面向对象分析与设计
Object-Oriented Analysis and Design

陈昊鹏
chen-hp AT sjtu.edu.cn

Fall-2011

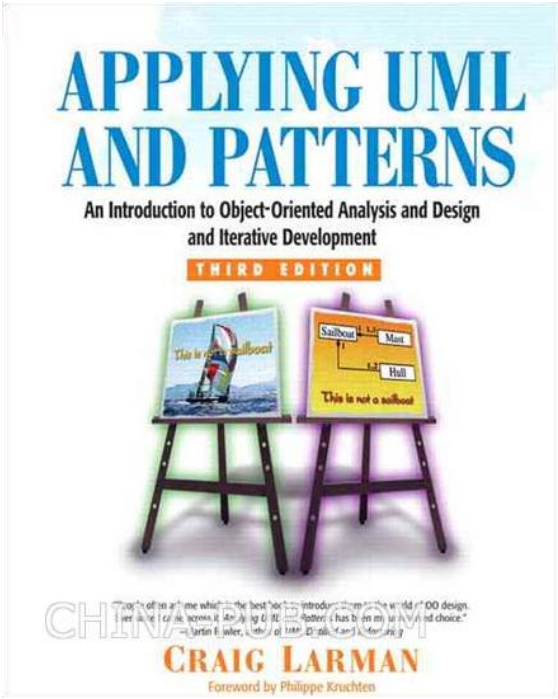
面向对象分析与设计

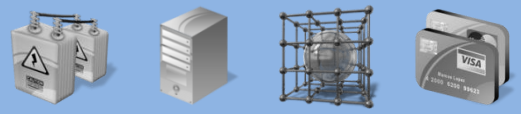
Object-Oriented Analysis and Design

第8章
持久化和数据库设计

Chapter Eight

Persistence Framework and Database Design





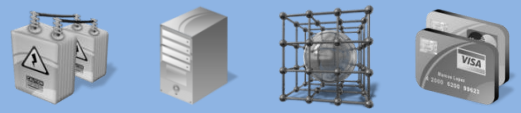
AGENDA

BASIC CONCEPTS

DATABASE DESIGN

PERSISTENCE FRAMEWORKS

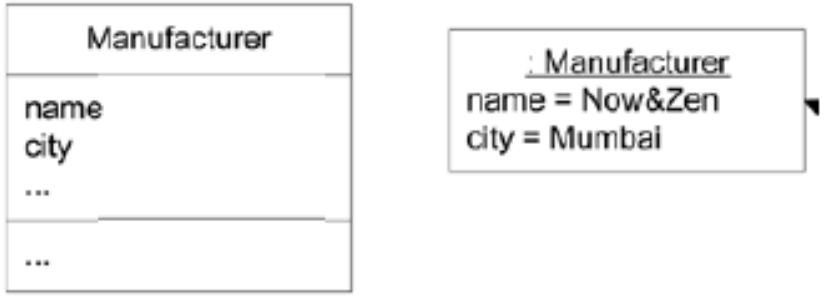
Basic Concepts



- **Persistence Object**
- **Relational Data Model**
- **Object Model**
- **O-R Mapping**
- **UML Profile for Data Modeling**

Persistence Objects

- **Persistent objects are those that require persistent storage**
  - **such as ProductDescription instances.**
- **Storage Mechanisms**
  - **Object databases**
  - **Relational databases**
  - **Other**
    - **flat files**
    - **XML structures**
    - **Palm OS PDB files**
    - **hierarchical databases**
    - **and so on**

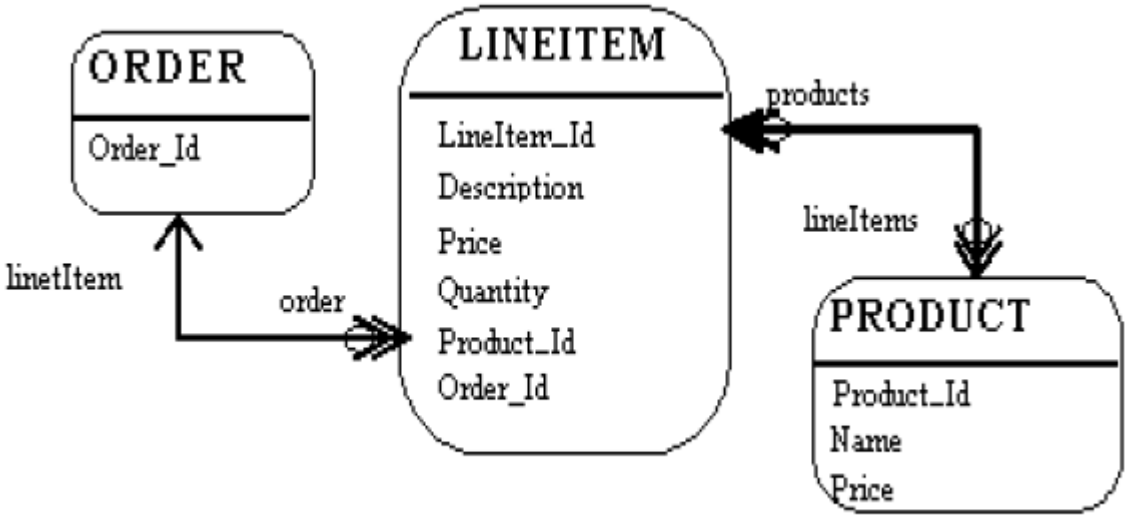


MANUFACTURER TABLE

name	city
Now&Zen	Mumbai
Celestial Shortening	San Ramon

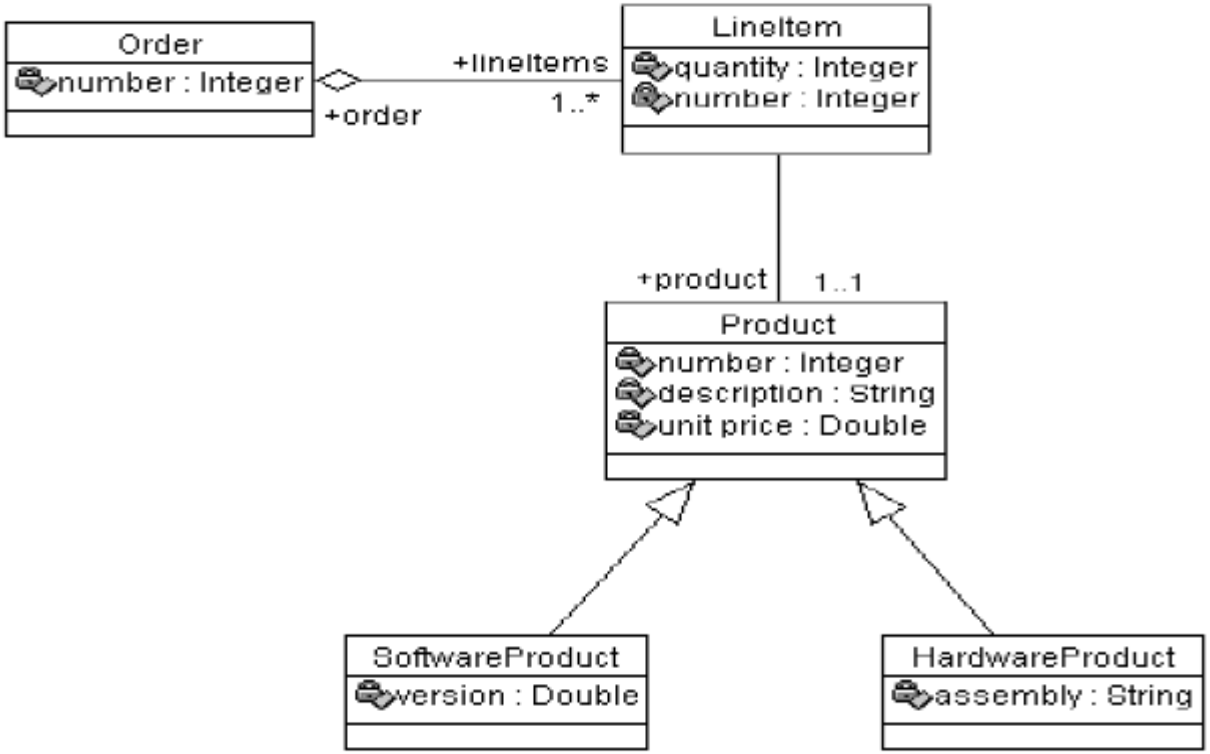
The Relational Data Model

- **The relational model is composed of entities and relations. An entity may be a physical table or a logical projection of several tables also known as a view.**



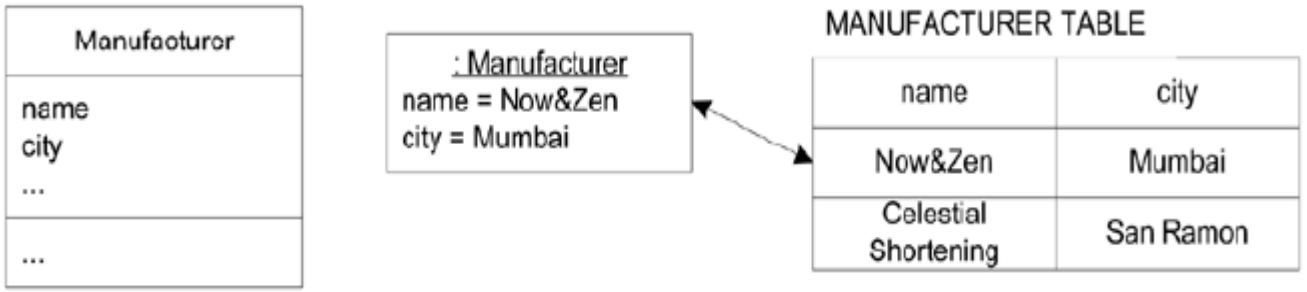
The Object Model

- **An object model contains classes. Classes define the structure and behavior of a set of objects, sometimes called objects instances. The structure is represented as attributes (data values) and associations (relationships between classes).**

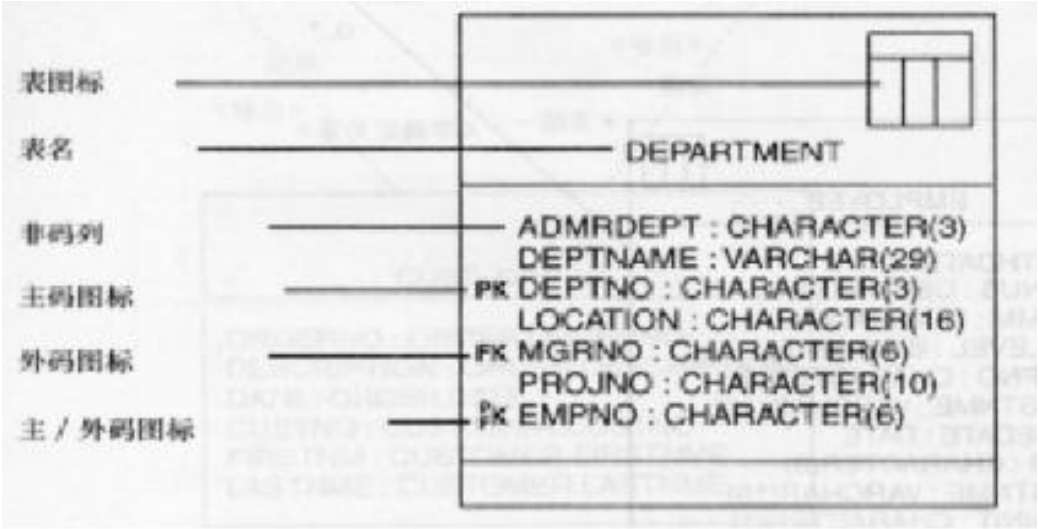


O-R Mapping

- **The Problem:**
  - **As with relational databases, a representation mismatch exists between objects and these non-object-oriented formats.**
- **The Solution:**
  - **O-R Mapping service**
    - **a persistence service**
      - **translate objects into records and save them in a database,**
      - **and translate records into objects when retrieving from a database**



