Java Persistence API (JPA)

Fritz Solms

March 9, 2015
Java Persistence API (JPA)

- Java API for accessing, persisting, and managing data
  - persisted in a relational database
  - or in any other database if JPA provider available.
- Persist POJOs (Plain Old Java Objects)
  - no interface to implement
  - no class to sub-class
  - flexible and light-weight
What is JPA?

- Std framework to persist entities to DB
- Initially std API for O/R mappers
  - abstract from
    - O/R framework (e.g. Hibernate, EclipseLink, ...),
    - from RDBMS & SQL flavour
  - use object cache to improve scalability & performance.
- Now also abstract fully from DB technology
  - JPA adapters for NOSQL DBs
  - JPA/JDO for OO DBs
Why Use JPA?

1. Decoupling/abstraction
   - from JPA-provider, Technology-specific query language and Persistence Technology
   - no vendor lock-in

2. Reduce code bulk
   - plumbing code removed (code only business logic)

3. Reduce errors & improve consistency
   - mapping complex object graphs onto relational databases is error-prone
   - database schemas and structure created from object graphs

4. Improve performance
   - through object caching

5. Maintainability & Portability
   - Through code reduction, decoupling and JPA being a widely supported public standard.

6. JPA Criteria
   - Have powerful framework for dynamic query construction
What Does JPA Provide?

- O/R mapping
- Persisting, removing, querying, updating
- Object caching
  - with eager and lazy data retrieval
- Value objects and merging
- Object-Oriented query language with mapping onto technology-specific query language
  - including pre-compiled queries
- Concurrency support
- Constraint validation support
- Custom converters
- Dynamic query construction
- Calling stored procedures
Wide range of JPA implementations

1. EclipseLink
2. Hibernate
3. OpenJPA
4. DataNucleus
5. ...

Compete on non-functional attributes
- and on extensions
- avoid/ring-fence extensions ⇒ remain portable & flexible
What is a Persistence Context?

**Definition**

A persistence context is a cache of objects whose persistence is managed.

- **Cache**
  - maintains objects in memory for efficient access
  - no need to consult DB every time.

- **Persistence context**
  - maintains set of non-shared DB connections
  - only 1 instance of object with ID in DB
What is an Entity Manager?

- Entity resource manager associated with a persistence context
  - used by application code to access entities
  - maintains cache for PC
  - incl life cycle of objects in cache
  - interacts with O/R mapper
  - uses connection pool to interact with persistence provider (e.g. DB)

- Change propagation to persistence provider
  - automatic on transaction boundaries

- Detaching entities as value objects
  - and merging back into PC
Optimistic Concurrency Control

- Commonly required for highly scalable systems.
  - Version or time-stamp checking used to
    - conflicting updates across transactions, and to
    - prevent lost updates within a transaction.

- Similar to concurrent version control system
  - different persistence contexts obtain own detached copies of entities
  - whichever commits first merged changes into global PC
  - conflicts result in exception
Two types of persistence contexts supported in JPA:

1. transaction-scoped persistence context
2. extended persistence context

which are across transactions.
Transaction-Scoped Persistence Contexts

- Mainly used in non-managed apps.
- Separate PC (and cache) for each transaction.
  - Entities detached at end of transaction
    - changes no longer persisted through to database.
  - No support for optimistic locking.
Extended Persistence Contexts

- PC maintains cache across transactions
  - App servers typically provide extended transactions
  - More efficient caching
  - Optimistic locking
  - Important for scalability
2 transaction types:

1. **RESOURCE LOCAL**
   - JPA delegates transaction management to local resource (e.g. DB)
   - Usually used in simple non-managed (stand-alone) apps
     - Often require only one resource (typically DB).
     - Use JTA API to demarcate transaction in code
     - start, commit, roll-back transaction

2. **JTA**
   - JPA uses its own or app-server’s JTA transaction manager
   - Can enlist multiple resources within transaction, e.g.
     - databases
     - queues, . . .
   - transaction boundaries usually determined by app server
     - from transaction requirements annotations
     - e.g. requires, requires-new, . . .
Scope of Persistence Context

- **Scope** = domain of entities managed by PC
  - collection of entities managed by PC

Specified in one of following ways:

1. **orm.xml**
   - persistence.xml specifies `<mapping-file>`
   - orm.xml specifies entities with
     - mappings onto DB (e.g. field to column, inheritance strategy, ...)
     - converters

2. **jar file**
   - persistence.xml specifies `<jar-file>` containing entities

3. **persistence.xml** specifies entities as list of `<class>` elements

4. Annotated entities contained in root of persistence unit
   - jar file or directory containing META-INF directory with persistence.xml file
How is an Entity Manager Obtained?

- **Managed context**
  - dependency injection or JNDI lookup

- **Non-managed Context**
  - Create entity manager manually
Obtaining Entity Manager in Container Managed Environment

- Use JNDI lookup or dependency injection
  - For JTA entity manager
    - annotate entity manager field with @PersistenceContext
  - For RESOURCE_LOCAL entity manager
    - annotate entity manager field with @PersistenceUnit
Manual Creation of Entity Manager in JavaSE App

Create entity manager manually:
- define persistence.xml or setup PC properties in code
  - persistence unit descriptor in META-INF
- Instantiate entity manager factory
  - specifying persistence unit name
- Request entity manager from factory

1. `EntityManagerFactory emf = Persistence.createEntityManagerFactory(
   persistenceUnitName);`
2. `EntityManager em = emf.createEntityManager();`
Detaching Objects from Persistence Context

