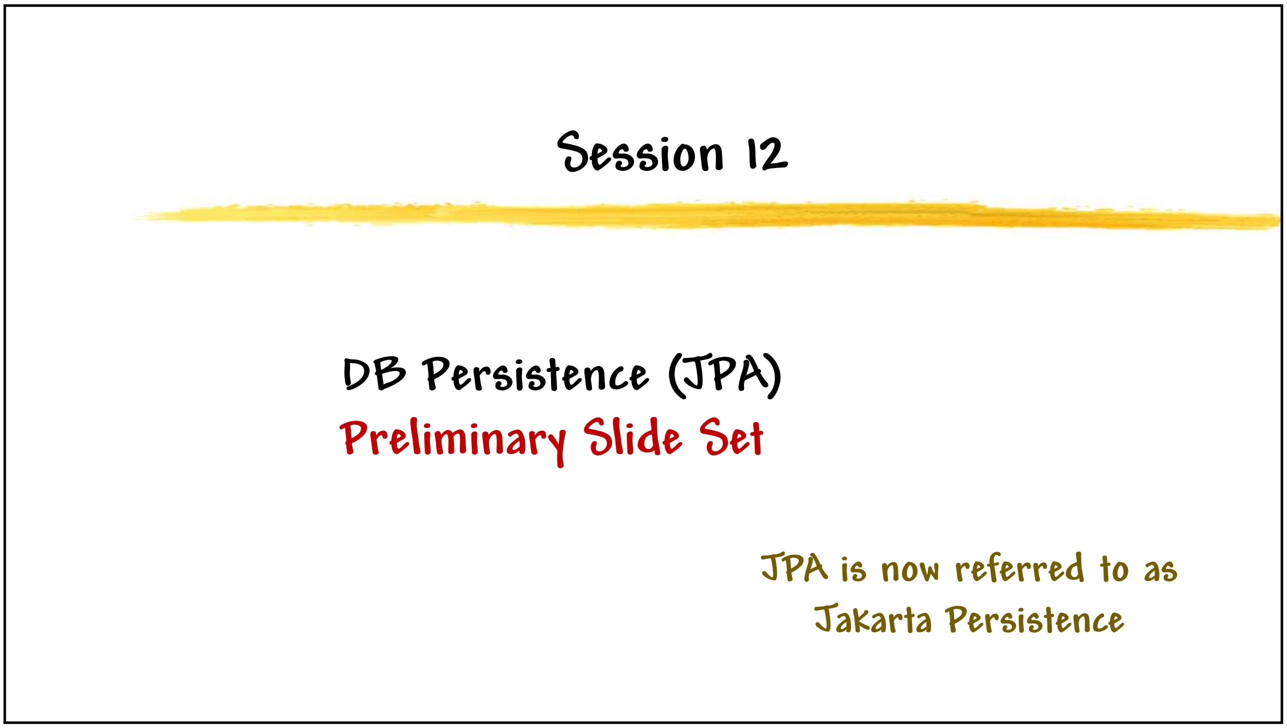


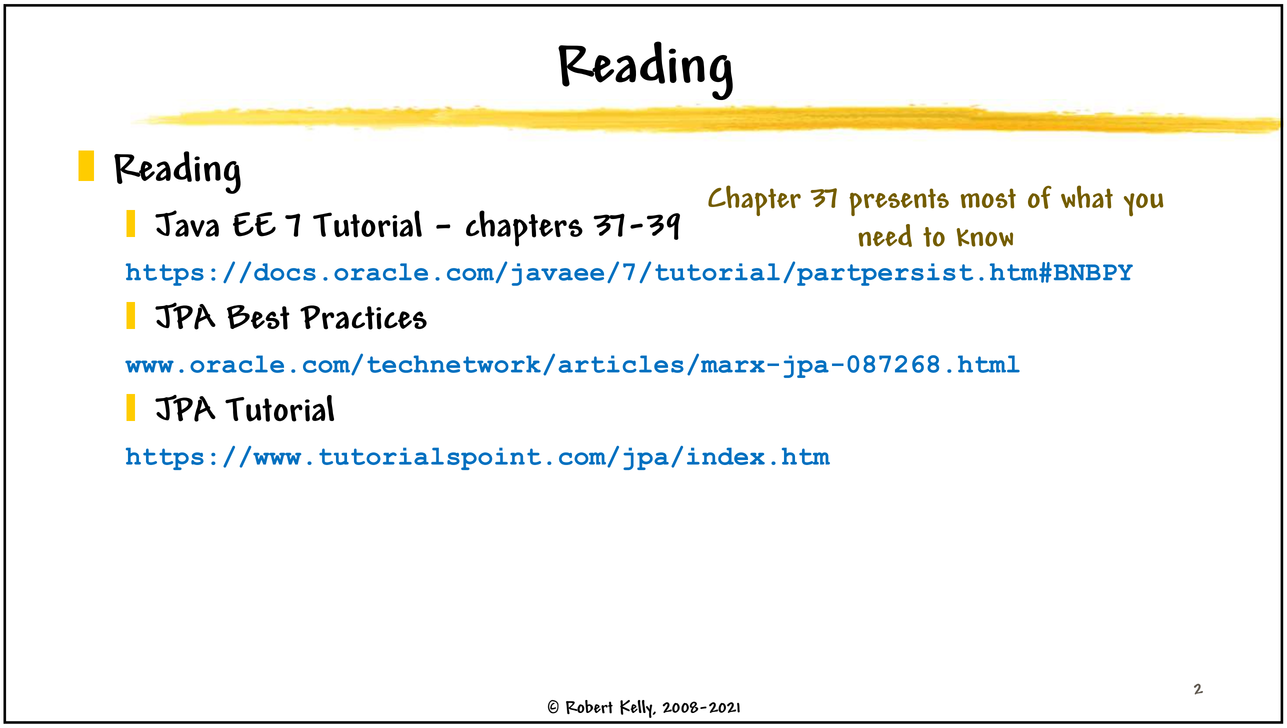
Session 12

DB Persistence (JPA)

Preliminary Slide Set

JPA is now referred to as
Jakarta Persistence

1



Reading

- Reading
