JPA Modelling: OneToMany, ManyToOne

Object Relational Mapping - ORM

- Object oriented programming languages Vs Relational DBMS

  - OO -> Classes, Objects, Methods, Inheritance, Polymorphism

  - Relational Model -> Tables, Row, Columns, Keys, Stored Procedures.

Impedance mismatch

OO model	Relational model
Class, object	Table, row
Attributes	Columns
Identity	Primary key
Methods	Stored procedures
Inheritance	Not supported
Polymorphism	Not supported

Java Persistence API

- JPA is an API

    - Implemented by a persistence provider

- Some persistence providers

    - Hibernate from JBoss

    - TopLink from Oracle

- JPA defined a runtime Entity Manger API processing queries and transaction on the objects against the database.

- It is defined a objects-level query language JPQL to allow querying of the objects from the database.

Persistence API-Entities

- Entity
  - A table in a database.
  - An Entity is represented by a class
  - Instance of entity corresponds to a row in the database.
  - A number of rules cover the implementation of an entity class.

Persistence API-Entity Relationships

- Relationships are the same as relationships between tables in a database:
  - One-To-One:
    - Each instance of the entity (row) is related to single instance of another entity (to a single row of another table).
  - One-To-Many:
    - An entity instance can be related to more than one instance of another entity. But an instance of the other Entity can only relate to one instance of the first.

Persistence API-Entity Relationships

- Many-To-One:
  - Multiple instances (rows) of an Entity can be related to one instance of another Entity. But an instance of the other Entity can relate to only one instance of the first Entity.

- Many-To-Many:
  - An instance of one Entity relates to many instances of another Entity and vice versa.

- For every relationship there is an *owning* side and *inverse* side.
  - The relationship can be unidirectional –it has only an owning side.
  - Or, bidirectional – it ha s both an owning side and an inverse side.

Example Relationships
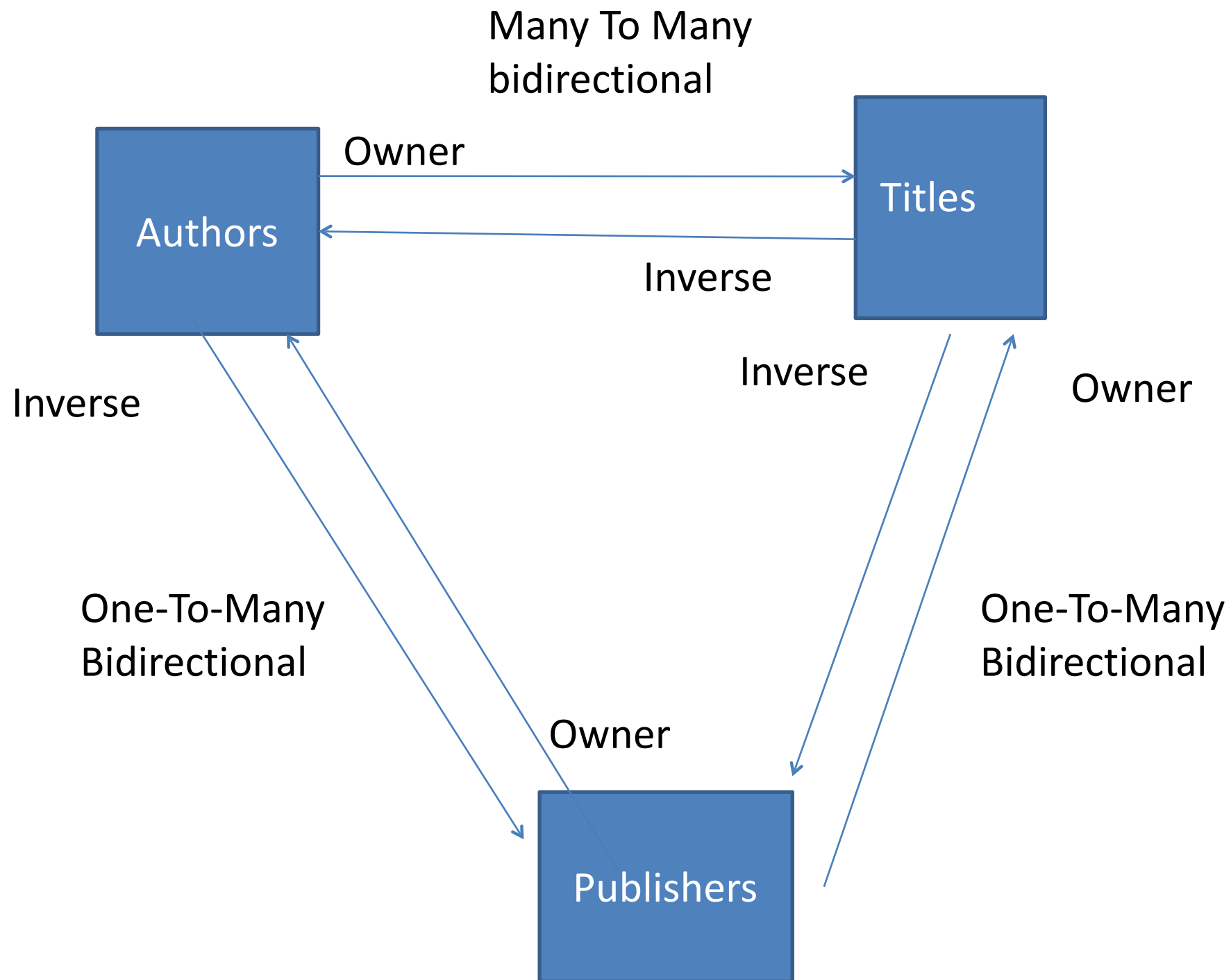
The class EntityManager