UML Profile for Data Modeling



表(Table)— 数据库中关于同一事物的一组信息，由列组成。

列(Column)— 表的一个组成部分，拥有表的一个单一属性。

主码(Primary key)— 被选择来标识表的一行的候选码。

外码(Foreign key)— 表中映射到另一张表的主键的一列或者一组列。

确定关系(Identifying relationship)— 两张表之间的关系，子表必须与父表共存。

不确定关系(Non-identifying relationship)—两张表之间的关系，每张表都独立存在。

视图(View)— 从用户角度看到的一个虚表，具有典型的表的行为，但是不能独立存在。

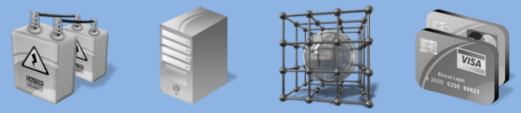
存储过程(Stored procedure)——一个独立的典型地在服务器上执行的函数过程。

域(Domains)——属性或列的一组有效赋值。

AGENDA

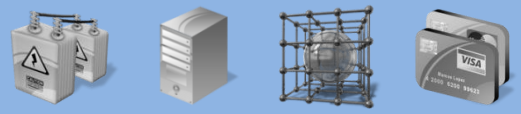
BASIC CONCEPTS
DATABASE DESIGN
PERSISTENCE FRAMEWORKS

Database Design



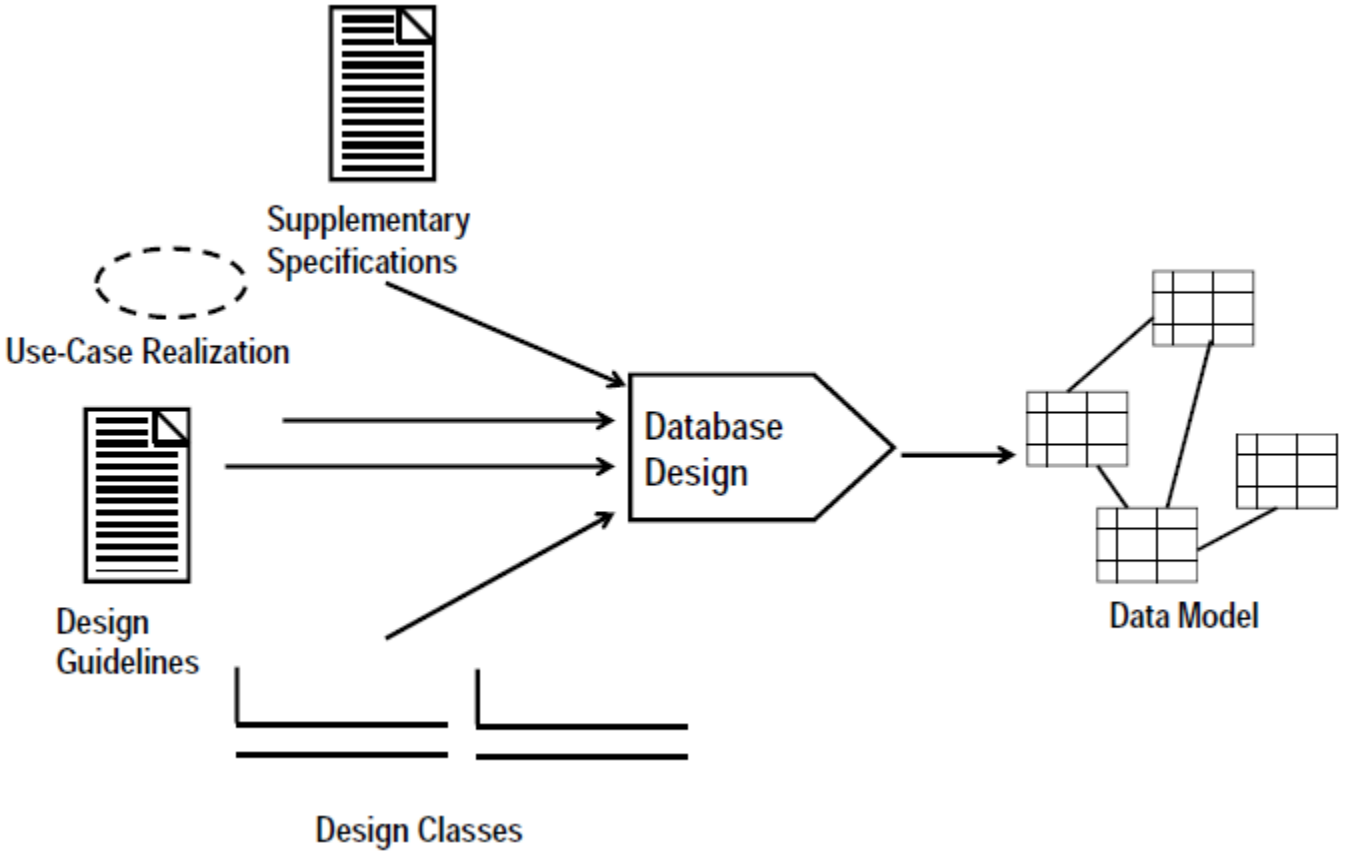
- **Purpose**
- **Overview**
- **Steps**
- **Map persistent design classes to tables**
- **Map associations to tables**
- **Map aggregation to tables**
- **Map inheritance to tables**
- **Distribute class behavior to the database**

Purpose

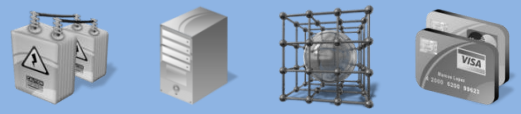


- **To ensure that persistent data is stored consistently and efficiently.**
- **To define behavior that must be implemented in the database.**

Database Design Overview



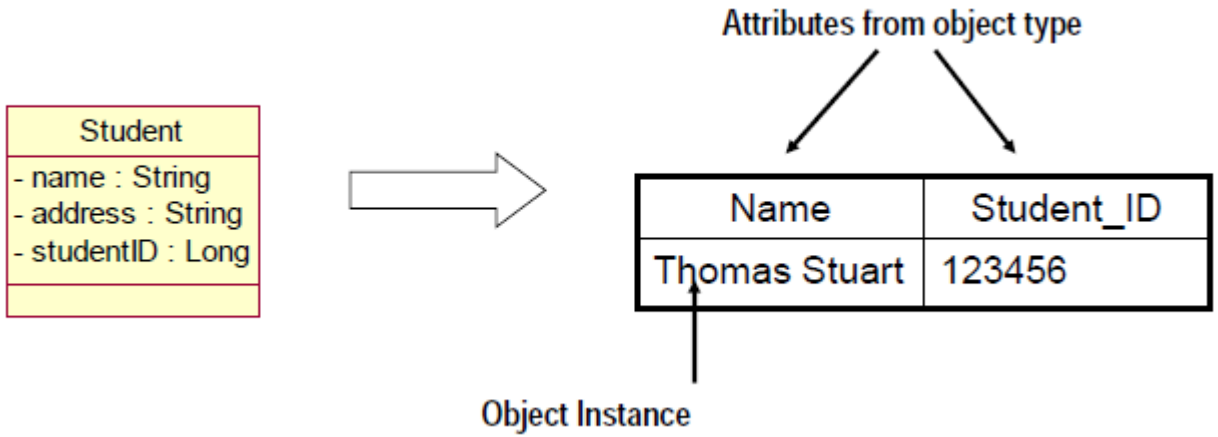
Steps



- Map Persistent Design Classes to the Data Model
- Optimize the Data Model for Performance
- Optimize Data Access
- Define Storage Characteristics
- Define Reference Tables
- Define Data and Referential Integrity Enforcement Rules
- Distribute Class Behavior to the Database
- Review the Results

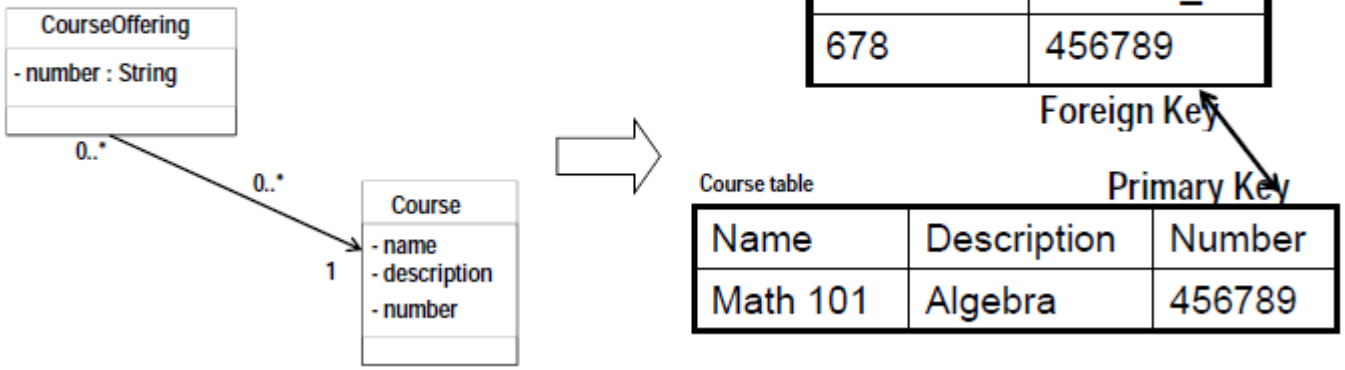
Mapping Persistent Classes to Tables

- **In a relational database**
  - **Every row is regarded as an object**
  - **A column in a table is equivalent to a persistent attribute of a class**

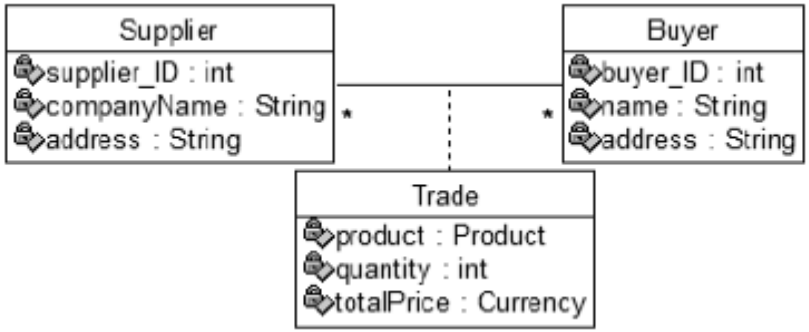


Modeling Associations in the Data Model

- **Associations between two persistent objects are realized as foreign keys to the associated objects.**
  - **A foreign key is a column in one table which contains the primary key value of associated object**



Associations: many to many



- 问题
  - 一次交易由一对卖家和买家唯一决定
  - 如果一对卖家和买家做了多次交易怎么办？

Primary Key — Supplier table

supplier_ID	companyName	address
11	ABCD	888 3rd Ave

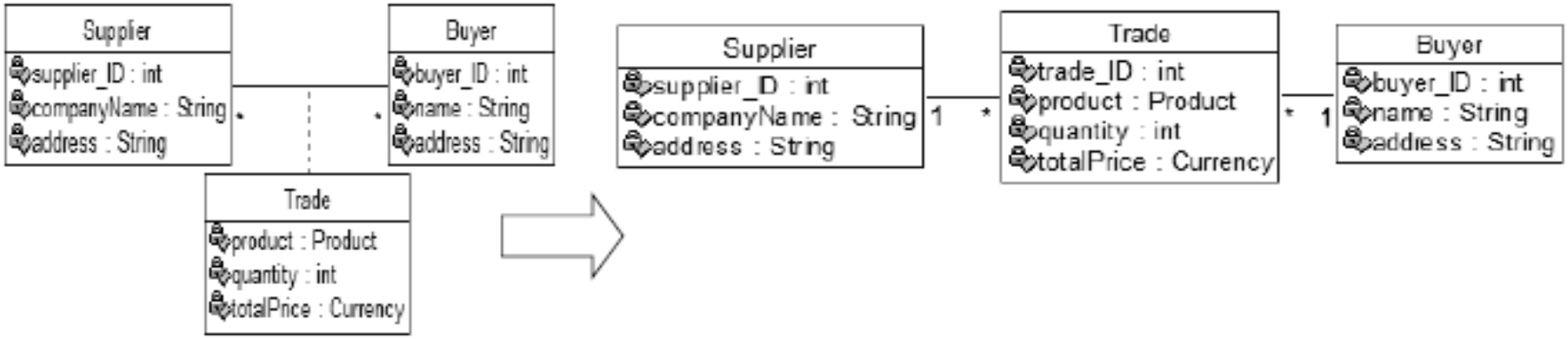
Primary Key — Buyer table

buyer_ID	Name	address
101	John	9 Center St.