  - Java EE 7 Tutorial - chapters 37-39

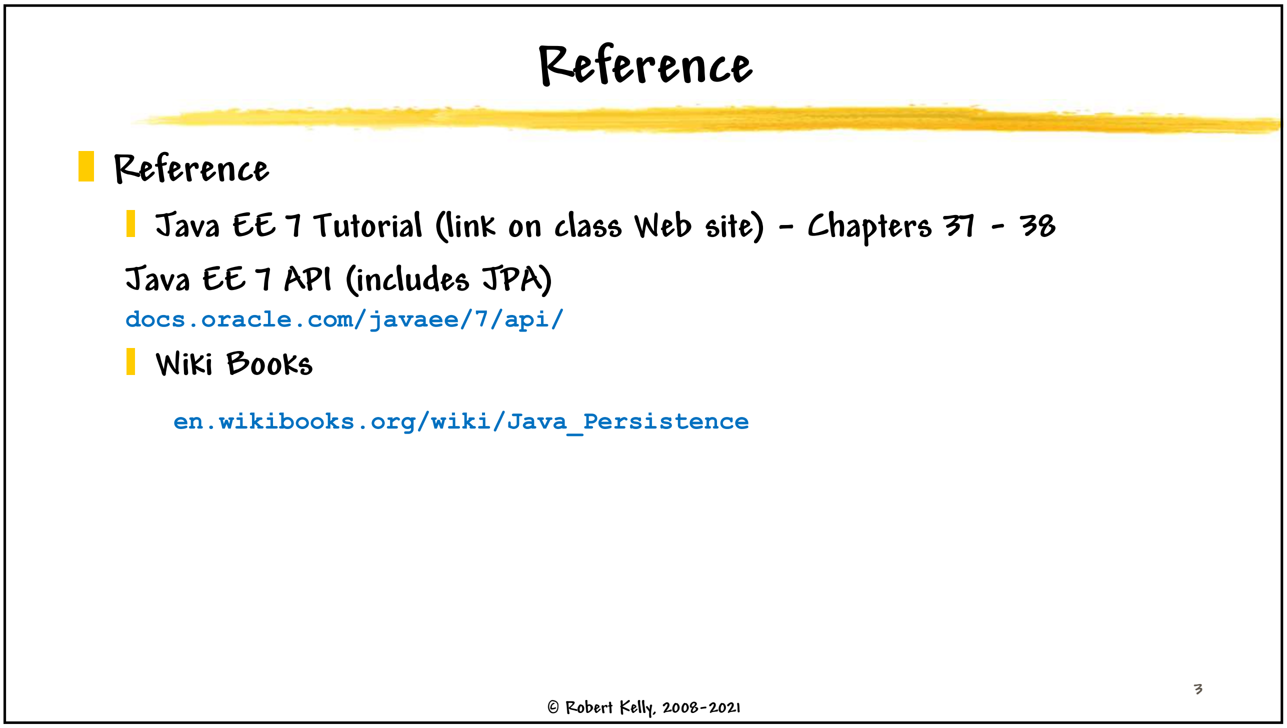
    Chapter 37 presents most of what you need to know

    https://docs.oracle.com/javaee/7/tutorial/partpersist.htm#BNBPY
  - JPA Best Practices