- Object detached from persistence context if
  - leaves persistence context (serialization)
    - Note: Due to lazy loading, detached object may not be fully populated.
  - manually detached via

```java
1 entityManager.detach(myEntity);
```

- entity out-lives persistence context
- detached object no longer managed by entity manager
- changes
  - not reflected in cache and
  - not propagated to DB on transaction commit.
- Merged back into persistence context via

```java
1 entityManager.merge(myEntity);
```
Persistence Context Configuration

- Persistence context configuration depends on whether
  - managed application
  - non-managed application
Configuring PC for Non-Managed App

- Need to specify
  - DB
  - DB driver
  - login credentials
  - scope of PC

- Scope:
  - can be specified in orm.xml (all configuration outside code)
  - Usually more convenient to specify with jar file containing annotated entities
Example persistence.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
    <persistence-unit name="myPersistenceUnit" transaction-type="RESOURCE_LOCAL">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <jar-file>myEntities.jar</jar-file>
        <properties>
            <property name="eclipselink.target-database" value="DERBY"/>
            <property name="eclipselink.ddl-generation" value="drop-and-create-tables"/>
            <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver"/>
            <property name="javax.persistence.jdbc.url" value="jdbc:derby://localhost:1527/myDB;create=true"/>
            <property name="javax.persistence.jdbc.user" value="myApp"/>
            <property name="javax.persistence.jdbc.password" value="myApp"/>
        </properties>
    </persistence-unit>
</persistence>
```
Configuring PC in Managed App

- Done in `persistence.xml` in META-INF directory
- Typically use
  - JTA-based TC
  - Container published datasource (& connection pool)
  - Usually applies to all entities in ejb-jar.
  - Additionally some configuration

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="myEnterpriseApp" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>jdbc/myDataSource</jta-data-source>
    <properties>
      <property name="eclipselink.ddl-generation" value="create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```
Entities

- Entities are persistent objects
  - exist from when created until explicitly removed.
  - persisted in DB & outlive span of session and potentially application
Simple entities

- Entities without relationships
  - to other entities
Declaring Entities

- annotate with `javax.persistence.Entity`

```java
import javax.persistence.Entity;

@Entity
public class Account {
    ...
}
```

- Can customize persistence
  - e.g. specify table:

```java
@Entity(name="ACCOUNTS")
public class Account {
    ...
}
```
Requirements for Entities

- public or protected default constructor
- Primary key (field or composite)
- Implement Serializable
- Neither entity class nor any methods final
- All annotations either on fields or on methods, not on both.
  - Can explicitly specify via \texttt{@Access(AccessType.FIELD)} or \texttt{@Access(AccessType.Property)}.
- Can subclass entities or non-entities.
- May be abstract.
What is persisted?

- persistent state of entity defined by its fields
  - fields accessed either
    - directly (through reflection or byte code weaving), or
    - via getters
  - determined by whether primary key annotation of field or accessor
Valid Persistent Field Types

The following are valid data types for persistent fields:

- Java primitives and primitives wrappers,
- The following built-in classes:
  - java.lang.String,
  - java.util.Date (requiring the @Temporal annotation),
  - java.math.BigDecimal and BigInteger,
  - java.sql.Time and Timestamp (requiring the @Temporal annotation),
  - byte[], Byte[], char[] and Character[],
  - java.sql.Blob and Clob,
- embedded classes,
- other entities,
- collections of primitives, and
- any other serializable objects.
Collection Variables

- Collections automatically persisted
  - Supported collections:
    - `java.util.List`, `java.util.Set` and `java.util.Map`
  - Should use generics
    - (e.g. `List<Account>`)  
  - Mapped onto collection tables.
  - Relationship annotations control mapping.
    - e.g. join tables for many-to-many
**Transient Fields**

- **Fields which should not be persisted**
  - declared *transient* by either
    - annotating accessor with `@javax.persistence.Transient`, or
    - in the case of field access, declaring field *transient*. 
Field Validation

- Can perform field validation in *setter* methods.
  - *setter* may throw exception
    - will cause controlling transaction (if any) to be rolled back.
- **Note:**
  - Consider using bean validation.
  - Questionable whether validation should be done on entity
    - rather enforced at services level (e.g. session beans)
Embeddable Classes

- Entity components objects
  - which are expanded in table for entity

- Example: Location
  - has geographic location with longitude & latitude as component.
  - want to save in location table
    - as geographicLocation.longitude/latitude columns

- Note:
  - Embedded objects have no persistent identity
    - identity = owner identity + role
  - Use only for composition relationships.
Defining an Embeddable Class

- Annotate with @Embeddable instead of @Entity

```java
@Embeddable
class GeographicLocation implements Serializable {
    // getters & setters
    private double degreesLongitude, degreesLattitude;
}
```
Specifying the Access Type

- Default:
  - access type of embeddable class determined by access type of enclosing entity.

- Can change/explicitly by annotating embeddable class with either
  - @Access(AccessType.Field) or
  - @Access(AccessType.Property)

- Should specify explicitly
  - prevent O/R mapping error
    - e.g. when enclosed within entity of one access type which is, in turn, enclosed in entity of another access type.
Embedding a Class within an Entity

- Annotate field or accessor as @Embedded

```java
@Entity
class Location implements Serializable {
    ...