- EntityManager is the most important class of JPA
  - Full name javax.persistence.EntityManager
- Some methods of EntityManager
  - T find(primaryKey)
  - Query createQuery(String jpql)
    - Creates a JPQL query
  - Query createNativeQuery(String sql)
    - Creates a SQL query
- Some methods of Query
  - List getResultList()
    - Executes a Query and returns a list of objects

Java Persistence Query Language (JPQL)

- Very much like ordinary SQL - But not specific to any DBMS

- JPA converts JPQL to ordinary SQL for the actual DBMS

Entity Lifecycle

- Entity Objects are carefully managed by JPA Implementations

- This involves support for transactions, commit, rollback and various operations required by a resilient enterprise applications
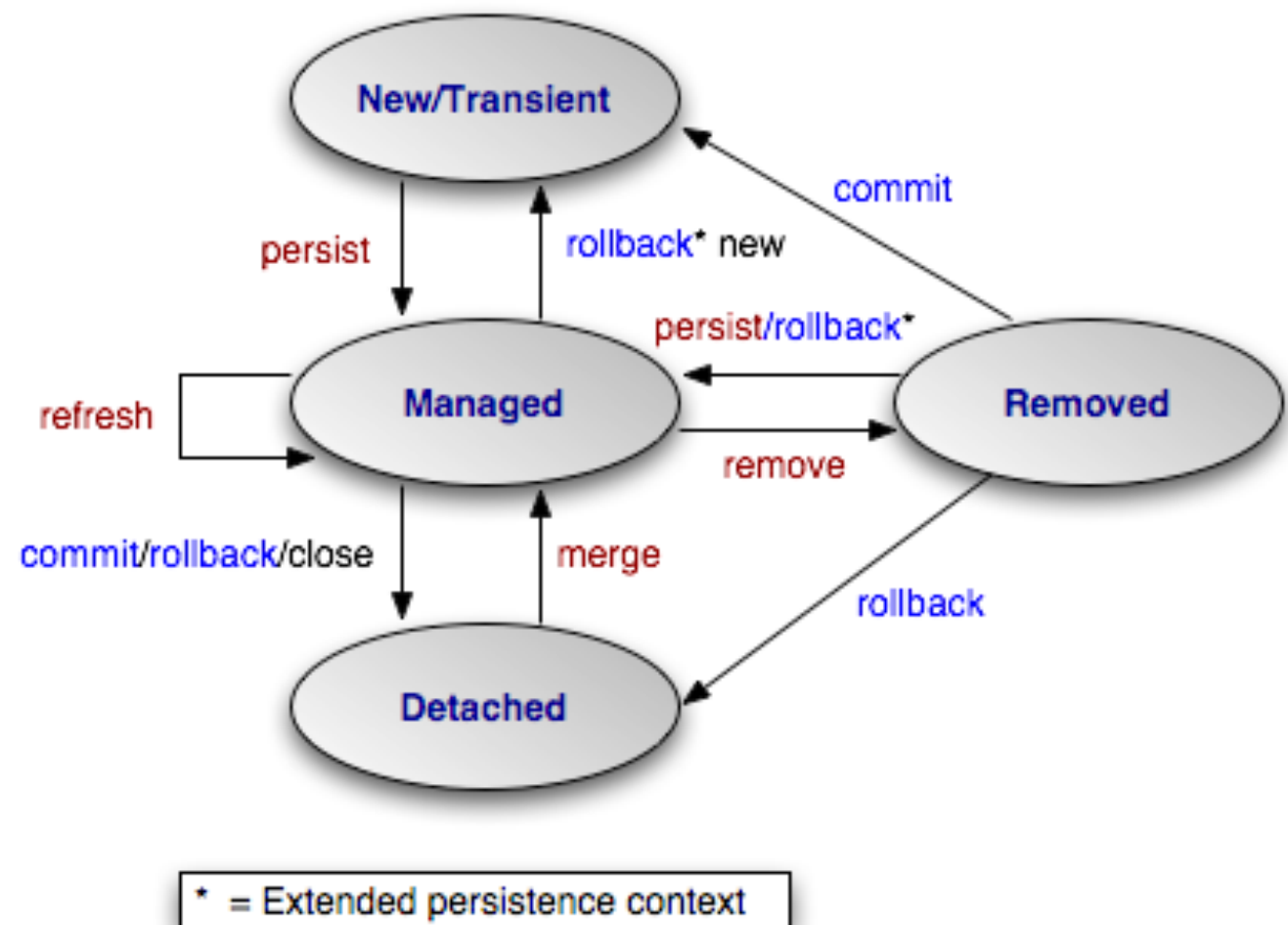
Play & JPA

