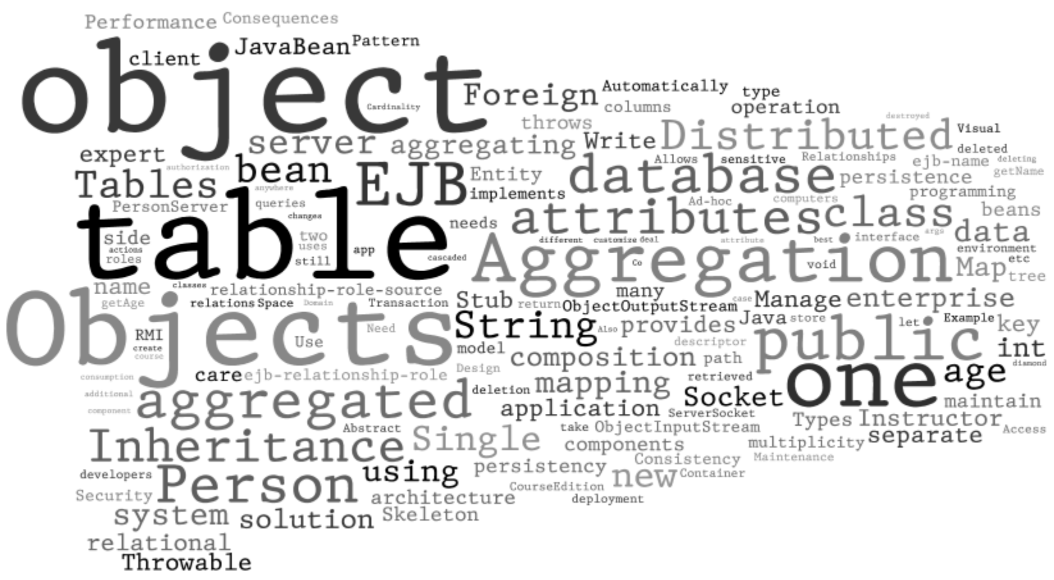
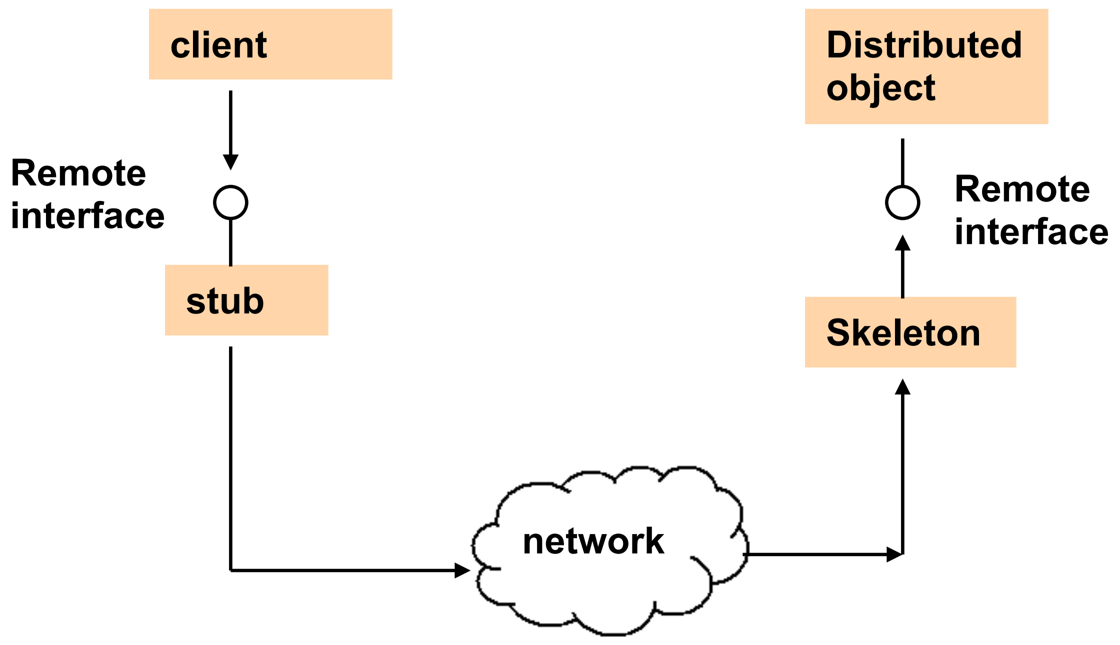
Distributed Objects and object persistence



Distributed objects

- Distributed computing: part of the system located on separate computers
- Distributed objects: allow objects running on one machine to be used by client applications on different computers
- Distributed object technologies:
  - Java RMI
  - CORBA
  - DCOM

Implementing distributed objects



– Skeleton: server side proxy object
– Stub: client side proxy object
– Stub and skeleton implements the same remote interface

Write your own distributed object

- Client only knows the interface

```
public interface Person {
  public int getAge() throws Throwable;
  public String getName() throws Throwable;
}
```

- Client uses the object just as if it were local

```
public class PersonClient {
  public static void main(String[] args) {
    try {
          Person person = new Person_Stub();
          int age = person.getAge();
          String name = person.getName();
          System.out.println(name + age);
    } catch (Throwable t) {
    }
  }
}
```

- Networking is taken care of by Person_Stub

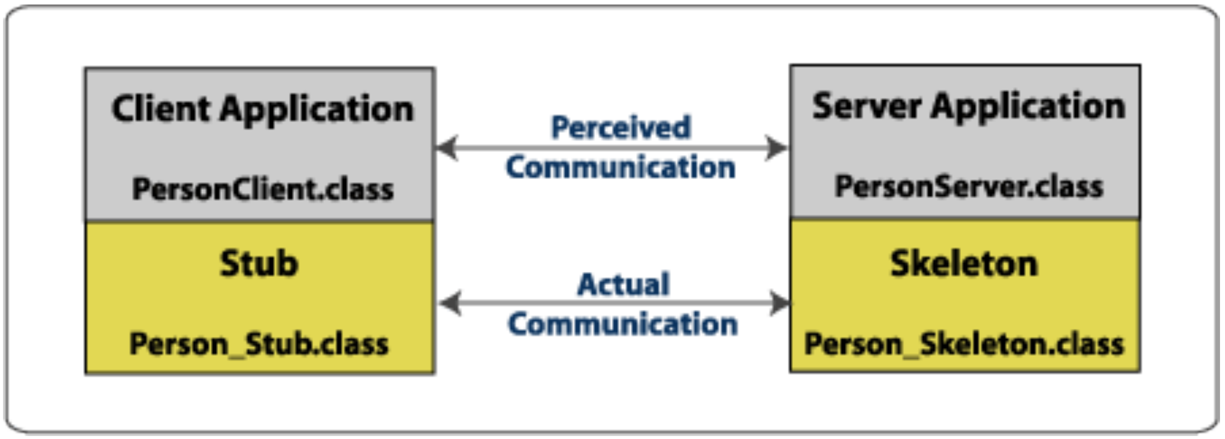
From the client side

```
public class Person_Stub implements Person{
    Socket socket;
    public Person_Stub() throws Throwable{
          socket=new Socket("ip address here", 8765);
    }

    public int getAge()throws Throwable{
       ObjectOutputStream outStream =
                              new ObjectOutputStream(socket.getOutputStream());
       outStream.writeObject("age");
       outStream.flush();
       ObjectInputStream inStream=
                           new ObjectInputStream(socket.getInputStream());
       return inStream.readInt();
    }

    public String getName()throws Throwable{ … }
}
```

From the server side

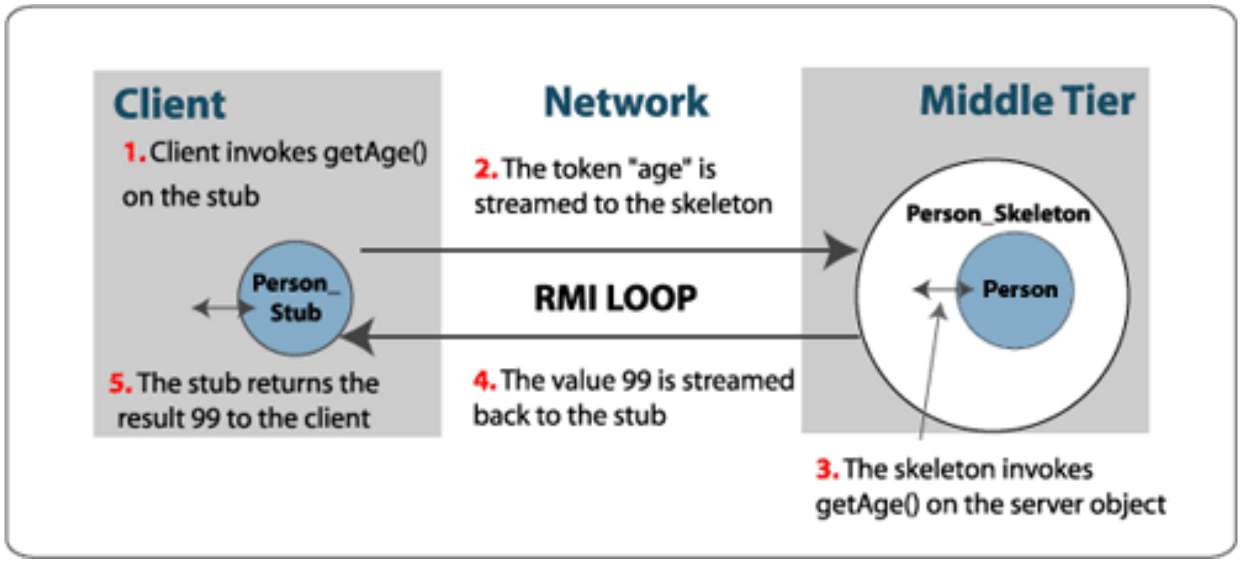
```
public class Person_Skeleton extends Thread { …

public Person_Skeleton(Person person) {  this.myPerson = person;}

public void run() { …
    ServerSocket serverSocket = new ServerSocket(8765);
    Socket socket = serverSocket.accept();
    while (socket != null) {
      ObjectInputStream inStream = new ObjectInputStream(socket.getInputStream());
       String method = (String) inStream.readObject();
       if (method.equals("age")) {
         int age = myPerson.getAge();
         ObjectOutputStream outStream =
                  new ObjectOutputStream(socket.getOutputStream());
         outStream.writeInt(age);
        outStream.flush();
       } else if (method.equals("name")) { … }
   } …
}

public static void main(String[] args){
    PersonServer person = new PersonServer("mike", 24);
    Person_Skeleton skel=new Person_Skeleton(person);
    skel.start();
}}
```

```
public class PersonServer implements Person {
  int age;
  String name;
  public PersonServer(String n, int a) { name=n; age=a;}
  public int getAge() {return age; }
  public String getName() {return name; }
}
```

