

Object-Oriented Databases

Object Persistence

- Object-Relational Mappings and Frameworks
- Serialisation
- Persistent Programming Languages



Principles of Persistence

- Data has to outlive the execution of the program
- Independence
 - persistence of data object independent of how the program manipulates that data object
- Data type orthogonality
 - all data types should be allowed the full range of persistence
- Identification
 - choice of how to provide and identify persistence at language level independent of choice of data objects in language
- Implicitness
 - data does not have to be moved or copied to be made persistent

Uniformity

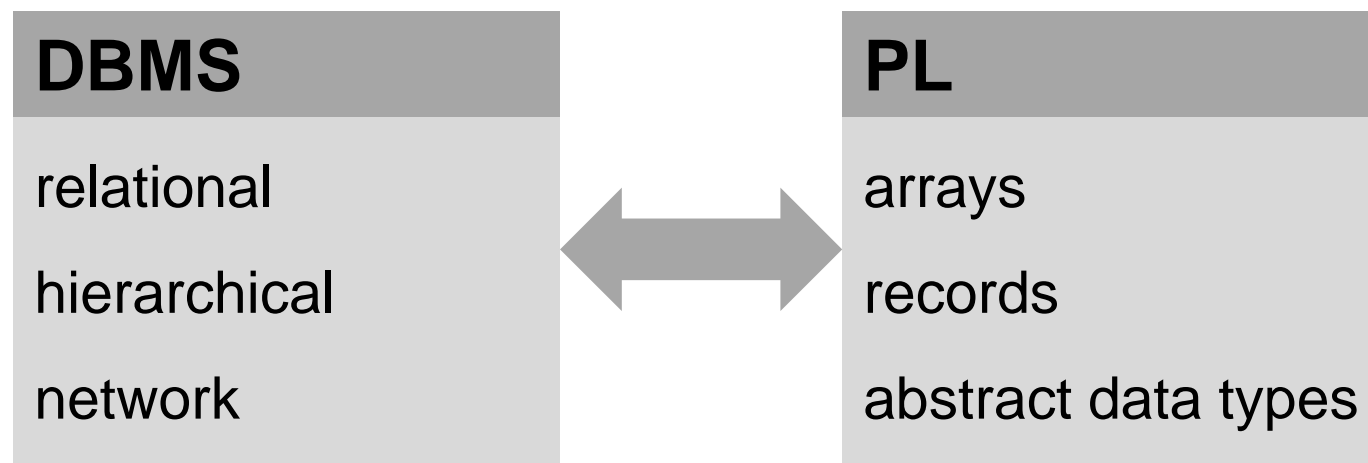
- Treat data values uniformly, independent of
 - longevity
 - size
 - type
- Achieve uniformity for all aspects of system services
 - data definition
 - operations
 - integrity
 - concurrency control
 - distribution

Range of Persistence

1	Transient results in expression evaluation
2	Local variables
3	Global variables and heap items
4	Data that lasts a whole execution of a program
5	Data that lasts for several executions of several programs
6	Data that lasts for as long as a program is being used
7	Data that outlives a successions of versions of such a program
8	Data that outlives versions of the persistent support system

Traditional Programming Languages

- Facilities for the manipulation of data whose lifetime does not extend beyond activation of the program
- Storage of data requires mapping to and from files or DBMS



Disadvantages

- Effort to understand and manage mappings from program data to stored data
 - IBM Report (1978)
«30% of application code is concerned with transferring data to and from files or DBMS»
- Data type protection of programming language system often lost in the mapping

Databases and Programming Languages

- Database and programming languages communities research and develop products independently despite having to provide many similar services
- Database focus
 - preserve large volumes of data reliably
 - support many processes operating against data efficiently
- Programming language focus
 - help programmers be precise
 - make programs understandable

Databases and Programming Languages

- Separate development and consequent inconsistencies tend to perpetuate and grow
- Intellectual and software investment in each camp goes against adoption of other's ideas
- View of database from programming language
 - Mess of incomprehensible ad hoc design
- View of programming language from database
 - Programming languages unhelpful with real problems such as bulk types, persistence, concurrency and transactions

