

Chapter 6 – Object Persistence, Relationships and Queries



Object Persistence

- Persistent object:
 - Lifetime of a persistent object may exceed the execution of individual applications
- Goals
 - 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, ...
 - eliminate "object/relational impedance mismatch"

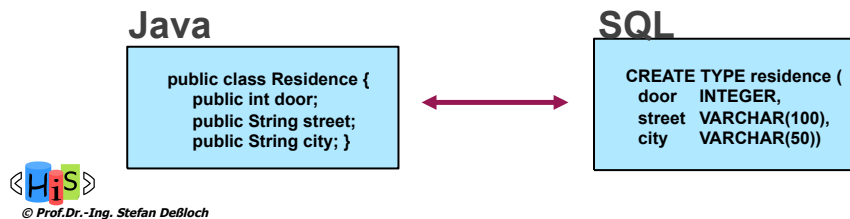
	objects	relations
structure	•complex values, collections •class hierarchies (inheritance)	•flat tables
relationships	•binary •1:1, 1:n, n:m (using collections) •uni-/bi-directional references	•binary •1:1, 1:n •value-based, symmetric
behavior	•methods	
access paradigm	•object navigation (follow references)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



Object-Relational DBMS and JDBC

- Materializing instances of SQL user-defined types as instances of corresponding Java classes
 - manipulated using existing result set or prepared statement interfaces
 - get/setObject(<column>) simply "works" for structured types
 - Example:

```
ResultSet rs = stmt.executeQuery("SELECT e.addr FROM Employee e");
rs.next( );
Residence addr = (Residence)rs.getObject(1);
```
- Still requires knowledge of DB-schema, explicit SQL statements for retrieval, insertion, update, deletion of objects
- No support for building Java object references from DB-object relationships



Object Persistence Services & Frameworks

- 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
 - see course "Informationssysteme" (EER -> RM) for possible mapping alternatives

Caution: inherent performance impact!

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)



© Prof.Dr.-Ing. Stefan DeBloch

Persistence Programming Model Design Points

- Persistence in application server middleware
 - surfaced at the distributed object programming model, *or*
 - supported at the programming language level
- 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)
- Locating/referencing persistent objects
 - by object key (lookup)
 - by query



© Prof.Dr.-Ing. Stefan DeBloch

Persistence Programming Model Design Points (2)

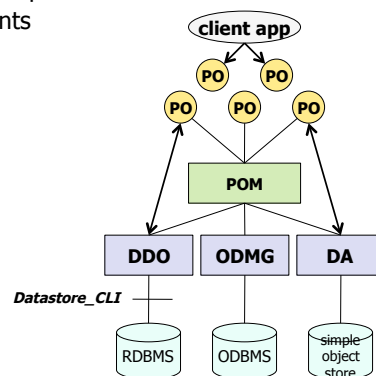
- Accessing object state (from client, from server/persistent object)
 - (public) member variables, *or*
 - object methods (getter/setter, ...)
- Updating persistent object state
 - explicit (methods for store, load, ...), *or*
 - automatic (immediate, deferred), *or*
 - combination
- Handling dependencies/relationships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



© Prof. Dr.-Ing. Stefan DeBloch

CORBA – Persistent Object Service

- Goal: uniform interfaces for realizing object persistence
- POS (Persistent Object Service) components
 - PO: Persistent Object
 - are associated with persistent state through a PID (persistent object identifier)
 - PID describes data location
 - POM: Persistent Object Manager
 - mediator between POs and PDS
 - realizes interface for persistence operations
 - interprets PIDs
 - implementation-independent
 - PDS: Persistent Data Service
 - mediator between POM/PO and persistent data store
 - data exchange between object and data store as defined by protocols
 - Datastore
 - stores persistent object data
 - may implement *Datastore_CLI* (encapsulates ODBC/CLI)