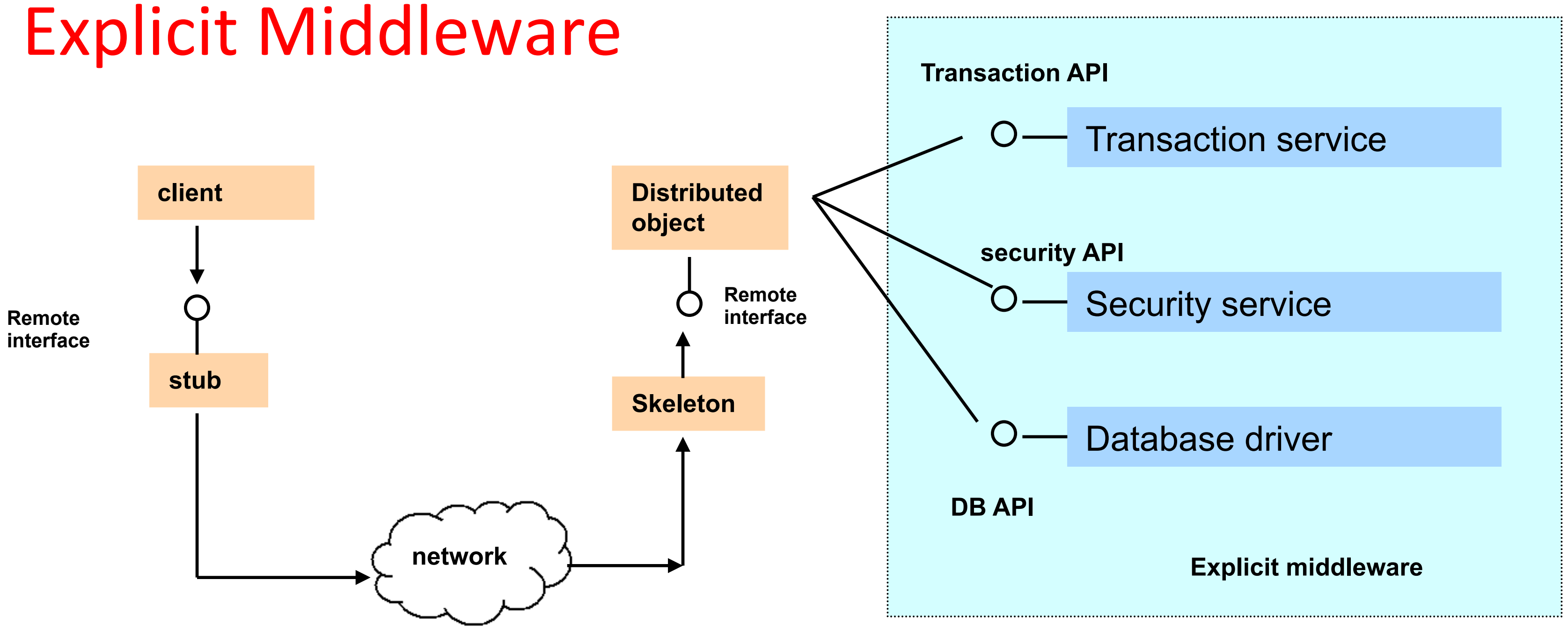


From hand-craft to RMI

- RMI technology
  - Automatically generate appropriate stubs and skeletons
  - Error and exception handling
  - Parameter passing
- RMI is not good enough in
  - Object persistence;
  - Transaction handling;
  - Security;
  - … …


Client
1. Client invokes getAge() on the stub
Person_ Stub
5. The stub returns the result 99 to the client
Network
2. The token "age" is streamed to the skeleton
RMI LOOP
4. The value 99 is streamed back to the stub
Middle Tier
Person_Skeleton
Person
3. The skeleton invokes getAge() on the server object

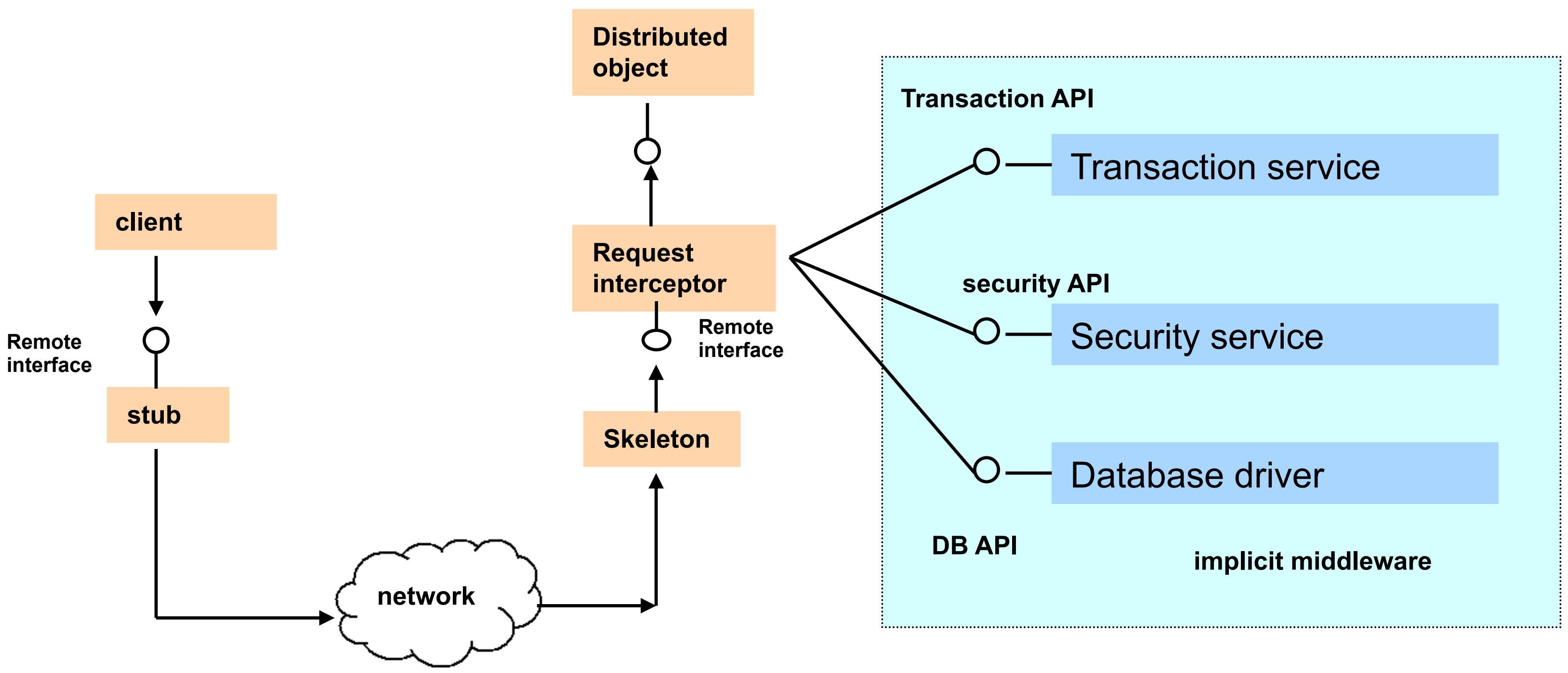
Explicit Middleware



```
Transfer(Account a1, Account a2, long amount) {
    Call middleware API to check security;
    call middleware API to start a transaction;
    if (a1.balance>amount)
              subtract the balance of a1 and add the amount to a2;
    call DB API to store the data;
    call middleware API to end the transaction;
}
```

- Difficult to write, maintain, and support

Implicit middleware



```
Transfer(Account a1, Account a2, long amount){
    if (a1.balance>amount)
        subtract the balance of a1 and add the amount to a2;
}
```

Declare the middle services needed in a text file.
Generate a Request Interceptor from this declaration

- Easy to write, maintain, and support

EJB (enterprise java bean)

- EJB 2.0 wants to provide basic services and environment to make enterprise systems easier to develop, by providing
  - automatically-managed *persistence* logic
  - *transaction* plumbing for components
  - an enforced-*authorization* framework
  - "best of breed" capabilities by providing all this in a vendor-neutral fashion
- This is the ambitious goal set by EJB version 2

three versions of EJB

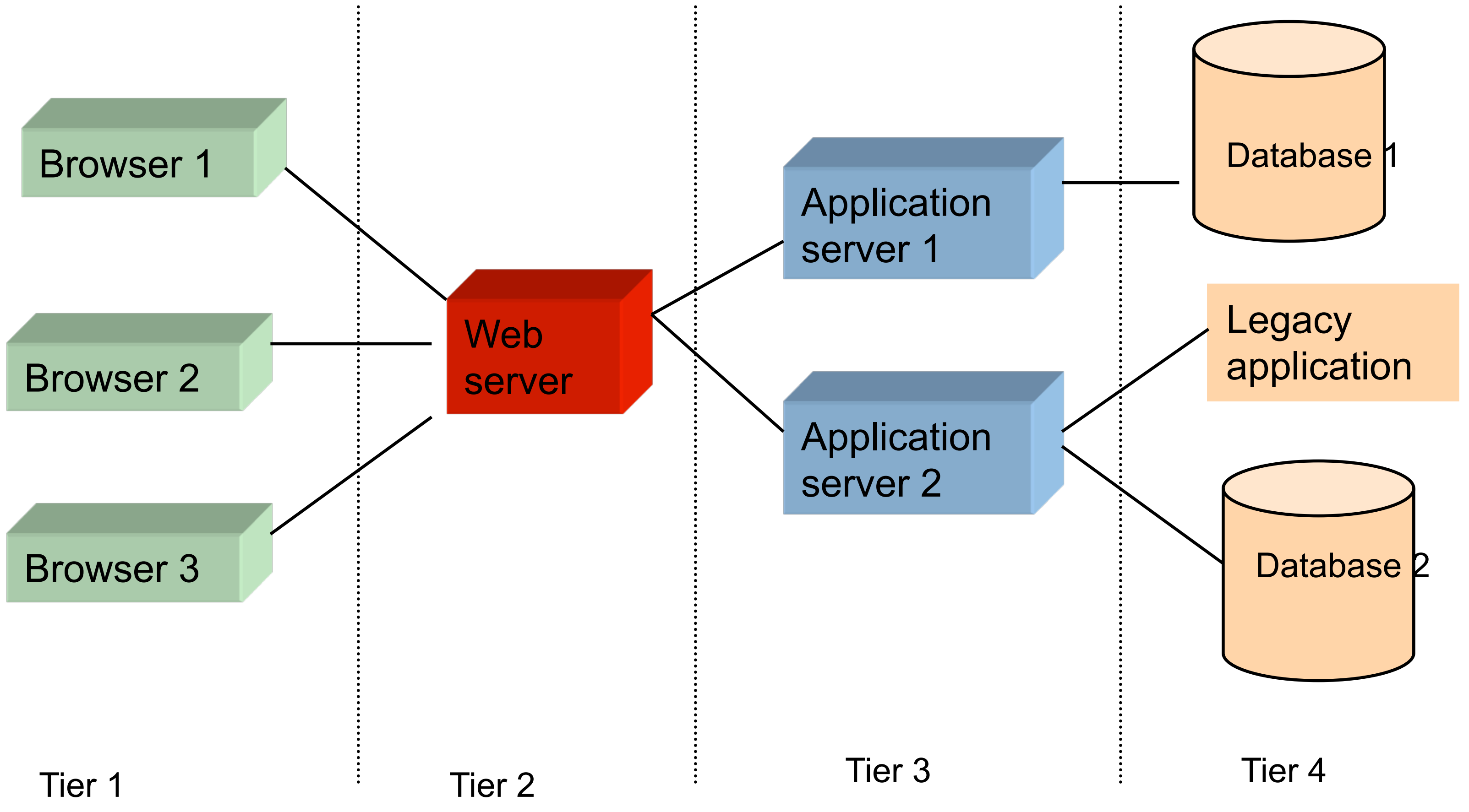
- EJB 1 (1998) advocated by IBM and Sun, quickly adopted in industry
- EJB 2 (2001) excessively complicated, widely criticized
- EJB 3 (2006) reinventing EJB, simplified, adopted tech from Hibernate etc.