    @Embedded
    public GeographicCoordinates getCoordinates() {
        return coordinates;
    }

    private String name;
    ...

    private GeographicCoordinates coordinates;
}
```
Primary Keys

- Every entity
  - must have primary key specified.
- JPA supports both,
  - simple keys, and
  - composite keys
Simple primary keys

- generally preferable
  - simple primary key
    - which is independent of business semantics.
- Persisted to single DB column
  - assigned primary key constraint on DB
Valid Data Types for Simple Primary Keys

- Valid types:
  - Java primitives
  - Primitive wrappers
  - java.lang.String

- Floating point types legal
  - Should not be used
Specifying the Primary Key Field

- Can be specified
  - on field, or
  - on accessor method

```java
@Entity
public class Account {
    public int getAccountNo() { return accountNo; }
    ...  
    @Id
    private int accountNo;
}
```
Commonly want entity manager / database
to automatically generate primary key

Annotate primary key field (or accessor):

```java
import java.io.Serializable;
import javax.persistence.*;

@Entity
public class Account implements Serializable {

@Id
@GeneratedValue(strategy=GenerationType.AUTO)
private long accountNumber;
}
```
Primary Key Classes
(Composite Keys)

- Composite primary keys
  - defined as *embeddable class*
    - properties form primary key fields.
  - must have
    - default constructor
    - accessors for primary key fields
Interface for Primary Key Class

```java
package za.co.solms.partsCatalog;

/**
 * Interface for a part identifier.
 */
public interface PartId {
    public String getCode();

    public String getManufacturerId();

    public interface Mutable extends PartId {
        public void setCode(String newCode);
        public void setManufacturerId(String newManufacturer);
    }
}
```
Implementation of a Primary Key Class

```java
package za.co.solms.partsCatalog;

import java.io.Serializable;
import javax.persistence.Embeddable;

@Embeddable
public class PartPK implements PartId.Mutable, Serializable {
    public PartPK(String code, String manufacturerId) {
        setCode(code); setManufacturerId(manufacturerId);
    }

    protected PartPK() {}

    public void setCode(String newCode) {this.code = newCode;}

    public void setManufacturerId(String newManufacturer) {
        this.manufacturerId = newManufacturer;
    }

    public String getCode() {return code;}

    public String getManufacturerId() {return manufacturerId;}

    private String code, manufacturerId;
}
```
package za.co.solms.partsCatalog;

import javax.persistence.EmbeddedId;
import javax.persistence.Entity;

@Entity public class PartBean implements Part.Mutable {

    public PartPK getPartId() {return partPk;}

    public String getDescription() {return description;}

    public String getName() {return name;}

    public void setDescription(String newDescription) {
        this.description = newDescription;
    }

    public void setName(String newName) {
        this.name = newName;
    }

    public void setPartId(PartId newPartId) {
        this.partPk = new PartPK(newPartId.getCode(), newPartId.getManufacturerId());
    }

    @EmbeddedId
    private PartPK partPk = new PartPK();

    private String name, description;
}
Specifying Column Mappings

- O/R mapping can be customized
  - in `orm.xml` entity descriptor
  - in code using annotations
- Can, for example, specify for column mapping
  - column name, length, precision, ...

```java
@Entity(name="VHCL")
public class Vehicle {
    @Column(name="REG_NO" length="10")
    public String getRegistrationNumber() {
        ...
    }
}
```
Typical example

- entries in column must be unique
- column entry may not be empty
  - i.e. entity object must have property populated

```java
@Entity(name="VHCL")
public class Vehicle
{
    @Column(name="REG_NO", length="10", nullable=false, unique=true)
    public String getRegistrationNumber()
    {
        ...
    }
}
```
Primitive Collections and Maps

- Can annotate collections & maps
  - as @ElementCollection
- If RDBMS
  - mapped onto separate table
    - with link & value columns
  - maps have additional key column.
- Can customize mapping: table name, map-key column name, ...

```java
@Entity
public class CarPriceList implements Serializable {

  @ElementCollection(fetch = FetchType.LAZY)
  @CollectionTable(name = "UnavailableCars")
  public List<String> withdrawnCars;

  @ElementCollection(fetch = FetchType.EAGER)
  public Map<String, double> activeCarPrices;
}
```
Relationships between entities
- persisted through to persistent storage
- if RDBMS, foreign keys, link tables, . . .

Limited support for standard OO relationships:
- association and aggregations,
- composition, and
- specialization.
Summary of UML Relationships

- **Customer**: user, "weak uses" - dependency
- **Automatic Teller Machine**: service provider
  - Makes, at times, use of.
  - No msg path maintained.
- **Speaker**: server
  - Client maintains msg path to server.
- **Portfolio**: aggregate object, "weak has a"
  - Component state affects state of aggregate object.
  - Encapsulation (component only accessible by owner).
  - Component does not survive owner.
- **DvdPlayer**: owner, "strong has a"
- **Laser**: component
  - Inheritance
  - Pluggability
- **Homloan Application**: specialized object, "strong is a"
- **Loan Application**: generalized object
- **Mobile Telephone**: service provider, "weak is a"
  - Realization
  - Contract
- **Java Persistence API (JPA)**
- **Entities**
- **Relationships**

Here we consider association for client server. Association is also used purely for navigability.
Composition

- Component does not survive owner.
  - supported in JPA via cascading relationship attribute.
    - cascading create, merge, remove.
JPA supports
- uni-directional relationships
- bi-directional relationships
- one-to-one relationships
- one-to-many relationships
- many-to-one relationships
- many-to-many relationships
For each relationship
  - one class/object = relationship owner
    - maintains pointer (e.g. foreign key)
  - If bi-directional
    - related entity also provides msg path to relationship owner.
Uni-Directional Single-Valueded Relationships

```java
@Entity
public class Address {
    ...
}

@Entity
public class Customer {
    public Address getAddress() {return address;}
    public void setAddress(Address addr) {address = addr;}