- Play simplifies and encapsulates JPA

  - Play will automatically start the Hibernate entity manager when it finds one or more classes annotated with the @javax.persistence.Entity annotation.

  - When the JPA entity manager is started you can get it from the application code, using the JPA helper.

```
public static index()
{
    Query query = JPA.em().createQuery("select * from Article");
    List<Article> articles = query.getResultList();
    render(articles);
}
```

Transaction management

- Play will automatically manage transactions.

- It will start a transaction for each HTTP request and commit it when the HTTP response is sent. If your code throws an exception, the transaction will automatically rollback.

- If you need to force transaction rollback from the application code, you can use the JPA.setRollbackOnly() method, which tells JPA not to commit the current transaction.

The play.db.jpa.Model support class

- This is the main helper class for JPA. If you make one of your JPA entities extend the Model class, it will give you a lot of helper methods to simplify the JPA access.

- The Model class automatically provides an auto generated Long id field.

```java
@Entity
public class Post extends Model
{
    public String title;
    public String content;
    public Date postDate;

    @ManyToOne
    public Author author;

    @OneToMany
    public List<Comment> comments;
}
```

Finding objects

- Find by ID

```
Post aPost = Post.findById(5L);
```

- Find all

```
List<Post> posts = Post.findAll();
List<Post> posts = Post.all().fetch();
```

- Find using a simplified query

```
// 100 max posts
List<Post> posts = Post.all().fetch(100);
// 100 max posts start at 50
List<Post> posts = Post.all().from(50).fetch(100);
```

```
Post.find("byTitle", "My first post").fetch();
Post.find("byTitleLike", "%hello%").fetch();
Post.find("byAuthorIsNull").fetch();
Post.find("byTitleLikeAndAuthor", "%hello%", connectedUser).fetch();
```

Simplified Queries

- Simple queries follow the syntax [Property] [Comparator]And?

- **LessThan** - less than the given value
- **LessThanEquals** - less than or equal a give value
- **GreaterThan** - greater than a given value
- **GreaterThanEquals** - greater than or equal a given value
- **Like** - Equivalent to a SQL like expression, except that the property will always convert to lower case.
- **like** - Similar to a Like, except case insensitive, meaning that your argument will convert to lower case too.
- **Elike** - Equivalent to a SQL like expression, no conversion.
- **NotEqual** - Negates equality
- **Between** - Between two values (requires two arguments)
- **IsNotNull** - Not a null value (doesn't require an argument)
- **IsNull** - Is a null value (doesn't require an argument)

Explicit Save

- All the persistent objects extending the Model class will not be saved without an explicit call to the save() method.