“One day, when God was looking over his creatures, he noticed a boy named Sadhu whose humor and cleverness pleased him. God felt generous that day and granted Sadhu three wishes. Sadhu asked for three reincarnations—one as a ladybug, one as an elephant, and the last as a cow. Surprised by these wishes, God asked Sadhu to explain himself. The boy replied, "I want to be a ladybug so that everyone in the world will admire me for my beauty and forgive the fact that I do no work. Being an elephant will be fun because I can gobble down enormous amounts of food without being ridiculed. I will like being a cow the best because I will be loved by all and useful to mankind." God was charmed by these answers and allowed Sadhu to live through the three incarnations. He then made Sadhu a morning star for his service to humankind as a cow.

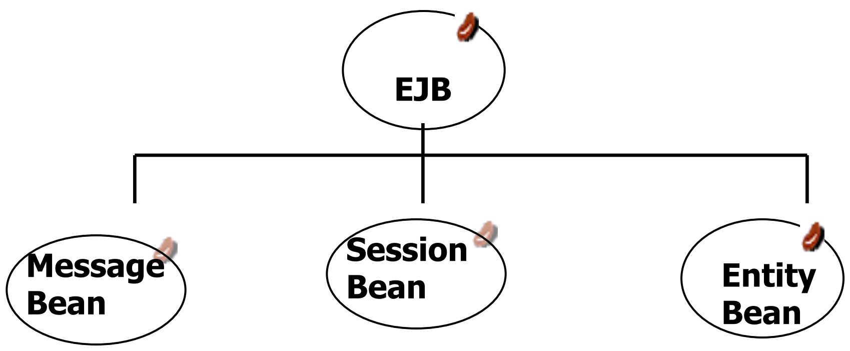
EJB too has lived through three incarnations.”

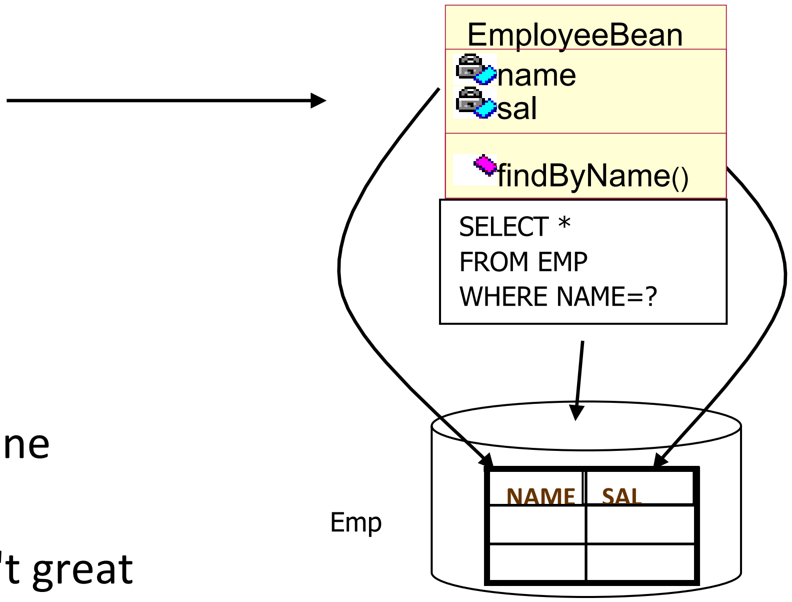
—from “EJB3 in action”

Multi-tier Enterprise Architecture



Types of beans





- Entity bean supports object persistence
  - Access to relational data is done through SQL
  - Most Java programmers aren't great database engineers
  - Therefore let the application server worry about how to obtain the data

Part 2: Object Persistence

Persistence

- Persistency: characteristic of data that outlives the execution of the program that created it.
- Almost all applications require persistence data
- Data are typically saved in relational database
- In Java, we can write SQL statement using JDBC API
  - Send a SQL query string in Java
  - Obtains rows of returned query results
  - Process the rows one by one

JDBC example

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery( "SELECT * FROM MyTable" );
while ( rs.next() ) {
     int numColumns = rs.getMetaData().getColumnCount();
     for ( int i = 1 ; i <= numColumns ; i++ ) {
       System.out.println("COLUMN" + i + " = "+ rs.getObject(i));
     }
}
```

- Disadvantages
  - The process is low level and tedious
  - Programmers love Objects

Object Persistence

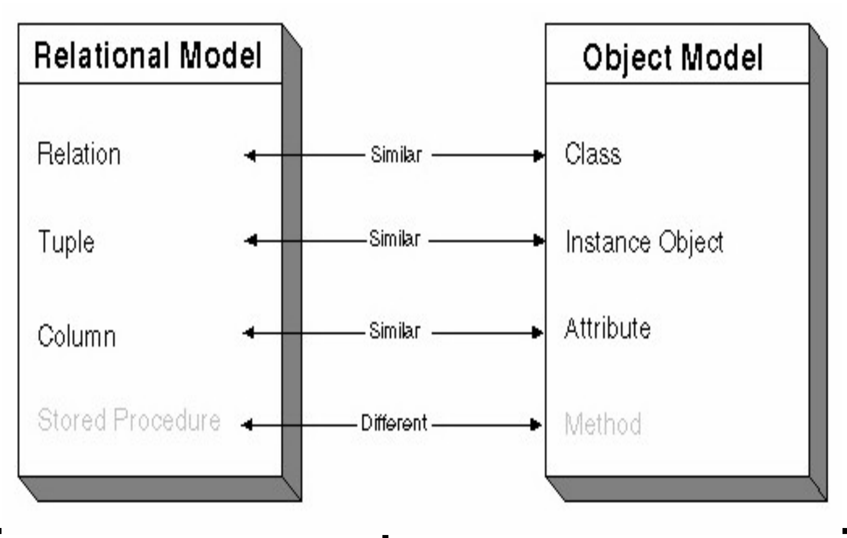
- Object persistence: Allow an object to outlive the process that creates it
  - State of an object can be stored in a permanent storage;
  - An object with the same state can be recreated later;
  - Most often the storage and retrieval involves a whole network of objects
- Not every object need to be persistent.
  - A typical application is a mix of persistent objects and transient objects
- Objects can't be directly saved to and retrieved from relational databases.
- Approaches to object persistence
  - Object serialization
    - Convert object into a byte stream and store it as a whole in database/ file system.
  - XML file based persistence, such as JAXB (Java Architecture for XML Binding)
  - Object database
    - Objects are the 1st class citizen in the database
  - Object relational mapping (ORM)
    - Supported by Hibernate, EJB, etc. ✓

Object-based persistence: java.io.Serializable

- Converts an object (and all internal objects) into a stream of bytes that can be later deserialized into a copy of the original object (and all internal objects).
- Pros:
  - Fast, simple, compact representation of an object graph.
  - May be great choice for *temporary* storage of data.
- Cons:
  - Creates long-term maintenance issues
  - Harder to evolve objects and maintain backward compatibility with serialized representation.

```
public class Address extends Serializable {
// Class Definition
}

// Serialise an object
FileOutputStream f = new FileOutputStream("tmp");
ObjectOutput out = new ObjectOutputStream(f);
out.writeObject(new Address());
out.flush();
out.close();

// Deserialise an object
FileInputStream f = new FileInputStream("tmp");
ObjectInput in = new ObjectInputStream(f);
Address address = (Address) in.readObject();
in.close();
```

Objects and relations



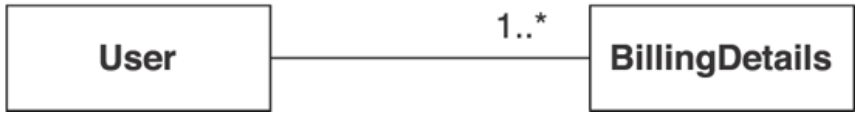
Relation	**Object**
goal of relational modeling is to normalize data (i.e., eliminate redundant data from tables)	goal of object-oriented design is to model a business process by creating real-world objects with data and behavior
RDBMS stores data only	objects have identity, state, and behavior
no inheritance	organized into inheritance hierarchy
tables are related via values in foreign and primary keys	objects are traversed using direct references

Paradigm mismatch

```
public class User {
    private String username;
    private String name;
    private String address;
    private Set billingDetails;

    // Accessor methods (getter/setter), business methods,
      etc.
    ...
}
public class BillingDetails {
    private String accountNumber;
    private String accountName;
    private String accountType;
    private User user;

    // Accessor methods (getter/setter), business methods,
      etc.
    ...
}
```

```
create table USERS (
    USERNAME varchar(15) not null primary key,
    NAME varchar(50) not null,
    ADDRESS varchar(100)
)
create table BILLING_DETAILS (
    ACCOUNT_NUMBER varchar(10) not null primary key,
    ACCOUNT_NAME varchar(50) not null,
    ACCOUNT_TYPE varchar(2) not null,
    USERNAME varchar(15) foreign key references user
)
```

ACCOUNT_ NUMBER	ACCOUNT_ NAME	ACCOUNT_ TYPE	USERNAME
111	aaa	01	david
222	bbb	02	david
…			



Example from "Java Persistence with Hibernate"

Association problem

- In OO, relationships are expressed as references.
  - Object references are directional. They are pointers. If you need to navigate in both directions, you must define the relationship twice.
  - Navigation example:
    - user.getBillingDetails().getAddress
    - billing.getUser().get…
- In RDBMS, relationships expressed as foreign keys.
  - FK associates are not directional. You can create arbitrary relationships with joins.