    @OneToOne(cascade=CascadeType.All)
    private Address address;
}
```

- specify cascade=CascadeType.All for composition
Bi-Directional One-To-One Relationships

```java
@Entity
public class PurchaseOrder {
    public Invoice getIssuedInvoice() { return invoice; }
    public void setIssuedInvoice(Invoice inv) { invoice = inv; }
    ...  
    @OneToOne
    private Invoice issuedInvoice;
}

@Entity
public class Invoice {
    public PurchaseOrder getPurchaseOrder() { return purchaseOrder; }
    public void setPurchaseOrder(PurchaseOrder order) { purchaseOrder = order; }
    ... 
    @ManyToOne
    @JoinColumn(name="customerInvoice")
    private PurchaseOrder order;
}
```
Bi-Directional Many-To-One Relationships

```java
@Entity
public class Client {

  public Collection<Account> getAccounts() {
    return accounts;
  }
  ...

  @OneToMany(mappedBy="client")
  private Collection<Account> accounts;

@Entity
public class Account {

  public Client getClient() {
    return client;
  }

  public void setClient(Client clnt) {
    client = clnt;
  }

  @ManyToOne
  private Client client;

  @ManyToMany
  private Client client;```
```
Cascade Types

- Cascade types supported in JPA:
  - **ALL**: Cascade all operations
  - **MERGE**: Cascade merge (update) operation
  - **PERSIST**: Cascade persist operation
  - **REFRESH**: Cascade refresh operation
  - **REMOVE**: Cascade remove operation
  - **DETACH**: Cascade detach operations

- From OO perspective
  - use none for association
  - use ALL for composition
Fetching Strategies

- Performance optimization based on usage expectations
  - How much of object graph is loaded into cache.
    - **EAGER/LAZY**
    - fetch/don’t-fetch associations entity

```
@Entity
public class Order {
    ...
    @ManyToOne(fetchType = FetchType.EAGER)
    public Client getClient()
    {
        ...
    }
}
```

- **Defaults:**
  - one-to-one & many-to-one → **EAGER**
  - one-to-many & many-to-many → **LAZY**
Specialization

- Specialization is not support in RDBMS
  - Need some mapping to support OO ↔ Relational mapping
    - JPA supports mapping of specialization
    - and polymorphism through to persistent storage.
  - JPA supports range of mapping strategies for specialization
    - None is perfect for all cases.
    - Selected based on quality attribute trade-off decisions
    - In particular: performance and scalability vs maintainability & improved semantics
Relational DB Mapping

- 4 types of mappings:
  1. Joined subclass
  2. Single table per class hierarchy
  3. Table per class
  4. Mapped superclass
Joined Subclass

- Usually the preferred mapping.
  - Each class in hierarchy persisted in its own table.
  - Subclass has primary key column
    - acts as foreign key to primary key column of its superclass.
    - object identity preserved across abstractions of object.

- Annotation inherited
  - i.e. mapping strategy need only be specified on ultimate base class of hierarchy.

```java
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Person {...}

@Entity
public class Employee extends Person {...}

@Entity
@InheritanceJoinColumn(name="EMPLOYEE_REF")
public class Contractor extends Employee {...}
```
Single Table Per Class Hierarchy

- Entire class hierarchy persisted in single table
  - Table has
    - primary key column
    - columns for fields of all classes in hierarchies
    - discriminator column identifying class
- Sparsely populated
- Single lookup
  - No table joins.

```java
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE
discriminatorType=DiscriminatorType.STRING
discriminatorValue="Person" /*default: fully qualified class name*/)
@DiscriminatorColumn(name="Type") // default: "TYPE"
public class Person {...}

@Entity @Inheritance(discriminatorValue="Employee")
public class Employee extends Person {...}

@Entity @Inheritance(discriminatorValue="Contractor")
public class Contractor extends Employee {...}
```
Each concrete subclass is represented in a table
  containing all defined and inherited fields + primary key

Implications
  Not normalized
  Modify superclass ⇒ modify tables for all concrete subclasses.
  Single table lookup for each object retrieval

```java
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Person {...}

@Entity
public class Employee extends Person {...}

@Entity
public class Contractor extends Employee {...}
```
At times, don’t want
  - separate table for abstract superclass
  - single table for class hierarchy
Want
  - embed superclass fields in table of concrete subclass
⇒ MappedSuperclass
  - Typically used if
    - abstract superclass has only very few fields
    - don’t want overheads of query on another small table
Implications

- Not normalized
  - maintainability traded off for performance and scalability
- Retain pluggability and polymorphism.
- Abstract superclass not annotated as entity
  - since no table is created.

Usage:

- Annotate abstract base entity as @MappedSuperclass
- Can still specify inheritance strategy for class hierarchy.

@MappedSuperclass is effectively @Embeddable

- Composition → @Embeddable class
- Specialization → @MappedSuperclass
Java Persistence API (JPA)

Entities

Example

```java
package mappedSuperclass [chargeables]

// Chargeable
<<mappedSuperclass>>
Chargeable

// Product
<<mappedSuperclass>>
Product
- price : double

// Service
<<mappedSuperclass>>
Service

// Account
<<entity>>
Account

// Disease
<<entity>>
Disease

// Consultation
<<entity>>
Consultation
- duration : double

// BloodTest
<<entity>>
BloodTest
```

Relationships

1..*
Chargeable

```java
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.MappedSuperclass;
import javax.persistence.OneToOne;

