Prof. Dr.-Ing. Stefan Deßloch
AG Heterogene Informationssysteme
Geb. 36, Raum 329
Tel. 0631/205 3275
dessloch@informatik.uni-kl.de

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Chapter 6 – Object Persistence Services

Object/Relational Impedance Mismatch

- Object-oriented programming/design is increasingly used for building information systems
  - general approach: design a domain object model that represents the data, structure and common behavior of the business objects
  - domain object state has to be retrieved from and written to an underlying DBS (usually a relational DBS)
- Problem: object-oriented and relational models have severe differences
  - ➔ impedance mismatch

	objects	**relations**
structure	•complex values, collections •class hierarchies (inheritance)	•flat tables
relationships	•binary •1:1, 1:n, n:m (using collections) •uni-/bi-directional references	•binary •1:1, 1:n •value-based, symmetric
behavior	•methods	
access paradigm	•object navigation (follow references)	•declarative, set-oriented (queries)

Data Access Layer

- The impedance mismatch needs to be addressed/resolved in the application program
  - requires detailed knowledge of the DB-schemas, DBMS capabilities
  - involves coding SQL statements, awareness of transaction processing concepts
- Data access layer
  - introduces a common infrastructure layer where all interactions with the DBMS are performed
  - common design approach to separate the business logic from the data access logic of the transaction server programs
  - helps increase program maintenance, programmer productivity
  - building a data access layer is a complex undertaking
- Middleware to help with this task
  - object/relational mappers (ORM), object persistence services/frameworks
  - shield the application from existing data stores
    - data model, query language, API, schema
  - simplification of programming model for persistent data access and management
    - no explicit interaction with data source using SQL, JDBC, …

Object Persistence Services & Frameworks

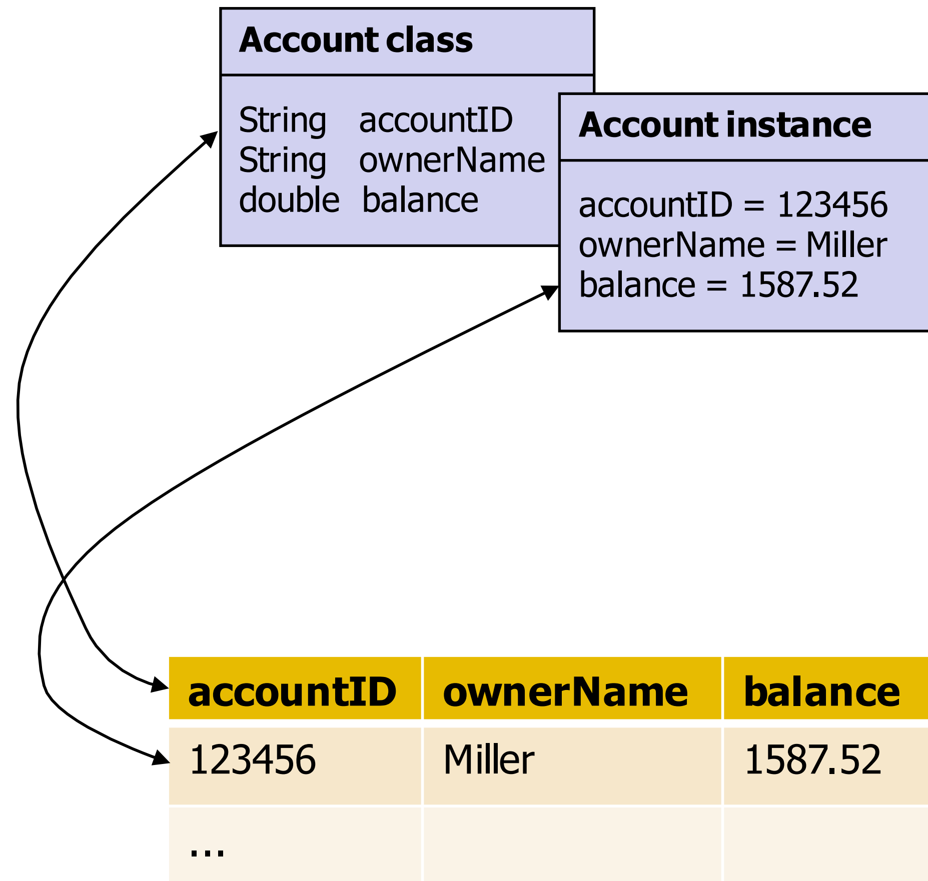
- Persistent object: lifetime of the object exceeds the execution of individual applications
- Basic approach (both in an application server and stand-alone appl. context)
  - application interacts only with objects
    - create, delete
    - access/modify object state variables
    - method invocation
  - persistence infrastructure maps interactions with objects to operations on data sources
    - e.g., INSERT, UPDATE, SELECT, DELETE
- May involve definition of a "mapping" from objects to data store schema
  - mapping has to cover
    - datatypes
    - classes, class hierarchies
    - identifiers
    - relationships

Caution: inherent performance impact!