© Prof. Dr.-Ing. Stefan DeBloch

CORBA Persistence Model

- CORBA object is responsible for realizing its own persistence
 - can use PDS services and functions
 - implicit persistence control
 - client is potentially unaware of object persistence aspects (client persistence independence)
 - explicit persistence control
 - persistent object implements PO interface, which can then be used by the client
- Explicit persistence control by CORBA client:
 - client creates PID, PO using factory objects
 - PO Interface
 - connect/disconnect – automatic persistence for the duration of a "connection"
 - store/restore/delete – explicit transfer of data
 - delegated to POM, PDS
 - caution!: CORBA object reference and PID are different concepts
 - client can "load" the same CORBA object with data from different persistent object states



© Prof. Dr.-Ing. Stefan DeBloch

Persistence Protocols

- CORBA Persistence Service defines three protocols
 - Direct Access (DA) protocols
 - PO stores persistent state using so-called *direct access data objects* (DADOs)
 - **CORBA objects** whose interfaces only have attributes
 - defined using Data Definition Language (IDL subset)
 - precompilation is specific to CORBA/PDS environment
 - DADOs may persistently reference other DADOs, CORBA objects
 - ODMG'93 protocols
 - utilizes ODMG standard for object-oriented databases
 - persistent objects are **programming language objects**, not CORBA objects
 - definition of persistence "schema" similar to DA protocol (is a superset)
 - own DDL (ODL) for defining POs
 - Dynamic Data Object (DDO) protocols
 - "generic", **self-describing DO**
 - methods for read/update/add of attributes and values
 - manipulation of meta data
 - used for accessing record-based data sources (e.g. RDBMS) using DataStore CLI interface
 - CLI for CORBA
- Protocols are employed in the implementation of POs



© Prof. Dr.-Ing. Stefan DeBloch

EJB Version 2 – Entity Beans

- Persistence is supported at the EJB/distributed object programming model
 - explicit type of EJB for (static) persistent objects
 - invocation of remote object methods
 - life-cycle interface (*Home* interface)
 - *create, retrieve, delete*
 - *findByPrimaryKey*
 - additional, bean-specific finder methods
 - primary-key class for uniquely identifying persistent bean objects
- Follows *transparent persistence* approach on the client
 - persistence-related operations (e.g., synchronizing object state with DB contents) are hidden from the client
 - automatic update of persistent object state



© Prof. Dr.-Ing. Stefan DeBloch

Entity Beans

- Object persistence logic is implemented separately from business logic
 - entity bean "implements" call-back methods for persistence
 - *ejbCreate* – insert object state into DB
 - *ejbLoad* – retrieve persistent state from DB
 - *ejbStore* – update DB to reflect (modified) object state
 - *ejbRemove* – remove persistent object state
- Manipulation of CMP fields through access methods (*getField()*, *setField(...)*)
 - access within methods of the same EB
 - client access can be supported by including access methods in the remote interface
 - provides additional flexibility for container implementation
 - lazy loading of individual attributes
 - individual updates for modified attributes



© Prof. Dr.-Ing. Stefan DeBloch

Container-Managed Persistence (CMP)

- Bean developer defines an *abstract persistence schema* in the deployment descriptor
 - persistent attributes (*CMP fields*)
- Mapping of CMP fields to DB-structures (e.g., columns) in deployment phase
 - depends on DB, data model
 - tool support
 - *top-down, bottom-up, meet-in-the-middle*
- Container saves object state
 - bean does not worry about persistence mechanism
 - call-back methods don't contain DB access operations
 - may be used to compress/decompress values, derive attribute values, ...