@MappedSuperclass @Inheritance(strategy=InheritanceType.JOINED)
public abstract class Chargeable
{
    public Chargeable() {}

    public Account getIncomeAccount() {return incomeAccount;}
    public void setIncomeAccount(Account incomeAccount) {
        this.incomeAccount = incomeAccount;
    }

    public long getCode() {return code;}
    public void setCode(long code) {this.code = code;}

    @OneToOne
    private Account incomeAccount;

    @Id @GeneratedValue
    private long code;
}
```
```java
import javax.persistence.MappedSuperclass;

@MappedSuperclass
public class Service extends Chargeable {
    public Service() {}
}
```
import javax.persistence.MappedSuperclass;

@MappedSuperclass
public class Product extends Chargeable {
    public Product() {}
    public double getPrice() {return price;}
    public void setPrice(double price) {this.price = price;}
    double price;
}
import javax.persistence.Entity;

@Entity
public class Consultation extends Service {
    public Consultation() {}

    public double getDuration() {
        return duration;
    }

    public void setDuration(double duration) {
        this.duration = duration;
    }

    private double duration;
}
import java.util.Set;

import javax.persistence.Entity;
import javax.persistence.OneToMany;

@Entity
public class BloodTest extends Service {
    public BloodTest() {}

    public Set<Disease> getDiseaseCheckList() {
        return diseaseCheckList;
    }

    public void setDiseaseCheckList(Set<Disease> diseaseCheckList) {
        this.diseaseCheckList = diseaseCheckList;
    }

    @OneToMany
    private Set<Disease> diseaseCheckList;
}
Java Persistence Query Language (JPQL)

- technology-neutral OO query language
  - supports querying across object graph
- object graph queries
  - mapped onto query language of persistence provider
  - e.g. SQL for a specific RDBMs
  - OQL
  - xpath
JPQL versus SQL

Structure of JPQL query similar to that of SQL query:

- **SELECT**
  - an entity,
  - a value object or
  - a primitive data type

- **FROM**
  - specifies domain to which query applies and

- **WHERE**
  - specifies constraints which restrict result collection.

e.g.

```
SELECT a FROM Account a WHERE a.balance > 0
```
Result Collections in JPQL

- SQL: result table
- JPQL:
  - collection of objects which are instances of either
    - Entities,
    - Instances of *embedded classes*,
    - Java primitives,
    - Instances of Java result objects populated from query,
  - either `java.util.Collection` or `java.util.Set`
Selecting entity Attributes

- can use element access operator to select specific attributes:
- e.g.

```
SELECT a.balance FROM Account a
```

- returns collection of all account balances.
  - of the data type of the balance
  - e.g. instance of Money or a Double
Similar to SQL: 3 statement types:

1. **Select statements**
   - select data from persistent storage,

2. **Update statements**
   - modify information maintained in persistent storage.

3. **Delete statements**
   - used information held in persistence storage.
Elements of JPQL Query Statements

- **SELECT clause:**
  - determines types of objects returned — either
    - objects retrieved from persistent storage, or
    - objects constructed from information retrieved from persistent storage

- **FROM clause:**
  - constrains domain from which selection is done,
  - an optional WHERE clause
    - constrain collection of objects selected from above domain,
  - an optional GROUP BY clause
    - supports grouping query results into specified groups,
  - an optional HAVING clause
    - used in conjunction with GROUP BY clause
    - in order to filter over aggregated groups, and

- **an optional ORDER BY clause**
  - to request ordering of returned result objects.
Elements of UPDATE and DELETE statements

- UPDATE & DELETE statement have only
  - UPDATE/DELETE clause
  - optional WHERE clause
Polymorphism

- JPQL are intrinsically polymorphic
  - all statement elements on target beans
    - automatically apply to all specializations
  - Polymorphic collections are commonly returned.
Navigating Object Graphs

- JPQL = OO query language
  - ⇒ query along associations
- Special types of associations
  - aggregation
  - composition
  - specialization
  - primitive fields
Simple Paths

Example JPQL query

```java
SELECT b.bondAccount FROM Bond b
```

Equivalent SQL statement

```sql
SELECT account from Account, Bond
WHERE Bond.bondAccount = Account.id
```

Query can span multiple nodes

```sql
SELECT bond.bondAccount.balance FROM Bond bond
```
Single-Valued versus Multi-Valued Paths

- Query paths must be single valued
  - branching at most on last node

```sql
1  SELECT client.bonds.account FROM Client client  -> INVALID
```
Further Single vs Multi-Valued Paths

- SQL:

  1. `SELECT Course.name from Course, Presentation WHERE Presentation.course = Course.id`

- Corresponding JPQL single valued:

  1. `SELECT p.course.name FROM Presentation p`

- But following is multi-values path (invalid):

  1. `SELECT p.presenters.course.name FROM Presentation p`
Specifying the Source of a Query

- **FROM clause**
  - specifies/constrains domain of query
  - as a particular entity type
## Selecting from Multiple Domains

- **FROM clause:**
  - can select from multiple domains
  - delimited by commas:

```java
1 SELECT DISTINCT election
2 FROM Election election, Election election94
3 WHERE election.turnout > election94.turnout AND
4    election.country = 'South Africa' AND
5    election94.year = '1994'
```

*Find all election results which had greater attendance than SA’s 1994 election*
Joins

- combine 2 or more entities which have some common property.
Inner Joins

- used to select from *inclusion set*
  - obtained by join condition over different entities
- Can be specified
  - explicitly
  - implicitly via Cartesian product
Implicit Inner Joins

- join multiple paths from multiple entities implicitly
- Example:

```java
1 SELECT DISTINCT c
2 FROM Customer c, ServiceProvider sp
3 WHERE c.companyRegistrationNo = sp.id
```

*Find companies which are both, customers and service providers.*
Explicit Inner Joins

- specify the join entities explicitly
- e.g.