© Prof.Dr.-Ing. Stefan Deßloch

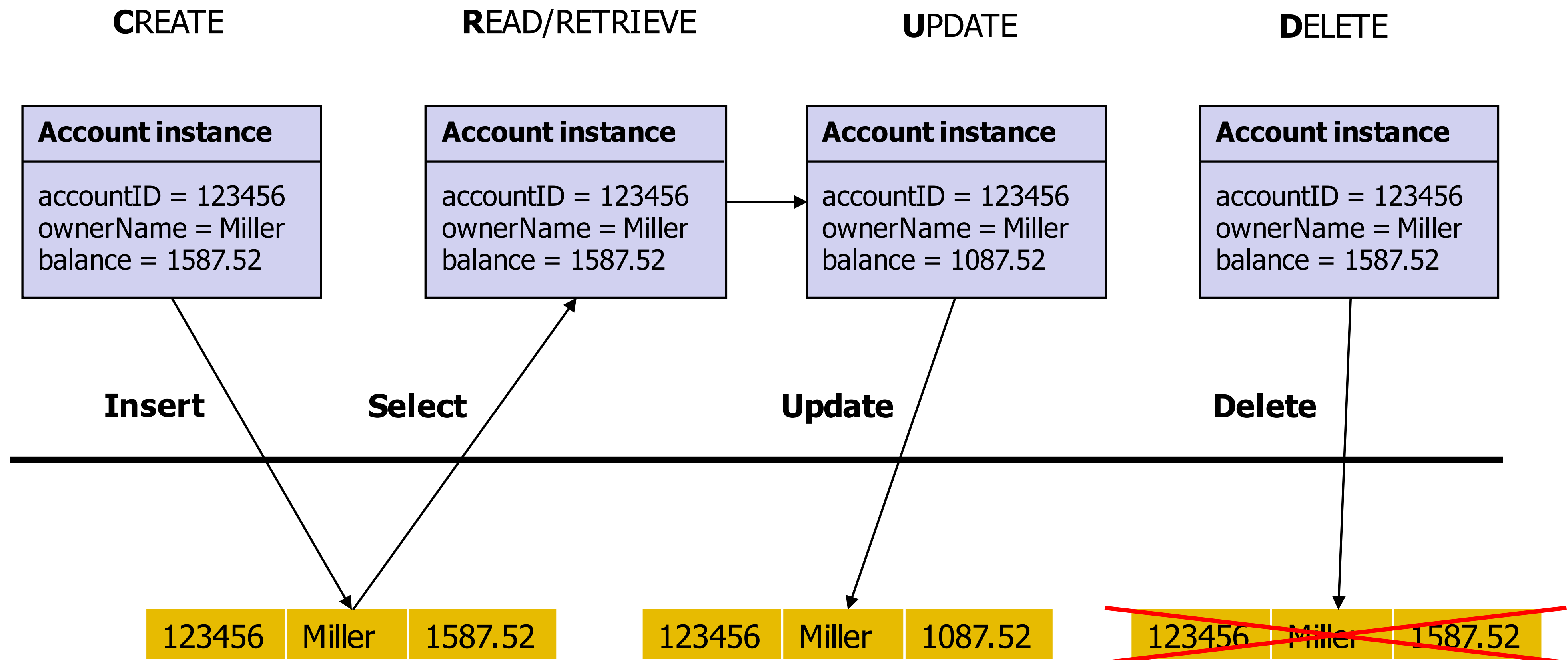
Object-Relational Mapping

- Object class
  - to single table
  - to multiple tables to support
    - inheritance
    - complex field values
- Object reference
  - to foreign key constraint
- Instance object
  - to one or more rows in a table
- Data types and values
  - mapping needs to consider variable length data (strings), differences in the type models, semantics
- Mapping tool support
  - *top-down, bottom-up, meet-in-the-middle*



The CRUD - Pattern

- Typical operation pattern provided by data access layer/persistence service



Object Persistence

- Aspects of persistence (Atkinson et.al, SIGMOD Record 1996)
  - Orthogonal persistence
    - persistence independent of data type, class
    - instances of the same class may be transient or persistent
  - Transitive persistence (aka persistence by reachability)
    - objects can be explicitly designated to become persistent (i.e., roots)
    - objects referenced by persistent objects automatically become persistent, too
  - Persistence independence (aka transparent persistence)
    - code operating on transient and persistent objects is (almost) the same
    - "client object" side: no impact when interacting with persistent objects
      - application may have to explicitly "persist" an object, but continues to use the same interface for interacting with the persistent object
      - interactions with a data store are not visible to/initiated by the client object, but happen automatically (e.g., when object state is modified or at EOT)
    - "persistent object" side: no special coding for "implementing" persistence
- Realizing the above aspects
  - requires significant efforts in programming language infrastructure
    - above goals are almost never fully achieved
  - may be considered "dangerous" (transitive persistence)