```
public class User {
  private Set billingDetails;
   ...
}
public class BillingDetails {
  private User user;
   ...
}
```

Many-to-many association

- OO association can have many-to-many multiplicity
- Table association is always one-to-one or one-to-many
- To represent many-to-many association, a link table must be introduced

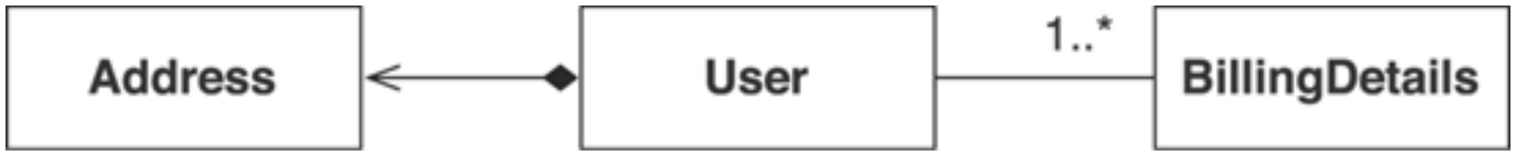
```
public class User {
    private Set billingDetails;
    ...
}
public class BillingDetails {
    private Set users;
    ...
}
```

```
create table USER_BILLING_DETAILS (
    USER_ID bigint foreign key references USERS,
    BILLING_DETAILS_ID bigint foreign key references BILLING_DETAILS,
    PRIMARY KEY (USER_ID, BILLING_DETAILS_ID)
)
```

The granularity problem

- Address is broken down to street, city, etc.
- In OO, It is natural to have an Address class
- Classes have multi levels of granularity
  - User: coarse-grained
  - Address: finer-grained
  - Zipcode(String): simple type
- RDBMS just has two levels of granularity visible
  - tables such as USERS
  - columns such as ADDRESS_ZIPCODE.

```
create table USERS (
   USERID varchar(15) not null primary key,
   NAME varchar(50) not null,
   ADDRESS_STREET varchar(50),
   ADDRESS_CITY varchar(15),
   ADDRESS_STATE varchar(15),

   ADDRESS_ZIPCODE varchar(5),
   ADDRESS_COUNTRY varchar(15)
)
```



Object identity

- Object 'sameness'
  - object identity: Objects are identical if they occupy the same memory location in the JVM. This can be checked by using the == operator.
  - Object equality: Objects are equal if they have the same value, as defined by the equals (Object o) method. Classes that don't explicitly override this method inherit the implementation defined by java.lang.Object, which compares object identity.
- Database identity:
  - Objects stored in a relational database are identical if they represent the same row, or
  - equivalently, if they share the same table and primary key value.

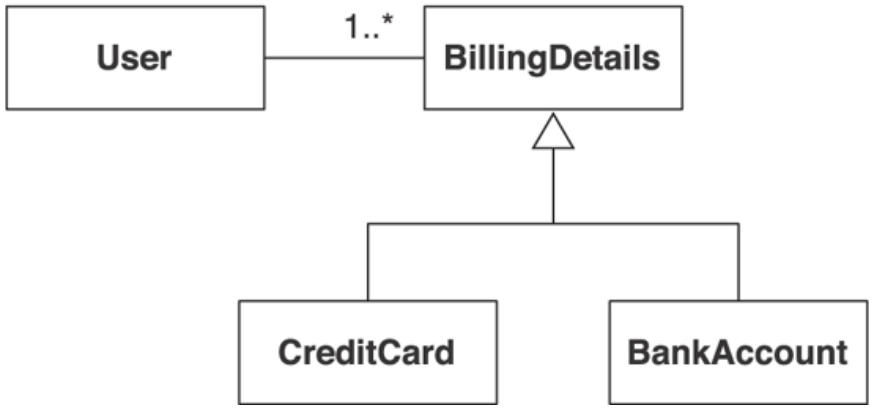
Object identity

- Primary keys are often system-generated
  - E.g. The USER_ID and BILLING_DETAILS_ID columns contain system-generated values.
- These columns were introduced purely for the benefit of the data model
- how should they be represented in the domain model?

```
create table USERS (
   USER_ID bigint not null primary key,
   USERNAME varchar(15) not null unique,
   NAME varchar(50) not null,
   ...
)
create table BILLING_DETAILS (
   BILLING_DETAILS_ID bigint not null primary key,
   ACCOUNT_NUMBER VARCHAR(10) not null unique,
   ACCOUNT_NAME VARCHAR(50) not null,
   ACCOUNT_TYPE VARCHAR(2) not null,
   USER_ID bigint foreign key references USER
)
```

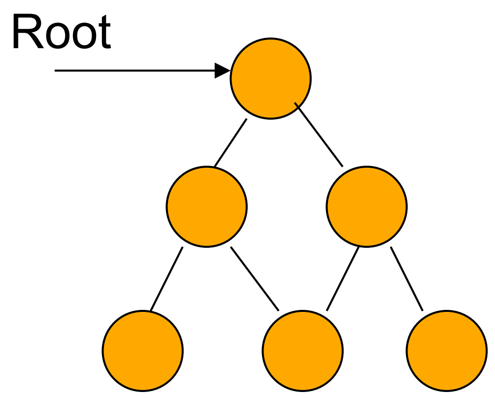
Subtype problem

- Inheritance problem:
  - RDBMS products don't support "table inheritance"
  - There is no standard way to map the class hierarchy to tables
- Polymorphism problem
  - User class associates to BillingDetails, which has two subclasses
  - It is a polymorphic association
  - The corresponding query may be also polymorphic, which has no obvious solution in DB.

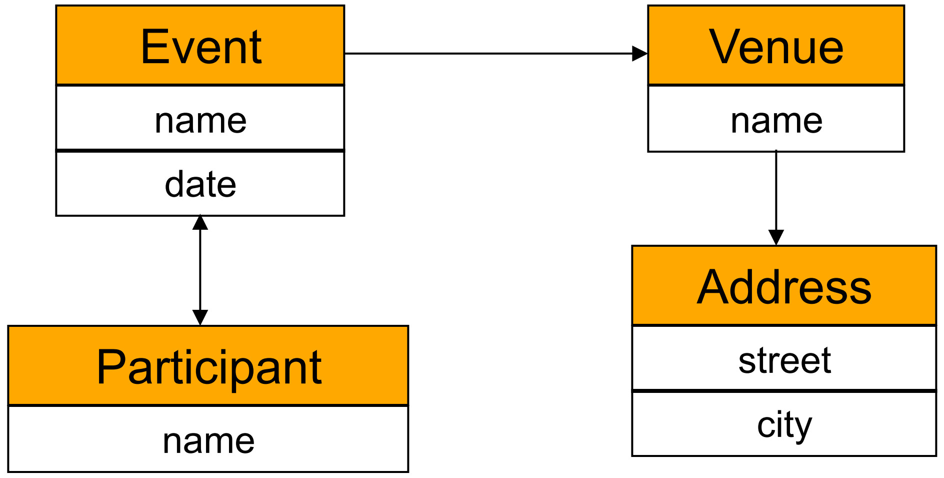


Object navigation problem

- In an object graph usually there's roots, where navigation starts.



Object navigation problem