```sql
SELECT bonds.account
FROM Client c INNER JOIN c.bonds bonds
WHERE c.address.city = 'Johannesburg'
```

- can abbreviate INNER JOIN to just JOIN
- Above equivalent to

```sql
SELECT bonds.account
FROM Client c IN (c.bonds) bonds
WHERE c.address.city = 'Johannesburg'
```
Left Outer Joins
(Fetch Joins)

- Inner join
  - retrieves objects satisfying join condition

- Outer (fetch) join does same PLUS
  - return objects from left collection
  - for which no matching object in right collection

- Example

  Assume: *one-to-zeroOrOne* relationship btw book & publisher

```
SELECT b FROM Book b LEFT JOIN b.publisher p WHERE p.address.country = 'South Africa'
```

- Retrieve all book entities, irrespective of whether they have publisher or not
- Also loads all publishers of books who are in South Africa into cache.
Collapsing Multi-Valued Paths into Single-Valued Paths

- **SELECT & WHERE clauses:**
  - restricted to single-valued paths
- **Collapse multi-valued path into single-valued path**
  - using collection variable using `IN` clause:
- **Example:**

```sql
1 SELECT client.bonds.account FROM Client client -> INVALID
2
3 SELECT bonds.account FROM Client c, IN(c.bonds) bonds -> VALID
```

- **Similarly**

```sql
1 SELECT p.presenters.course.name FROM Presentation p ->&gt; INVALID
2
3 SELECT ps.course.name FROM Presentation p, IN(p.presenters) ps ->&gt; VALID
```
Constraining a result set via a WHERE clause

- restrict elements returned in result collection.
- e.g.

```
1 SELECT Object(c)
2     FROM Course c
3 WHERE c.presenters NOT EMPTY
```
Java Persistence API (JPA)
The Java Persistence Query Language (JPQL)
Constraining a result set via a WHERE clause

Comparison & Logical Operators

- Operators
  - ==, <, >, <=, >=, <>
  - BETWEEN, LIKE, IN, IS NULL, EMPTY, MEMBER OF which can all be inverted by combining them with a
    logical operators AND, NOT, OR.

- Examples:

```sql
1 SELECT a FROM album a WHERE a.year NOT BETWEEN 1980 AND 2005
2 3 SELECT s FROM soccer_club s where s.home.city IN ('London', 'Madrid', 'Rio de Janeiro')
4 5 select c FROM customer c WHERE c.email LIKE '%ac.za'
```
Calculation Operators

- Arithmetic operators:
  - + - * /
- Calculation operators:
  - MAX, MIN, SUM, MOD, AVG, COUNT, SQRT
- String operators’:
  - LENGTH, LOCATE, SUBSTRING, UPPER, LOWER, CONCAT
Using Collection Variables in WHERE Clauses

- Commonly used to collapse multi-valued paths into single value paths.
- For example
  - Convert the invalid multi-valued path
    
    ```sql
    SELECT Object(c)
    FROM Course c
    WHERE c.prerequisites.name = 'Programming in Java'
    ```
    
    into a single-valued path
    ```sql
    SELECT Object(c)
    FROM Course c, IN(c.prerequisites) p
    WHERE p.name = 'Programming in Java'
    ```
Constructing Result Objects

- Result of query
  - always single object or collection of objects
  - instead of returning persisted objects (e.g. entities, embedded objects, ...)
  - return collection of new objects
    - populated from entities.

- Example:

```sql
1 SELECT NEW za.co.academics.UniversityInfo (u.name, u.address, c.name c.registeredStudents)
   JOIN u.course c WHERE c.registeredStudents > 100
```

- creates a list of result objects
  - populated from university and course entities.
JPQL supports nested queries
  - which have sub-queries embedded
    - within the conditional expression of a **WHERE** or **HAVING** clause.

**Example:**
- Select best student(s) on a course:

```sql
1  SELECT s FROM student s where s.courseResults.average
2    = (SELECT MAX(s.courseResults.average) from student s)
```
Ordering

- Can use ORDER BY clause
  - followed by either ASC or DESC

Example:

```
SELECT s FROM stadium s WHERE s.numSeats >= 10000 ORDER BY numSEATS DESC
```

- return list of stadiums which can seat at least 10000 spectators in order of no of spectators

- Can use multiple comma-separated criteria:

```
SELECT s FROM stadium s WHERE s.numSeats >= 10000 ORDER BY s.numSEATS DESC, s.age ASC
```
**Grouping**

- **GROUP BY**
  - aggregation of values according to set of properties
- **HAVING**
  - conditions to further restrict query result
- **Example:**

```java
1 SELECT j.publisher, count(j.circulation)
2 FROM journal j
3 GROUP BY j.publisher
4 HAVING COUNT(j.circulation) > 100000
```

*Retrieve*
Query Parameters

- Can specify query input as
  - positional parameters
  - named parameters
- Query input can be used only in
  - WHERE clause
  - HAVING clause
Positional Parameters

- specified as \(?1, \?2, \?3, \ldots\)
  - numbering starts at 1
- positional parameter can be used multiple times
  - in same query

```sql
SELECT sf FROM soccerFixture sf WHERE (sf.date \(\geq\) :date1) AND (sf.date \(\leq\) :date2)
```
Named Parameters

- prefixed with colon
- names are case-sensitive
- Example