Primary Key (supplier_ID, buyer_ID) — Trade table

supplier_ID	buyer_ID	product	quantity	totalPrice
11	101	毛巾	12	￥120

Associations: many to many



Primary Key

Supplier table

supplier_ID	companyName	address
11	ABCD	888 3rd Ave

Primary Key

Buyer table

buyer_ID	Name	address
101	John	9 Center St.

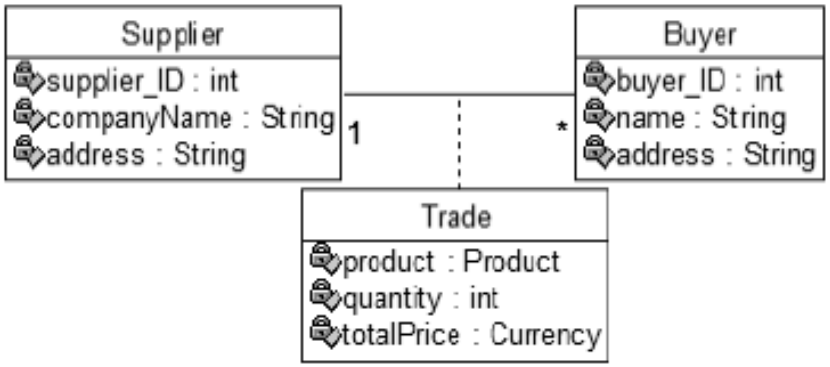
Primary Key : trade_ID

Foreign Key: (supplier_ID, buyer_ID)

trade_ID	supplier_ID	buyer_ID	product	quantity	totalPrice
1	11	101	毛巾	12	￥120
2	11	101	电池	4	￥40

Trade table

Association: one to many



Primary Key

Supplier table

supplier_ID	companyName	address
11	ABCD	888 3rd Ave

Primary Key

Foreign Key

Buyer table

buyer_ID	name	address	supplier_ID	product	quantity	totalPrice
101	John	9 Center St.	11	毛巾	12	￥120

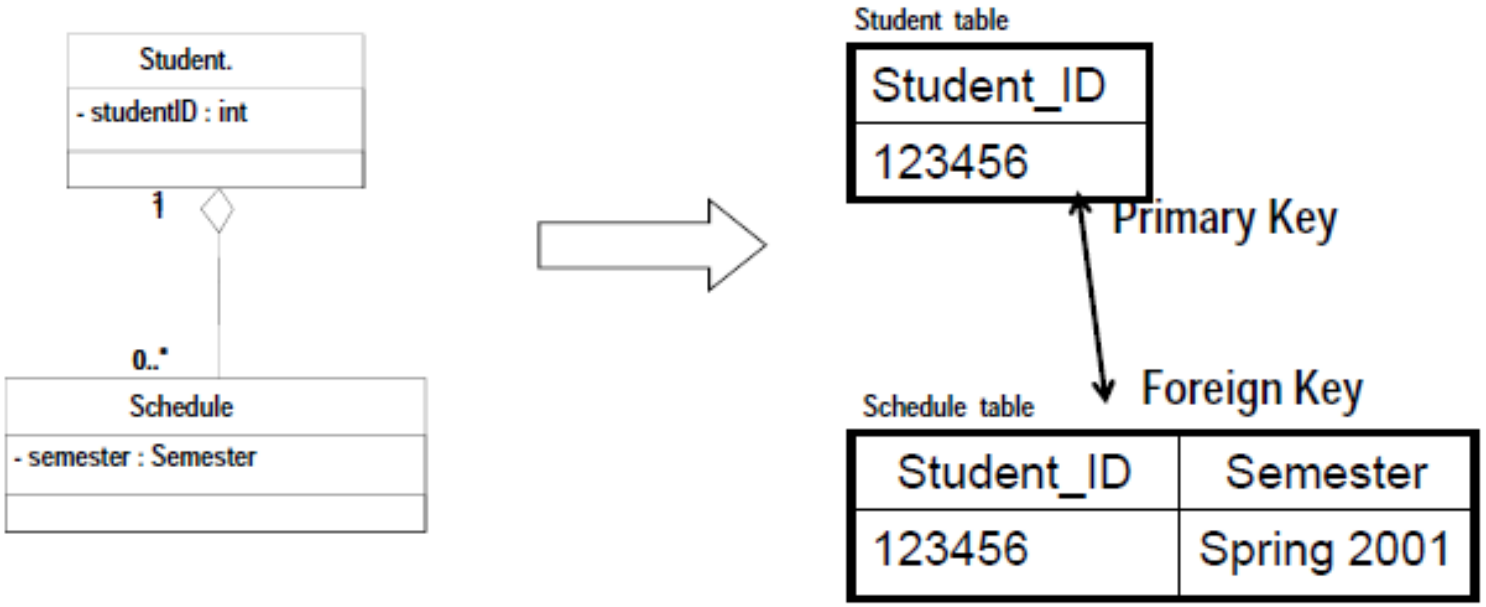
- 外键嵌入

(foreign key embedding)

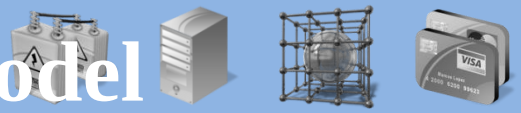
  - 优点
    - 减少一个表，性能提高
  - 缺点
    - 虽然没违反第三范式
    - 但背离“一类一表”的映射原则

Modeling Aggregation in the Data Model

- **Aggregation is also modeled using foreign key relationships**
  - **Using composition implements a cascading delete constraint**



Modeling Inheritance in the Data Model



- **A data model does not support modeling inheritance in a direct way**
- **Three Strategies**
  - **table per subclass**
  - **table per concrete class**
  - **table per class hierarchy**

Modeling Inheritance in the Data Model - Example

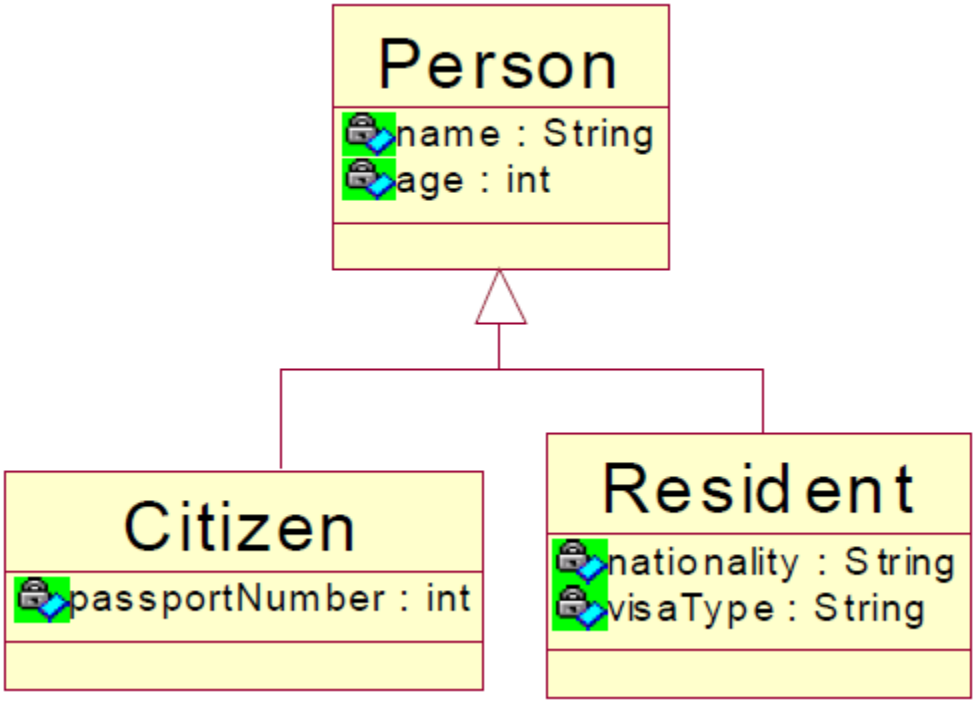
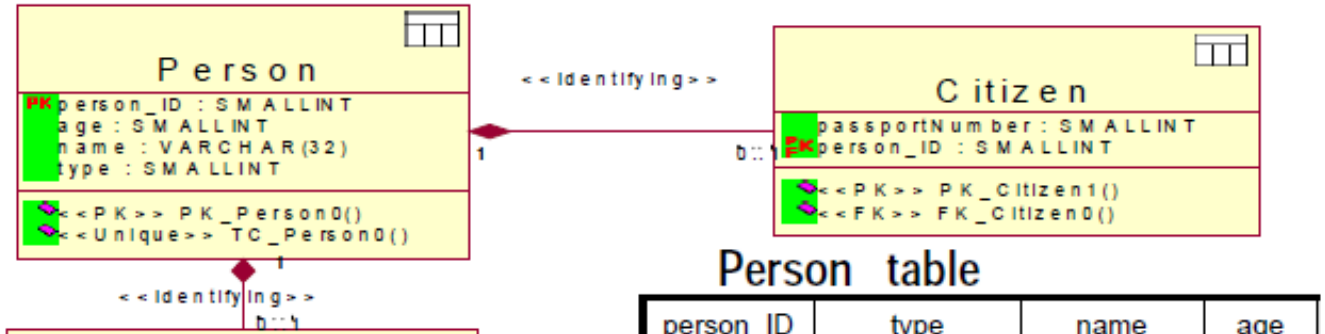


Table per subclass



Person table

person_ID	type	name	age
1	citizen	John	19
2	resident	Mike	73
3	person	Julie	20

Citizen table

person_ID	passportNumber
1	123-456

Resident table

person_ID	nationality	visaType
2	China	H-1

- 查询：找出age>=65 .and.visatype="H-1"的居民（resident）
  - 3步：1）先在Person表中找age>=65的居民，结果放临时表中；
  - 2）对临时表每行，由主键person_ID在Resident表中找到其visaType，如果不是“H-1”,删掉该行；
  - 3）临时表中剩下的行，就是结果