```
event.getVenue().getAddress().getStreet();
```

```
SELECT street FROM Addresses WHERE AddressId=
(SELECT VenueAddress FROM Venues WHERE VenueId=
(SELECT EventVenue FROM Events WHERE EventId=1));
```

The cost of mismatch

- the main purpose of up to 30 percent of the Java application code written is to handle the tedious SQL/JDBC and manual bridging of the object/relational paradigm mismatch.

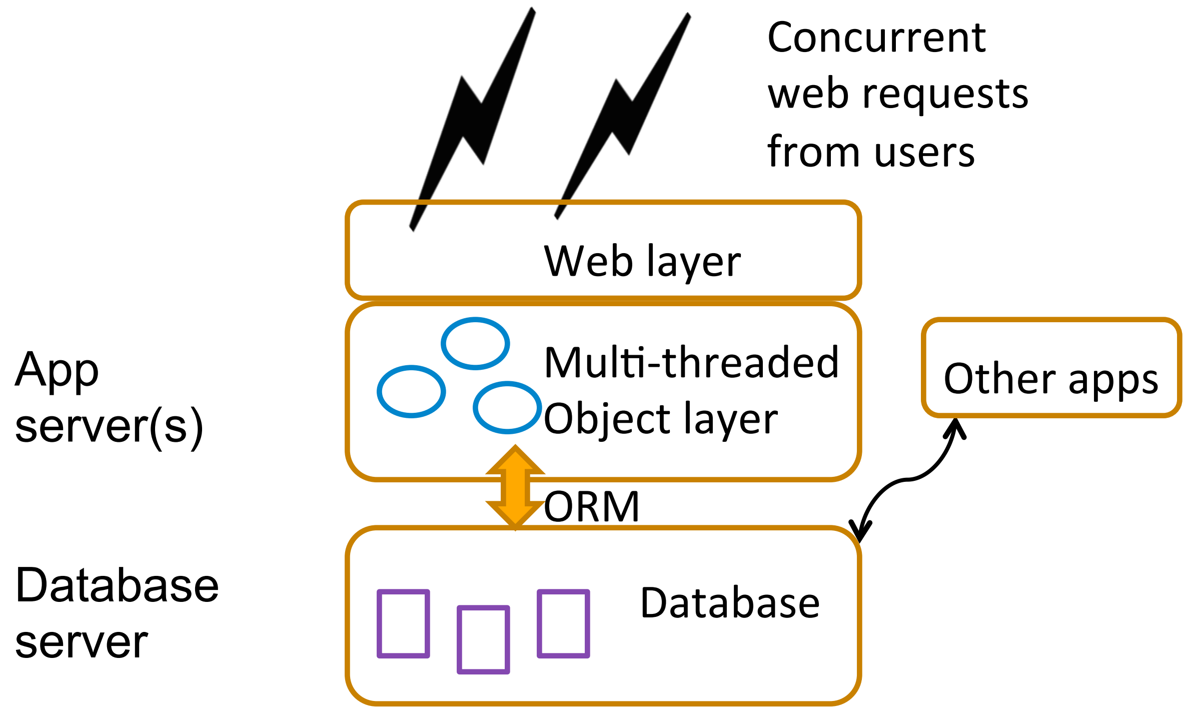
- There should be a systematic way

What is ORM (object/relational mapping)

- automated (and transparent) persistence of objects in an application to the tables in a relational database, using metadata that describes the mapping between the objects and the database.
- works by (reversibly) transforming data from one representation to another. This implies certain performance penalties
- An ORM solution consists of the following four pieces:
  - An API for performing basic CRUD (create, read, update, delete) operations on objects of persistent classes;
  - A language or API for specifying queries that refer to classes and properties of classes;
  - A facility for specifying mapping metadata
  - A technique for the ORM implementation to interact with transactional objects to perform dirty checking, lazy association fetching, and other optimization functions
- Has a long history, but widely adopted only since 2001.

Current ORM Impetus: the web app

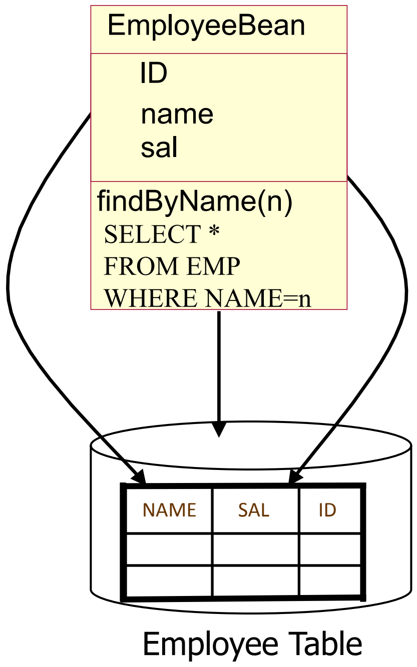
- A web app, with its multi-threaded object layer, particularly needs help with the correct handling of persistent data


Concurrent web requests from users
Web layer
App server(s)
Multi-threaded Object layer
Other apps
ORM
Database server
Database

Mapping objects to Tables

- Simple case
  - Map entities to tables
  - Map attributes (with primitive data types) in an entity bean to columns in the table
- Synchronizing the object with the table in EJB 2:

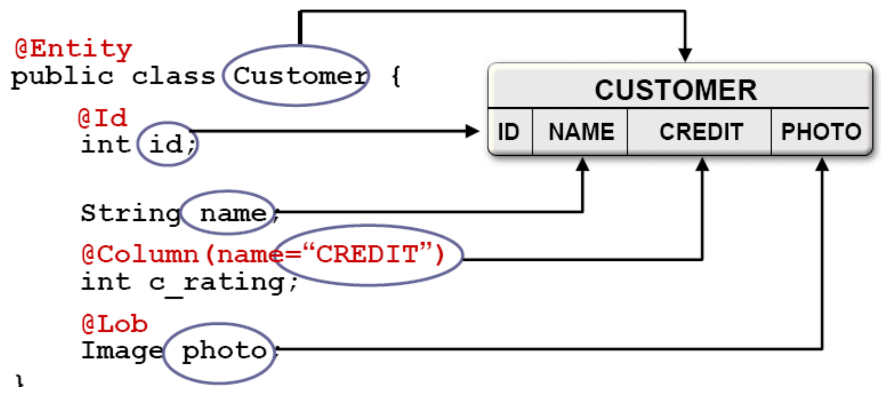
```
ejbLoad:
  SELECT name, sal FROM employee WHERE id=?
ejbStore:
   UPDATE employee SET name=?, sal=? WHERE id=?
ejbPostCreate:
  INSERT INTO employee VALUES(?,?,?)
ejbRemove:
  DELETE FROM employee WHERE id=?
```



Object attributes and table columns

- Not all attributes are mapped to table columns
- Describe the mapping using deployment descriptor using XML (or using annotation in EJB3.0)

```
<enterprise-beans>
    <entity>
        <ejb-name>Employee</ejb-name>
        <cmp-field> name </cmp-field>
        <cmp-field> sal     </cmp-field>
        … …
    </entity>
…
  </enterprise-beans>
```



```
@Entity
public class Customer {
    @Id
    int id;