```java
SELECT sf FROM soccerFixture sf WHERE (sf.date >= :date1) AND (sf.date <= :date2)
```
Constructing and Executing Queries

Use Entity manager to

- construct JPQL query
- execute queries

```java
1 List<Product> products = entityManager.createQuery
2     ("SELECT p FROM Product p WHERE p.description like :descr")
3     .setParameter("descr", description)
4     .setMaxResults(30)
5     .setFirstResult(pageNo*30)
6     .getResultList();
```
Java Persistence API (JPA)
Constructing and executing queries
Named queries

**Named Queries**

- statically defined queries with
  - predefined unchangeable query string
- typically pre-compiled.
- defined via `@NamedQuery` (e.g. on entities or session beans)

```java
@NamedQuery(name="bonds.getAllAbove",
            query="select b from Bond b where b.balance &gt;= :amount")
```

- named query instantiated & executed via

```java
Query query = entityManager.createNamedQuery("bonds.getAllAbove");
query.setParameter(0, new Double(500000));
List<Bond> list = (List<Bond>)query.getResultList();
```
JPA Converters

- At times data type in DB $\neq$ data type in Java class
  - e.g. Boolean $\rightarrow$ 0, 1 or list $\rightarrow$ comma-delimited string
  - $\Rightarrow$ Upon accessing persistent storage.
    - need to convert between data types
    - $\Rightarrow$ Need custom converter.
@Converter
public class BooleanToIntConverter implements AttributeConverter<Boolean, String>
{
    public int convertToDatabaseColumn(Boolean value)
    {
        if (value)
            return 1;
        else
            return 0;
    }

    public Boolean convertToEntityAttribute(int value)
    {
        return (value == 1);
    }
}
Applying Custom Converters

```java
@Entity
public class Order {
    ...
    @Convert(converter=BooleanToIntConverter.class)
    private Boolean priorityShipping;
    ...
}
```
Default Converters

```java
@Entity
public class Order {

... // No custom converter specified
private Boolean priorityShipping;
}

@Converter(autoApply=true)
public class BooleanToIntConverter implements AttributeConverter<Boolean, String> {
    public int convertToDatabaseColumn(Boolean value) {...}
    public Boolean convertToEntityAttribute(int value) {...}
}
```
Calling Stored Procedures

Define named query for stored procedure

```java
@Entity
@NamedStoredProcedureQuery(
        name = "calcAverageTemperature",
        procedureName = "CalcAvgTemp")
public class WeatherReading {
    ..
}
```

Instantiate and execute query

```java
StoredProcedureQuery query = EntityManager.createNamedStoredProcedureQuery("calcAverageTemperature");
query.registerStoredProcedureParameter(1, String.class, ParameterMode.IN);
query.setParameter(1, locationId);
query.registerStoredProcedureParameter(2, Timestamp.class, ParameterMode.IN);
query.setParameter(2, date1, TemporalType.TIMESTAMP);
query.registerStoredProcedureParameter(3, Timestamp.class, ParameterMode.IN);
query.setParameter(3, date2, TemporalType.TIMESTAMP);
query.execute();
Double averageTemp = query.getSingleResult();
```
Overview of the JPA Criteria API

- as of JPA 2.0
- JPA Criteria API provides
  - object-based API for defining JPQL queries
    - in Java code — NOT text based
Why JPA Criteria API?

- simpler dynamic query construction
- compile-time checking of query syntax (against object graph)
  - query errors not necessarily obvious:

```java
1 String jpqlQuery = "select weatherReading from WeatherReading where
weatherReading.temperature > 40";
2 Query query = em.createQuery(jpqlQuery);
3 List<WeatherReading> result = query.getResultList();
```

Correct query:

```java
1 String jpqlQuery = "select weatherReading from WeatherReading
weatherReading where weatherReading.temperature > 40";
```
Why JPA Criteria API?

- type-safe queries
  - string-based queries not type safe
  - typically suppress warnings:

```java
@SuppressWarnings("unchecked")
public List<WeatherReading> getAllWeatherReadings() {
    Query query = entityManager.createNamedQuery("findAllWeatherReadings");
    return query.getResultList();
}
```
Using Criteria API

- one assembled a query tree
  - with root node representing starting from of query

Nodes represent semantic query element, e.g.

- WHERE clauses,
- GROUP BY or ORDER BY clauses,
- sub-queries,
- ...
Generating a JPA Metamodel

- JPA criteria queries
  - assembled from instances of classes describing entities
  - ⇒ need to generate these descriptor classes
    - from entities
    - → persistence metamodel
  - Typically done via annotations processor
    - generates canonical metamodel classes when annotated entities are loaded.
- For EclipseLink, use

```java
org.eclipse.persistence.internal.jpa.modelgen.CanonicalModelProcessor
```
Maven Build Declarations to Generate the Metamodel classes

- Done in Maven's `generate-sources` life cycle phase
  - ⇒ classes available in `compile phase`
- Include in Maven build
  - Dependency on some JPA implementation
    - Not provided by Java-EE API
  - Dependency on O/R mapper used
    - e.g. *EclipseLink*
  - Configure Maven compiler plugin to compile for Java 7 or later
    - supporting annotations processing
  - Configuring Annotations processor plugin to generate canonical model
    - e.g. *Eclipse*'s `CanonicalModelProcessor`
- Required Maven repositories and plugin repositories.
Required Repositories

```xml
<repositories>
  <repository>
    <id>maven-repo</id>
    <name>Maven repository</name>
    <url>http://repo1.maven.org/maven2/</url>
  </repository>
  <repository>
    <id>EclipseLink Repo</id>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
</repositories>