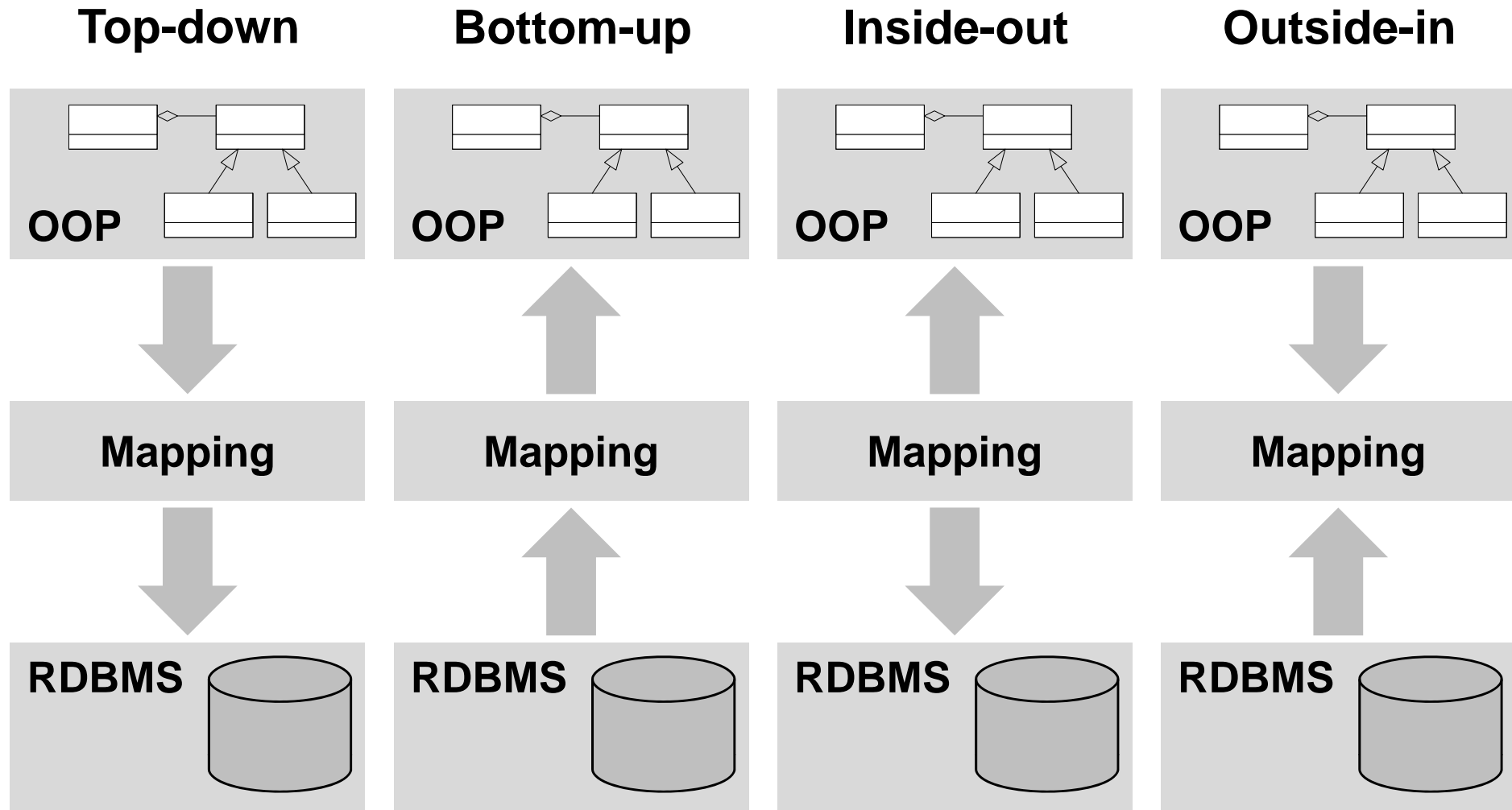
Two Approaches

- Glue current underlying technologies together
 - “glue-ware”, e.g. object-relational mappings and frameworks
 - hide technologies behind sufficient “standard” interface
 - underlying differences in semantics ultimately show through
- Complete computational environments
 - Java object serialisation
 - persistent programming languages

