Java Persistence API

Established in Being Perform Action

How data is handled

RDBM

- Data is stored in tables of rows and columns.
- Primary keys identify data.
- Process data through triggers, stored procedures.
- Data exists even if the RDMS is not running.
- Foreign keys & joins establish relations.

Java

- Stores business data in attributes (either primitive types or instances of objects).
- Manipulates data through methods using objects.
- Data exists only if the JVM is running.
- Inheritance, interfaces, enumerations, abstract classes, …

ORM – Object Relational Mapping

- Maps the object oriented world to the relational database world.
- A world of classes, objects, and attributes is mapped to tables of rows and columns.
- Delegate the access to database and conversion from DB to OO to an external framework.
- Metadata is used to describe mapping.
  - Annotations
  - XML
- Convention over configuration, configuration by exception, programming by exception.

Java Persistence API

- JPA components
  - ORM
  - Entity Manager API (CRUD) operations.
  - JPQL (Java Persistence Query Language) OOQL.
  - Transactions, JTA & non-JTA
  - Callbacks and listeners.
  - Optional XML mapping.
- JPA 1 was introduced with Java EE5 – matched Hibernate
- JPA 2 was introduced with Java EE 6
- Built on top of JDBC - but JDBC is transparent to us

Entity

- Objects that live shortly in memory, and persistently in a database.
- Mapped to database.
- Can be concrete or abstract.
- Support inheritance, associations, and so on.
- Manageable by JPA.
- Can be queried using JPQL, HQL
- Follows a defined life cycle.
- An annotated Java Class (or an XML descriptor).
  - @Entity
- Must:
  - Have id @Id @GeneratedValue
  - Have a default constructor.
  - Not final.
  - Implement Serializable when it has to be passed as detached objects.

Entity Lifecycle

- Entity is a POJO.
- When managed by Entity Manager (attached)
  - it has a persistence identity.
  - It has a synchronized state with a database.
- When not managed they are detached
  - Used as regular java classes (garbage collected)
- Operations on entity:
  - Persist (Create)
  - Load (Read)
  - Update (Update)
  - Delete (Delete)

Identity

- Choosing Primary Key:

If the property can change then it is not a good selection, always better to use a surrogate key.

Very simple to do with annotations:

```
@Id@GeneratedValue
  private Long id;
```

Object Relationships

- One-to-One
  - Unidirectional
  - Bidirectional
- One-to-Many
  - Unidirectional
  - Bidirectional
- Many-to-One
  - Unidirectional
  - Bidirectional
- Many-to-Many
  - Unidirectional
  - Bidirectional

One-to-One

Student

PK	name	…	FK
1	Jack	…	101
2	Jill	…	201
3	John	…	205

Course

PK	title	…	…
101	Into to	…	…
201	Data ..	…	…
205	Prog..	…	…

Does this look right?

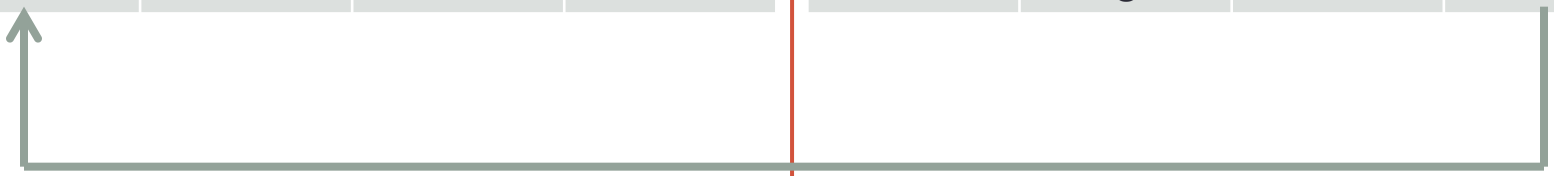
One-to-One

Student

PK	name	…	…
1	Jack	…	…
2	Jill	…	…
3	John	…	…

Course

PK	title	…	FK
101	Into to	…	1
201	Data ..	…	2
205	Prog..	…	3



Does this look right?

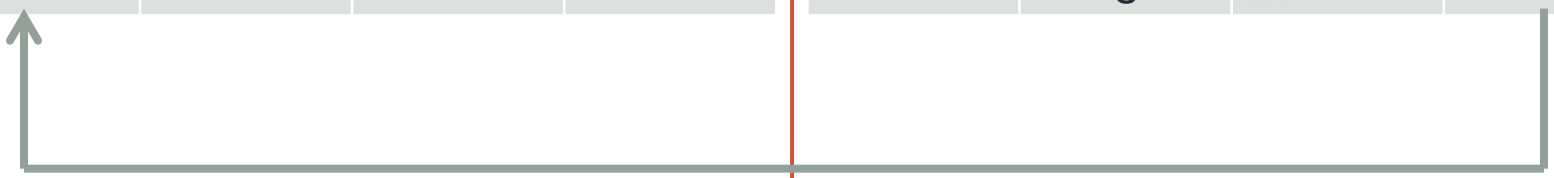
One-to-Many

Student

PK	name	…	…
1	Jack	…	…
2	Jill	…	…
3	John	…	…

Course

PK	title	…	FK
101	Into to	,,,	1
201	Data ..	,,,	1
205	Prog..	,,,	3



Look right?