    String name;
    @Column(name="CREDIT")
    int c_rating;
    @Lob
    Image photo;
}
```

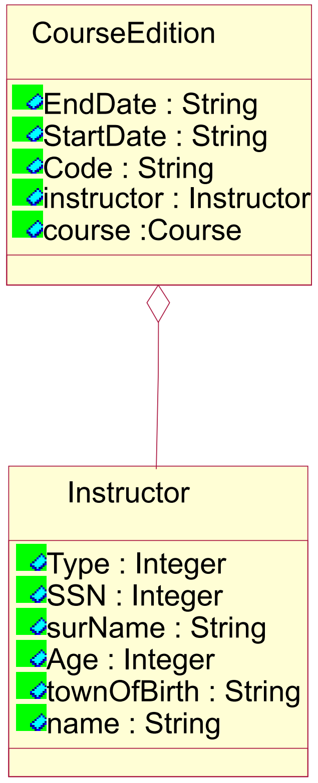
- What if the attribute in an entity bean is an entity bean itself?
  - E.g., `Employee` bean may have `Address` attribute
  - It is a relations between objects

Relationships between objects

- Aggregation
  - Mapping strategies
    - Single table aggregation
    - Foreign key aggregation
  - Manage cardinalities
  - Manage many-to many relations
  - Manage composition
- Inheritance
  - One table for one inheritance tree
  - One table for one class
  - One table for one inheritance path

Single Table Aggregation Pattern(1/4)

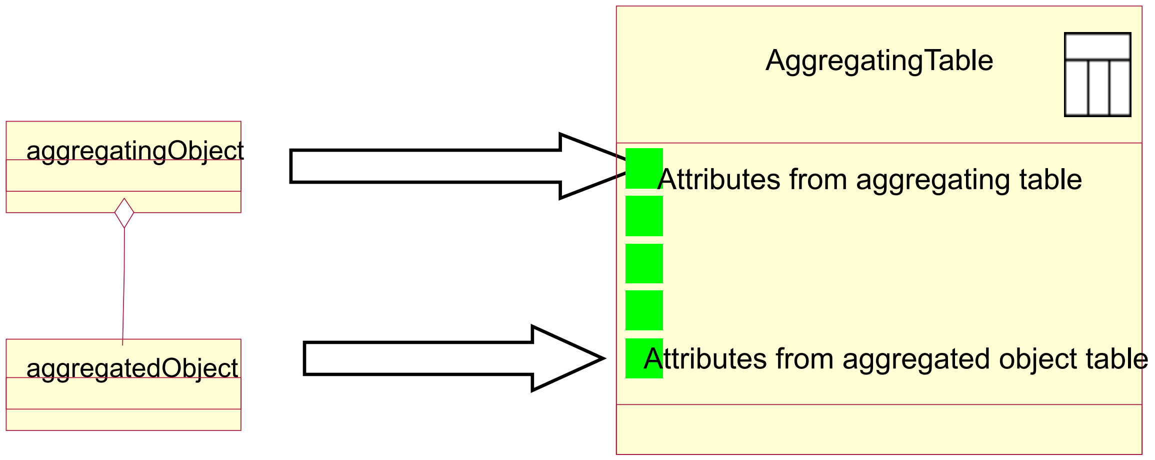
- **Abstract:** map aggregation to a relational data model by integrating all aggregated objects' attributes into a single table.
- **Example:** `CourseEdition` has `Instructor` attribute



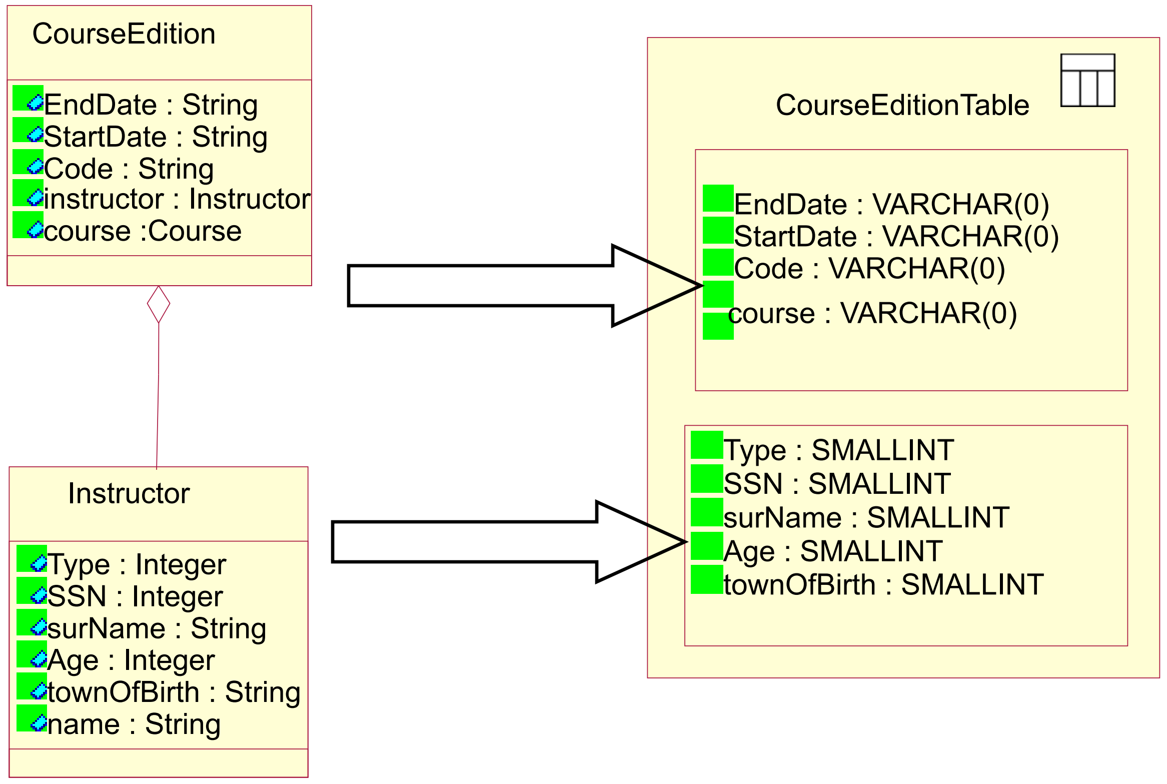
- Design pattern: a general repeatable solution to a commonly-occurring problem in software design

Single Table Aggregation(2/4)

- **Solution**: Put the aggregated object's attributes into the same table as the aggregating object's.


aggregatingObject
aggregatedObject
AggregatingTable
Attributes from aggregating table
Attributes from aggregated object table

Single Table Aggregation(3/4)

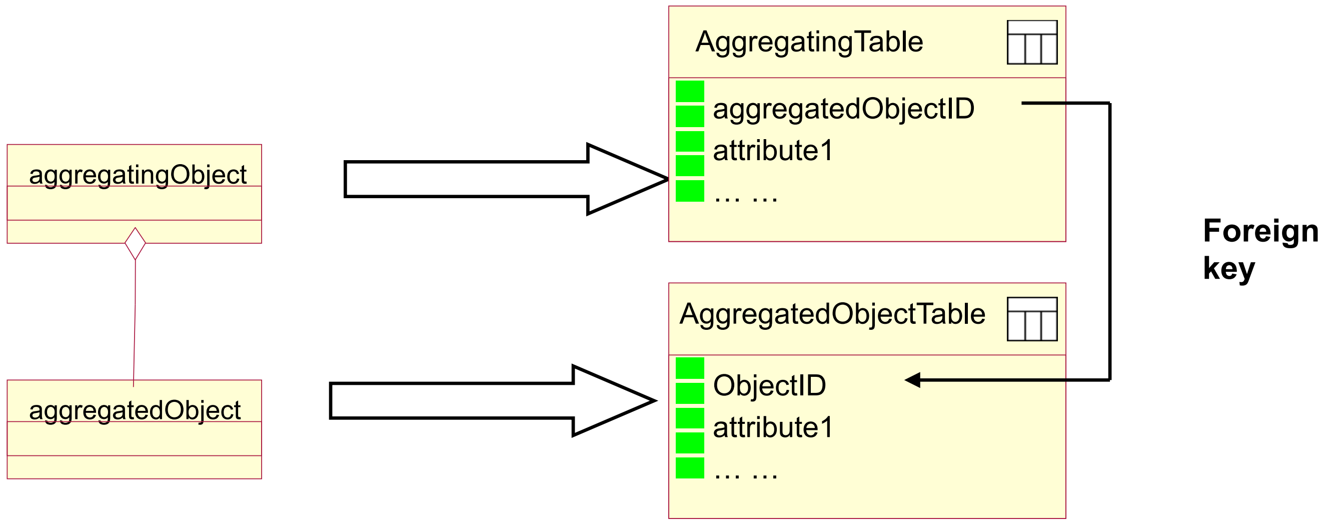


Single Table Aggregation(4/4) Consequences

- *Performance:*
  - Pro: only one table needs to be accessed. No join operation is needed.
  - Con: the table may be too big with many duplicate cells.
- *Maintenance and flexibility:*
  - If an aggregated type occurs in multiple classes/tables, a change in the aggregated class needs to be handled for all the tables.
  - E.g., Both Course and HumanResource tables contain Instructor class.
  - A change in Instructor class ripples to all the aggregating classes
- *Consistency of the database:*
  - Aggregated objects are automatically deleted on deletion of the aggregating objects.
  - e.g. The deletion of Course table/rows causes the deletion of Instructor infor
- *Ad-hoc queries*:
  - E.g., If you want to form a query that scans all `Instructor` objects in the database, this is very hard to formulate.

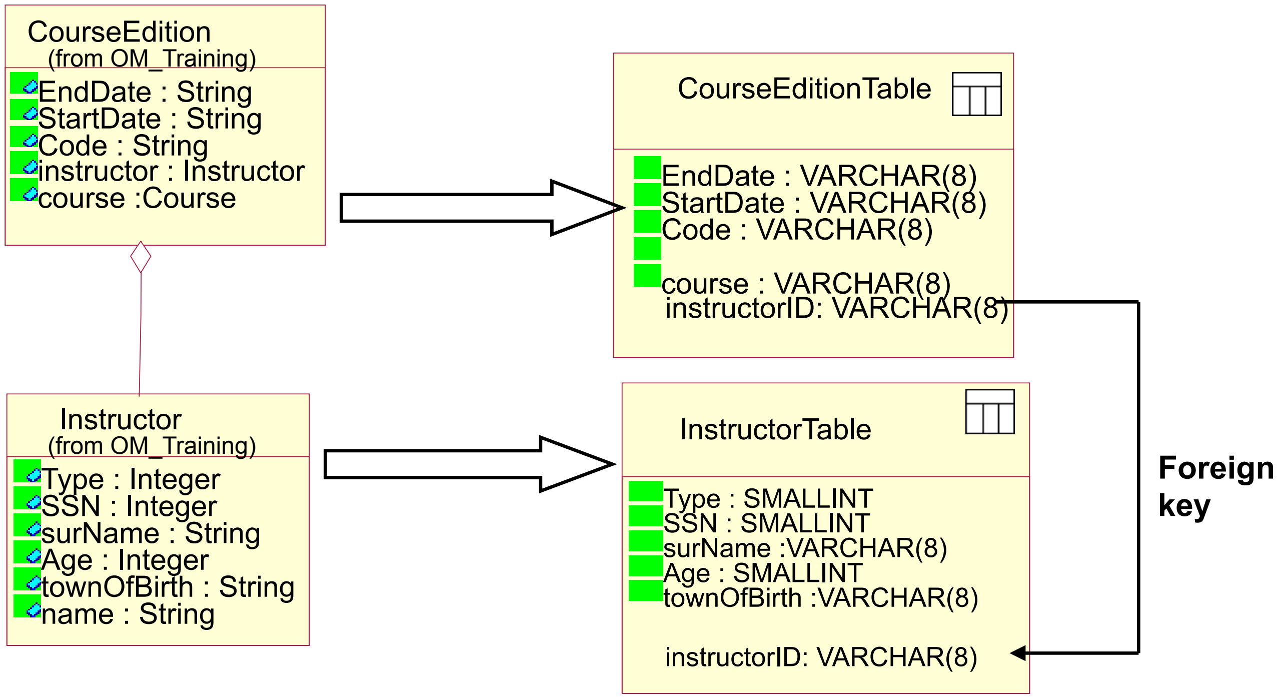
Foreign Key Aggregation (1/3)

- **Abstract:** The pattern shows how to map aggregation to a relational data model using foreign keys.
- **Solution:** Use a separate table for the aggregated type.
  - Insert an synthetic object identity into the table
  - use this object identity in the table of the aggregating object to make a foreign key link to the aggregated object.



Foreign Key Aggregation (2/3)

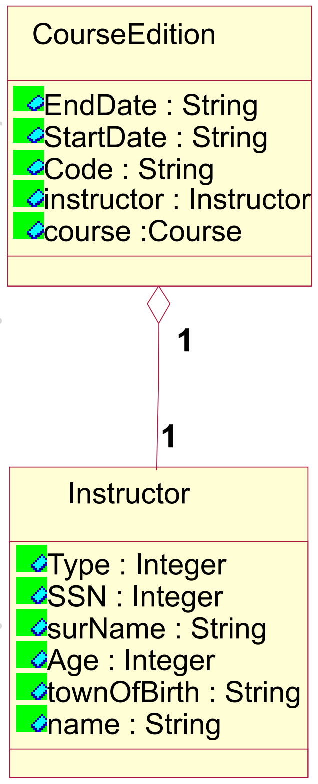
- **Example:** `Instructor` and `CourseEdition` are mapped to two tables



Foreign Key Aggregation: Consequences

- *Performance:*
  - needs a join operation. at least two database accesses
  - *Single Table Aggregation* needs a single database operation.
  - If accessing aggregated objects is a statistical rare case this is acceptable.
  - If the aggregated objects are always retrieved together with the aggregating object, you have to have a second look at performance here.
- *Maintenance:*
  - Factoring out objects like `Instructor` into tables of their own makes them easier to maintain and hence makes the mapping more flexible.
- *Consistency of the database:*
  - Aggregated objects are not automatically deleted on deletion of the aggregating objects.
- *Ad-hoc queries:* Factoring out aggregated objects into separate tables allows easy querying these tables with ad-hoc queries.

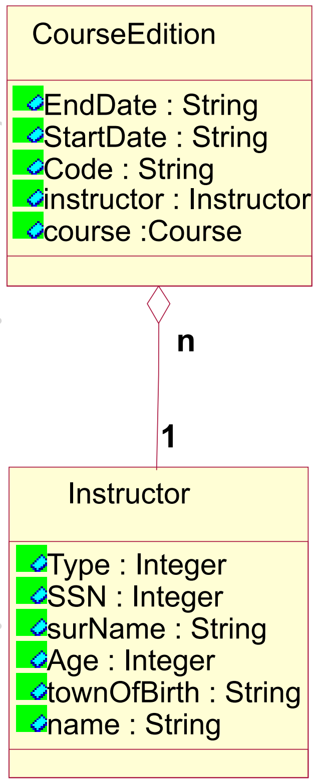
Specify relations using deployment descriptor



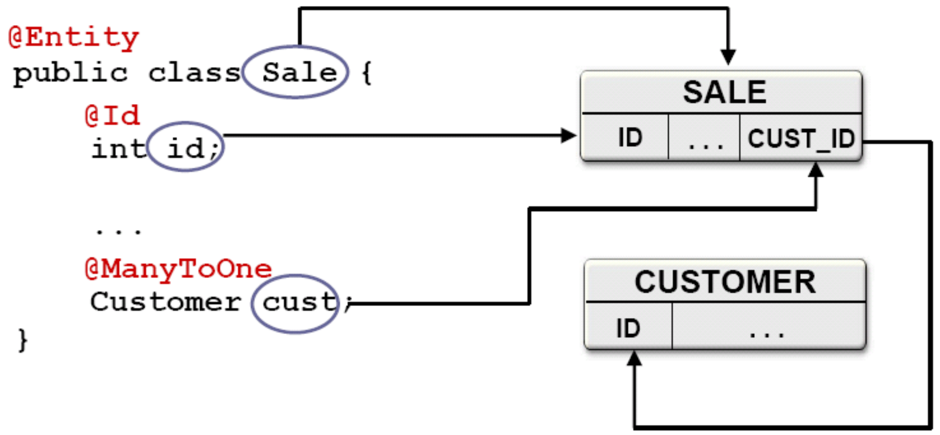
```
<enterprise-beans>
    <entity>
      <ejb-name>CourseEdition</ejb-name>
        …
    </entity>
    <entity>
      <ejb-name>Instructor</ejb-name>
        …
    </entity>
    …
  </enterprise-beans>
```

```
<ejb-relation>
   <ejb-relation-name>Instructor-CourseEdition</ejb-relation-name>

   <ejb-relationship-role>
         <ejb-relationship-role-name> Instructor-For </ejb-relationship-role-name>
         <multiplicity>One</multiplicity> … …
         <relationship-role-source> <ejb-name> Instructor </ejb-name>  </relationship-role-source>
   </ejb-relationship-role>