- This differs from default Hibernate implementations, which implicitly manage the object lifecycle, synchronising with transactions support

- The Play model is simpler to grasp, but can lead to some complexities when editing / changing objects in relationships

Explore JPA via JUnit Testing

- The most direct way of learning JPA - or any persistence technology - is to write simple unit tests.

- These tests should explore all aspects of the potential relationships, particularly:

  - Create

  - Read

  - Update

  - Delate

- Rely on the unit tests to yield verified API sequence calls.

JPA Model Project

Start by creating a brand new Play project. Do this by determining the parent folder (most likely your workspace) and running a command prompt. Then type:

```
play new jpamodel
```

Once this has completed, change into the folder just created (jpamodel) and run the eclipsify command:

```
cd jpamodel
play eclipsify
```

You can now import the project into eclipse in the usual way.
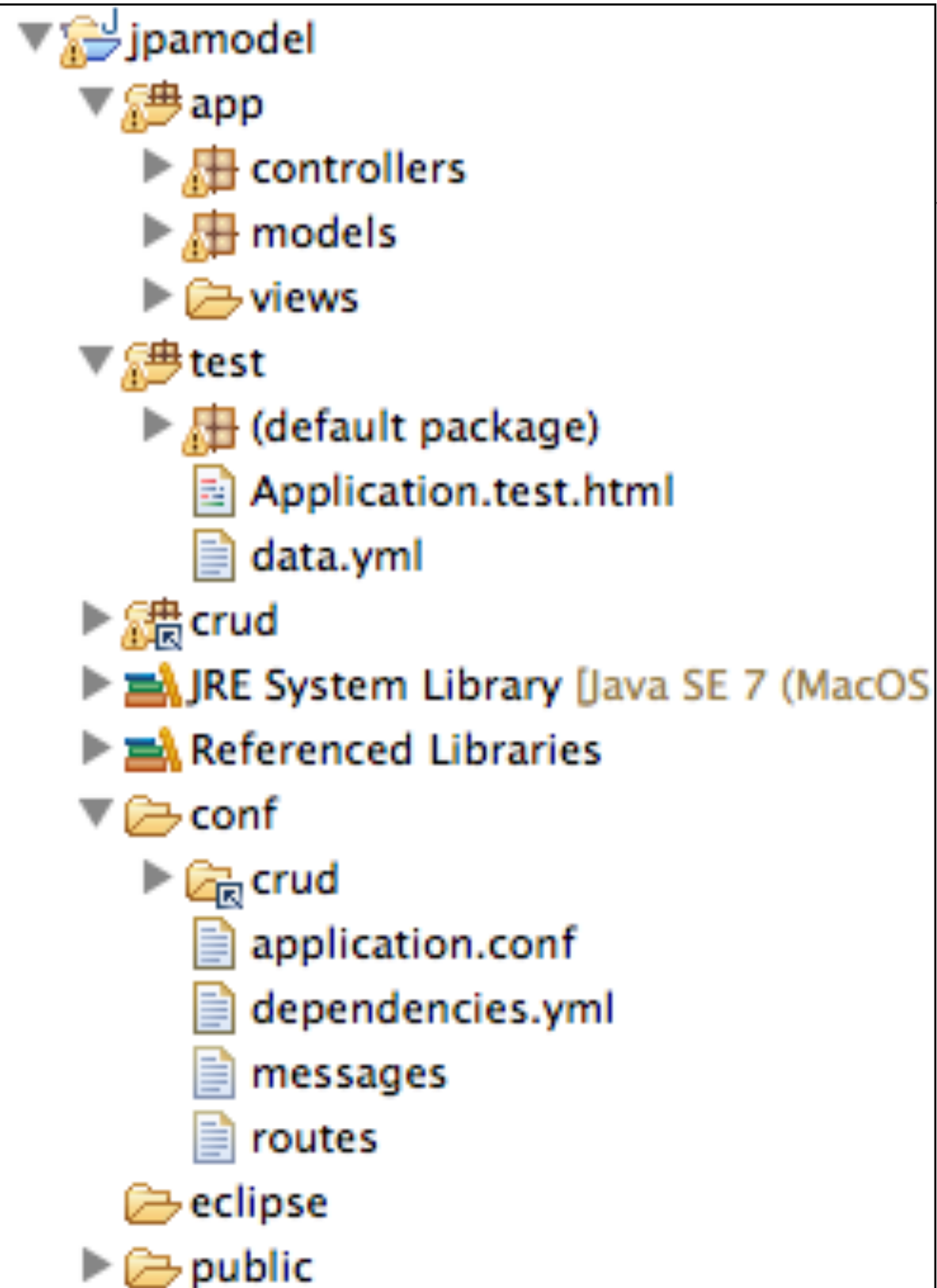
jpamodel
- app
  - controllers
  - models
  - views
- test
  - (default package)
  - Application.test.html
  - data.yml
- crud
- JRE System Library [Java SE 7 (MacOS
- Referenced Libraries
- conf
  - crud
  - application.conf
  - dependencies.yml
  - messages
  - routes
- eclipse
- public