Table per concrete class

- 查询：找出age>=65 .and.visatype="H-1"的居民（resident） 1步即可

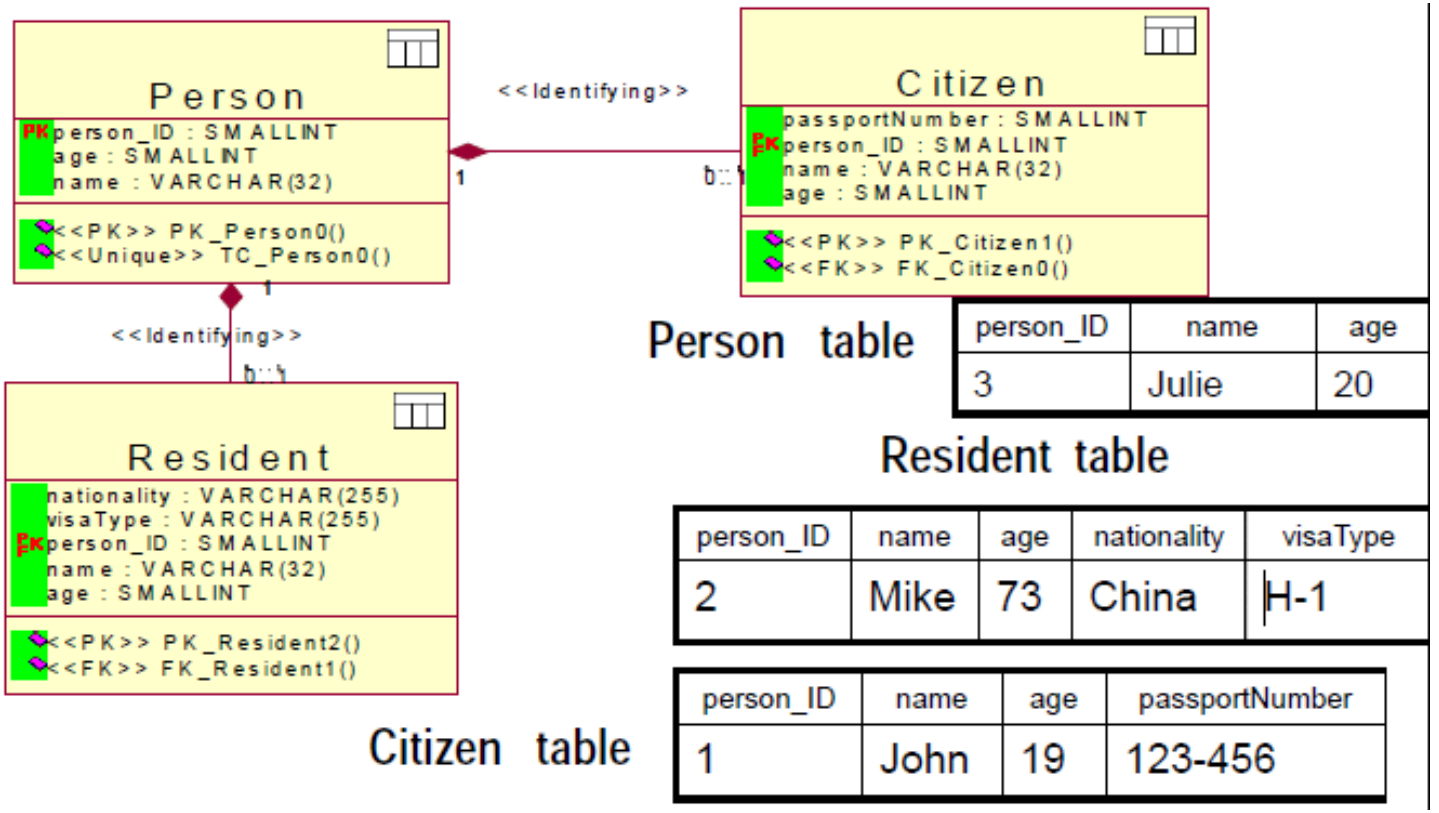
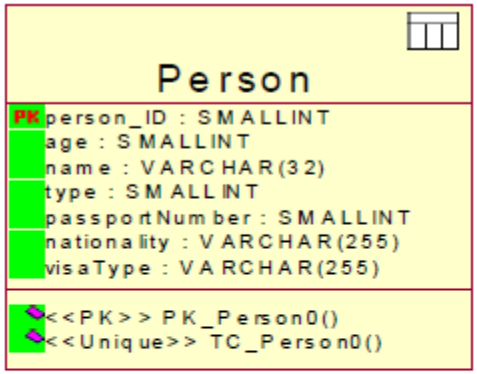


Table per class hierarchy (using discriminator 使用辨别标志)

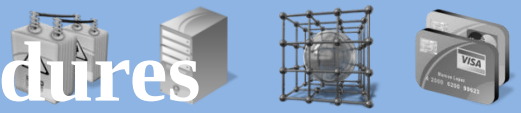


Person table

person_ID	type	name	age	passportNumber	nationality	visaType
1	citizen	John	19	123-456		
2	resident	Mike	73		China	H-1
3	person	Julie	20			

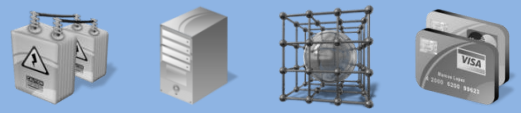
- 查询：找出**age>=65 .and.visatype="H-1"**的居民（**resident**） **1**步即可
- 优点：性能全面优化
- 缺点：违背第三范式

Map Class Behavior to Stored Procedures



- **Determine if any operations can be implemented as a stored procedure**
- **Candidate operations**
  - **Deal with persistent data**
  - **Any operations where a query is involved in a computation**
  - **Need to access the database to validate data**

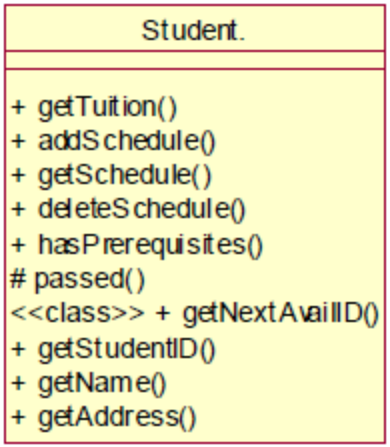
What Are Stored Procedures?



- **A stored procedure is executable code which runs under the RDBMS**
- **Two types of stored procedures**
  - **Procedures: Executed explicitly by an application**
  - **Triggers: Invoked implicitly when some database event occurs**

Example: Map Class Behavior to Stored Procedures

Class



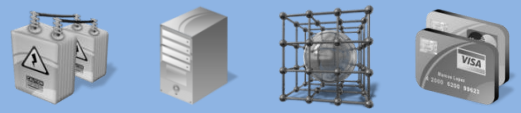
Candidate Operations

- getTuition
- addSchedule
- getSchedule
- deleteSchedule
- getStudentID
- getName
- getAddress

AGENDA

BASIC CONCEPTS
DATABASE DESIGN
PERSISTENCE FRAMEWORKS

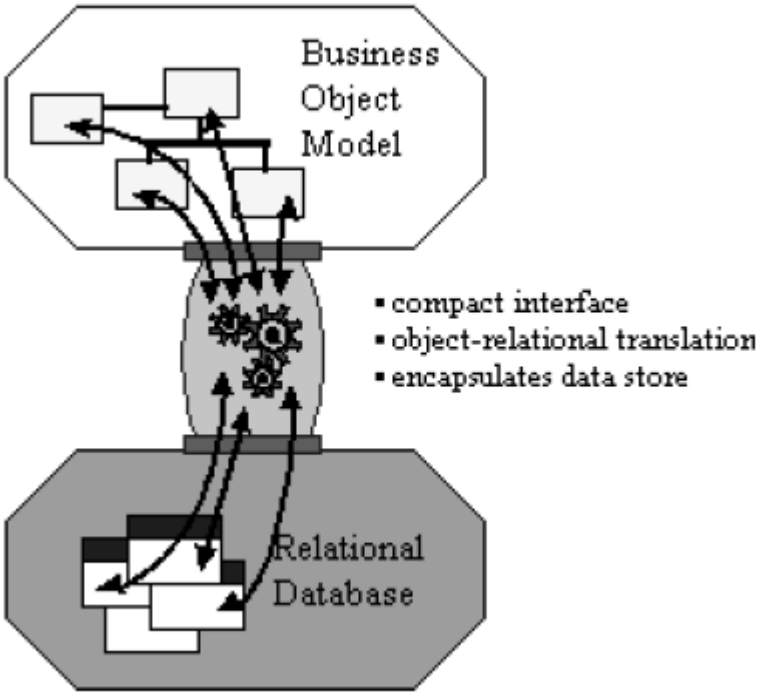
Persistence Frameworks



- **Persistence Framework**
- **Hibernate**

Persistence Framework

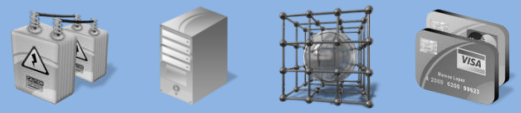
- **A persistence framework**
  - **is a general-purpose, reusable, and extendable set of types that provides functionality to support persistent objects**



Requirements for the Persistence Framework

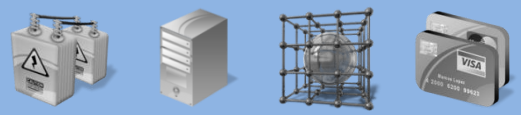
- **The framework should provide functions such as:**
  - **store and retrieve objects in a persistent storage mechanism**
  - **commit and rollback transactions**
- **key ideas:**
  - **Mapping**
  - **Object identity**
  - **Database mapper**
  - **Materialization and dematerialization**
  - **Caches**
  - **Transaction**
  - **Lazy materialization**
  - **Virtual proxies**

Hibernate Overview



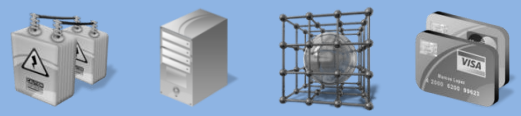
- **O/R mapping**
  - **Basic O/R Mapping**
  - **Working with Objects**
  - **Transactions and Concurrency**
  - **Improving Performance**

O/R Mapping



- **Working with both Object-Oriented software and Relational Databases can be cumbersome and time consuming.**
- **Development costs are significantly higher due to a paradigm mismatch between how data is represented in objects versus relational databases.**
- **Hibernate is an Object/Relational Mapping solution for Java environments.**
- **The term Object/Relational Mapping refers to the technique of mapping data from an object model representation to a relational data model representation (and visa versa)**

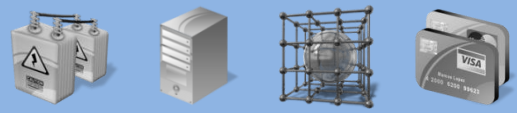
Hibernate – first class



- **Event.class**

```
package org.hibernate.tutorial.domain;
import java.util.Date;
public class Event {
 private Long id;
 private String title;
 private Date date;
 public Event() {}
 public Long getId() { return id; }
 private void setId(Long id) { this.id = id; }
 public Date getDate() { return date; }
 public void setDate(Date date) { this.date = date; }
 public String getTitle() { return title; }
 public void setTitle(String title) { this.title = title; }
}
```