    www.oracle.com/technetwork/articles/marx-jpa-087268.html
  - JPA Tutorial

    https://www.tutorialspoint.com/jpa/index.htm

2

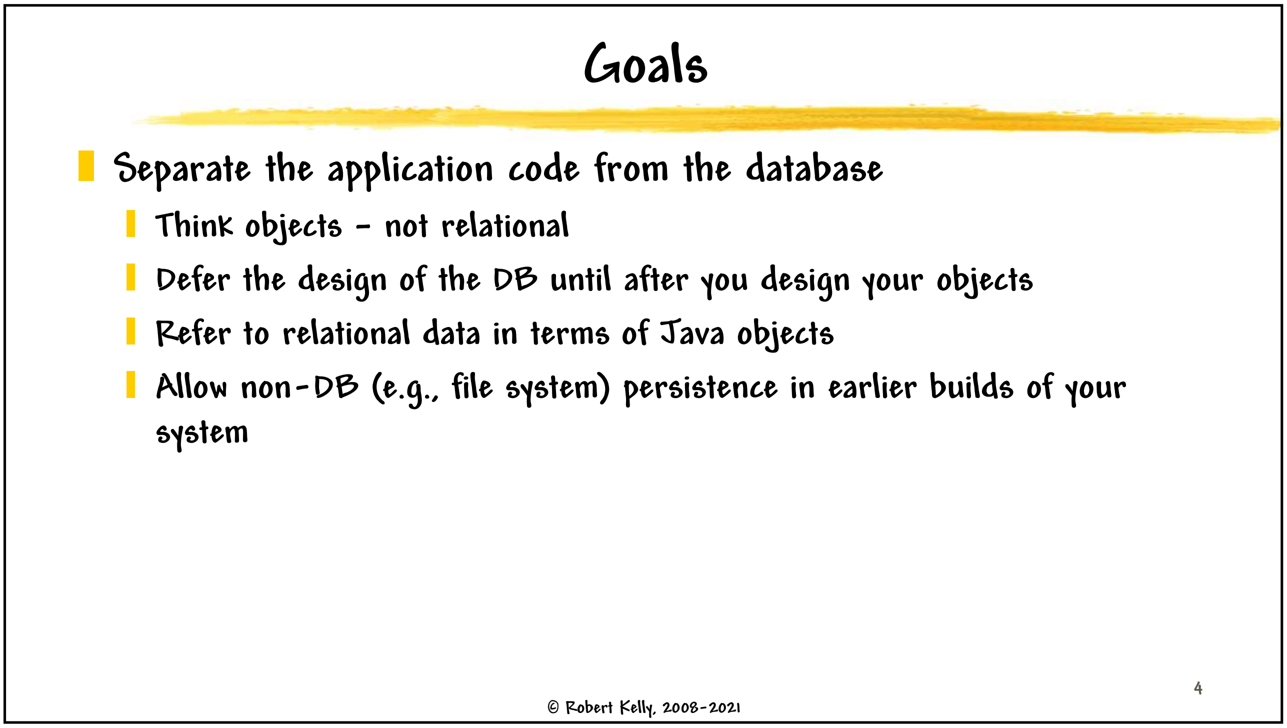


Reference

- Reference
  - Java EE 7 Tutorial (link on class Web site) – Chapters 37 - 38

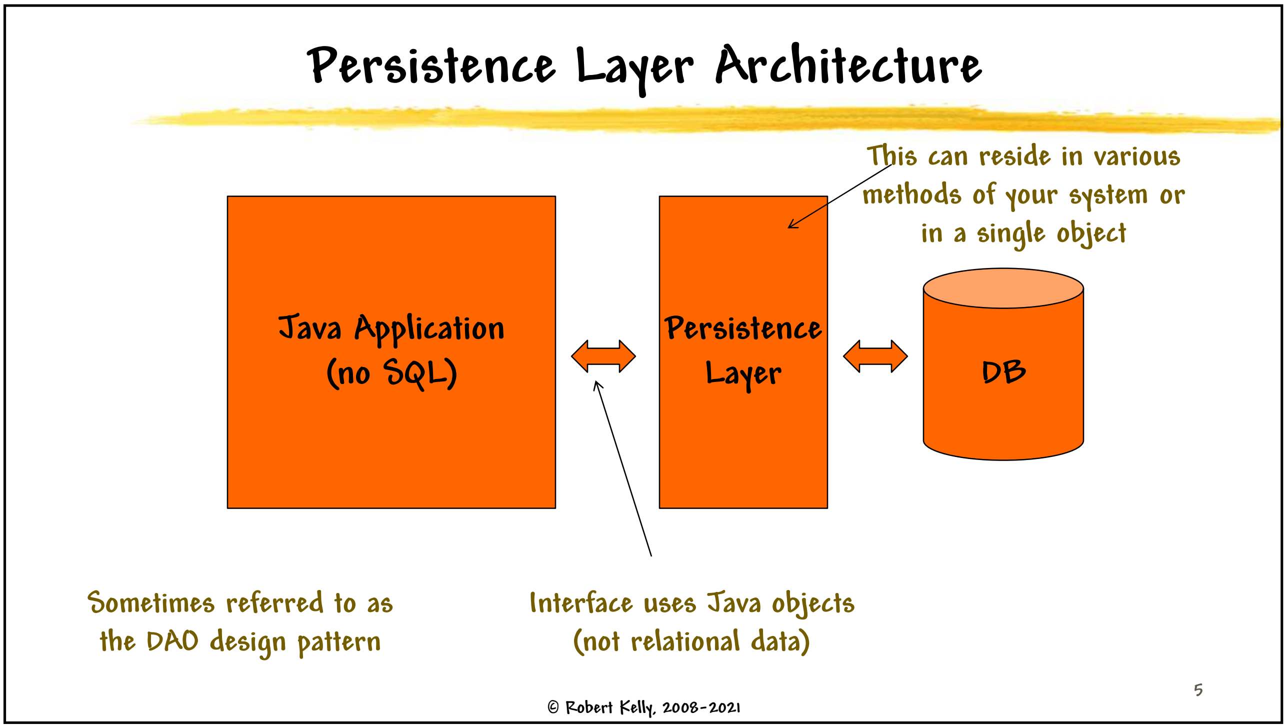
  Java EE 7 API (includes JPA)
  docs.oracle.com/javaee/7/api/
  - Wiki Books