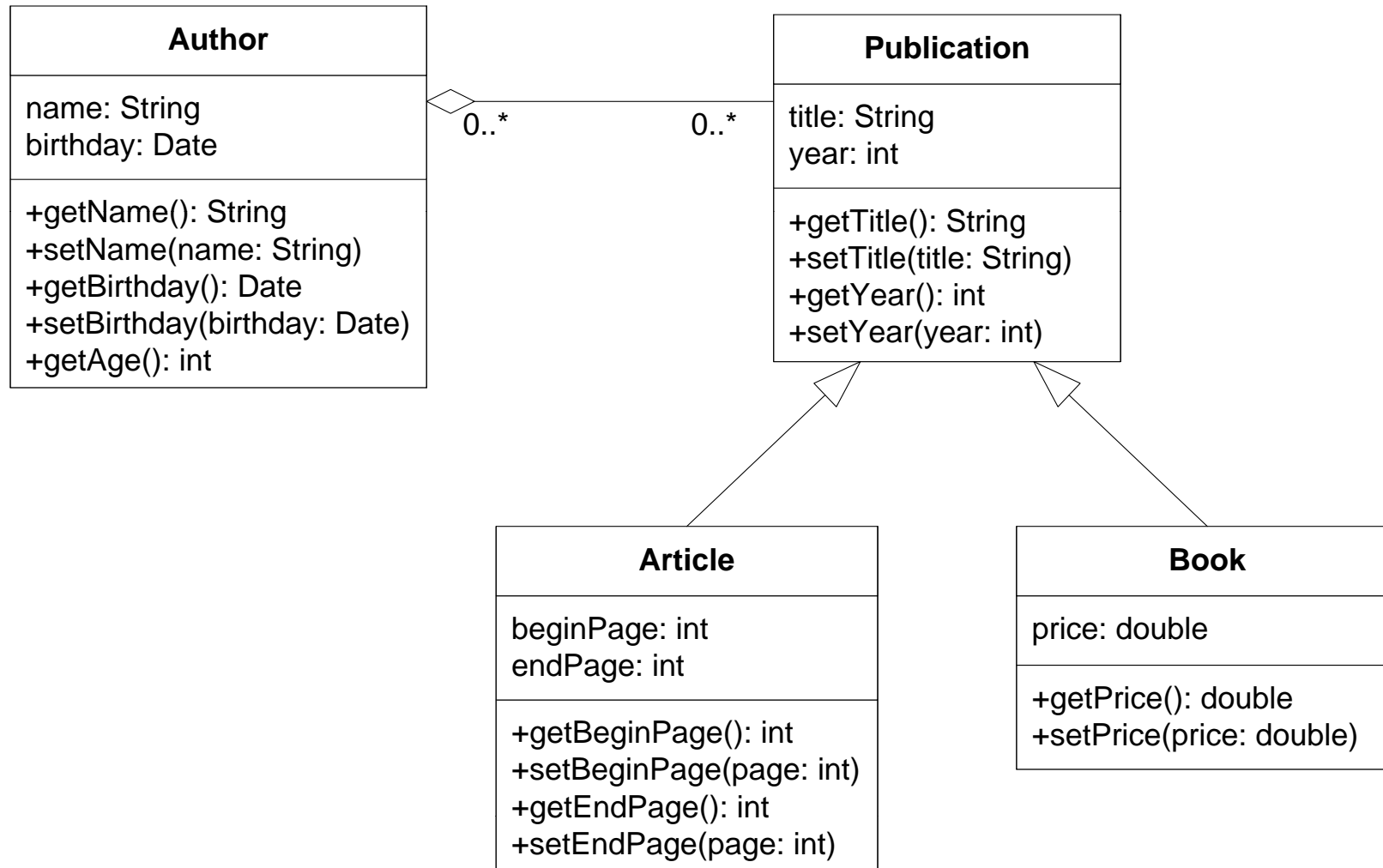
Object-Relational Mappings

- 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

Designing a Object-Relational Mapping



Example Class Hierarchy



Mapping Classes

```
public class Author {  
  
    private long id;  
    private String name;  
    private Date birthday;  
    private Set<Publication> publications;  
  
    /**  
     * No-argument constructor is a required by Hibernate.  
     */  
    Author() { }  
  
    public Author(String name) {  
        this.name = name;  
        this.publications =  
            new HashSet<Publication>();  
    }  
    ...  
}
```

