popular Databases: MS-SQL server, MySQL
- Unfortunately recent versions of Hibernate seem to no-longer default to Identity for MySQL
  - See: https://hibernate.atlassian.net/browse/HHH-11014
- To fix this behavior you can add the following to the persistence.xml file:

```
<property name="hibernate.id.new_generator_mappings" value="false" />
```

```
@Entity(name = "MEMBER")
public class Member {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="ID")
    private long id;

    @Column(name="FNAME", length = 16)
    private String firstName;
}
```

Sequence

- A sequence is a separate DB object that provides 'next' values
  - Can be used as identity source by multiple tables
  - Ensuring unique ID column with unique values even when these tables are combined into a single view (or resultset)
- Popular databases that use sequences:
  - Oracle, PostgreSQL

```
@Entity(name = "MEMBER")
public class Member {

    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE)
    @Column(name="ID")
    private long id;

    @Column(name="FNAME", length = 16)
    private String firstName;
}
```

Sequences Names

- Each sequence has its own name
- If you don't specify a sequence name
- Hibernate defaults to "hibernate_sequence"

```
@Entity(name = "MEMBER")
public class Member {

   @Id
   @GeneratedValue(strategy=GenerationType.SEQUENCE)
   @Column(name="ID")
   private long id;

   @Column(name="FNAME", length = 16)
   private String firstName;
}
```

Specifies Sequence but **not <u>which</u> one**!

Specifying a Sequence

- Specify the existance of a sequence
  - Then tell JPA to use that one for generation

Specifies that the MEMBER_SEQUENCE exists in the DB

```
@Entity(name = "MEMBER")
@SequenceGenerator(name="member", sequenceName="MEMBER_SEQUENCE")
public class Member {

    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="member")
    @Column(name="ID")
    private long id;

    @Column(name="FNAME", length = 16)
    private String firstName;
}
```

Indicates that we should use the customer generator

Table

- JPA can use a Table to emulate a Sequence
  - Slow because it requires an additional transaction
  - Sometimes useful on Databases that don't have sequences

```
@Entity(name = "MEMBER")
public class Member {

   @Id
   @GeneratedValue(strategy=GenerationType.Table)
   @Column(name="ID")
   private long id;

   @Column(name="FNAME", length = 16)
   private String firstName;
}
```

Data Types

- JPA has decent defaults for most types
  - Java and SQL data types are not that different
  - Ints become ints, Strings become varchar(255), …
  - You can customize things (length of varchar)
- Not all types always map correctly
  - Specifically date and time related types

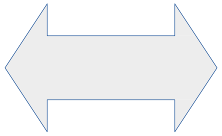
@Basic

- @Basic indicates that a property should be persisted and the default type should be used
  - JPA assumes these are there
    - (you don’t have to add them)
- Also has options for:
  - Indicating that a property is Nullable
  - Indicating if a property should be fetched lazily

Hibernate mostly ignores this it doesn’t make sense from an optimization point of view

Exactly the same

```
package cs544.hibernate01.basic;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;  import
javax.persistence.Id;

@Entity
public class Customer {
        @Id
        @GeneratedValue
        private Integer id;
        @Basic
        private String firstName;
        @Basic
        private String lastName;
        ...

}
```



```
package cs544.hibernate01.basic;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;  import
javax.persistence.Id;

@Entity
public class Customer {
        @Id
        @GeneratedValue
        private Long id;
        private String firstName;
        private String lastName;

        ...

}
```

@Column

- @Column allows us to specify several optional additional values for this column
- Name: column name can differ from property name
- Length: for string valued properties
- Scale and Precision for decimal columns
- Nullable: if the column should be nullable
- Unique: if the column values should be unique
- Table (for secondary tables, discussed later)
- ColumnDefinition: raw DDL to be used for this column

```
@Entity
public class Customer {  @Id
          @GeneratedValue
          private Long id;
          @Column(name="first", length=45, nullable=false)
          private String firstName;
          @Column(name="last", length=60, nullable=true)
          private String lastName;

          ...
}
```

Date and Time

- Date and Time related data-types allways default to the SQL type: TimeStamp
  - Includes: java.util.Date, java.sql.Date, java.util.Calendar
  - But you may not always want it stored as a Timestamp!

- java.time.* officially supported by JPA 2.2

@Temporal

- @Temporal converts date and time values from Java object to compatible database type and retrieving back to the application.
- java.util.Date or java.util.Calendar require @Temporal to map to database types
- Not required when using java.sql.Date or java.sql.Time

```
@Temporal(TemporalType.DATE)              Same as:
 private java.util.Date lastLogin;         java.sql.Date lastLogin;

@Temporal(TemporalType.TIME)              Same as:
 private java.util.Date lastLogin;         java.sql.Time lastLogin;

@Temporal(TemporalType.TIMESTAMP)         Same as:
 private java.util.Date lastLogin;         java.sql.Timestamp lastLogin;

WITHOUT @Temporal                          Same as:
private java.util.Date lastLogin;         java.sql.Timestamp lastLogin;
```

@Transient

- JPA automatically includes all the instance variables of a class
  - Auto-maps them to columns of the same name

- What if you do not want to persist an variable?
  - @Transient specifies that it should not be stored