Persistence Programming Model Design Points

- Object-relational mapping
  - explicit mapping meta-data (descriptor files, annotations, ...)
  - hand-crafted implementation by developer (i.e., implementing CRUD-methods)
- Determining object persistence
  - statically (compile-time) – all/no objects of a certain class/type/programming model concept are persistent, *or*
  - semi-dynamic – objects of preselected classes (persistence-capable) may become persistent dynamically at runtime, *or*
  - dynamic (also: orthogonal persistence) – any object may be transient or persistent
- Identifying objects
  - implicit OID*, or* explicit (visible) object key (primary key)
  - object/identity cache support
- Locating/referencing persistent objects
  - by object key (lookup)
  - by query

Persistence Programming Model Design Points (2)

- Accessing object state (from client, from server/persistent object)
  - (public) member variables*, or*
  - object methods (getter/setter, …)
- Updating persistent object state
  - explicit (methods for store, load, …)*, or*
  - automatic (immediate, deferred)*, or*
  - combination
- Handling dependencies/relationships
  - Referential integrity
  - Lazy vs. eager loading
  - "Pointer swizzling"

Java Persistence API (JPA)

- Java standard for persistence frameworks
- Result of a major 'overhaul' of EJB specification for persistence, relationships, and query support
  - simplified programming model
  - standardized object-to-relational mapping
  - inheritance, polymorphism, "polymorphic queries"
  - enhanced query capabilities for static and dynamic queries
- API usage
  - from within an EJB environment/container
  - outside EJB, e.g., within a standard Java SE application
- Support for pluggable, third-party persistence providers

➔ We use JPA throughout this chapter to illustrate concepts and design points

Entities in JPA

- *"An entity is a lightweight persistent domain object"*
  - in EJB, entities are **not** remotely accessible (i.e., they are local objects)
- Simple programming model for EJB entities
  - entity is a POJO (plain old Java object)
    - no additional interfaces or implementation of generic (CRUD-support) methods required
    - class has to be designated (e.g., annotated) as *Entity* class
  - entity state (instance variables) is encapsulated, client access only through accessor methods (get*X*(), set*X*()) or other methods
- Explicit mapping meta-data
  - use of annotations for persistence and relationship aspects
  - alternative: XML deployment descriptor
- Entities and inheritance
  - abstract and concrete classes can be entities
  - entities may extend both non-entity and entity classes, and vice versa

➔ Does JPA provide orthogonal persistence?

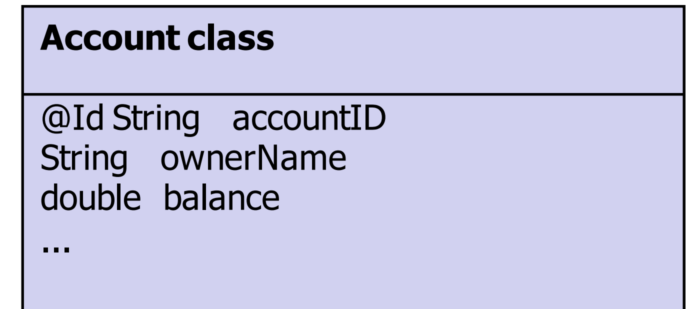
Requirements on Entity Class

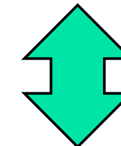
- Public, parameter-less constructor
- Top-level class, not final, methods and persistent instance variables must not be final
- Entity state is made accessible to the persistence provider runtime
  - either via instance variables (protected or package visible)
  - or via (bean) properties (get*Property*/set*Property* methods)
  - consistently throughout the entity class hierarchy
- Collection-valued state variables have to be based on (generics of) specific classes in java.util

➔ Does JPA provide transparent persistence?

Mapping to RDBMS

- Entities must have primary keys
  - defined at the root, exactly once per class hierarchy
  - may be simple or composite
    - key class required for composite keys
  - must not be modified by the application
    - more strict than primary key in the RM
- Entity mapping
  - default table/column names for entity classes and persistent fields
    - can be customized using annotations, deployment descriptor
  - mapping may define a primary table and one or more secondary tables for an entity
    - state of an entity/object may be distributed across multiple tables
    - need to specify join columns for joining tuples from primary and secondary tables to "build" the entity state