Club Class

```java
package models;

import javax.persistence.Entity;
import play.db.jpa.Model;

@Entity
public class Club extends Model
{
  public String name;

  public Club(String name)
  {
    this.name = name;
  }
}
```

Player Class

```java
package models;

import javax.persistence.Entity;
import play.db.jpa.Model;

@Entity
public class Player extends Model
{
  public String name;

  public Player(String name)
  {
    this.name = name;
  }
}
```

ClubTest

```java
import org.junit.*;

import java.util.*;
import play.test.*;
import models.*;

public class ClubTest extends UnitTest
{
  @Before
  public void setup()
  {
  }


  @After
  public void teardown()
  {
  }


  @Test
  public void testCreate()
  {

  }
}
```

PlayerTest

```java
import org.junit.*;

import java.util.*;
import play.test.*;
import models.*;

public class PlayerTest extends UnitTest
{
  @Before
  public void setup()
  {
  }


  @After
  public void teardown()
  {
  }


  @Test
  public void testCreate()
  {

  }
}
```

Run the app now in 'test' mode:

```
play test
```

...and navigate to the test runner page:

- `http://localhost:9000/@tests`

Select the Club and Player tests - and they should be green.

Also try the database interface:

- `http://localhost:9000/@db`

PlayerTest

```java
public class PlayerTest extends UnitTest
{
  private Player p1, p2, p3;

  @Before
  public void setup()
  {
    p1 = new Player("mike");
    p2 = new Player("jim");
    p3 = new Player("frank");
    p1.save();
    p2.save();
    p3.save();
  }

  @After
  public void teardown()
  {
    p1.delete();
    p2.delete();
    p3.delete();
  }

  @Test
  public void testCreate()
  {
  }
}
```

N⟂ | ⟳ | ☑ Auto commit ↶ ↷ | Max rows: `1000` ⇕ ▶ ■ | ⬆ | Auto complete `Normal` ⇕ ?

jdbc:h2:mem:play
- ⊟ club
  - ⊞ id
  - ⊞ name
  - ⊞ Indexes
- ⊟ player
  - ⊞ id
  - ⊞ name
  - ⊞ Indexes
- ⊞ information_schema
- ⊞ Sequences
- ⊞ Users
- ⓘ H2 1.3.166 (2012-04-08)

Run (Ctrl+Enter) Clear SQL statement:

```
SELECT * FROM CLUB
```

SELECT * FROM CLUB;

ID	NAME

(no rows, 25 ms)

Edit

toString + //@After

```java
public class Player extends Model
{
  public String name;

  @ManyToOne
  public Club club;

  public Player(String name)
  {
    this.name = name;
  }
}
```

- We can use @db interface while project is in 'test' mode

- Enables us to understand model as we evolve classes and their relationships

```java
public class PlayerTest extends UnitTest
{
  private Player p1, p2, p3;

  @Before
  public void setup()
  {
    p1 = new Player("mike");
    p2 = new Player("jim");
    p3 = new Player("frank");
    p1.save();
    p2.save();
    p3.save();
  }

  //@After
  public void teardown()
  {
    p1.delete();
    p2.delete();
    p3.delete();
  }

  @Test
  public void testCreate()
  {
    Player a = Player.findByName("mike");
    assertNotNull(a);
    assertEquals("mike", a.name);
    Player b = Player.findByName("jim");
    assertNotNull(b);
    assertEquals("jim", b.name);
    Player c = Player.findByName("frank");
    assertNotNull(c);
    assertEquals("frank", c.name);
  }
}
```

jdbc:h2:mem:play

Run (Ctrl+Enter) Clear SQL statement:

⊟ 📋 club
  ⊞ 🔲 id
  ⊞ 🔲 name
  ⊞ ↓ᵃᶻ Indexes
⊟ 📋 player
  ⊞ 🔲 id
  ⊞ 🔲 name
  ⊞ ↓ᵃᶻ Indexes
⊞ 📁 information_schema
⊞ ⬚ Sequences
⊞ 👥 Users
ⓘ H2 1.3.166 (2012-04-08)

SELECT * FROM CLUB

SELECT * FROM CLUB;

ID	NAME
1	tramore
2	dunmore
3	fenor
4	tramore
5	dunmore
6	fenor

(6 rows, 3 ms)

Edit

```java
private Player p1, p2, p3;

public void setup()
{
  p1 = new Player("mike");
  p2 = new Player("jim");
  p3 = new Player("frank");
  p1.save();
  p2.save();
  p3.save();
}
```

30

Some Player Tests

```java
@Test
public void testCreate()
{
  Player a = Player.findByName("mike");
  assertNotNull(a);
  assertEquals("mike", a.name);
  Player b = Player.findByName("jim");
  assertNotNull(b);
  assertEquals("jim", b.name);
  Player c = Player.findByName("frank");
  assertNotNull(c);
  assertEquals("frank", c.name);
}

@Test
public void testNotThere()
{
  Player a = Player.findByName("george");
  assertNull(a);
}
```

ClubTest

```java
public class ClubTest extends UnitTest
{
  private Club c1, c2, c3;

  @Before
  public void setup()
  {
    c1 = new Club("tramore");
    c2 = new Club("dunmore");
    c3 = new Club("fenor");
    c1.save();
    c2.save();
    c3.save();
  }

  @After
  public void teardown()
  {
    c1.delete();
    c2.delete();
    c3.delete();
  }
```

```java
  @Test
  public void testCreate()
  {
    Club a = Club.findByName("tramore");
    assertNotNull(a);
    assertEquals("tramore", a.name);
    Club b = Club.findByName("dunmore");
    assertNotNull(b);
    assertEquals("dunmore", b.name);
    Club c = Club.findByName("fenor");
    assertNotNull(c);
    assertEquals("fenor", c.name);
  }

  @Test
  public void testNotThere()
  {
    Club a = Club.findByName("bunmahon");
    assertNull(a);
  }
}
```

Multiplicity & Navigation
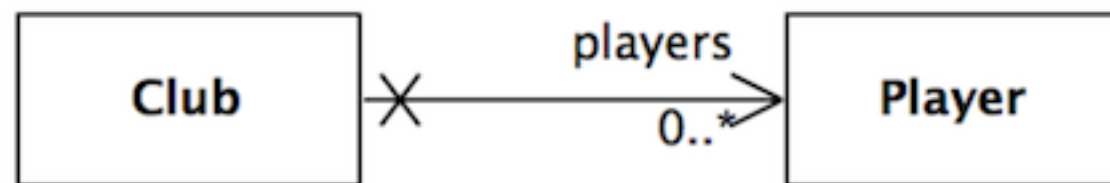
- Club has a collection of zero or more players

- Players are unaware of Club

Implementation Relationship in Java Classes

```java
public class Club extends Model
{
  public String name;

  @OneToMany(cascade=CascadeType.ALL)
  public List<Player> players;


  public Club(String name)
  {
    this.name = name;
    this.players = new ArrayList<Player>();
  }


  public String toString()
  {
    return name;
  }


  public void addPlayer(Player player)
  {
    players.add(player);
  }
}
```

```java
public class Player extends Model
{
  public String name;


  public Player(String name)
  {
    this.name = name;
  }
}
```