    en.wikibooks.org/wiki/Java_Persistence



Goals

- Separate the application code from the database
  - Think objects – not relational
  - Defer the design of the DB until after you design your objects
  - Refer to relational data in terms of Java objects
  - Allow non-DB (e.g., file system) persistence in earlier builds of your system



Persistence Layer Architecture

Java Application (no SQL)

Persistence Layer

DB

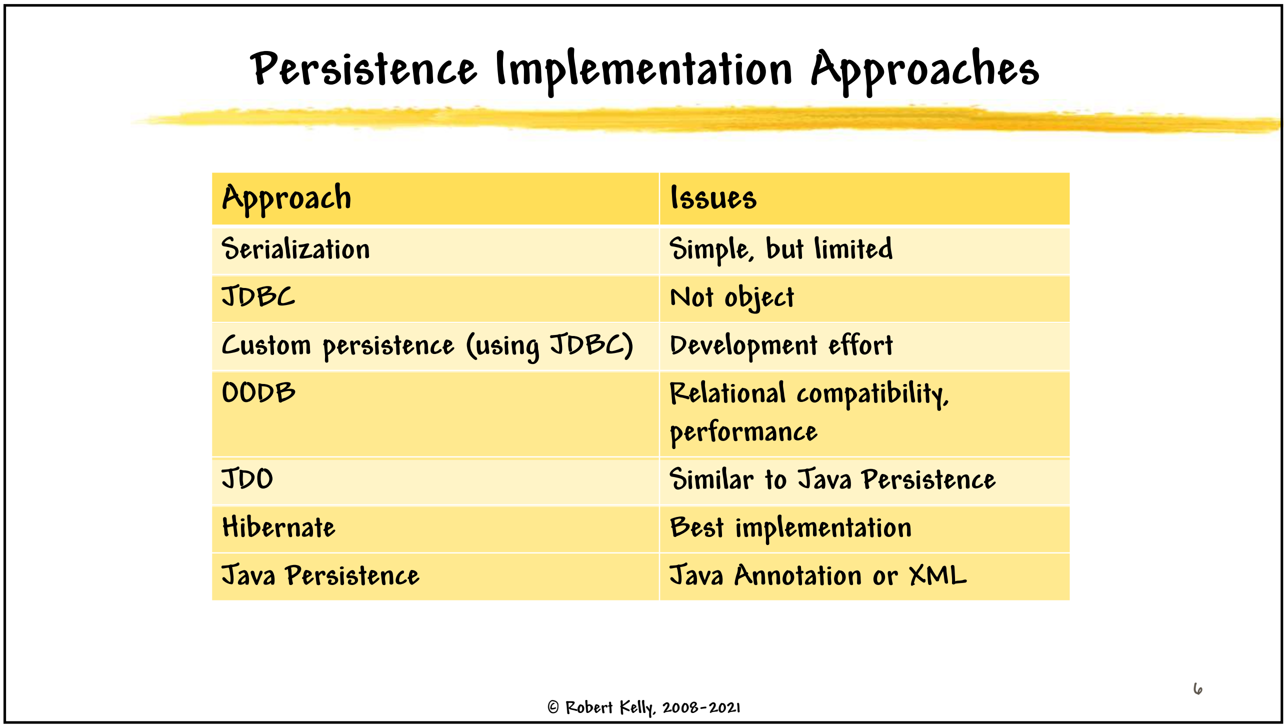
This can reside in various methods of your system or in a single object

Sometimes referred to as the DAO design pattern

Interface uses Java objects (not relational data)