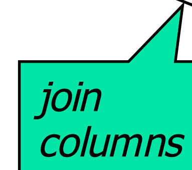


Accounts table

accountID	ownerName	balance

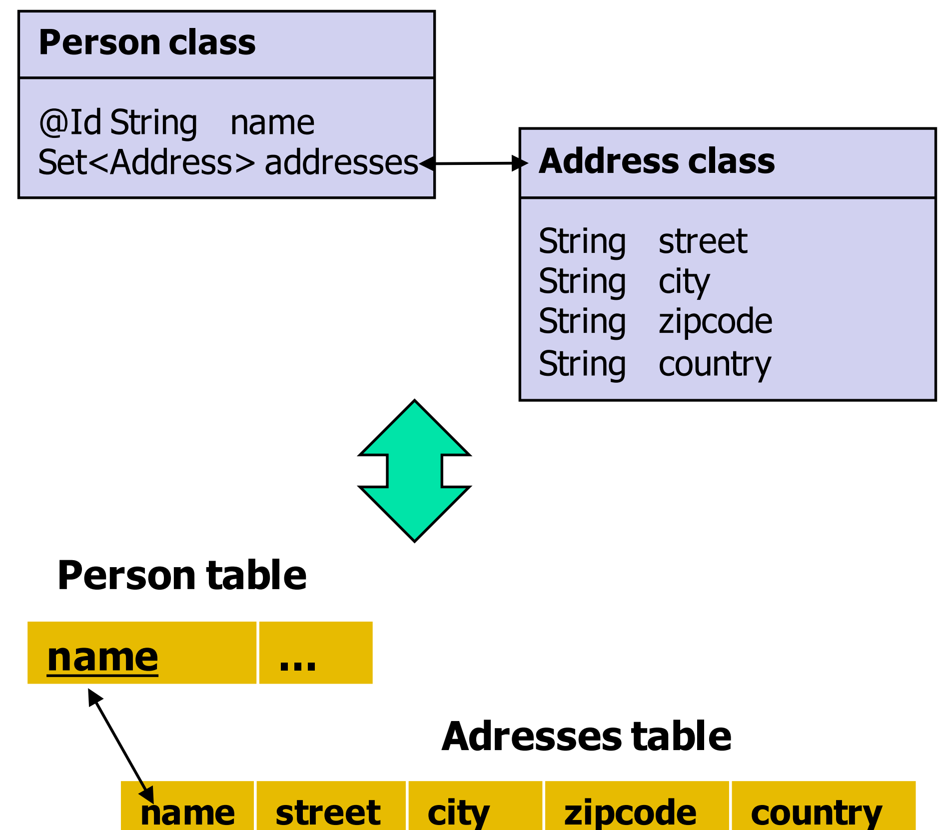
AccountDetails table

accountID	...

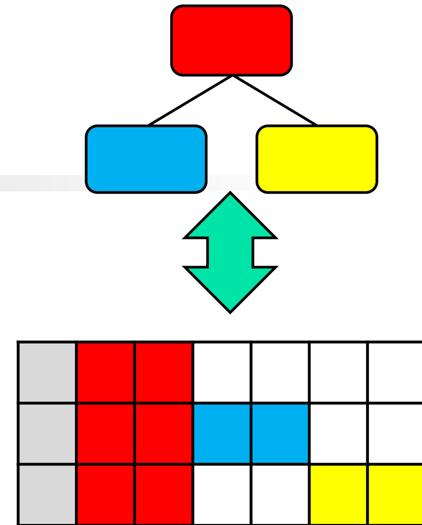


Embeddable Classes

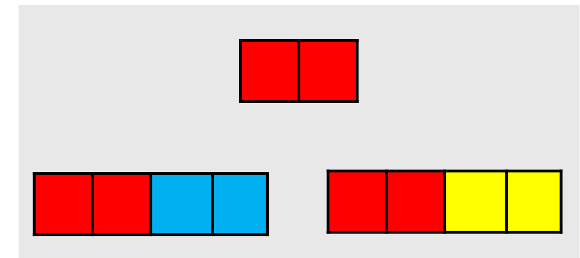
- Embeddable classes
  - "fine-grained" classes used by an entity to represent state
  - instances are seen as embedded objects, do not have a persistent identity
    - mapped with the containing entities
    - not sharable across persistent entities
- Used as field variable type in embedding class
  - single-valued or collection-valued
- Mapping to the same table as the containing entity, or to a collection table



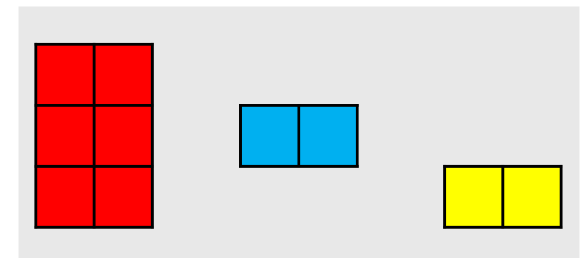
Inheritance Mapping Strategies



- Single table with discriminator column (default)
  - has columns for all attributes of any class in the hierarchy
  - stores **all** instances of the class hierarchy
  - has a special discriminator column identifying the class within the hierarchy to which a specific instance belongs



- Horizontal partitioning (single table per concrete entity class)
  - one table per entity class, with columns for all attributes (incl. inherited)
  - table stores only the **direct** instances of the class



- Vertical partitioning (separate table per subclass)
  - one table per entity class, with columns for newly defined attributes (i.e., attributes specific to the class), plus ID column
  - table stores **partial** information about **all** (i.e., **transitive**) instances of the class

➔ Advantages/disadvantages?