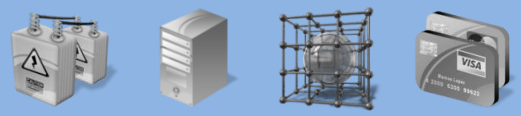
Hibernate – mapping



- **Event.hbm.xml**

```
<hibernate-mapping package="org.hibernate.tutorial.domain">
    <class name="Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
        <property name="date" type="timestamp"
                column="EVENT_DATE"/>
        <property name="title"/>
    </class>
</hibernate-mapping>
```

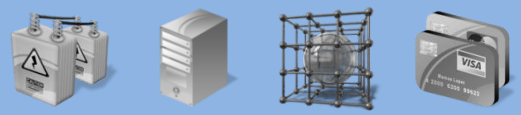
Hibernate – configuration



- **hibernate.cfg.xml**

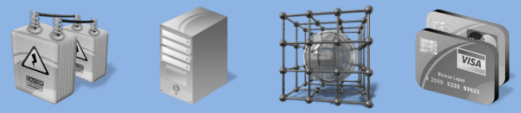
```
<hibernate-configuration>
 <session-factory>
    <!-- Database connection settings >
    <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
    <property name="connection.url">jdbc:hsqldb:hsql://localhost</property>
    <property name="connection.username">sa</property>
    <property name="connection.password"></property>
    <!-- JDBC connection pool (use the built-in) -->
    <property name="connection.pool_size">1</property>
    <!-- SQL dialect -->
    <property name="dialect">org.hibernate.dialect.HSQLDialect</property>
    <!-- Enable Hibernate's automatic session context management -->
    <property name="current_session_context_class">thread</property>
```

Hibernate – configuration



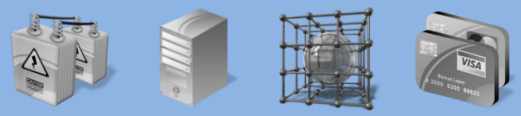

```
    <!-- Disable the second-level cache -->
    <property name="cache.provider_class">
        org.hibernate.cache.internal.NoCacheProvider
    </property>
    <!-- Echo all executed SQL to stdout -->
    <property name="show_sql">true</property>
    <!-- Drop and re-create the database schema on startup -->
    <property name="hbm2ddl.auto">update</property>
    <mapping resource="org/hibernate/tutorial/domain/Event.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

Helper class



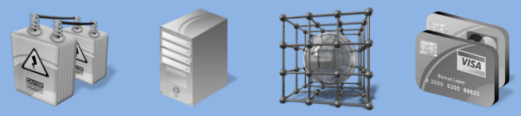

```
package org.hibernate.tutorial.util;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
   private static final SessionFactory sessionFactory = buildSessionFactory();
   private static SessionFactory buildSessionFactory() {
            try {
               // Create the SessionFactory from hibernate.cfg.xml
               return new Configuration().configure().buildSessionFactory(); }
            catch (Throwable ex) {
                 // Make sure you log the exception, as it might be
               System.err.println("Initial SessionFactory creation failed." + ex);
                throw new ExceptionInInitializerError(ex);
            }
   }
   public static SessionFactory getSessionFactory() {
            return sessionFactory;
   }
}
```

Loading and storing objects



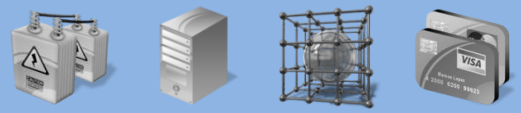

```
package org.hibernate.tutorial;
import org.hibernate.Session;
import java.util.*;
import org.hibernate.tutorial.domain.Event;
import org.hibernate.tutorial.util.HibernateUtil;
public class EventManager {
   public static void main(String[] args) {
      EventManager mgr = new EventManager();
      if (args[0].equals("store"))
         mgr.createAndStoreEvent("My Event", new Date());
      HibernateUtil.getSessionFactory().close();
   }
   private void createAndStoreEvent(String title, Date theDate) {
      Session session = HibernateUtil.getSessionFactory().getCurrentSession();
      session.beginTransaction();
      Event theEvent = new Event();
      theEvent.setTitle(title);
      theEvent.setDate(theDate);
      session.save(theEvent);
      session.getTransaction().commit();
   }
}
```

Loading and storing objects




```
if (args[0].equals("store"))
    mgr.createAndStoreEvent("My Event", new Date());
else if (args[0].equals("list")) {
    List events = mgr.listEvents();
    for (int i = 0; i < events.size(); i++) {
        Event theEvent = (Event) events.get(i);
        System.out.println( "Event: " + theEvent.getTitle() + " Time: "
                                    + theEvent.getDate() );
}

private List listEvents() {
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();
    List result = session.createQuery("from Event").list();
    session.getTransaction().commit();
    return result;
}
```

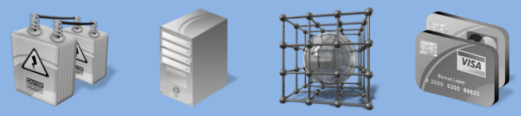
Mapping associations



- **Person.class**

```
package org.hibernate.tutorial.domain;
public class Person {
    private Long id;
    private int age;
    private String firstname;
    private String lastname;
    public Person() {}
    // Accessor methods for all properties, private setter for 'id'

    private Set events = new HashSet();
    public Set getEvents() { return events; }
    public void setEvents(Set events) { this.events = events; }
}
```

Unidirectional association



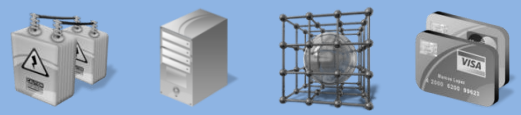
- **Person.hbm.xml**

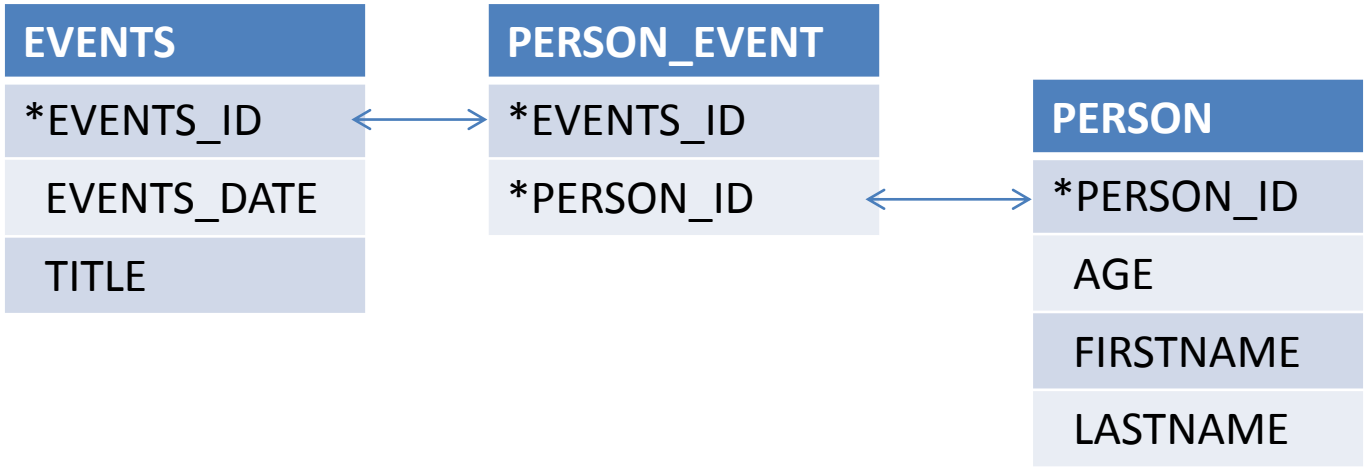
```
<class name="Person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="native"/>
    </id>
    <property name="age"/>
    <property name="firstname"/>
    <property name="lastname"/>
    <set name="events" table="PERSON_EVENT">
        <key column="PERSON_ID"/>
        <many-to-many column="EVENT_ID" class="Event"/>
    </set>
</class>
```

- **Hibernate's configuration**

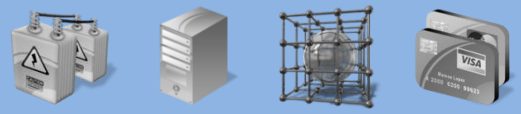
```
<mapping resource="org/hibernate/tutorial/domain/Event.hbm.xml"/>
<mapping resource="org/hibernate/tutorial/domain/Person.hbm.xml"/>
```

Database Schema



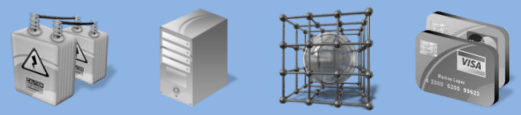


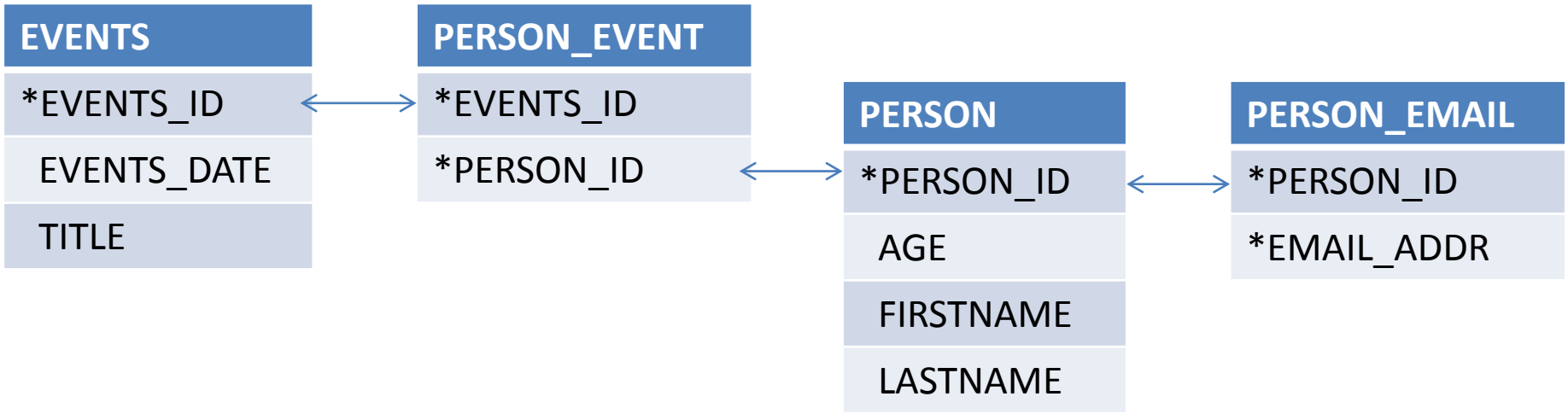
Working the association



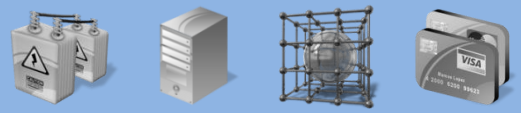

```
private void addPersonToEvent(Long personId, Long eventId) {
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session .createQuery("select p from Person p left join fetch
                                p.events where p.id = :pid") .setParameter("pid",
                                personId) .uniqueResult();
    // Eager fetch the collection so we can use it detached
    Event anEvent = (Event) session.load(Event.class, eventId);
    session.getTransaction().commit();
    // End of first unit of work

    aPerson.getEvents().add(anEvent); // aPerson (and its collection) is detached

   // Begin second unit of work
   Session session2 = HibernateUtil.getSessionFactory().getCurrentSession();
   session2.beginTransaction();
   session2.update(aPerson); // Reattachment of aPerson
   session2.getTransaction().commit();
}
```

Database Schema





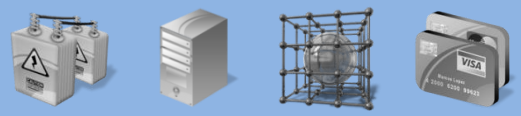
Mapping associations



- **Person.class**