Large Objects

- Certain things need more space in the DB
  - Images are usually stored as BLOBs
  - Large amounts of text as CLOBs
- JPA offers the @LOB annotation
  - Placed on text related properties makes CLOB
  - Placed on binary related properties makes BLOB

```
@Entity
public class Customer {
        @Id
        @GeneratedValue
        private Long id;

        @Column(name="first", length=45, nullable=false)  private
        String firstName;

        @Column(name="last", length=60, nullable=true)  private
        String lastName;

        @Temporal(TemporalType.TIMESTAMP)
        private Date birthDate;

        @Transient
        private String temp;

        @Lob
        private String biography;

        ...
}
```

Access

- Hibernate can either:
  - Use the getters / setters methods
  - Or use reflection to get/set directly

- Which one it uses depends on where you place your @Id annotation

Field Access

- Examples so far always used field access
  - @Id placed directly on the field
  - Hibernate uses reflection to directly get/set field
  - All other annotations also have to be on the fields

```
@Entity
public class Customer {
        @Id
        @GeneratedValue
        private Long id;
        @Column(name="first", length=45, nullable=false)  private
        String firstName;
        @Column(name="last", length=60, nullable=true)  private
        String lastName;

        ...
}
```

@Id on the field indicates Field Access

Property Access

- Place @Id on a getter for Property Access
  - All other annotations also have to be on the getters

```
@Entity
public class Customer2 {
        private Long id;
        private String firstName;
        private String lastName;
        private Date birthDate;  private
        String temp;  private String
        biography;

        @Id
        @GeneratedValue
        public Long getId() { return id; }
        @Column(name = "first", length = 45, nullable = false)
        public String getFirstName() { return firstName; }
         @Column(name = "last", length = 60, nullable = true)
         public String getLastName() { return lastName; }
        @Temporal(TemporalType.TIMESTAMP)
        public Date getBirthDate() { return birthDate; }
        @Transient
        public String getTemp() { return temp; }
        @Lob
        public String getBiography() { return biography; }
```

@Id on the getter indicates
Property Access
Hibernate will use
methods to get and set

Changing Access

- You can change access for individual fields

```
@Entity
public class Customer {  @Id
        @GeneratedValue
        private Long id;
        @Access(AccessType.PROPERTY)
        private String firstName;  private
        String lastName;


        ...
}
```

Everything will be accessed through field except firstName will use getters / setters

```
@Entity
public class Customer2 {
        private Long id;
        private String firstName;
        private String lastName;

        @Id
        @GeneratedValue
        public Long getId() { return id; }
        @Access(AccessType.FIELD)
        public String getFirstName() { return firstName; }
        public String getLastName() { return lastName; }

        ...
}
```

Everything will be accessed through methods (property) except firstName will use the field directly

Reflection

- Here is an example of how reflection works:

```
package cs544.hibernate01.data;

import java.lang.reflect.Field;  public

class TestReflection {
        public static void main(String[] args)
                throws NoSuchFieldException, SecurityException,
                        IllegalArgumentException, IllegalAccessException {

                Customer c = new Customer("George", "Simpson");
                // Possible NoSuchFieldException
                Field f = c.getClass().getDeclaredField("firstName");
                f.setAccessible(true);
                // Possible IllegalAccessException  String
                fieldData = (String) f.get(c);
                System.out.println(fieldData);
        }
}
```

Encapsulation

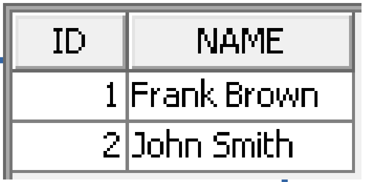
- Using reflection breaks the OO principle of encapsulation
  - Property access hides implementation details
  - Next slide shows an example of hiding the implementation details

Encapsulation (cont.)

```
@Entity
public class Person {
        private Long id;
        private String firstname;
        private String lastname;

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

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

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

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

Encapsulation used to map 3 instance variables to 2 database columns

Table: PERSON



ID	NAME
1	Frank Brown
2	John Smith

XML Mapping

- Like Spring it's also possible to provide all meta-data in XML instead of annotations
  - Same benefits: separate from code, no recompile
  - If both annotation and XML, XML wins
  - Less popular than Spring XML configuration

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-
3.0.dtd">
<hibernate-mapping package="edu.mum.domain">
    <class name="Member" table="MEMBER">
        <id name="id" column="ID">
            <generator class="native"/>
        </id>
        <property name="firstName" column="FIRSTNAME"/>
        <property name="lastName" column="LASTNAME"/>
        <property name="age" column="AGE"/>
        <property name="title" column="TITLE"/>
        <property name="memberNumber"
column="MEMBERNUMBER"/>
    </class>
</hibernate-mapping>
```

⇔

```java
@Entity(name = "MEMBER")
public class Member {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  @Column(name="ID")
  private int id;

  @Column(name="FIRSTNAME")
  private String firstName;
  @Column(name="LASTNAME")
  private String lastName;
  @Column(name="AGE")
  private int age;
  @Column(name="TITLE")
  private String title;
  @Column(name="MEMBERNUMBER")
  private int memberNumber;
```

Main Point

- The mapping of simple object structures to a database is done through configuration files and/or annotations. This simple configuration is enough to instruct the framework about the objects it has to control and store.

- ***Science of Consciousness:*** *The simple mechanics of the TM technique allow [instruct] the mind to transcend to the home [store] of all knowledge.*