Mapping Classes

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

<hibernate-mapping>
  <class name="ch.ethz.globis.oodb.hibernate.domain.Author"
    table="AUTHORS">
    <id name="id" column="AUTHOR_ID">
      <generator class="native" />
    </id>
    <property name="name" />
    <property name="birthday" />
    <set name="publications" table="AUTHORSPUBLICATIONS" cascade="all">
      <key column="AUTHOR_ID" />
      <many-to-many column="PUBLICATION_ID"
        class="ch.ethz.globis.oodb.hibernate.domain.Publication" />
    </set>
  </class>
</hibernate-mapping>
```

Mapping Associations

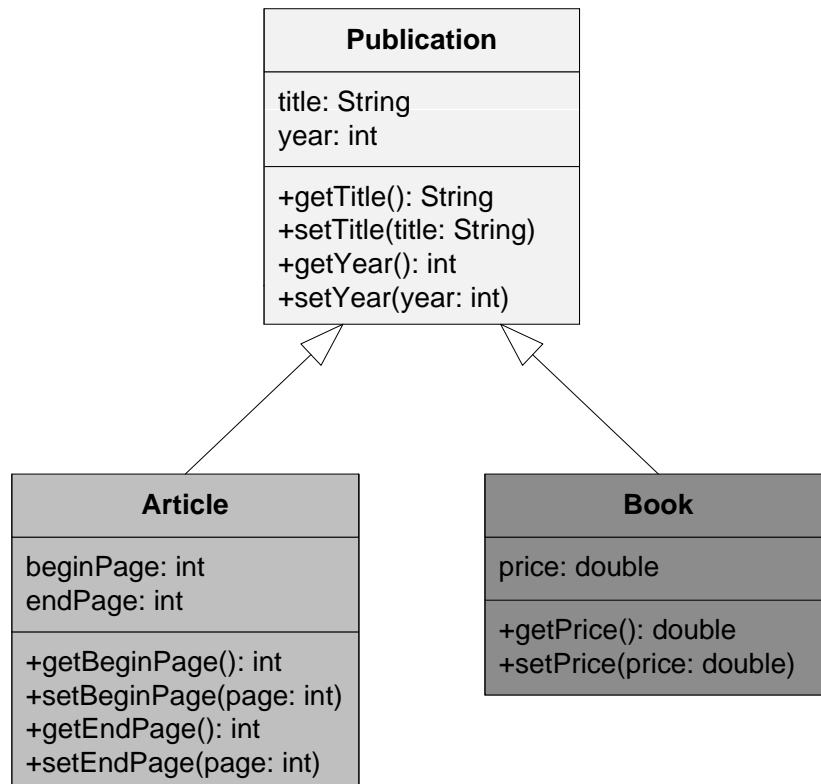
- Unidirectional and bidirectional associations
- Unordered and ordered associations
- Association cardinality types
 - one-to-one
 - many-to-one and one-to-many
 - many-to-many
- Join Tables to map complex associations

```
CREATE TABLE AUTHOR(AUTHOR_ID BIGINT NOT NULL PRIMARY KEY, ...)  
CREATE TABLE AUTHORS PUBLICATIONS(  
    AUTHOR_ID BIGINT NOT NULL,  
    PUBLICATION_ID BIGINT NOT NULL,  
    PRIMARY KEY(AUTHOR_ID, PUBLICATION_ID))  
CREATE TABLE PUBLICATION(PUBLICATION ID BIGINT NOT NULL PRIMARY KEY, ... )
```

Mapping Inheritance

- Multiple strategies to map inheritance
 - one table per class hierarchy
 - one table per subclass
 - one table per concrete class
- Mapping strategies can be mixed for different branches of an inheritance hierarchy
- Implicit polymorphism
 - one table per concrete class
 - common interface is not mentioned in the mapping
 - common properties are mapped in every table

Mapping Strategies



- **One Table Per Class Hierarchy**

P	D	A	B

- **One Table Per Subclass**

P	A	B

- **One Table Per Concrete Class**

P	A	P	B

One Table Per Class Hierarchy

```
<class name="Publication" table="PUBLICATION">
  <id name="id" type="long" column=" PUBLICATION_ID">
    <generator class="native"/>
  </id>
  <discriminator column="PUBLICATION_TYPE" type="string"/>
  <property name="title" column="TITLE"/>
  <property name="year" column="YEAR"/>
  <subclass name="Article" discriminator-value="ARTICLE">
    <property name="beginPage" column="BEGIN_PAGE"/>
    <property name="endPage" column="END_PAGE"/>
  </subclass>
  <subclass name="Book" discriminator-value="BOOK">
    <property name="price" column="PRICE"/>
  </subclass>
</class>
```

One Table Per Subclass