```
package org.hibernate.tutorial.domain;
public class Person {
   private Long id;
   private int age;
   private String firstname;
   private String lastname;
   public Person() {}
   // Accessor methods for all properties, private setter for 'id'

   private Set events = new HashSet();
   public Set getEvents() { return events; }
   public void setEvents(Set events) { this.events = events; }

   private Set emailAddresses = new HashSet();
   public Set getEmailAddresses() { return emailAddresses; }
   public void setEmailAddresses(Set emailAddresses)
   { this.emailAddresses = emailAddresses; }
}
```

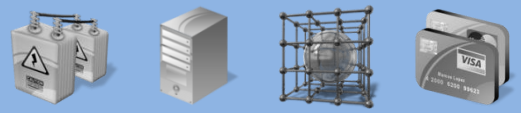
Unidirectional association



- **Person.hbm.xml**

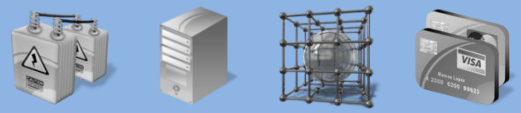
```
<class name="Person" table="PERSON">
   <id name="id" column="PERSON_ID">
      <generator class="native"/>
   </id>
   <property name="age"/>
   <property name="firstname"/>
   <property name="lastname"/>
   <set name="events" table="PERSON_EVENT">
      <key column="PERSON_ID"/>
      <many-to-many column="EVENT_ID" class="Event"/>
   </set>
   <set name="emailAddresses" table="PERSON_EMAIL_ADDR">
      <key column="PERSON_ID"/>
      <element type="string" column="EMAIL_ADDR"/>
   </set>
</class>
```

Collection of values




```
private void addEmailToPerson(Long personId, String emailAddress) {
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session.load(Person.class, personId);
    // adding to the emailAddress collection might trigger a lazy load of the
    // collection
   aPerson.getEmailAddresses().add(emailAddress);
   session.getTransaction().commit();
}
```

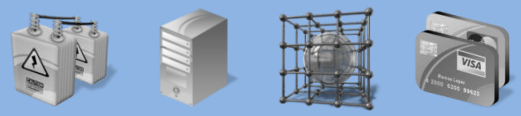
Bi-directional association



- **Event.class**

```
package org.hibernate.tutorial.domain;
import java.util.Date;
public class Event {
 private Long id;
 private String title;
 private Date date;
 public Event() {}
 public Long getId() { return id; }
 private void setId(Long id) { this.id = id; }
 public Date getDate() { return date; }
 public void setDate(Date date) { this.date = date; }
 public String getTitle() { return title; }
 public void setTitle(String title) { this.title = title; }

 private Set participants = new HashSet();
 public Set getParticipants() { return participants; }
 public void setParticipants(Set participants) { this.participants = participants; }
}
```

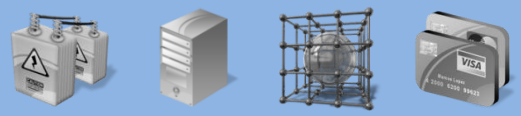
Hibernate – mapping

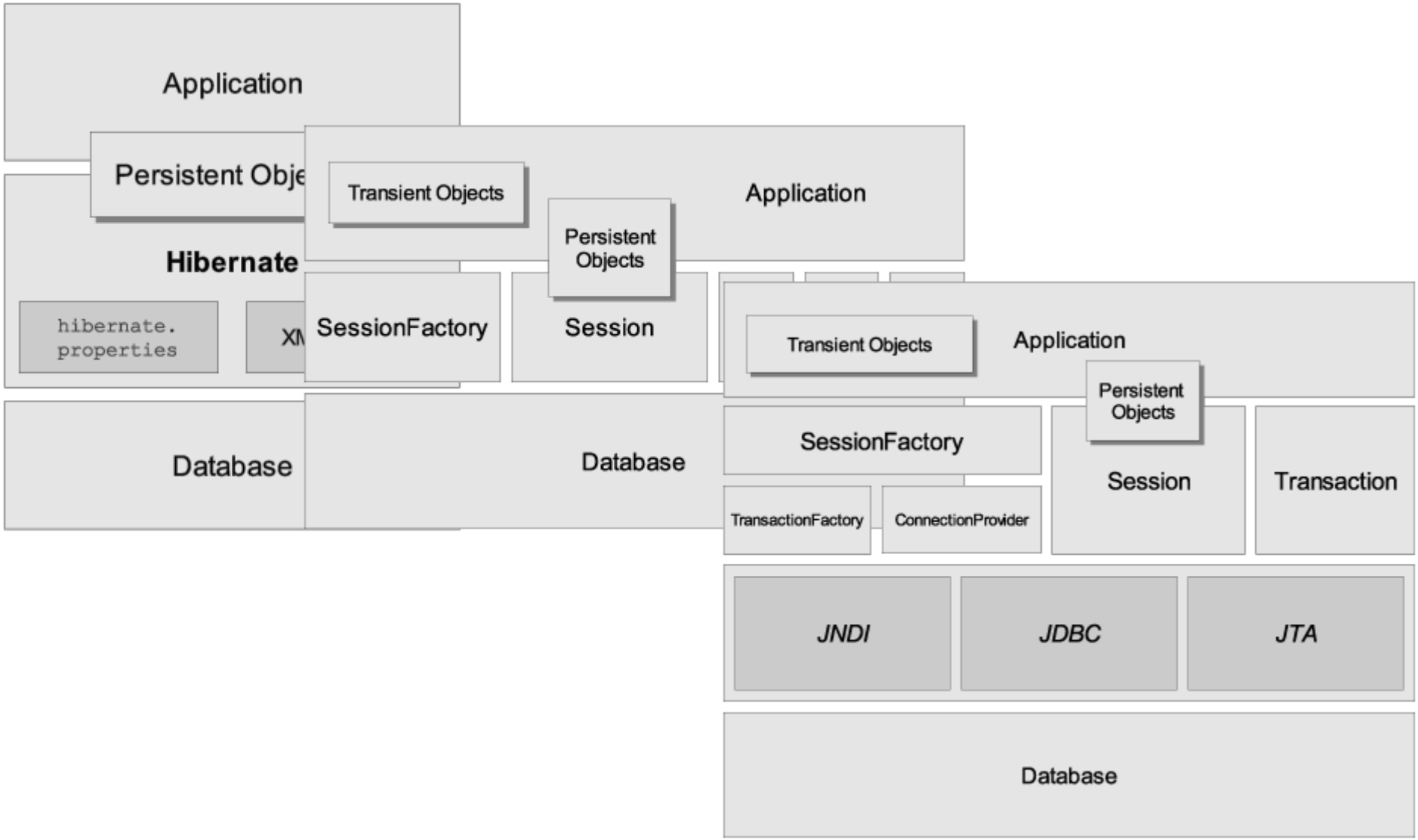


- **Event.hbm.xml**

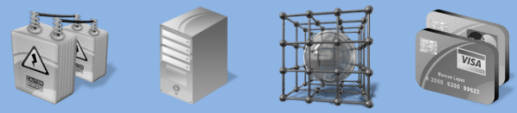
```
<hibernate-mapping package="org.hibernate.tutorial.domain">
    <class name="Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
        <property name="date" type="timestamp"
                column="EVENT_DATE"/>
        <property name="title"/>
        <set name="participants" table="PERSON_EVENT" inverse="true">
          <key column="EVENT_ID"/>
          <many-to-many column="PERSON_ID" class="Person"/>
        </set>
    </class>
</hibernate-mapping>
```

Architecture



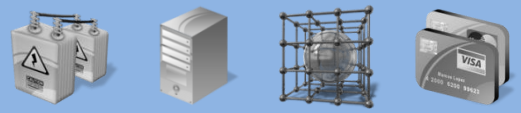


Basic API



- **SessionFactory (org.hibernate.SessionFactory)**
- **Session (org.hibernate.Session)**
- **Persistent objects and collections**
- **Transient and detached objects and collections**
- **Transaction (org.hibernate.Transaction)(Optional)**
- **ConnectionProvider (org.hibernate.connection.ConnectionProvider)(Optional)**
- **TransactionFactory (org.hibernate.TransactionFactory)(Optional)**
- ***Extension Interfaces***

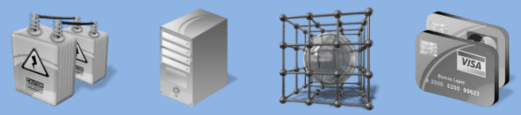
Entity



- **An entity is a regular Java object (aka POJO) which will be persisted by Hibernate.**

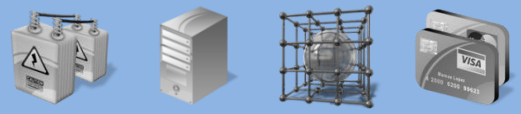
```
@Entity
public class Flight implements Serializable {
    Long id;

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

Entity



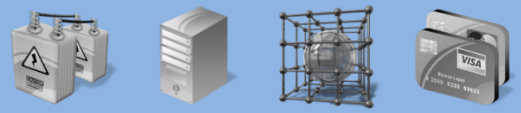

```
@Entity
@Table(name="TBL_FLIGHT",
        schema="AIR_COMMAND",
        uniqueConstraints=
                @UniqueConstraint(
                    name="flight_number",
                  columnNames={"comp_prefix", "flight_number"} ) )
public class Flight implements Serializable {

   @Column(name="comp_prefix")
   public String getCompagnyPrefix() { return companyPrefix; }

   @Column(name="flight_number")
   public String getNumber() { return number; }
}
```

Entity



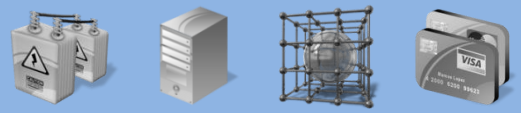
name="ClassName“
table="tableName“
discriminator-value="discriminator_value“
mutable="true|false“
schema="owner“
catalog="catalog“
proxy="ProxyInterface“
dynamic-update="true|false“
dynamic-insert="true|false“
select-before-update="true|false“
polymorphism="implicit|explicit“
where="arbitrary sql where condition
persister="PersisterClass“
batch-size="N“
optimistic-lock="none|version|dirty|all“
lazy="true|false"
entity-name="EntityName"
check="arbitrary sql check condition"
rowxml:id="rowid"
subselect="SQL expression"
abstract="true|false"
node="element-name"

Identifiers




```
@Entity
public class Person {
    @Id Integer getId() { ... }
    ...
}

<id
    name="propertyName"
    type="typename"
    column="column_name"
    unsaved-value="null|any|none|undefined|id_value"
    access="field|property|ClassName" >
        node="element-name|@attribute-name|element/@attribute|."
    <generator class="generatorClass"/>
</id>
```

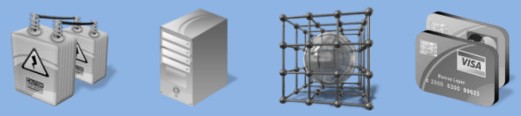
Composite identifier



- **id as a property using a component type**

```
@Entity
class User {
   @EmbeddedId
   @AttributeOverride(
        name="firstName", column=@Column(name="fld_firstname")
   UserId id;
   Integer age;
}

@Embeddable
class UserId implements Serializable {
   String firstName;
   String lastName;
}
```