© Prof.Dr.-Ing. Stefan DeBloch

Bean-Managed Persistence (BMP)

- Callback-methods contain explicit DB access operations
 - useful for interfacing with legacy systems or for realizing complex DB-mappings (not supported directly by container or CMP tooling)
- No support for container-managed relationships
- Finder-methods
 - have to be implemented in Java
 - no support for EJB-QL



© Prof.Dr.-Ing. Stefan DeBloch

Entity Beans (and CORBA) - Problems

- Distributed component vs. persistent object
 - granularity
 - potential overhead (and possible performance problems)
 - solution in EJB 2.0: local interfaces
 - but: semantic differences (*call-by-value* vs. *call-by-reference*)
 - complexity of development process
- Missing support for class hierarchies with inheritance



© Prof.Dr.-Ing. Stefan DeBloch

JDO – Java Data Objects

- JDO developed as new standard for persistence in Java-based applications
 - first JDO specification 1.0 released in March 2002 (after ~ 3 years) through Sun's JCP (Java Community Process)
 - > 10 vendor implementations plus open-source projects
 - *mandatory features* and *optional features* in the specification (i.e., some optional features are „standardized“ → promises better portability).
- Features, elements
 - orthogonal, transitive persistence
 - native Java objects (inheritance)
 - byte code enhancement
 - mapping to persistence layer using XML-metadata
 - transaction support
 - JDO Query Language
 - JDO API
 - JDO identity
 - JDO life cycle
 - integration in application server standard (J2EE)



© Prof.Dr.-Ing. Stefan DeBloch

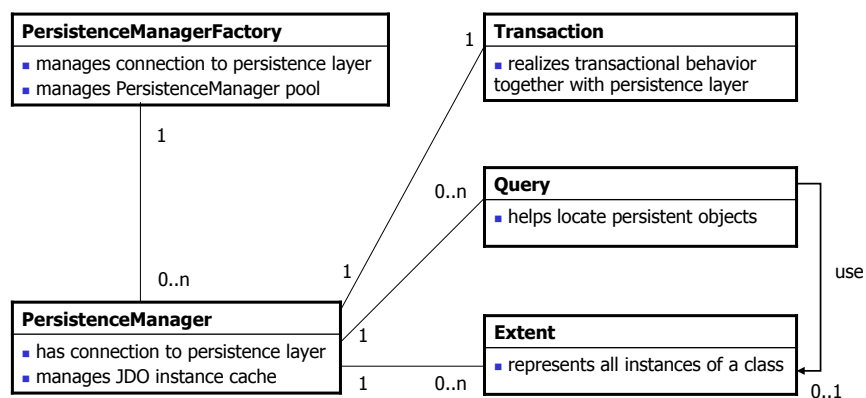
Persistence in JDO

- (Semi-)dynamic persistence
 - Java class supports (optional) persistence (implements PersistenceCapable)
 - not all instances of the class need to be persistent
 - application can/must explicitly turn a transient object into a persistent object (and vice versa)
- Persistence logic is transparent for client at the Java level
 - interacting with transient and persistent objects is the same
- Transitive persistence (i.e., by reachability)



© Prof. Dr.-Ing. Stefan DeBloch

JDO API



© Prof. Dr.-Ing. Stefan DeBloch

PersistenceManager API - Example

```
1 Author author1 = new Author("John", "Doe");
2 PersistenceManager pm1 = pmf.getPersistenceManager();
3 pm1.currentTransaction.begin();
4 pm1.makePersistent(author1);
5 Object jdoID = pm1.getObjectId(author1);
6 pm1.currentTransaction.commit();
7 pm1.close();

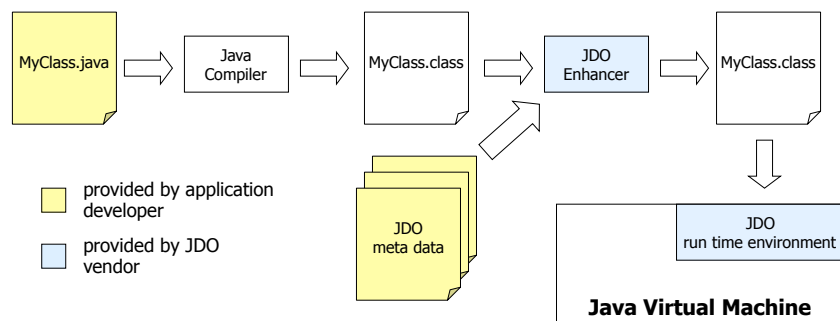
8 // Application decides that author1
9 // must be deleted
10 PersistenceManager pm2 = pmf.getPersistenceManager();
11 pm2.currentTransaction.begin();
12 Author author2 = (Author)pm2.getObjectById(jdoID);
13 pm2.deletePersistent(author2);
14 pm2.currentTransaction.commit();
15 pm2.close();
```