Many-to-Many

Student

PK	name	…	…
1	Jack	…	…
2	Jill	…	…
3	John	…	…

Course

PK	title	…	…
101	Into to	,,,	…
201	Data ..	,,,	…
205	Prog..	,,,	…

Student PK	Course PK
1	101
2	201
3	205

Mapping cardinality

- Unidirectional: the owner has the reference.
- Bidirectional:
  - The owner has the @JoinColumn or @JoinTable definition
  - The inverse owner has the mappedBy definition

- Annotations:
- @OneToOne, @OneToMany, @ManyToMany
  - @JoinColumn
  - @JoinTable
- Bidirectional
  - mappedBy element.

One-to-One

Student

```
@Entity
public class Student {

   @Id@GeneratedValue
   private Long id;
   private String name;
   …
   @OneToOne
   @JoinColumn(name=“book_fk”)
   private Book bookReading;
}
```

Book

```
@Entity
public class Book{

   @Id@GeneratedValue
   private Long id;
   private String title;
   …
   @OneToOne(mappedBy
“bookReading”)
   private Student student;
}
```

One-to-Many One Direction

Student

```
@Entity
public class Student {

   @Id@GeneratedValue
   private Long id;
   private String name;
   …
   @OneToMany
   @JoinTable(name = “jnd_std_bk”)
   private List <Book > bookList;
}
```

Book

```
@Entity
public class Book{

   @Id@GeneratedValue
   private Long id;
   private String title;
   …
//unidirectional – comment out:
//private Student student;
}
```

One-to-Many Bi-directional

Block

```
@Entity
public class Block {
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
private long id;

@NotEmpty
private String blockName;

@ManyToOne
private Entry myEntry;
//Block is the owner
```

Entry

```
@Entity
public class Entry{

   @Id@GeneratedValue
   private Long id;

@NotEmpty
private String entryName;

@OneToMany(mappedBy = "myEntry")
    private List<Block> myBlockList = new ArrayList<Block>();;

//Entry is inverse owner
```

Many-to-Many Bi-directional

Course

```
@Entity
public class Course {

 @Id@GeneratedValue
   private Long id;
    private String courseName;
    private String courseNumber;

@ManyToMany
    private List<Faculty>
myFacultyList;

}  // owner of association
```

Faculty

```
@Entity
public class Faculty{

   @Id@GeneratedValue
   private Long id;
   private String firstName;
   private String lastName;

@ManyToMany(mappedBy =
"myFacultyList")
    private List<Course>
myCourseList;}  //inverse owner
```

Mapping Cardinality

- Issues:
- Performance -- we generally assume that creating join tables are expensive. We use join tables for one to many and many to many – just join columns for one-to-one
- mappedBy causes it to be treated as bidirectional and identifies the owning classes attribute.

Fetching related entities

- Loading related entities can be done
  - Eagerly (when owning entity is read)
  - Lazily (when attribute is read)
- It depends on the frequency of accessing entities.
- Fetching method affects performance.

Fetching Related Entities

- Use EAGER – all loaded at the same time
- Use LAZY – must request the related object
- For our project it is fine to use Defaults

Fetching Defaults

- One-to-One: EAGER
- Many-to-One: EAGER
- One-to-Many: LAZY
- Many-to-Many: LAZY

Entity Manager

- Orchestrates and Manages entities.
- CRUD operations on entities.
- Execute complex queries using JPQL.
- It is an interface, that is implemented by different Persistence Providers.
- We use the simplest interface to the Entity Manager (i.e. let the Entity Manager do all the transactions)

Java Persistence Query Language

- Is similar to SQL except you use objects instead of tables.
- SQL

```
SELECT s.name FROM Student s
  WHERE s.gpa > 3
```

- JPQL/HQL

```
SELECT s.name FROM Student s
  WHERE s.gpa > 3
```

- JPQL /HQL query types
  - Dynamic – created at run time
  - Static - Named queries
  - Native – SQL statements

Java Persistence API - Main Points

• Java Persistence API implements the Data Access Object and Data Transfer Object design patterns to streamline the development of enterprise applications.

• By using the JPA design mechanism and the Hibernate implementation mechanism we build an efficient persistence layer for our application with minimal database design effort.

We can then concentrate on developing our business logic for our application.

- **Science of Consciousness**

.By experiencing pure consciousness during TM we create a rested and coherent brain as our most useful tool in building our software. This is the key to "do less, accomplish more".