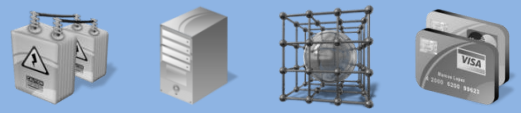
Composite identifier



- **id as a property using a component type**

```
@Entity class Customer {
   @EmbeddedId CustomerId id;
   boolean preferredCustomer;
   @MapsId("userId")
   @JoinColumns({
      @JoinColumn(name="userfirstname_fk", referencedColumnName="firstNam
e"),
      @JoinColumn(name="userlastname_fk", referencedColumnName="lastName
")
   })
   @OneToOne User user;
}

@Embeddable class CustomerId implements Serializable {
   UserID userId;
   String customerNumber;
   //implements equals and hashCode
}
```

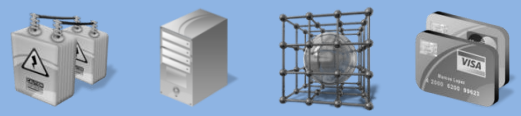
Composite identifier



- **id as a property using a component type**

```
@Entity  class User {
    @EmbeddedId UserId id;
    Integer age;
 }

 @Embeddable
 class UserId implements Serializable {
    String firstName;
    String lastName;
   //implements equals and hashCode
}
```

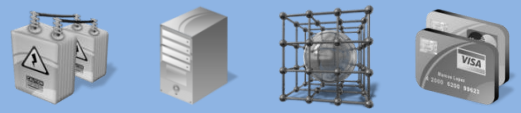
Composite identifier



- **Multiple id properties without identifier type**

```
@Entity  class Customer implements Serializable {
  @Id @OneToOne
  @JoinColumns({
    @JoinColumn(name="userfirstname_fk",
        referencedColumnName="firstName"),
    @JoinColumn(name="userlastname_fk",
        referencedColumnName="lastName")
  })
  User user;
  @Id String customerNumber;
  boolean preferredCustomer;
  //implements equals and hashCode
}
```

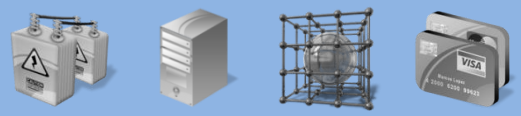
Composite identifier



- **Multiple id properties without identifier type**

```
@Entity  class User {
    @EmbeddedId UserId id;
    Integer age;
 }
@Embeddable class UserId implements Serializable {
    String firstName;
    String lastName;
   //implements equals and hashCode
}
```

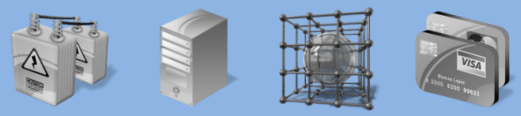
Composite identifier



- **Multiple id properties with a dedicated identifier type**

```
@Entity
@IdClass(CustomerId.class)
class Customer implements Serializable {
   @Id @OneToOne
   @JoinColumns({
      @JoinColumn(name="userfirstname_fk", referencedColumnName="firstName"),
      @JoinColumn(name="userlastname_fk", referencedColumnName="lastName")
   })
   User user;
   @Id String customerNumber;
   boolean preferredCustomer;
}

class CustomerId implements Serializable {
   UserId user;
   String customerNumber
  //implements equals and hashCode
}
```

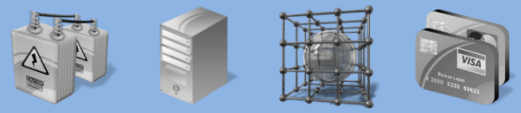
Composite identifier



- **Multiple id properties with a dedicated identifier type**

```
@Entity
class User {
    @EmbeddedId UserId id;
    Integer age;
    //implements equals and hashCode
 }

 @Embeddable class UserId implements Serializable {
    String firstName;
    String lastName;
   //implements equals and hashCode
}
```

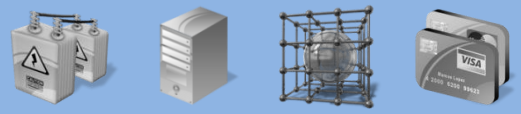
Identifier generator



- IDENTITY
- SEQUENCE (called seqhilo in Hibernate)
- TABLE (called MultipleHiLoPerTableGenerator in Hibernate)
- AUTO

```
@Entity
public class Customer {
    @Id @GeneratedValue
    Integer getId() { ... };
}

@Entity
public class Invoice {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    Integer getId() { ... };
}
```

Inheritance strategy

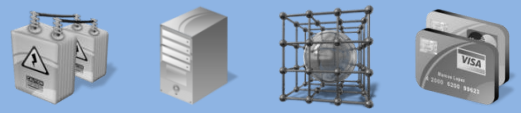


- **Single table per class hierarchy strategy**

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

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

```
<subclass name="ClassName" discriminator-value="discriminator_value"
          proxy="ProxyInterface" lazy="true|false"
          dynamic-update="true|false" dynamic-insert="true|false"
          entity-name="EntityName" node="element-name"
          extends="SuperclassName">
          <property .... /> .....
</subclass>
```

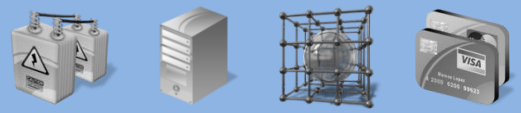
Inheritance strategy



- **Joined subclass strategy**

```
@Entity
@Table(name="CATS")
@Inheritance(strategy=InheritanceType.JOINED)
public class Cat implements Serializable {
    @Id @GeneratedValue(generator="cat-uuid")
    @GenericGenerator(name="cat-uuid", strategy="uuid")
    String getId() { return id; }
    ...
}

@Entity
@Table(name="DOMESTIC_CATS")
@PrimaryKeyJoinColumn(name="CAT")
public class DomesticCat extends Cat {
   public String getName() { return name; }
}
```

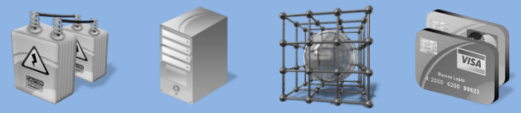
Inheritance strategy



- **Joined subclass strategy**

```
<hibernate-mapping package="eg">
  <class name="Cat" table="CATS">
    <id name="id" column="uid" type="long">
      <generator class="hilo"/>
    </id>
    <property name="birthdate" type="date"/>
    <property name="color" not-null="true"/>
    <property name="sex" not-null="true"/>
    <property name="weight"/>
    <many-to-one name="mate"/>
    <set name="kittens">
      <key column="MOTHER"/>
      <one-to-many class="Cat"/>
    </set>
    <joined-subclass name="DomesticCat" table="DOMESTIC_CATS">
       <key column="CAT"/>
       <property name="name" type="string"/>
    </joined-subclass>
  </class>
</hibernate-mapping>
```

Inheritance strategy

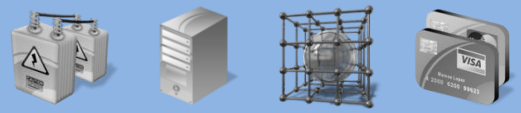


- **Table per class strategy**

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

- **No discriminator column or key column is required for this mapping strategy.**

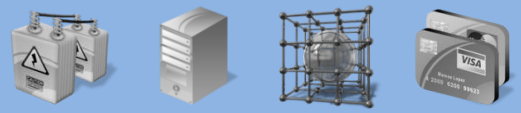
Inheritance strategy



- **Inherit properties from superclasses**

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

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

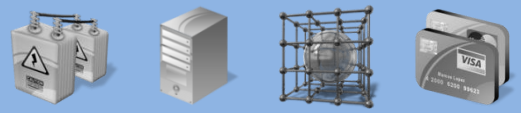
Inheritance strategy



- **Inherit properties from superclasses**

```
@MappedSuperclass
public class FlyingObject implements Serializable
    public int getAltitude() {  return altitude;  }
    @Transient
    public int getMetricAltitude() {  return metricAltitude; }
    @ManyToOne
    public PropulsionType getPropulsion() {  return metricAltitude; }
    ...
}

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

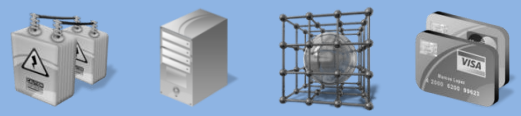
Inheritance strategy



- **Mapping one entity to several tables**

```
@Entity @Table(name="MainCat")
@SecondaryTables({
   @SecondaryTable(name="Cat1", pkJoinColumns={
      @PrimaryKeyJoinColumn(name="cat_id", referencedColumnName="id")    ),
   @SecondaryTable(name="Cat2", uniqueConstraints={
      @UniqueConstraint(columnNames={"storyPart2"})}) })
public class Cat implements Serializable {
   private Integer id;
   private String name;
   private String storyPart1;
   private String storyPart2;
   @Id @GeneratedValue
   public Integer getId() {       return id;    }
   public String getName() {       return name;    }
   @Column(table="Cat1")
   public String getStoryPart1() {       return storyPart1;    }
   @Column(table="Cat2")
   public String getStoryPart2() {       return storyPart2;    }
}
```

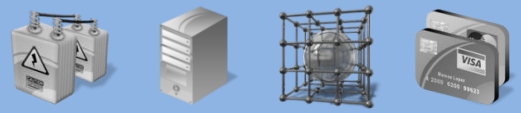
Mapping Associations



- **Using a foreign key or an association table**

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

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

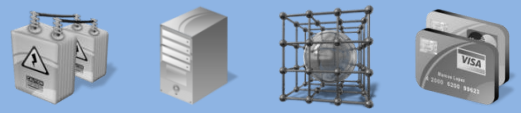
Mapping Associations



- **Sharing the primary key with the associated entity**

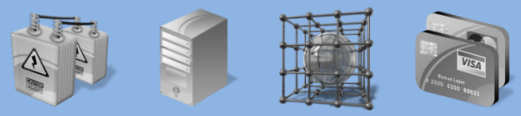
```
@Entity
public class Body {
    @Id
    public Long getId() { return id; }
    @OneToOne(cascade = CascadeType.ALL)
    @MapsId
    public Heart getHeart() {
        return heart;
    }
    ...
}

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

Hibernate Overview



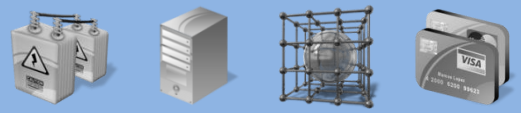
- **O/R mapping**
  - **Basic O/R Mapping**
  - **Working with Objects**

Hibernate object states



- **Hibernate defines and supports the following object states:**
  - ***Transient***
    - **an object is transient if it has just been instantiated using the new operator, and it is not associated with a Hibernate Session.**
  - ***Persistent***
    - **a persistent instance has a representation in the database and an identifier value.**
  - ***Detached***
    - **a detached instance is an object that has been persistent, but its Session has been closed.**