```
<class name="Publication" table="PUBLICATION">
  <id name="id" type="long" column="PUBLICATION_ID">
    <generator class="native"/>
  </id>
  <property name="title" column="TITLE"/>
  <property name="year" column="YEAR"/>
  <joined-subclass name="Article" table="ARTICLE">
    <key column="PUBLICATION_ID"/>
    <property name="beginPage" column="BEGIN_PAGE"/>
    <property name="endPage" column="END_PAGE"/>
  </joined-subclass>
  <joined-subclass name="Book" table="BOOK">
    <key column="PUBLICATION_ID"/>
    <property name="price" column="PRICE"/>
  </joined-subclass>
</class>
```

One Table Per Concrete Class

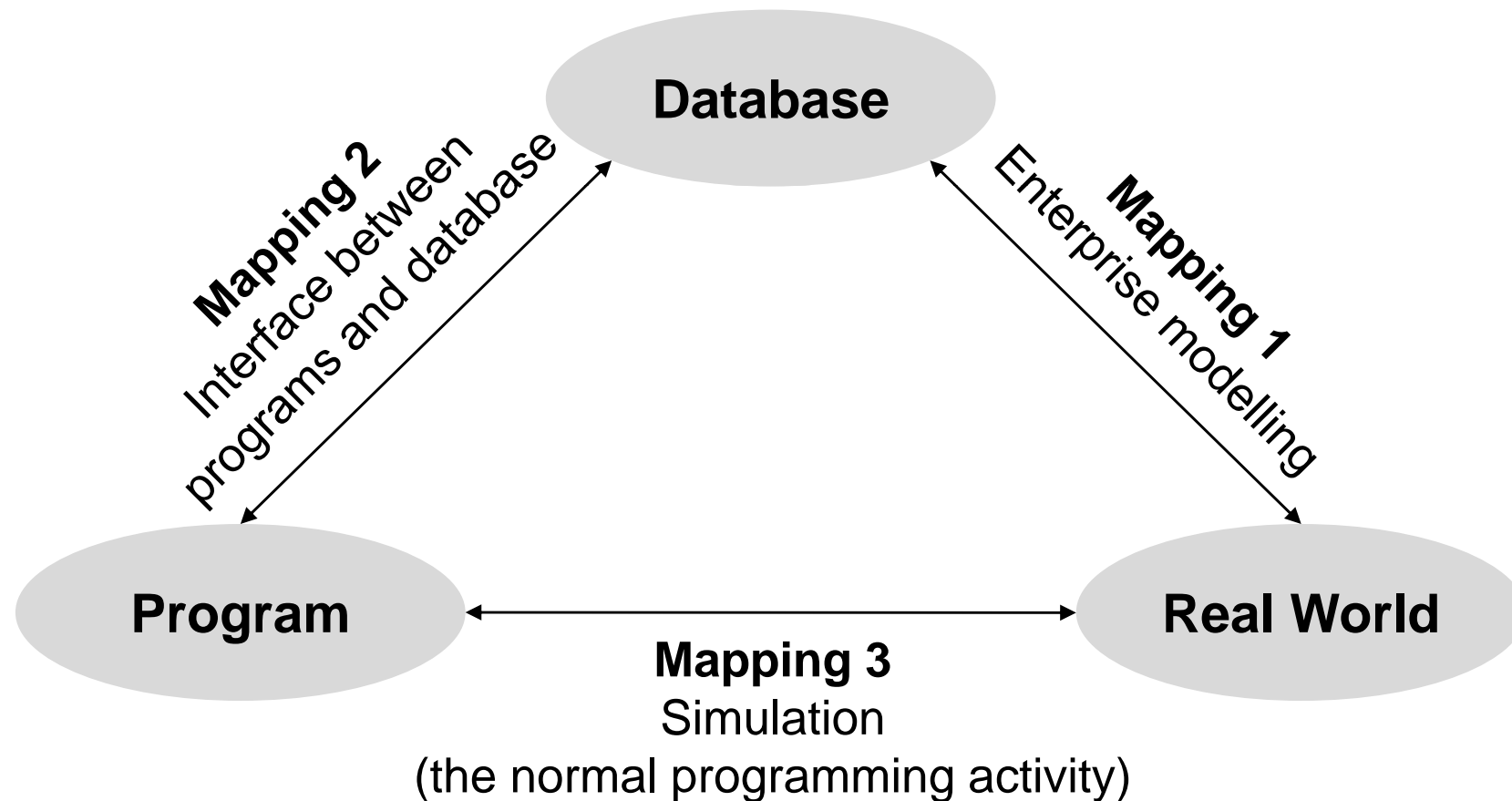
```
<class name="Publication">
  <id name="id" type="long" column="PUBLICATION_ID">
    <generator class="sequence"/>
  </id>
  <property name="title" column="TITLE"/>
  <property name="year" column="YEAR"/>
  <union-subclass name="Article" table="ARTICLE">
    <property name="beginPage" column="BEGIN_PAGE"/>
    <property name="endPage" column="END_PAGE"/>
  </union-subclass>
  <union-subclass name="Book" table="BOOK">
    <property name="price" column="PRICE"/>
  </union-subclass>
</class>
```

Using Annotations

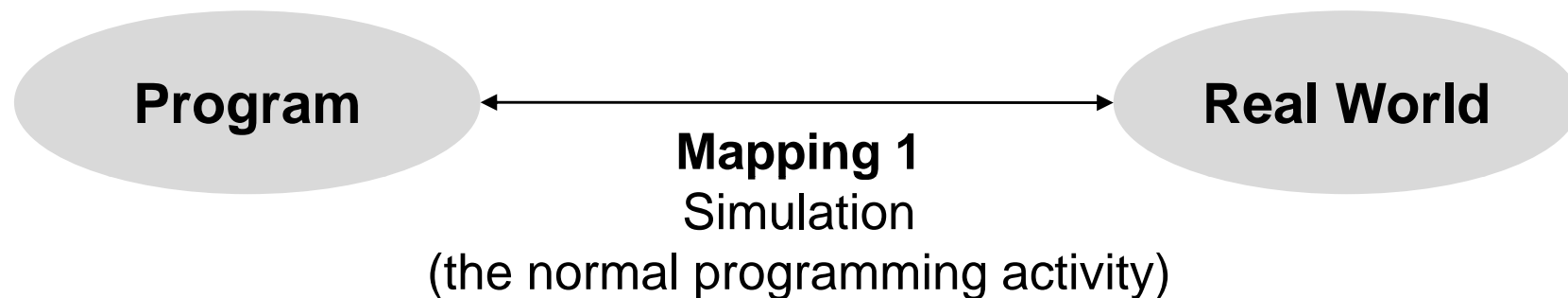
- Java annotations have been introduced in Java 5
- Enterprise Java Beans 3.0 includes Java Persistence API
- Uses Java annotations instead of XML descriptors to capture mappings
- Standardises object-relational mappings
- Hibernate implements JPA