<pluginRepositories>
  <pluginRepository>
    <id>maven-annotation-plugin</id>
    <url>http://maven-annotation-plugin.googlecode.com/svn/trunk/mavenrepo</url>
  </pluginRepository>
  <pluginRepository>
    <id>maven2-repository.dev.java.net</id>
    <name>Java.net Repository for Maven</name>
    <url>http://download.java.net/maven/2/</url>
  </pluginRepository>
</pluginRepositories>
```
Required Dependencies

```xml
<dependencies>
  <dependency>
    <groupId>org.eclipse.persistence</groupId>
    <artifactId>javax.persistence</artifactId>
    <version>2.0.0</version>
  </dependency>

  <dependency>
    <groupId>org.eclipse.persistence</groupId>
    <artifactId>eclipselink</artifactId>
    <version>${eclipselink.version}</version>
  </dependency>

  <dependency>
    <groupId>javax</groupId>
    <artifactId>javaee-api</artifactId>
    <version>7.0</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```
Configuring CanonicalModelProcessor

```xml
<build>
  <plugins>
    <plugin>
      <groupId>org.bsc.maven</groupId>
      <artifactId>maven-processor-plugin</artifactId>
      <executions>
        <execution>
          <id>process</id>
          <goals>
            <goal>process</goal>
          </goals>
          <phase>generate-sources</phase>
          <configuration>
            <outputDirectory>${project.build.directory}/generated-sources/apt</outputDirectory>
            <compilerArguments> -Declipselink.persistencexml=src/main/resources/META-INF/persistence.xml</compilerArguments>
            <processors>
              <processor>org.eclipse.persistence.internal.jpa.modelgen.CanonicalModelProcessor</processor>
            </processors>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```
Generated Metamodel Classes

- Metamodel → infrastructure for specifying queries

```java
package za.co.solms.weather;

import javax.annotation.Generated;
import javax.persistence.metamodel.SingularAttribute;
import javax.persistence.metamodel.StaticMetamodel;
import za.co.solms.location.Location;

@StaticMetamodel(WeatherReading.class)
public class WeatherReading_
{
    public static volatile SingularAttribute<WeatherReading, Integer> id;
    public static volatile SingularAttribute<WeatherReading, Double> humidity;
    public static volatile SingularAttribute<WeatherReading, Date> dateTime;
    public static volatile SingularAttribute<WeatherReading, Location> location;
    public static volatile SingularAttribute<WeatherReading, Ambiance> ambiance;
    public static volatile SingularAttribute<WeatherReading, Double> temperature;
}
```
A Simple Criteria-Based Query

- Get weather reading for location

```java
public List<WeatherReading> getWeatherReadingsForLocation(Location location) {
    CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
    CriteriaQuery<WeatherReading> criteriaQuery = criteriaBuilder.createQuery(WeatherReading.class);
    Root<WeatherReading> weatherReading = criteriaQuery.from(WeatherReading.class);
    Predicate predicate = criteriaBuilder.equal(weatherReading.get(WeatherReading_.location), location);
    criteriaQuery.where(predicate);
    TypedQuery<WeatherReading> query = entityManager.createQuery(criteriaQuery);
    return query.getResultList();
}
```
Query Operators

- **arithmetic operators** like sum, diff, prod, quot, min, max, avg, abs, and sqrt,
- **relational operators** like gt, ge, lt, le, equal, like and notLike,
- **logical operators** like and, or, xor, not
- **set and collection operators** like count, countDistinct, isEmpty, between, isMember, isNotMember, exists(subQuery), any(subQuery), all(subQuery)
- **sorting operators** like asc, desc
- **date/time operators** like currentDate, currentTime, and currentTimeStamp,
- **string operators** like upper, lower, concat, substring, and trim,
- **data conversion operators** like toDouble, toInteger, toLong, toString,
- and some general testOperators like isNull, isNotNull, isTrue, and isFalse.
Criteria builder can be used to create composite logical expressions from multiple predicates.

For example, to find all weather readings for a location where the temperature was above 40 degrees C:

```
1 Predicate atLocation = criteriaBuilder.equal(weatherReading.get(WeatherReading_.location), location);
2 Predicate tempGe40 = criteriaBuilder.ge(weatherReading.get(WeatherReading_.temperature), 40);
3 criteriaQuery.where(criteriaBuilder.and(atLocation, tempGe40));
4 TypedQuery<WeatherReading> query = entityManager.createQuery(criteriaQuery);
5 return query.getResultList();
```
Ordering

- add `orderBy` node to the query tree
  - `asc` or `desc` operator from criteria builder:

```java
1  CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
2  CriteriaQuery<WeatherReading> criteriaQuery
3      = criteriaBuilder.createQuery(WeatherReading.class);
4
5  Root<WeatherReading> weatherReading = criteriaQuery.from(WeatherReading.class);
6
7  Predicate atLocation
8      = criteriaBuilder.equal(weatherReading.get(WeatherReading_.location), location);
9  criteriaQuery.where(atLocation);
10
11 criteriaQuery.orderBy(criteriaBuilder.desc(weatherReading.get(WeatherReading_.temperature)));
12
13  TypedQuery<WeatherReading> query = entityManager.createQuery(criteriaQuery);
14  return query.getResultList();
```
find all students
  who have one or more enrollments
  which are not canceled:

```java
CriteriaQuery<Student> q = cb.createQuery(Student.class);
Root<Student> c = q.from(Student.class);
SetJoin<Student, Enrollment> o = c.join(Student_.enrollments);
Predicate p = cb.equal(o.get(Enrollment_.status), Status.Canceled).negate();
c.where(p);
```