Relationships

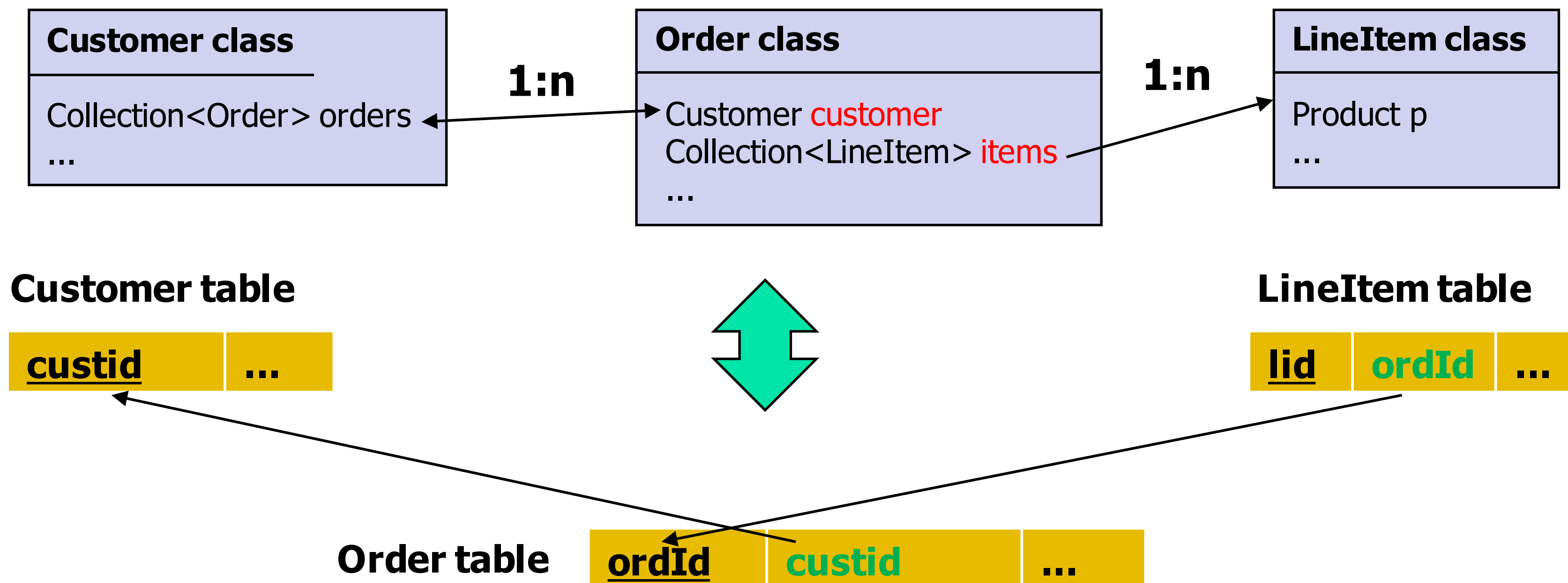
- Persistence model needs to be complemented by relationship support
  - represent relationships among data items (e.g., tuples) at the object level
  - support persistence of native programming language concepts for "networks" of objects
    - references, pointers
- Possible alternatives
  - value-based relationships at the object level (see relational data model)
    - requires to issue a query (over objects) to locate related object(s)
    - no "navigational" access
  - ➔ relationships are part of persistent object interface(s) or implementation
    - getter/setter methods or properties/fields to represent relationship roles of participating entities
    - relationships are always binary, collection support required for 1:n, n:m
    - uni-directional or bi-directional representation
      - consistency?

Relationships in Java Persistence API

- Relationships are represented in the same way as persistent attributes
  - member variables, get/set method pairs are annotated as relationship attributes
  - variable refers to an instance of the referenced Entity class
- Relationship types: 1:1, 1:n, n:1, n:m
  - 1:1, n:1 – variable type is the Entity class
  - 1:n, n:m – variable type is a collection type with Entity class as member type
- Supports uni- and bi-directional binary relationships
  - bi-directional
    - has a designated owning side and inverse side
    - for 1:n and n:1, the "many" side has to be the owning side
    - does not provide automatic maintenance of inverse relationships!
    - the designated owning side determines the state at the persistent data store
  - uni-directional relationship only has an owning side