© Prof. Dr.-Ing. Stefan DeBloch

Byte-Code-Enhancement

- Java bytecode (*.class) and metadata (*.jdo)
- Same object class (now implements [PersistenceCapable](#))
- O/R-mapping specification is vendor-specific



© Prof. Dr.-Ing. Stefan DeBloch

Java Persistence API

- 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



© Prof.Dr.-Ing. Stefan DeBloch

Entities

- *"An entity is a lightweight persistent domain object"*
 - entities are not remotely accessible (i.e., they are local objects)
 - no relationship with the EntityBeans concept, but co-existence
- Simplified programming model for EJB entities
 - entity is a POJO (plain old Java object)
 - marked as *Entity* through annotations or deployment descriptor
 - no additional local or home interfaces required
 - no implementation of generic EntityBean methods needed
 - entity state (instance variables) is **encapsulated**, client access only through accessor or other methods
 - use of annotations for persistence and relationship aspects
 - no XML deployment descriptor required
- Entities and inheritance
 - abstract and concrete classes can be entities
 - entities may extend both non-entity and entity classes, and vice versa



© Prof.Dr.-Ing. Stefan DeBloch

Identity and Embeddable Classes

- 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
- 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



© Prof. Dr.-Ing. Stefan DeBloch

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 (*getProperty/setProperty* methods)
 - consistently throughout the entity class hierarchy
- Collection-valued state variables have to be based on (generics of) specific classes in `java.util`



© Prof. Dr.-Ing. Stefan DeBloch

Mapping to RDBMS

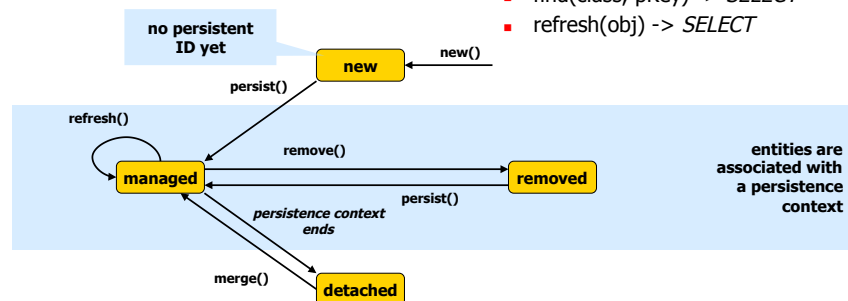
- Entity mapping
 - default table/column names for entity classes and persistent fields
 - can be customized using annotations, deployment descriptor
 - mapping may define a primary table and one or more secondary tables for an entity
 - state of an entity/object may be distributed across multiple tables
- Inheritance mapping strategies supported for the mapping
 - single table with discriminator column (default)
 - table has columns for all attributes of any class in the hierarchy
 - tables stores all instances of the class hierarchy
 - horizontal partitioning
 - one table per entity class, with columns for all attributes (incl. inherited)
 - table stores only the **direct** instances of the class
 - vertical partitioning
 - one table per entity class, with columns for newly defined attributes (i.e., attributes specific to the class)
 - table stores information about **all** (i.e., **transitive**) instances of the class



© Prof. Dr.-Ing. Stefan DeBloch

Entity Life Cycle and Persistence

- Orthogonal 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*



© Prof. Dr.-Ing. Stefan DeBloch

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
- 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?
 - separate relationship concept/service, independent of persistent object interfaces