```
public class Author {  
  
    @Id @GeneratedValue  
    private long id;  
    private String name;  
    private Date birthday;  
  
    @ManyToMany(fetch=FetchType.EAGER)  
    @JoinTable(  
        name="PUBLICATIONSAUTHORS",  
        joinColumns=@JoinColumn(  
            name="AUTHOR_ID",  
            referencedColumnName="id"),  
        inverseJoinColumns=@JoinColumn(  
            name="PUBLICATION_ID",  
            referencedColumnName="id")  
        )  
    )  
    private Set<Publication> pubs;  
  
}
```

Use of Database and Programming Language



Complete Computational Environments



- All data supported consistently whatever happens
- Programmers only have to understand one model and maintain one mapping

Java Programming Language

- Powerful object model
 - Strong typing
 - Automatic storage management
 - Concurrency support
-
- Objects do not outlive execution of virtual machine
 - Java object serialisation

Java Object Serialisation

- Stores and retrieves objects in serial form
- Maintain type safety
- Extensible mechanism
 - provide default mechanism
 - per class implementation for customisation
 - allow object to define its external format
- Persistence by reachability handles complex objects
- Intention
 - data exchange
 - "lightweight persistence"
 - object archiving for later use by same program

Java Object Serialisation Framework

- Interfaces for persistent object
 - `Serializable`
 - `Externalizable`
- Object streams to handle output and input
 - `ObjectOutputStream`
 - `ObjectInputStream`
- Interfaces defining output and input
 - `ObjectOutput` extends `DataOutput`
 - `ObjectInput` extends `DataInput`

Java Object Serialisation

- No special methods have to be implemented
- Method `writeObject` of class `ObjectOutputStream`
 - serialises objects
 - traverses references to other objects in the object graph
 - uses handles to preserve sharing and circular references
- Type information is stored together with objects
- Entire object graphs are read and written at same time
- Special handling is only required for
 - arrays
 - enum constants
 - objects of type `Class`, `ObjectStreamClass` and `String`

Java Object Serialisation

- A serialisable class must do the following
 - implement the `java.io.Serializable` interface
 - identify the fields that should be serialisable
 - non-transient and non-static fields are serialised by default
 - use the `serialPersistentField` member or the `transient` keyword
 - have access to the no-argument constructor of its first non-serialisable superclass
- Optionally, the class can define the following methods
 - `writeObject` controls saved data or appends information
 - `readObject` reads data corresponding to `writeObject`
 - `writeReplace` nominates a replacement object to be written
 - `readResolve` designates a replacement object when reading from the input stream

Java Object Serialisation Example

```
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();
```

Versioning of Serialisable Objects

- Simple versioning of serialised objects supported
- Bidirectional communication between versions of a class
- Evolved class is responsible to maintain contract established by non-evolved class