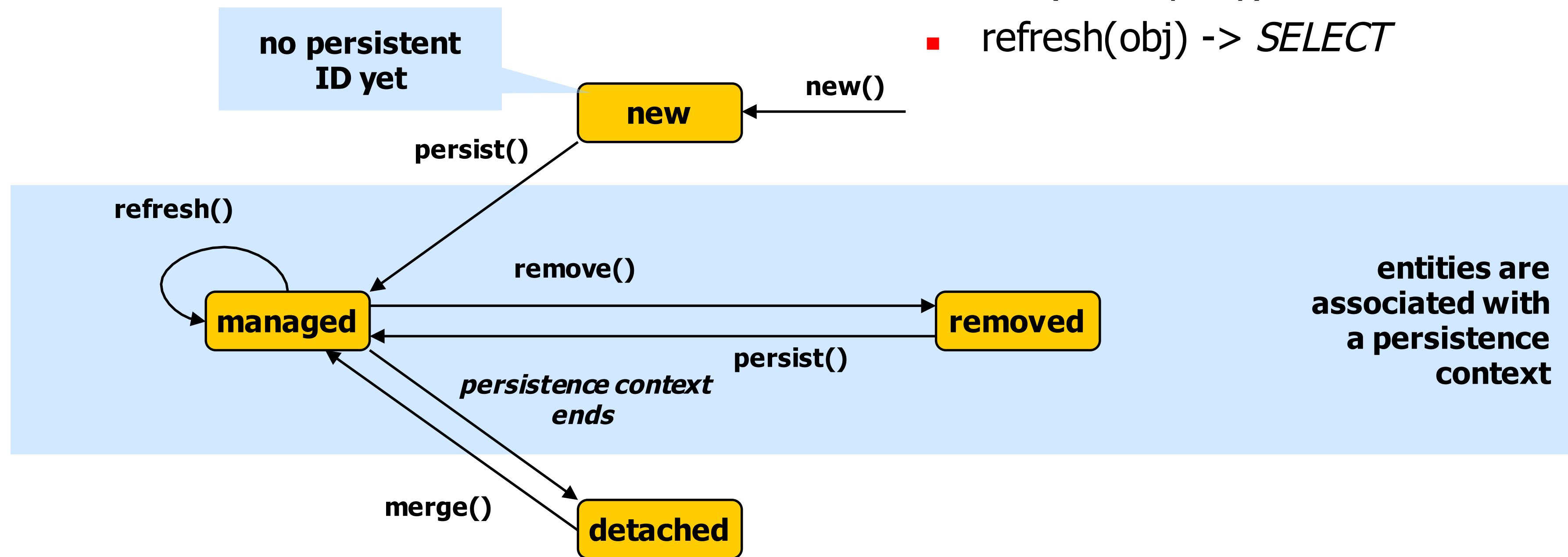
Relationship Mapping in JPA

- Standard relationship mapping
  - represented using primary key/foreign key relationships
  - table for the designated "owning" side has to contain the foreign key
    - exception: for unidirectional 1:n-relationship, foreign key is on the table for the "n" side!
  - N:M-relationships represented using a relationship table ("join table")
- Additional mapping strategies can involve "join tables" for 1:1, 1:n, n:1
- Example



Entity Life Cycle and Persistence

- Determining persistence
  - instances of entity classes may be transient or persistent
  - persistence property controlled by application/client (e.g., a SessionBean)
- Entity manager manages entity state and lifecycle within persistence context
  - persist(obj) -> *INSERT*
  - merge(obj) -> *UPDATE*
  - remove(obj) -> *DELETE*
  - find(class, pKey) -> *SELECT*
  - refresh(obj) -> *SELECT*



Example – Client Perspective

```
@Stateless
@TransactionManagement(CONTAINER)
@TransactionAttribute(REQUIRED)
public class OrderEntryBean implements OrderEntry {
    @PersistenceContext private EntityManager em;
    public void enterOrder(int custID, Order newOrder) {

      Customer cust = em.find(Customer.class, custID);

      cust.getOrders().add(newOrder);

      newOrder.setCustomer(cust);

      em.persist(newOrder);
    }
```

Client is a stateless session bean with transaction attribute REQUIRED

Persistence context (entity manager functions) is provided and scoped within the transaction

EM is used to find a customer entity using primary key;
➔ cust is a managed entity

newOrder (state is „new") is connected to cust via „orders" (inverse) relationship.
newOrder is still transient (state is „new").

Now the owning side of the relationship is updated.
newOrder is still transient (state is „new").

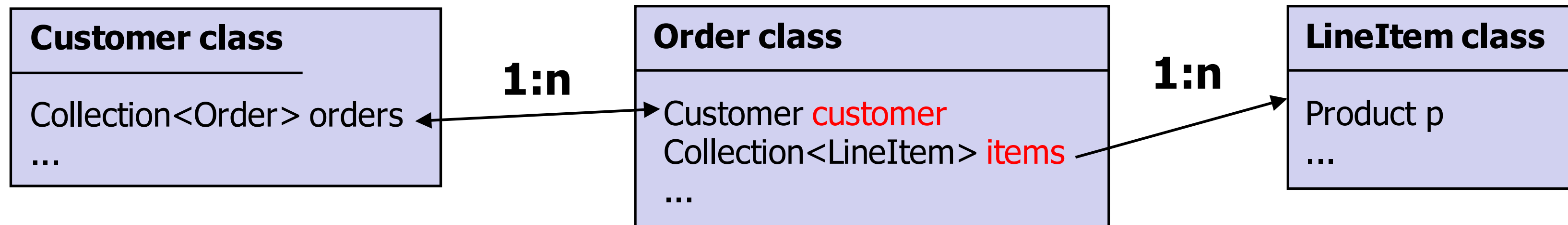
newOrder is made persistent (state is „managed").