© Robert Kelly, 2008-2021

5

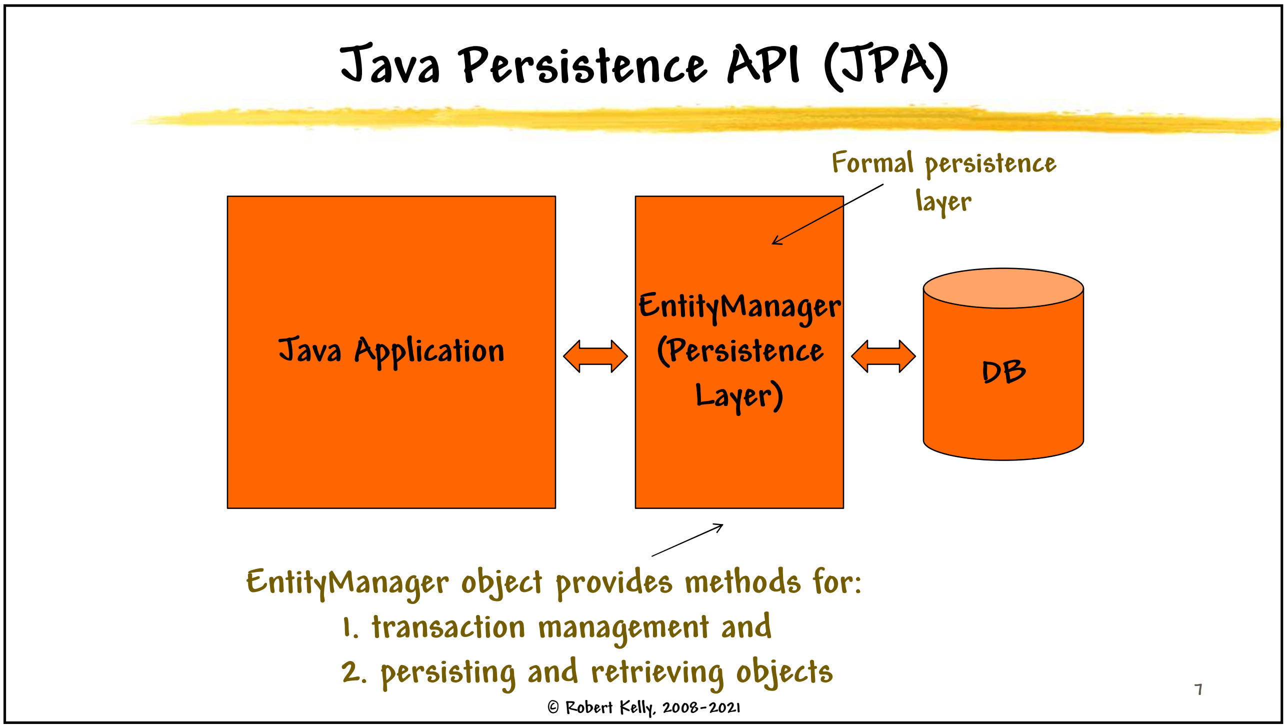


Persistence Implementation Approaches

Approach	Issues
Serialization	Simple, but limited
JDBC	Not object
Custom persistence (using JDBC)	Development effort
OODB	Relational compatibility, performance
JDO	Similar to Java Persistence
Hibernate	Best implementation
Java Persistence	Java Annotation or XML

© Robert Kelly, 2008-2021

6



Java Persistence API (JPA)