 - evolved class must not break assumptions about the interface provided by original version
 - later version must provide sufficient and equivalent information to allow earlier version to continue to satisfy non-evolved contract
- Compatible changes are changes that do not affect the contract between the class and its callers
- Field `serialVersionUID` to identifies class version

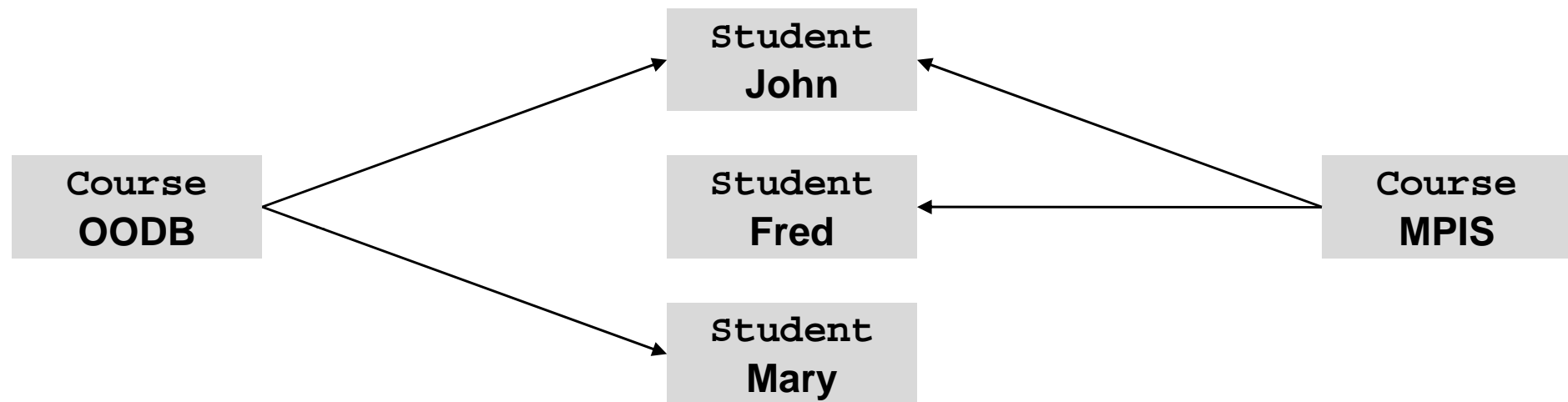
Incompatible and Compatible Changes

- Delete fields
- Move classes within the hierarchy
- Change non-static fields to static or non-transient fields to transient
- Change declared type of a field
- Change `writeObject` and `readObject` methods
- Change class from `Serializable` to `Externalizable` or vice-versa
- Change from non-enum type to enum type
- Remove either `Serializable` or `Externalizable`
- Adding `writeReplace` or `readResolved` method
- Add fields
- Add classes
- Remove classes
- Adding `writeObject` or `readObject` method
- Remove `writeObject` or `readObject` method
- Add `Serializable`
- Change access to a field
- Change static fields to non-static or transient fields to non-transient

Problems of Java Object Serialisation

- Not orthogonal
 - serialisable classes need to implement a special interface
- Not complete
 - class definition is not serialised along with objects
 - problems with evolution and versioning
- Not persistent
 - object identity is lost
 - relationship between static and instance variables is lost
- Not scalable
 - entire object graphs are serialised and deserialised
- Not transactional, recoverable nor concurrent

Problems of Object Identity



- If two object graphs are stored in separate serialisations, common substructures are duplicated when deserialised
- Similar effect occurs if a program re-reads data structure while holding parts of the original structure in memory
- Programmer must take great care when hashing objects

Persistent Programming Languages

- Orthogonal persistence
 - all objects may be made persistent
- Completeness or transitivity
 - everything needed to use persistent data must be preserved
 - object behaviour must also be preserved
 - persistence by reachability from named, persistent root objects
- Persistence independence
 - indistinguishable whether code operating on transient or persistent data
 - semantics of the language must not change
 - minimise what programmers have to learn to use persistence

PJama

- Persistent Java (PJama)
 - University of Glasgow
 - Sun Microsystems
- Assumptions
 - Java is used as implementation language for many applications
 - many applications will require long-term data management
- Goals
 - Orthogonality, persistence independence, durability, scalability, schema evolution, platform migration, endurance, openness, transactional, performance

PJama Architecture



- Standard Java applications
- Persistence is provided by a modified Java virtual machine
 - object faulting
 - promotion to persistence
 - recoverable and transactional operation
- Sphere
 - persistent object store
 - general purpose
 - supports disk garbage collection, evolution, ...

Creating Persistent Data