Method ends ➔ INSERT newOrder in the database; transaction commits, persistence context ends, newOrder becomes „detached"!

Transactions and Persistence Contexts

- Access of persistent data resulting from persistent object manipulation always occurs in the scope of a transaction
- What happens at transaction roll-back?
  - state of entities in the application is not guaranteed to be rolled back, only the persistent state
- What happens if a transaction terminates and objects become "detached"?
  - objects can still be modified "offline"
- What happens when objects are merged "re-attached" to a new transaction context?
  - objects are NOT automatically refreshed
  - potential for lost updates
  - can be controlled by explicit refresh or using optimistic locking

Transitive Persistence



- What happens in previous example, when *em.persist(newOrder)* is executed?
  - *newOrder* becomes a managed entity
- What about referenced order items?
  - goal: should be persisted as well
- What happens when we associate newOrder with the (managed) customer?
  - *cust.getOrders().add(newOrder);* should *newOrder* become persistent?
  - *newOrder.setCustomer(cust);* should *newOrder* become persistent now?
  - goal: establishing a relationship with (persistent) customer should make the order persistent as well
    - and transitively persist the order items, too

➔ Transitive persistence (persistence by reachability) would take care of that!

Relationships And Transitive Persistence

- Persistence by reachability: all objects reachable from persistent object through standard Java references are made persistent, too!
- Benefits and
  - powerful, easy to use from a development perspective
  - takes care of "dependent" objects, allowing to "encapsulate" the referenced object network
- Drawbacks: implicit definition of persistence
  - is this the correct semantics for all references?
  - developer needs to understand what to expect in terms of number of resulting insert operations
- What about the "reverse" semantics for object deletion: when should an object that was implicitly made persistent be deleted?
  - when the originally referencing object causing implicit persistence is deleted or removes the reference?
  - when the object is no longer referenced by other persistent objects (garbage collection)?
    - still could be retrieved using its primary key value
  - when it is explicitly deleted?

CASCADE Semantics Of Relationships

- CASCADE rules/annotations are usually the only mechanism offered to
  - specified as metadata on specific relationship attributes
  - allow realize selective transitive persistence
  - implement automatic selective transitive deletion
    - relationship attribute can be flagged to cause deletion, if "parent" object is deleted
    - often mapped to referential integrity constraints in the DB-mapping

    ➔ what is the resulting object state in the application, if the deleted object is still referenced?
- JPA supports CASCADE annotations
  - possible values: PERSIST, MERGE, REMOVE, REFRESH, ALL

Realizing Automatic Persistence

- Strategies for "loading" objects from the persistent store during navigational access
  - "lazy" loading – object is retrieved only when accessed based on primary key or reference (relationship)
    - easy to implement
    - may cause increased communication with data source, resulting in performance drawbacks
  - "eager" loading
    - when an object is requested, transitively load all the objects reachable through references
    - requires construction/generation of complex data store queries
    - may cause a lot of unnecessary objects to be loaded
- Persistence frameworks usually offer a combination of the above strategies
  - relationships can be explicitly designated as eager or lazy
    - at deployment time? separate definitions depending on the application scenario?
  - can be generalized to arbitrary persistent attributes
    - e.g., to pursue lazy loading of large objects
  - in JPA: fetch type LAZY or EAGER

Realizing Automatic Persistence (2)

- How to write object changes back to the data store
  - there may be many fine-grained (i.e., attribute-level) updates on a persistent object during a transaction
  - immediate update: write changes to the DB after every attribute modification
    - easy to implement/support, but many interactions with the DBMS
  - deferred update: record changes and combine them into a single update per tuple at the end of the transaction
    - more complex to implement, unless one always updates the complete tuple
      - the latter will result in unnecessary processing overhead at the DBMS
    - approach needs to be refined to account for consistent query results
      - write back changes also before any object query statements are executed
- Concurrency control strategy (determined in combination with the persistent data store)
  - pessimistic, using locking at the DBMS-level
    - requires long read locks to avoid lost updates
  - optimistic, by implementing "optimistic locking"

Optimistic Locking and Concurrency

- Note: most DBMSs don't support optimistic concurrency control
- Example JPA: *optimistic locking* is assumed, with the following requirements for application portability
  - isolation level "read committed" or equivalent for data access
    - no long read locks are held, DBMS does not prevent lost updates, inconsistent reads
  - declaration of a *version* attribute for all entities to be enabled for optimistic locking
    - persistence provider uses the attribute to detect and prevent lost updates
      - provider changes/increases the version during a successful update
      - compares original version with the current version stored in the DB, if the version is not the same, a conflict is detected and the transaction is rolled back
  - inconsistencies may arise if entities are not protected by a version attribute
  - does not guarantee consistent reads
  - conflicts can only be detected at the end of a (possibly long) transaction