34

Testing the Player / Club Relationship

- Use the fixture to set up some club / relationships

```java
@Before
public void setup()
{
  p1 = new Player("mike");
  p2 = new Player("jim");
  p3 = new Player("frank");

  c1 = new Club("tramore");
  c2 = new Club("dunmore");
  c3 = new Club("fenor");

  c1.addPlayer(p1);
  c1.addPlayer(p2);

  c1.save();
  c2.save();
  c3.save();
}
```

testPlayers

- In the test, see if these relationships have been established

```
@Test
public void testPlayers()
{
  Club tramore = Club.findByName("tramore");

  assertEquals (2, tramore.players.size());

  Player mike  = Player.findByName("mike");
  Player jim   = Player.findByName("jim");
  Player frank = Player.findByName("framk");

  assertTrue  (tramore.players.contains(mike));
  assertTrue  (tramore.players.contains(jim));
  assertFalse (tramore.players.contains(frank));
}
```

testRemovePlayers

- Removing relationships must also be tested

```java
@Test
public void testRemovePlayer()
{
  Club tramore = Club.findByName("tramore");
  assertEquals(2, tramore.players.size());

  Player mike  = Player.findByName("mike");
  assertTrue(tramore.players.contains(mike));
  tramore.players.remove(mike);
  tramore.save();

  Club c = Club.findByName("tramore");
  assertEquals(1, c.players.size());

  mike.delete();
}
```

Explore the Relationship in the Database

```java
@Before
public void setup()
{
  p1 = new Player("mike");
  p2 = new Player("jim");
  p3 = new Player("frank");

  c1 = new Club("tramore");
  c2 = new Club("dunmore");
  c3 = new Club("fenor");

  c1.addPlayer(p1);
  c1.addPlayer(p2);

  c1.save();
  c2.save();
  c3.save();
}
```
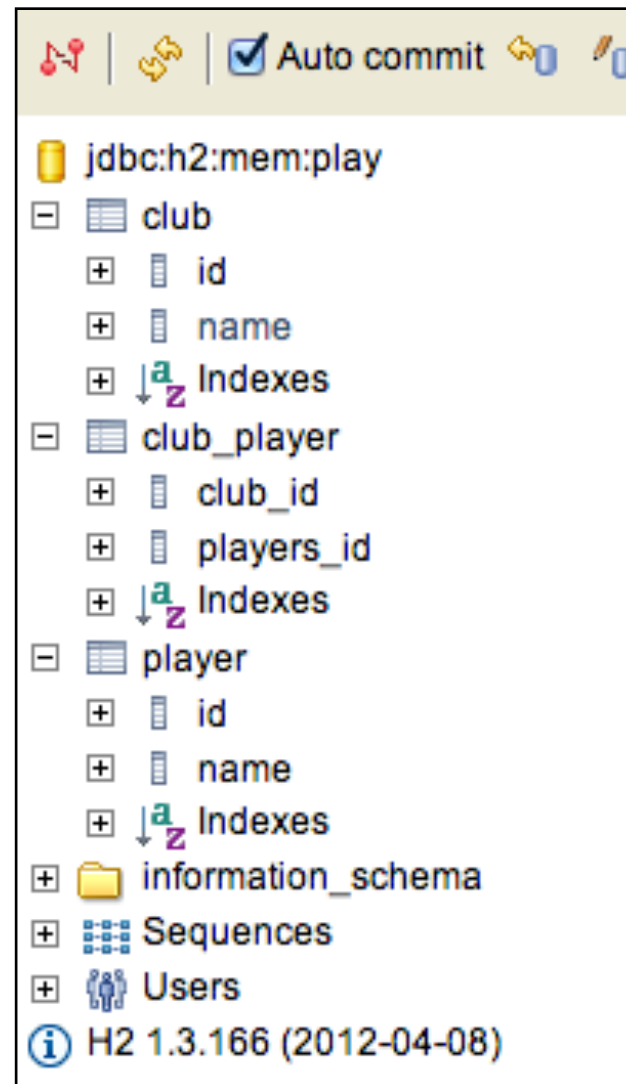

SELECT * FROM CLUB;

ID	NAME
1	tramore
2	dunmore
3	fenor

(3 rows, 3 ms)

SELECT * FROM PLAYER;

ID	NAME
1	mike
2	jim

(2 rows, 2 ms)

Edit

SELECT * FROM CLUB_PLAYER;

CLUB_ID	PLAYERS_ID
1	1
1	2

(2 rows, 4 ms)