Formal persistence layer

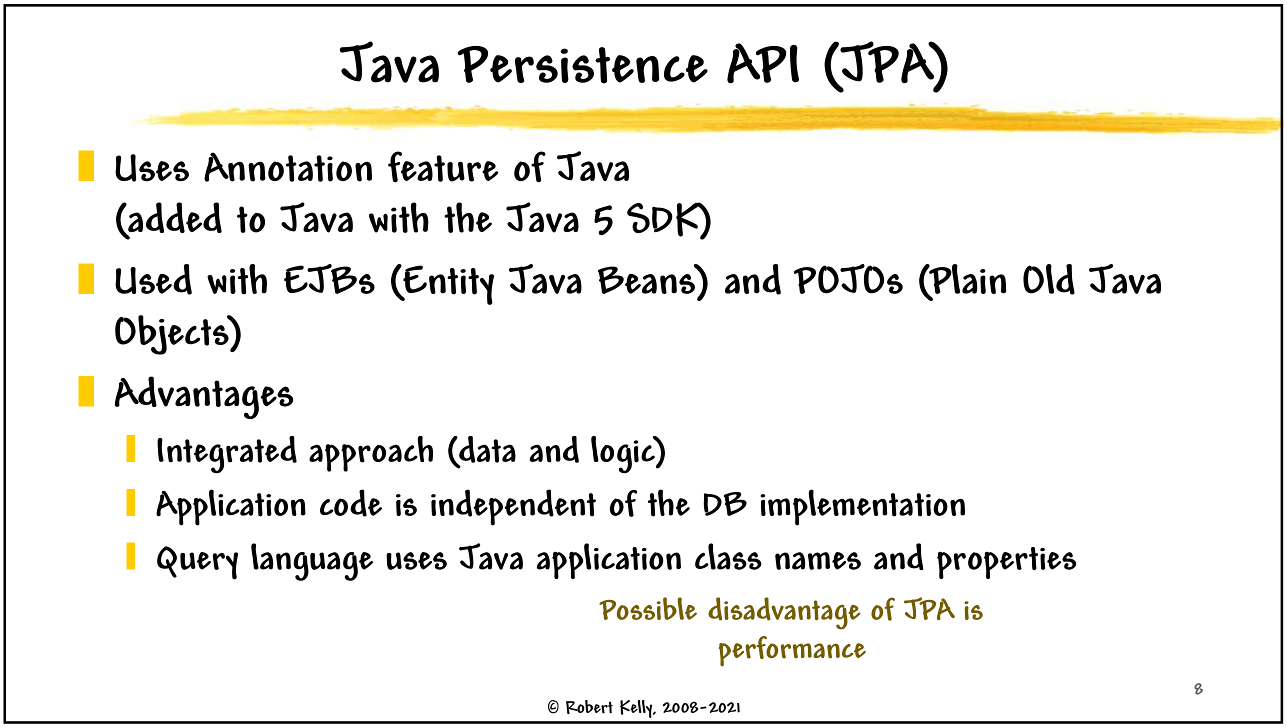
Java Application

EntityManager (Persistence Layer)

DB

EntityManager object provides methods for:

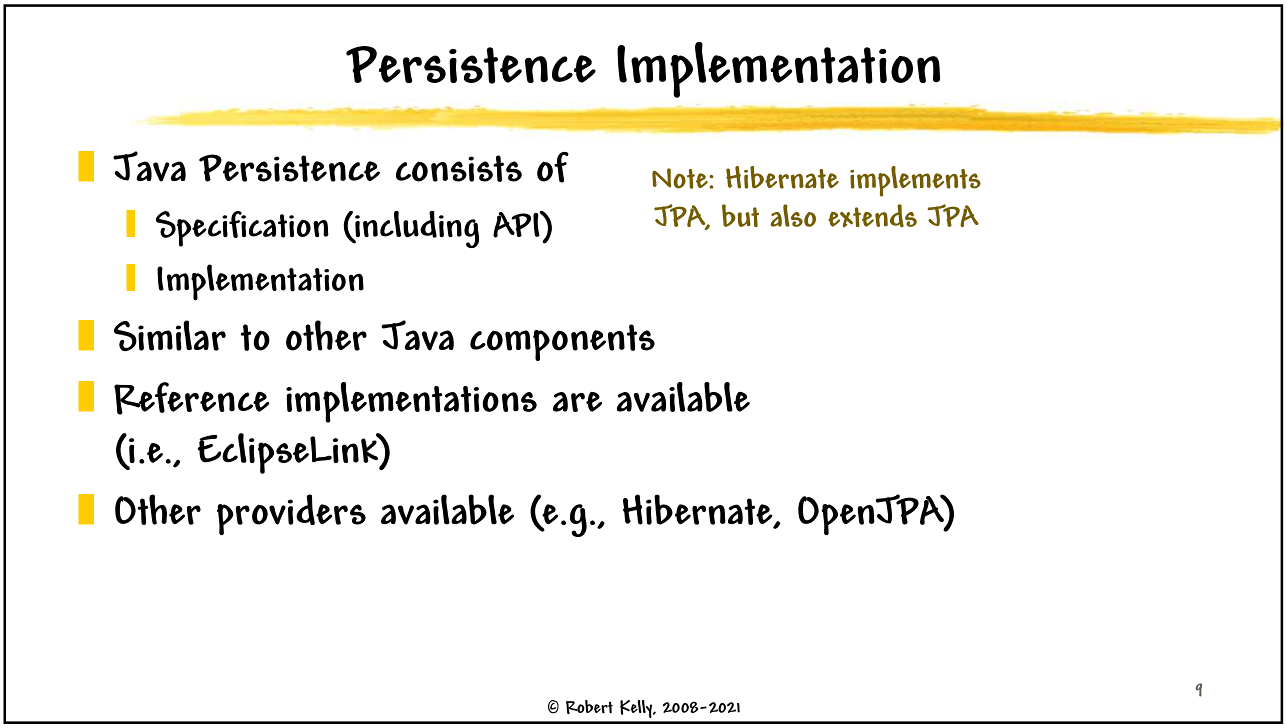
1. transaction management and
2. persisting and retrieving objects



Java Persistence API (JPA)

- Uses Annotation feature of Java (added to Java with the Java 5 SDK)
- Used with EJBs (Entity Java Beans) and POJOs (Plain Old Java Objects)
- Advantages
  - Integrated approach (data and logic)
  - Application code is independent of the DB implementation
  - Query language uses Java application class names and properties