   <ejb-relationship-role>
         <ejb-relationship-role-name>   Courses  </ejb-relationship-role-name>
         <multiplicity> One </multiplicity>
         <relationship-role-source><ejb-name> CourseEdition </ejb-name></relationship-role-
source>
   </ejb-relationship-role>
 </ejb-relation>
```

Manage Cardinality: 1 to many relationships



```
<ejb-relation>
    <ejb-relation-name>Instructor-CourseEdition</ejb-relation-name>

    <ejb-relationship-role>
        <ejb-relationship-role-name> Instructor-For </ejb-relationship-role-name>
        <multiplicity> Many </multiplicity>
        <relationship-role-source> <ejb-name> Instructor </ejb-name>  </relationship-role-source>
    </ejb-relationship-role>

    <ejb-relationship-role>
        <ejb-relationship-role-name>   Courses  </ejb-relationship-role-name>
        <multiplicity> One </multiplicity> … …
        <relationship-role-source> <ejb-name>  CourseEdition  </ejb-name>
        </relationship-role-source>
        <cmr-field> <cmr-field-name> courses </cmr-field-name>
                    <cmr-field-type> java.util.Collection </cmr-field-type>
        </cmr-field>
    </ejb-relationship-role>
</ejb-relation>
```



```
@Entity
public class Sale {
    @Id
    int id;

    ...
    @ManyToOne
    Customer cust;
}
```

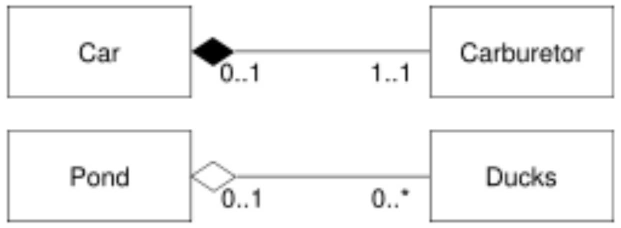
Manage cardinality: many to many relationships

- **Example:** A `Trainee` may register several `CourseEditions`, and one `CourseEdition` has a number of `Trainees`.
- **Solution:** Create a separate table containing the object identifiers (or Foreign Keys) of the two object types participating in the association. Map the rest of the two object types to tables using any other suitable mapping pattern.

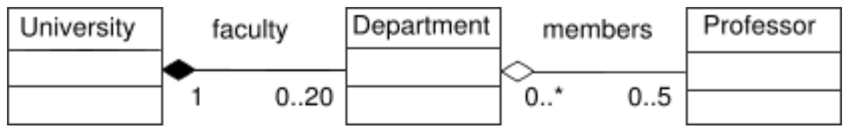
```
<ejb-relation>
      <ejb-relation-name> Trainee-CourseEdition</ejb-relation-name>

      <ejb-relationship-role>
          <ejb-relationship-role-name> Trainee-EnrollIn-Courses </ejb-relationship-role-name>
          <multiplicity> Many </multiplicity>
          <relationship-role-source> <ejb-name> Instructor </ejb-name>  </relationship-role-source>
          <cmr-field> <cmr-field-name> courses </cmr-field-name>
                      <cmr-field-type> java.util.Collection </cmr-field-type>
           </cmr-field>
      </ejb-relationship-role>

      <ejb-relationship-role>
          <ejb-relationship-role-name>   Courses-HaveEnrolled-Trainees  </ejb-relationship-role-name>
          <multiplicity> Many </multiplicity>
          <relationship-role-source> <ejb-name> CourseEdition </ejb-name> </relationship-role-source>
          <cmr-field> <cmr-field-name> trainees </cmr-field-name>
                      <cmr-field-type> java.util.Collection </cmr-field-type>
           </cmr-field>
      </ejb-relationship-role>
 </ejb-relation>
```

Object aggregation and composition

- Aggregation and composition
  - Composition is a stronger form of aggregation.
  - In composition, when deleting the owning objects, the contained object will also be deleted.
  - In UML, composition relation is drawn as filled diamond, while aggregation as unfilled diamond.
- Examples:
  - When a car is destroyed, so is its carburetor. When a pond is destroyed, the ducks are still alive.



  - When a university closes, the departments will be closed as well. However, data about professors should still be there.



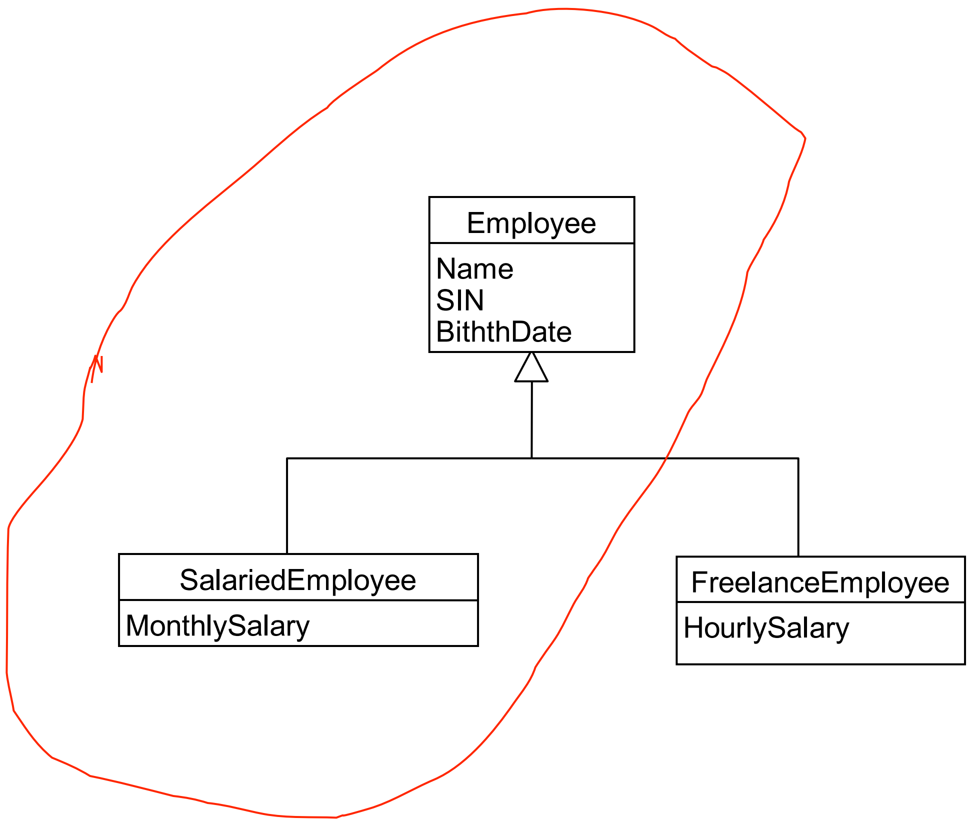
Manage composition in EJB 2.0

- Use cascaded delete in deployment descriptor, to indicate that deletion operation on `Instructor` is cascaded down to associated `Telephone` objects.

```
<ejb-relation>
    <ejb-relation-name>Instructor-Telephone</ejb-relation-name>

    <ejb-relationship-role>
        <multiplicity>One</multiplicity> … …
        <relationship-role-source> <ejb-name> Instructor </ejb-name>  </
  relationship-role-source>
    </ejb-relationship-role>