Edit

Bidirectional Relationship
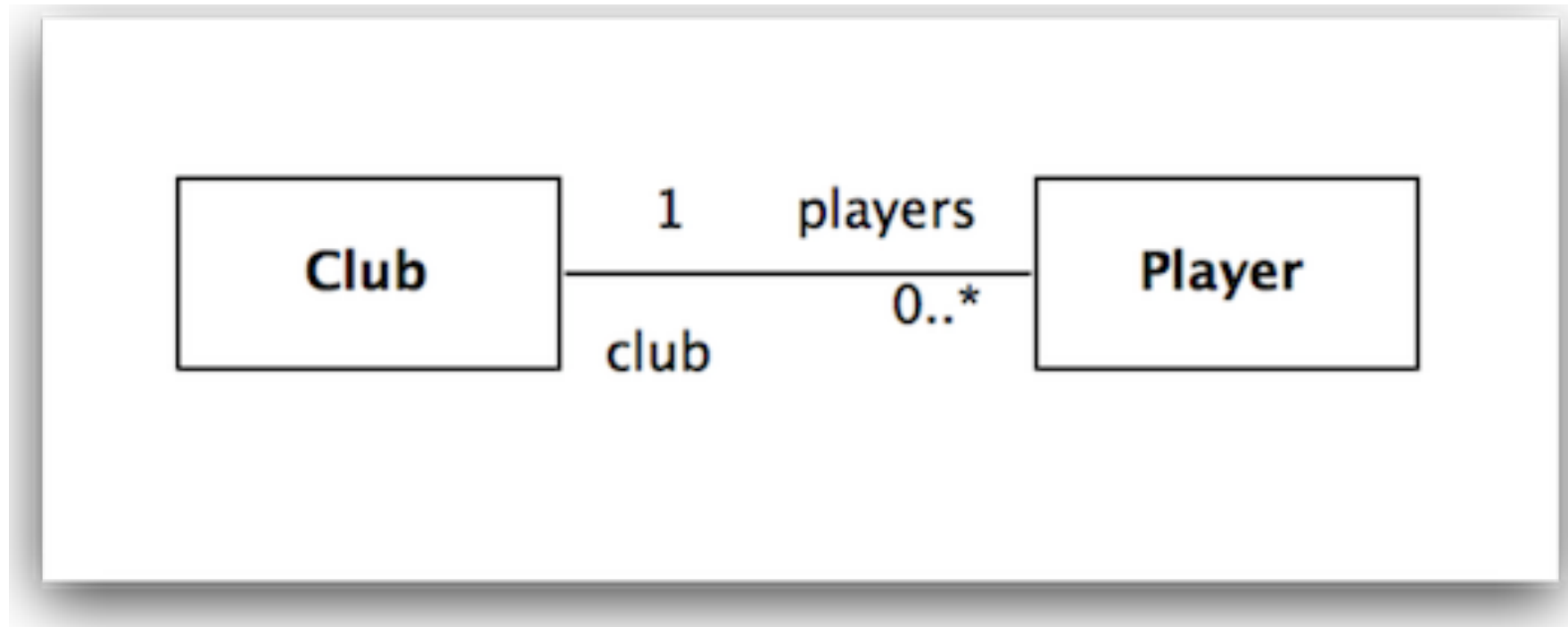
- Club has a 'one to many' relationship with players

- Player has a 'many to one' relationship with club

Bidirectional Relationship in Java Classes

```java
public class Club extends Model
{
  public String name;

  @OneToMany(mappedBy="club", cascade=CascadeType.ALL)
  public List<Player> players;


  public Club(String name)
  {
    this.name = name;
    this.players = new ArrayList<Player>();
  }

  public String toString()
  {
    return name;
  }

  public void addPlayer(Player player)
  {
    player.club = this;
    players.add(player);
  }
}
```

```java
public class Player extends
Model
{
  public String name;

  @ManyToOne
  public Club club;

  public Player(String name)
  {
    this.name = name;
  }

  public String toString()
  {
    return name;
  }
}
```

SELECT * FROM CLUB;

ID	NAME
1	tramore
2	dunmore
3	fenor

(3 rows, 3 ms)

Edit

SELECT * FROM PLAYER;

ID	NAME	CLUB_ID
1	mike	1
2	jim	1

(2 rows, 2 ms)

Edit

Database tree:

- jdbc:h2:mem:play
  - club
    - id
    - name
    - Indexes
  - player
    - id
    - name
    - club_id
    - Indexes
  - information_schema
  - Sequences
  - Users
  - H2 1.3.166 (2012-04-08)

Auto commit