© Prof. Dr.-Ing. Stefan DeBloch

CORBA Relationships

- Relationship Service
 - management of object dependencies, separate from object state or interface
 - relationship involves: type, role, cardinality
 - type: types of objects that may participate in a specific relationship type
 - role: role names of participating entities
 - major goals
 - multi-directional use/navigation and relationship maintenance
 - decouple relationship from CORBA object reference maintained by each participating object
 - graph traversal
 - attributes and behavior for relationships
 - generic IDL interfaces for roles, relationships, ...
 - to be subtyped for application-specific relationships (e.g., *Emp-Dept*)
 - supplemented by additional interfaces for relationship graph traversal
- Relationships are separate (CORBA) objects
 - highly dynamic, powerful, but very complex to use
 - not really suitable for (fine-grained) data-level relationships



© Prof. Dr.-Ing. Stefan DeBloch

EJB - Container-managed Relationships

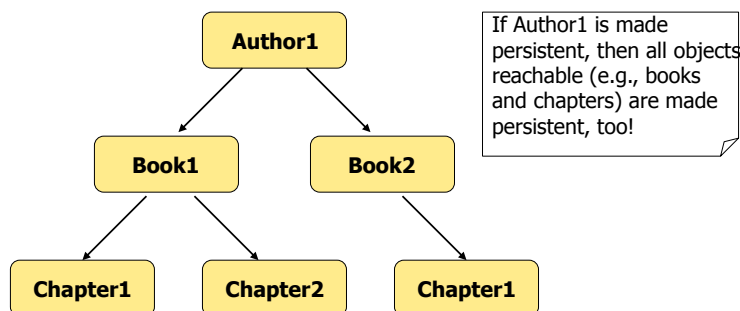
- Relationships can be defined in deployment descriptor or through annotations
 - part of abstract persistence schema
- Relationships may be uni-directional ("reference") or bi-directional
- Relationship types: 1:1, 1:n, n:1, n:m
- Access methods for accessing objects participating in a relationship
 - like CMP field methods (get/set)
 - Java Collection interface for set-valued reference attributes
- Container generates code for
 - relationship maintenance
 - cardinality, inverse relationship field consistency are guaranteed
 - persistent storage, involves mapping definition as well
- No transitive persistence
 - relationship can only be established among entityBeans, which are already persistent
- Only supported for CMP EntityBeans



© Prof. Dr.-Ing. Stefan DeBloch

JDO – Relationships and Transitive Persistence

- All **PersistenceCapable** objects reachable from persistent object through standard Java references within an object graph are made persistent, too
- No managed inverse relationships



© Prof. Dr.-Ing. Stefan DeBloch

Relationships in Java Persistence API

- Relationships are represented in the same way as persistent attributes
 - member variables, get/set method pairs
- Supports uni- and bi-directional binary relationships of the same types as EJB CMR
 - but does not provide automatic maintenance of inverse relationships
 - a designated owning side "wins" at the persistent data store
- Selective transitive persistence
 - defined using CASCADE options on relationships
- Relationship mapping
 - represented using primary key/foreign key relationships
 - table for the "owning" side of the relationship contains the foreign key
 - N:M-relationships represented using a relationship table



© Prof. Dr.-Ing. Stefan DeBloch

Relationships – Additional Aspects

- Discussions about benefits and drawbacks of transitive persistence
 - easy to use from a development perspective, but
 - implicit definition of persistence
 - developer needs to understand what to expect in terms of number of resulting insert operations
 - and 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?
- Cascading delete rules are usually the only mechanism offered to implement automatic deletion
 - relationships 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?



© Prof. Dr.-Ing. Stefan DeBloch

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



© Prof. Dr.-Ing. Stefan DeBloch

CORBA Queries

- Query Service
 - set-oriented queries for locating CORBA objects
 - SQL, OQL (ODMG) can be used as query languages
 - query results are represented using Collection objects
 - iterators
 - not restricted to persistent query objects
- Query can be optionally delegated to a "query evaluator" (e.g., the query engine of a RDBMS or ODBMS) or to a "queryable collection"
 - a query evaluator may iterate over a collection of CORBA objects and access attributes or evaluate methods, or
 - it may involve other queryableCollections to evaluate subqueries and then do the join processing after retrieving the results
- Queries can only access the public attributes of CORBA objects
 - everything is based on the remote interfaces of objects
 - performance? optimization?
- There is no conceptual mapping from query language concepts (e.g., tables, object collections) to CORBA concepts provided