Making objects persistent

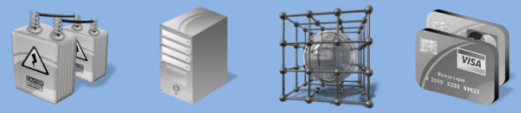



```
DomesticCat fritz = new DomesticCat();
fritz.setColor(Color.GINGER);
fritz.setSex('M');
fritz.setName("Fritz");
Long generatedId = (Long) sess.save(fritz);

DomesticCat pk = new DomesticCat();
pk.setColor(Color.TABBY);
pk.setSex('F');
pk.setName("PK");
pk.setKittens( new HashSet() );
pk.addKitten(fritz);
sess.save( pk, new Long(1234) );
```

or

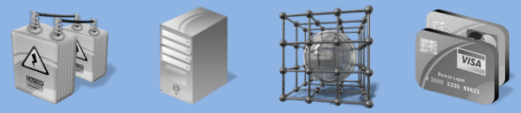
```
sess.persist( pk, new Long(1234) );
```

Loading an object



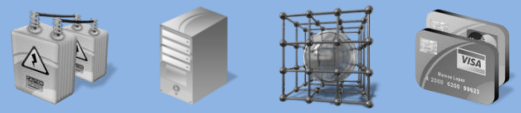

```
Cat fritz = (Cat) sess.load(Cat.class, generatedId);

// you need to wrap primitive identifiers
long id = 1234;
DomesticCat pk = (DomesticCat) sess.load( DomesticCat.class, new Long(id) );

Cat cat = new DomesticCat();
// load pk's state into cat
sess.load( cat, new Long(pkId) );
Set kittens = cat.getKittens();

Cat cat = (Cat) sess.get(Cat.class, id);
 if (cat==null) {
   cat = new Cat();
   sess.save(cat, id);
}
return cat;

sess.save(cat);
sess.flush(); //force the SQL INSERT
sess.refresh(cat); //re-read the state (after the trigger executes)
```

Executing queries



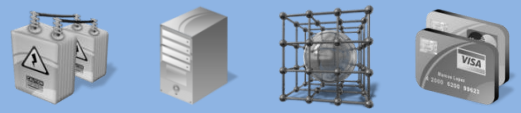

```
List cats = session.createQuery(
    "from Cat as cat where cat.birthdate < ?")
    .setDate(0, date)
    .list();
List mothers = session.createQuery(
    "select mother from Cat as cat join cat.mother as mother where cat.name = ?")
    .setString(0, name)
    .list();
List kittens = session.createQuery(
    "from Cat as cat where cat.mother = ?")
    .setEntity(0, pk)
    .list();
Cat mother = (Cat) session.createQuery(
    "select cat.mother from Cat as cat where cat = ?")
    .setEntity(0, izi)
    .uniqueResult();
Query mothersWithKittens = (Cat) session.createQuery(
    "select mother from Cat as mother left join fetch mother.kittens");
Set uniqueMothers = new HashSet(mothersWithKittens.list());
```

Iterating results




```
Iterator kittensAndMothers = sess.createQuery(
    "select kitten, mother from Cat kitten join kitten.mother mother")
   .list()
   .iterator();
while ( kittensAndMothers.hasNext() ) {
   Object[] tuple = (Object[]) kittensAndMothers.next();
   Cat kitten = (Cat) tuple[0];
   Cat mother = (Cat) tuple[1];
   ....
}

Iterator results = sess.createQuery(
    "select cat.color, min(cat.birthdate), count(cat) from Cat cat " + "group by cat.color")
    .list()
    .iterator();
while ( results.hasNext() ) {
   Object[] row = (Object[]) results.next();
   Color type = (Color) row[0];
   Date oldest = (Date) row[1];
   Integer count = (Integer) row[2];
   .....
}
```

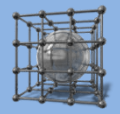
Bind parameters




```
//named parameter (preferred)
Query q = sess.createQuery("from DomesticCat cat where cat.name = :name");
q.setString("name", "Fritz");
Iterator cats = q.iterate();

//positional parameter
Query q = sess.createQuery("from DomesticCat cat where cat.name = ?");
q.setString(0, "Izi");
Iterator cats = q.iterate();

//named parameter list
List names = new ArrayList();
names.add("Izi");
names.add("Fritz");
Query q = sess.createQuery("from DomesticCat cat where cat.name in
(:namesList)");
q.setParameterList("namesList", names);
List cats = q.list();
```

Pagination and Scrollable iteration

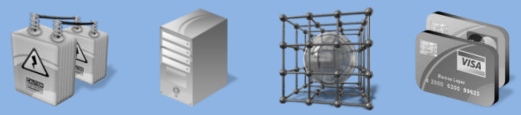






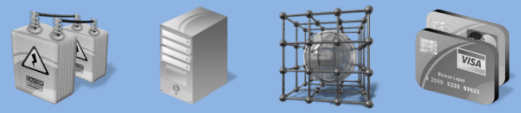
```
Query q = sess.createQuery("from DomesticCat cat"); q.setFirstResult(20);
q.setMaxResults(10);
List cats = q.list();

Query q = sess.createQuery("select cat.name, cat from DomesticCat cat " + "order by
cat.name");
ScrollableResults cats = q.scroll();
if ( cats.first() ) {
    // find the first name on each page of an alphabetical list of cats by name
    firstNamesOfPages = new ArrayList();
    do {
        String name = cats.getString(0);
        firstNamesOfPages.add(name);
    } while ( cats.scroll(PAGE_SIZE) );
    // Now get the first page of cats
    pageOfCats = new ArrayList();
    cats.beforeFirst();
    int i=0;
    while( ( PAGE_SIZE > i++ ) && cats.next() ) pageOfCats.add( cats.get(1) );
}
cats.close()
```

Criteria queries and native SQL



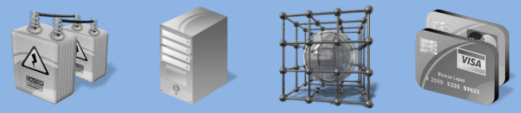

```
Criteria crit = session.createCriteria(Cat.class); crit.add(
Restrictions.eq( "color", eg.Color.BLACK ) ); crit.setMaxResults(10);
List cats = crit.list();


List cats = session.createSQLQuery("SELECT {cat.*} FROM CAT {cat}
WHERE ROWNUM<10")
   .addEntity("cat", Cat.class)
   .list();
List cats = session.createSQLQuery(
   "SELECT {cat}.ID AS {cat.id}, {cat}.SEX AS {cat.sex}, " +
   "{cat}.MATE AS {cat.mate}, {cat}.SUBCLASS AS {cat.class}, ... " +
   "FROM CAT {cat} WHERE ROWNUM<10")
   .addEntity("cat", Cat.class)
   .list()
```

Modifying objects




```
DomesticCat cat = (DomesticCat) sess.load( Cat.class, new Long(69) );
cat.setName("PK");
sess.flush(); // changes to cat are automatically detected and persisted

// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catId);
Cat potentialMate = new Cat();
firstSession.save(potentialMate);

// in a higher layer of the application
cat.setMate(potentialMate);

// later, in a new session
secondSession.update(cat); // update cat
secondSession.update(mate); // update mate
or
secondSession.merge(cat); // merge cat
secondSession.merge(mate); // merge mate
```

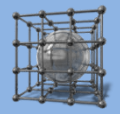
Automatic state detection




```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catID);

// in a higher tier of the application
Cat mate = new Cat();
cat.setMate(mate);

// later, in a new session
secondSession.saveOrUpdate(cat);
// update existing state (cat has a non-null id)
secondSession.saveOrUpdate(mate);
// save the new instance (mate has a null id)
```

Deleting persistent objects

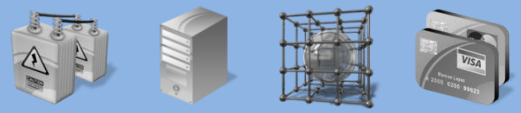






```
sess.delete(cat);
```

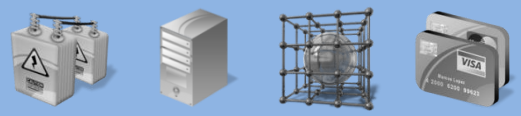
Replicating object




```
//retrieve a cat from one database
Session session1 = factory1.openSession();
Transaction tx1 = session1.beginTransaction();
Cat cat = session1.get(Cat.class, catId);
tx1.commit();
session1.close();

//reconcile with a second database
Session session2 = factory2.openSession();
Transaction tx2 = session2.beginTransaction();
session2.replicate(cat, ReplicationMode.LATEST_VERSION);
tx2.commit();
session2.close();
```

- **ReplicationMode:**
  - **IGNORE**
  - **OVERWRITE**
  - **EXCEPTION**
  - **LATEST_VERSION**

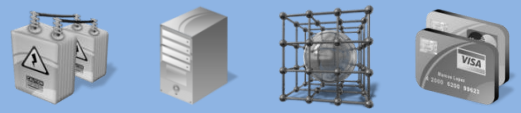
Flushing the Session



- ***flush*, occurs by default at the following points:**
  - **before some query executions**
  - **from org.hibernate.Transaction.commit()**
  - **from Session.flush()**

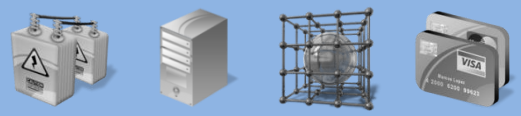
- **The SQL statements are issued in the following order:**
  - **all entity insertions in the same order the corresponding objects were saved using Session.save()**
  - **all entity updates**
  - **all collection deletions**
  - **all collection element deletions, updates and insertions**
  - **all collection insertions**
  - **all entity deletions in the same order the corresponding objects were deleted usingSession.delete()**

Flushing the Session



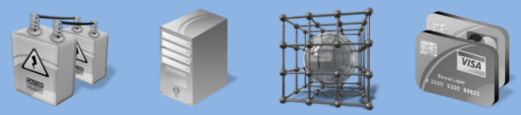

```
sess = sf.openSession();
Transaction tx = sess.beginTransaction();
sess.setFlushMode(FlushMode.COMMIT);
// allow queries to return stale state

Cat izi = (Cat) sess.load(Cat.class, id);
izi.setName(iznizi);

// might return stale data
sess.find("from Cat as cat left outer join cat.kittens kitten");

// change to izi is not flushed!
...
tx.commit(); // flush occurs
sess.close();
```

Transitive persistence



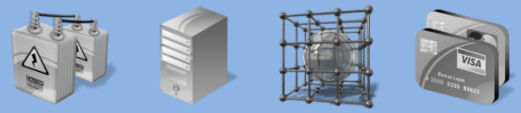
- For each basic operation of the Hibernate session there is a corresponding cascade style
- including persist(), merge(), saveOrUpdate(), delete(), lock(), refresh(), evict(), replicate()
  - CascadeType.PERSIST
  - CascadeType.MERGE
  - CascadeType.REMOVE
  - CascadeType.REFRESH
  - CascadeType.DETACH
  - CascadeType.ALL

Transitive persistence

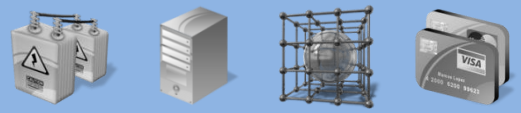



```
@Entity
public class Customer {
    private Set<Order> orders;
    @OneToMany(cascade=CascadeType.ALL, orphanRemoval=true)
   public Set<Order> getOrders() { return orders; }
   public void setOrders(Set<Order> orders) { this.orders = orders; }
   [...]
}

@Entity
public class Order { ... }

Customer customer = em.find(Customer.class, 1l);
Order order = em.find(Order.class, 1l);
customer.getOrders().remove(order); //order will be deleted by cascade
```

References



- **HIBERNATE - Relational Persistence for Idiomatic Java, http://docs.jboss.org/hibernate/orm/4.1/manual/en-US/html_single/#preface**



面对对象分析与设计

Object-Oriented Analysis and Design

下课！