Possible disadvantage of JPA is performance



Persistence Implementation

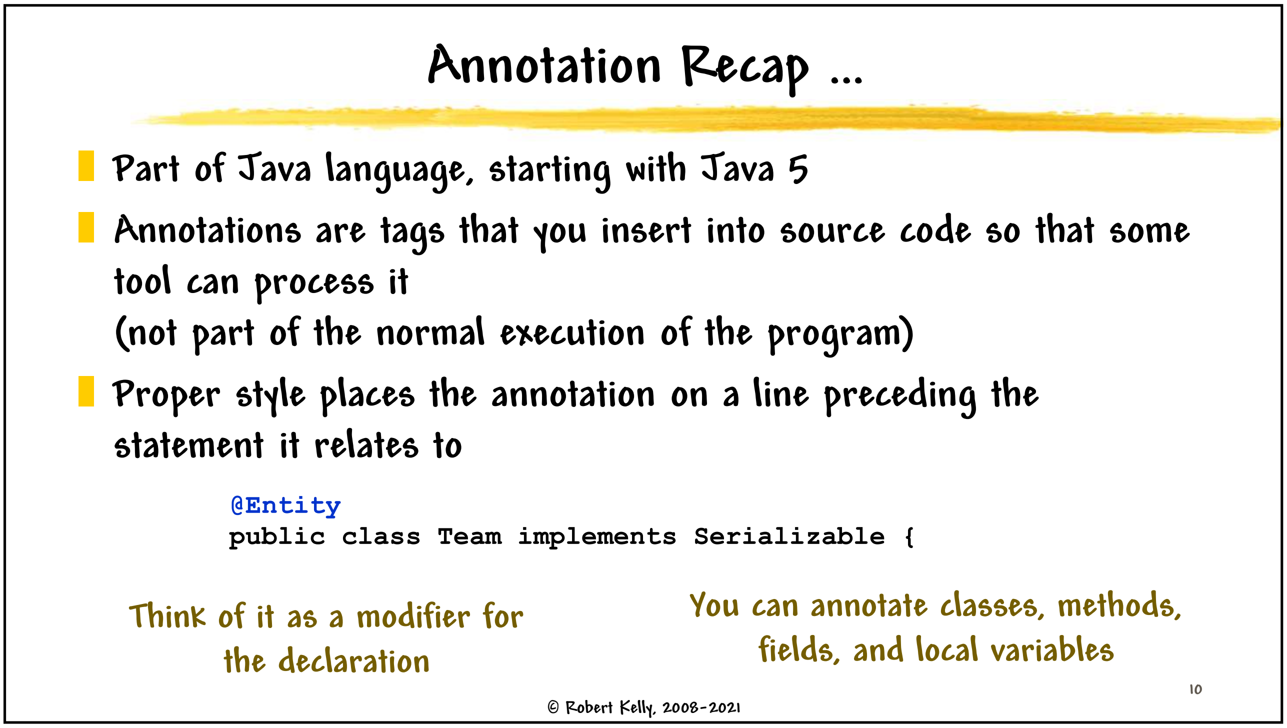
- Java Persistence consists of
  - Specification (including API)
  - Implementation
- Similar to other Java components
- Reference implementations are available (i.e., EclipseLink)
- Other providers available (e.g., Hibernate, OpenJPA)

Note: Hibernate implements JPA, but also extends JPA

© Robert Kelly, 2008-2021

9

9



Annotation Recap …

- Part of Java language, starting with Java 5
- Annotations are tags that you insert into source code so that some tool can process it
  (not part of the normal execution of the program)
- Proper style places the annotation on a line preceding the statement it relates to

```
@Entity
public class Team implements Serializable {
```