```
public class Department {  
    ...  
    public static void main (String[] args) {  
        // start transaction  
        Course c = new Course("OODB");  
        Person p = new Person("Fred");  
        try {  
            PJavaStore pjs = PJavaStore.getStore();  
            pjs.newPRoot("OODB", c);  
        } catch (PJSEException e) {  
            ...  
        }  
        // implicit commit  
    }  
}
```

Persistence Independence

```
Hashtable courses = new Hashtable();
try {
    PJavaStore pjs = PJavaStore.getStore();
    pjs.newPRoot("Courses", courses);
} catch (PJSEException e) {
    ...
}
...
Student student = new Student("Fred");
Course oodb = new Course("Object Oriented Databases");
Course webeng = new Course("Web Engineering");
courses.add(oodb.getTopic(), oodb);
oodb.attendedBy(student);
webeng.attendedBy(student);
...
courses.add(webeng.getTopic(), webeng);
...
```

Accessing Persistent Objects

```
...  
try {  
    PJavaStore pjs = PJavaStore.getStore();  
    Hashtable courses = (Hashtable) pjs.getPRoot("courses");  
} catch (PJSEException e) {  
    ...  
}  
...  
Course oodb = (Course) courses.get("Object Oriented Databases");  
oodb.display();  
...
```

Achievements of PJama

- Type safety
 - class information also stored in the persistent store
 - direct or indirect object access through named persistent roots
 - matching then performed of expected type and actual type
- Orthogonality
 - achieved approximation good enough for many applications
 - open issues with JDBC, CORBA and `java.lang.Thread`
- Persistence independence
 - no changes to language, core classes or compiler
 - persistence provided via additional API consisting mainly of methods of class `PJavaStore`

Achievements of PJama

- Durability
 - recovery points through explicit "stabilize" calls
- Endurance
 - Issues with recovery, schema evolution and platform migration that require application to be restarted
- Transactional
 - simple default model with implicit start and commit
 - different transaction models possible through specialisation of the class `TransactionShell`
- Performance
 - modified JVM/JIT is 15%-20% slower than unmodified JVM/JIT

Literature

- Christian Bauer and Gavin King: **Java Persistence with Hibernate**, *Manning Publications 2006*
- M. Atkinson: **Persistence and Java – A Balancing Act**, In: *Proceedings of Conference on Objects and Databases*, Springer Verlag, 2000

Next Week

db4o: Part 1

- Managing Databases, Storing and Retrieving Objects
- Query by Example, Native Queries, SODA
- Simple and Complex Objects, Activation, Transactions