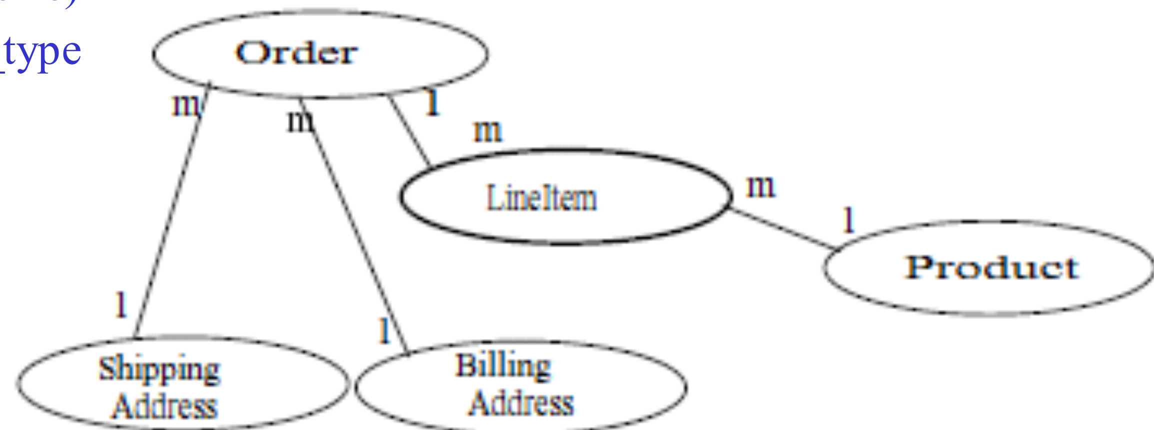
Queries Over Persistent Objects

- Accessing persistent objects through primary key or navigation over relationships
  - is a useful basic mechanism that fits the OO programming model
  - but is a severe restriction when accessing collections of persistent objects
  - and can cause severe performance impact through tuple-by-tuple operations
- Object retrieval through a query language
  - required to solve the above problems
  - but should not force the developer to drop down to the data store query language (and schema) again
- Object query language
  - continues to shield the developer from data store (and mapping) details
  - requires persistence framework to transform object queries into corresponding data store queries based on the object-to-relational mapping

EJB Query Language (EJB-QL)

- Introduced as a query language for CMP EntityBeans
  - used in the definition of user-defined Finder methods of an EJB Home interface
    - no arbitrary (embedded or dynamic) object query capabilities!
  - uses abstract persistence schema as its schema basis
  - SQL-like
- Example:

```
SELECT DISTINCT OBJECT(o)
FROM Order o, IN(o.lineItems) l
WHERE l.product.product_type
    = ‘office_supplies’
```



Java Persistence Query Language

- Extension of EJB-QL
  - named (static) and dynamic queries
  - range across the class extensions including subclasses
    - a *persistence unit* is a logical grouping of entity classes, all to be mapped to the same DB
    - queries can not span across persistence units
  - includes support for
    - bulk updates and delete
    - outer join
    - projection
    - subqueries
    - group-by/having
- Prefetching based on outer joins
  - Example:

```
SELECT d
FROM Department d LEFT JOIN FETCH d.employees
WHERE d.deptno = 1
```

Historic Perspective

- Object persistence supported at various levels of abstraction
  - CORBA
    - standardized "low-level" APIs
    - powerful, flexible, but no uniform model for component developer
      - various persistence protocols
    - explicit vs. implicit (client-side transparent) persistence
  - EJB/J2EE Entity Beans
    - persistent components
      - CMP: container responsible for persistence, maintenance of relationships
    - uniform programming model
    - transparent persistence
  - JDO
    - persistent Java objects
    - orthogonal, transparent, transitive persistence
  - Java Persistence API
    - successor of EJB entity beans
    - standardized mapping of objects to relational data stores
    - influenced partly by JDO, Hibernate
    - can be used outside the EJB context as well

Summary

- Object/relational mapping, object persistence service middleware
  - provide abstraction capabilities for developing a object-oriented data access layer
  - goal: increase programmer productivity
  - potential performance impact
  - complexity/learning curve
- Bridging the object/relational impedance mismatch is hard!
  - mapping alternative/complexity for classes, relationships
  - appropriate level of support for orthogonal, transparent and transitive persistence
    - object lifecycle
    - optimizations for loading/storing object state
    - transaction and concurrency semantics
- Mandates appropriate object query support
  - Example: Java Persistence Query Language
    - based on EJB-QL (and therefore on SQL)
    - numerous language extensions for query, bulk update
    - static and dynamic queries
  - Queries over multiple, distributed data sources are not mandated by the above approaches!