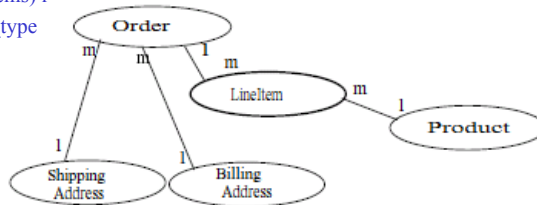
© Prof. Dr.-Ing. Stefan DeBloch

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'
```



© Prof. Dr.-Ing. Stefan DeBloch

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
```



© Prof. Dr.-Ing. Stefan DeBloch

JDO Query Language

- A JDOQL query has 3 parts
 - *candidate class*: class(es) of expected result objects
→ restriction at the class level
 - *candidate collection*: collection/extent to search over
→ (optional) restriction at the object extent level
 - *filter*: boolean expression with JDOQL (optional: other query language)
- JDOQL characteristics
 - read-only (no INSERT, DELETE, UPDATE)
 - returns JDO objects (no projection, join)
 - query submitted as string parameter → dynamic processing at run-time
 - logical operators, comparison operators: e.g. `! , = , > =`
 - JDOQL-specific operators: type cast using "`()`", navigation using "`.`"
 - no method calls supported in JDOQL query
 - sort order (*ascending/descending*)
 - variable declarations



© Prof. Dr.-Ing. Stefan DeBloch

Query

- JDO-Query with JDOQL for locating JDO instances:

```
1 String searchname = "Doe";
2 Query q = pm.newQuery();
3 q.setClass(Author.class);
4 q.setFilter("name == \"\" + searchname + \"\"");
5 Collection results = (Collection)q.execute();
6 Iterator it = results.iterator();
7 while (it.hasNext()){
8     // iterate over result objects
9 }
10 q.close(it);
```



© Prof. Dr.-Ing. Stefan DeBloch

JDOQL Examples

- Sorting:

```
1 Query query = pm.newQuery(Author.class);
2 query.setOrdering("name ascending, firstname ascending");
3 Collection results = (Collection) query.execute();
```

- Variable declaration

```
1 String filter = "books.contains(myBook) && " +
2               "(myBook.name == \"Core JDO\")";
3 Query query = pm.newQuery(Author.class, filter);
4 query.declareVariables("Book myBook");
5 Collection results = (Collection) query.execute();
```



© Prof. Dr.-Ing. Stefan DeBloch

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



© Prof. Dr.-Ing. Stefan DeBloch

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"



© Prof. Dr.-Ing. Stefan DeBloch

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



© Prof. Dr.-Ing. Stefan DeBloch

Transactions in JDO

- JDO transactions supported at the object level
- Datastore Transaction Management (standard):
 - JDO synchronizes transaction with the persistence layer
 - transaction strategy of persistence layer is used
- Optimistic Transaction Management (optional):
 - JDO progresses object transaction at object level
 - at commit time, transaction is synchronized with persistence layer
- Transactions and object persistence are orthogonal

object characteristics	transactional	non-transactional
persistent	standard	optional
transient	optional	standard (JVM)



© Prof. Dr.-Ing. Stefan DeBloch

Transactions and Concurrency Control

- 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



© Prof. Dr.-Ing. Stefan DeBloch

Summary

- 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



© Prof. Dr.-Ing. Stefan DeBloch

Summary (2)

- Query Support
 - CORBA: queries over object collections
 - no uniform query language
 - uses SQL, OQL
 - persistent object schema?
 - EJB-QL: queries over abstract persistence schema
 - limited functionality, only for definition of Finder methods
 - more or less a small SQL subset
 - JDO: queries over collections, extents
 - limited functionality
 - proprietary query language
 - 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!



© Prof. Dr.-Ing. Stefan DeBloch