    <ejb-relationship-role>
        <multiplicity>Many</multiplicity>  <cascade-delete/>  … …
        <relationship-role-source> <ejb-name>  Telephone  </ejb-name>   </
  relationship-role-source>
     </ejb-relationship-role>
 </ejb-relation>
```

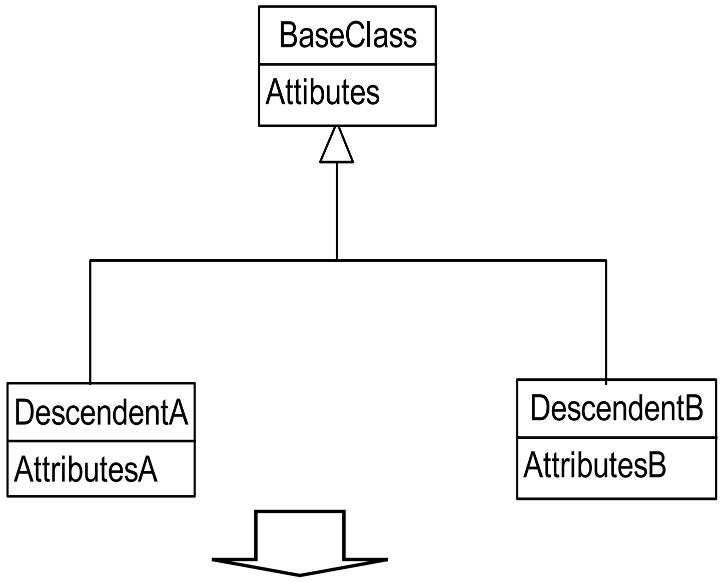
Mapping inheritance

- Strategies of mapping inheritance to tables:
  - One table for the inheritance tree
  - One table for each class
  - One table for each inheritance path



One table for the inheritance tree

- Solution: use the union of all attributes of all objects in the inheritance hierarchy as the columns of a single database table.



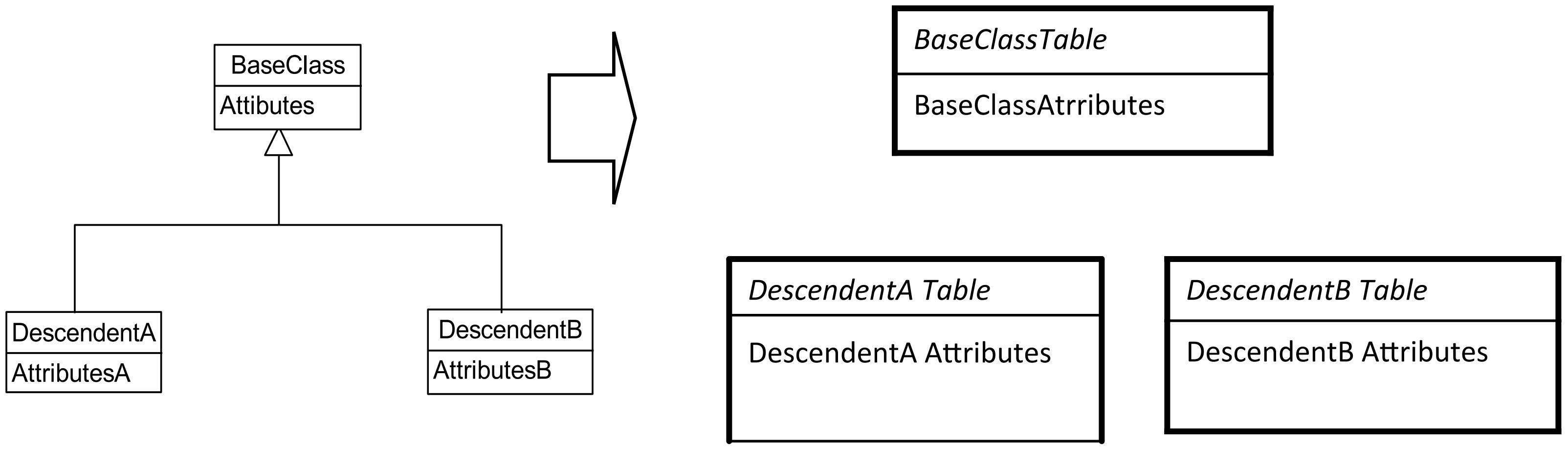
	BaseClass Attributes	DescendantA Attributes	DescendantB Attributes
Base class instance	Attribute Values from Base	Null Values	Null Values
DescendantA instance	Attribute Values from Base	Attribute Values from A	Null Values
DescendantB instance	Attribute Values from Base	Null values	Attribute Values from B

One table for the inheritance tree

- Consequences
  - **Write/update performance**: Reading/writing any objects in the hierarchy with one database operation
  - **Space consumption**: Requires more space to store the objects

One table for each class

- Solution
  - Map the attributes of each class to a separate table

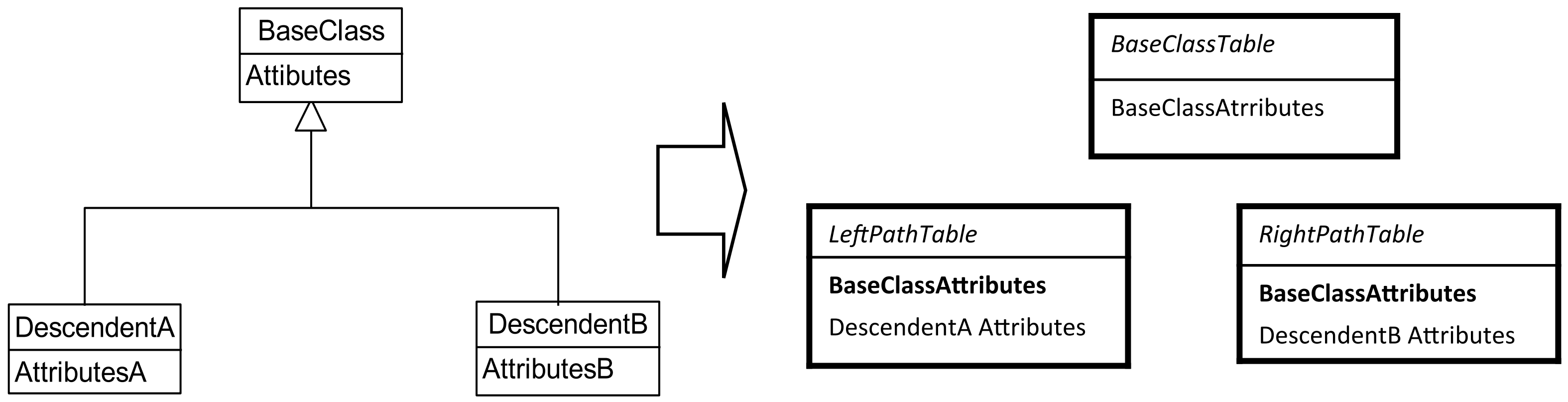


One table for each class

- Consequences
  - **Write and update performance**: More database operations involved
  - **Space consumption**: has near optimal consumption
  - **Maintenance cost:** As the mapping is straightforward and easy to understand, schema evolution is straightforward and easy.

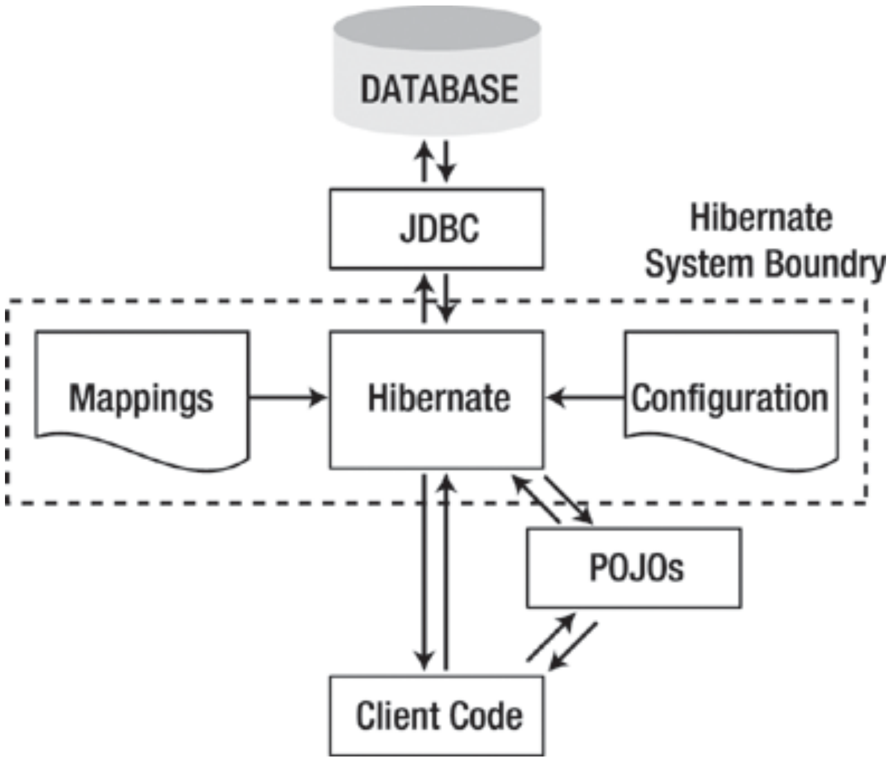
One table for each inheritance path

- Solution:
  - Map attributes of each class to a separate table. add all inherited attributes.
  - If `BaseClass` is abstract, `BaseClassTable` is not generated.
- Consequences
  - **Write and update performance**: One database operation to read or write an object
  - **Maintenance**: Adding or deleting attributes of a superclass results in changes to the tables of all derived classes.

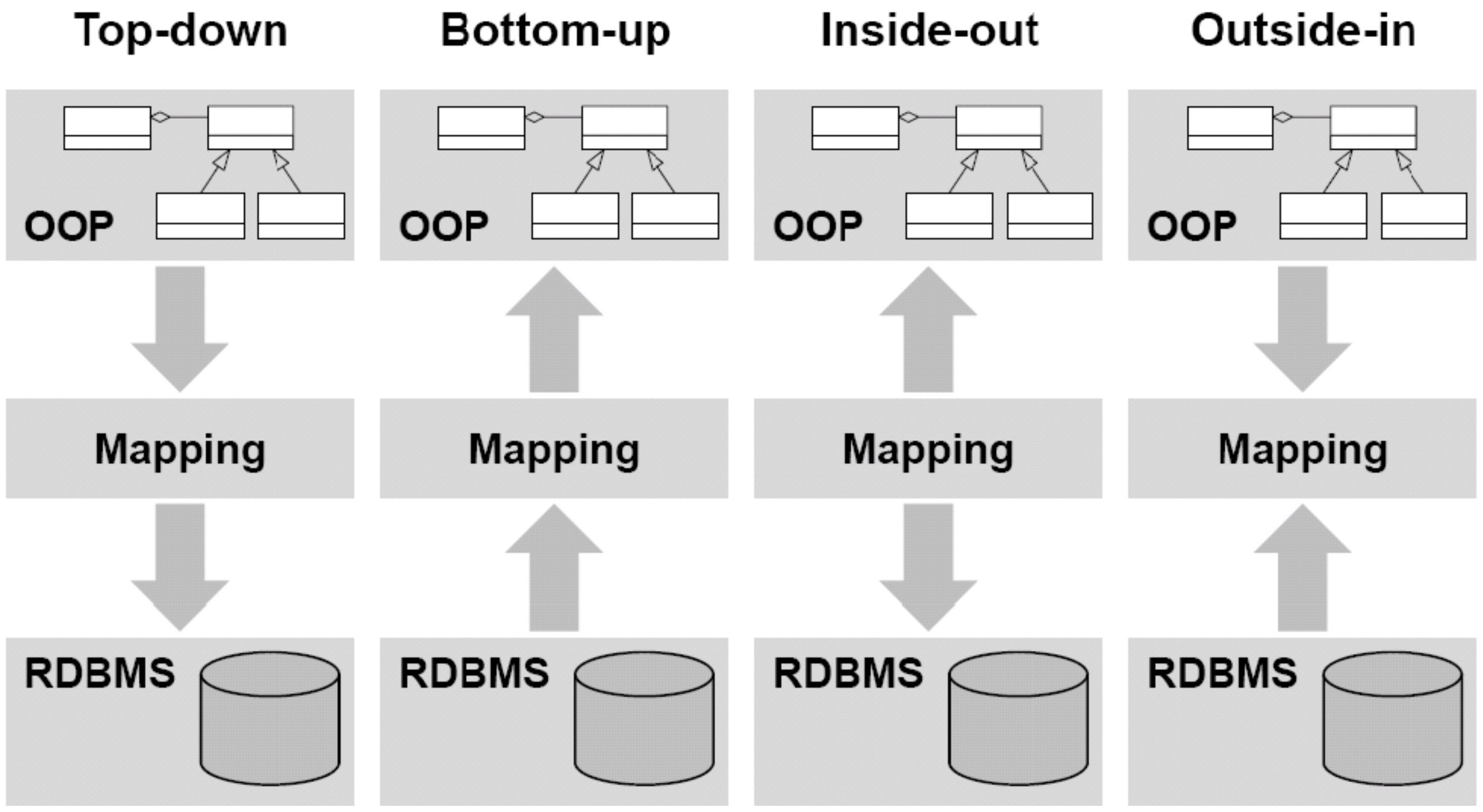


ORM tools

- Map object-oriented domain model to relational database
- Free developer of persistence-related programming task
- Hibernate
  - maps Java types to SQL types
  - transparent persistence for classes meeting certain requirements
  - generates SQL for more than 25 dialects behind the scenes
  - provides data query and retrieval using either HQL or SQL
  - can be used stand-alone with Java SE or in Java EE applications
- Java Persistence API (JPA)
  - Enterprise Java Beans Standard 3.0
  - introduced annotations to define mapping
  - javax.persistence package



Mapping strategies


Top-down
Bottom-up
Inside-out
Outside-in
OOP
OOP
OOP
OOP
Mapping
Mapping
Mapping
Mapping
RDBMS
RDBMS
RDBMS
RDBMS