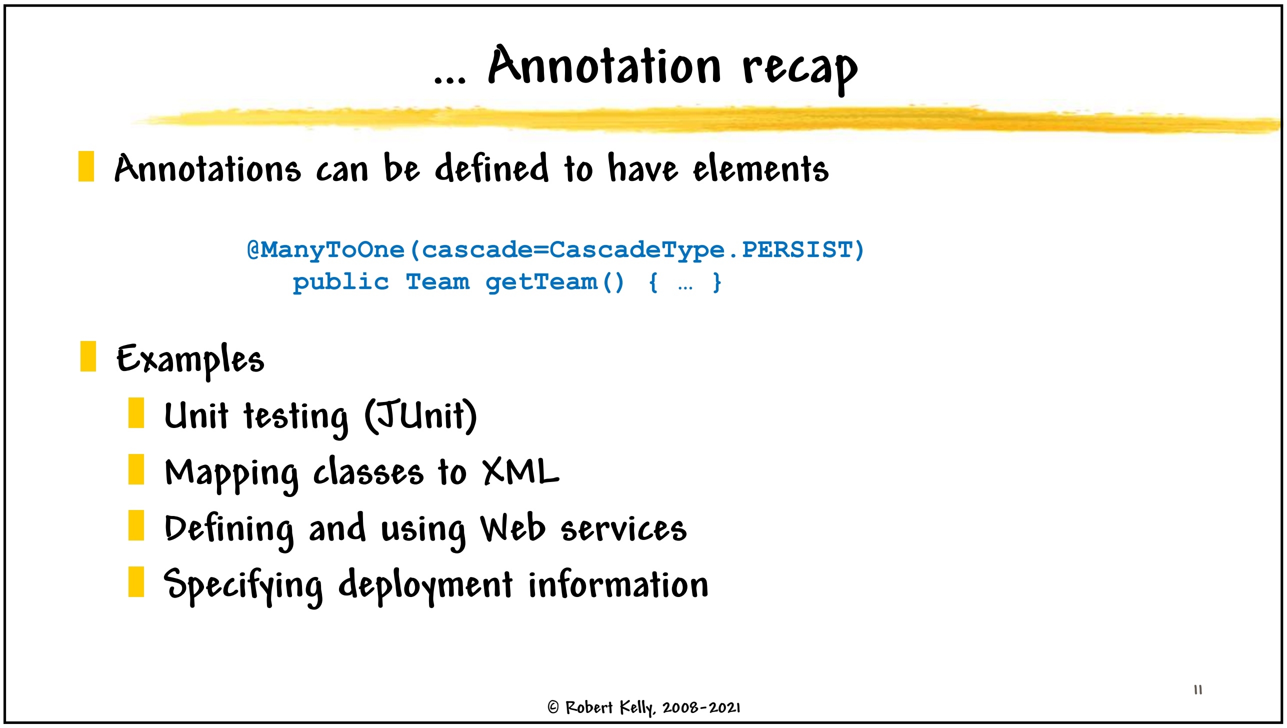
Think of it as a modifier for the declaration

You can annotate classes, methods, fields, and local variables

© Robert Kelly, 2008-2021

10

10

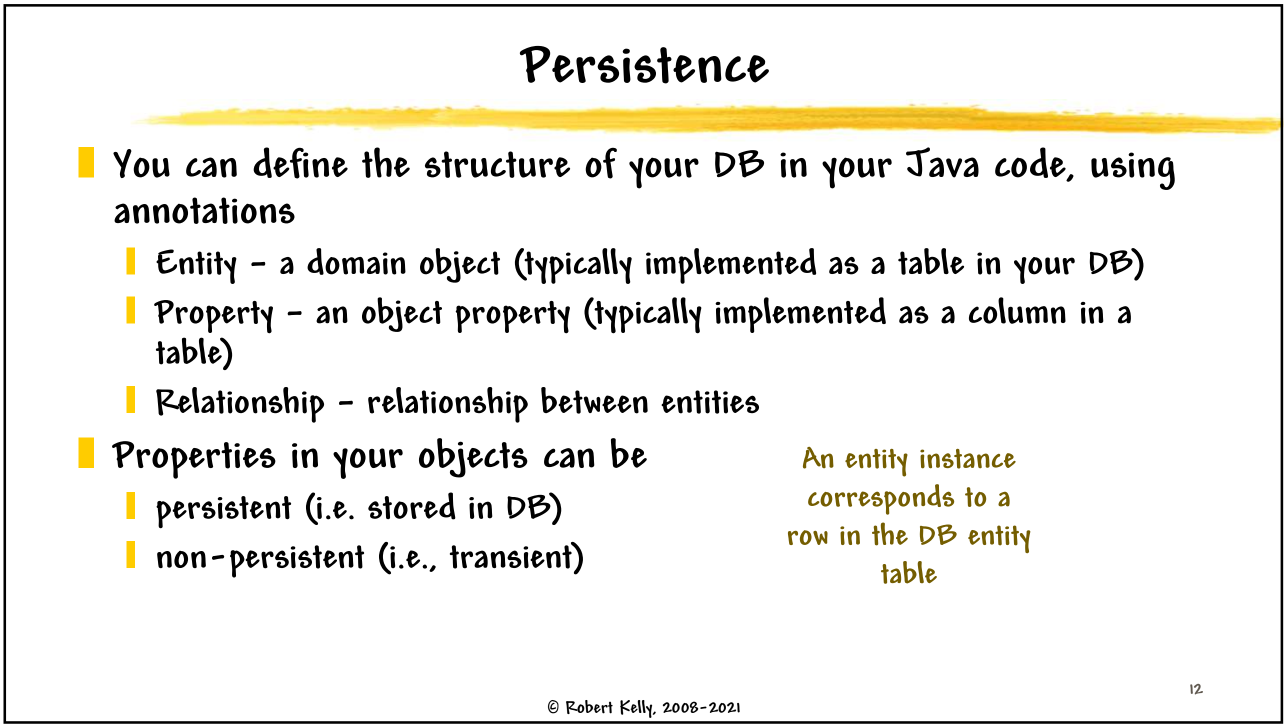


… Annotation recap

- Annotations can be defined to have elements

```
@ManyToOne(cascade=CascadeType.PERSIST)
   public Team getTeam() { … }
```

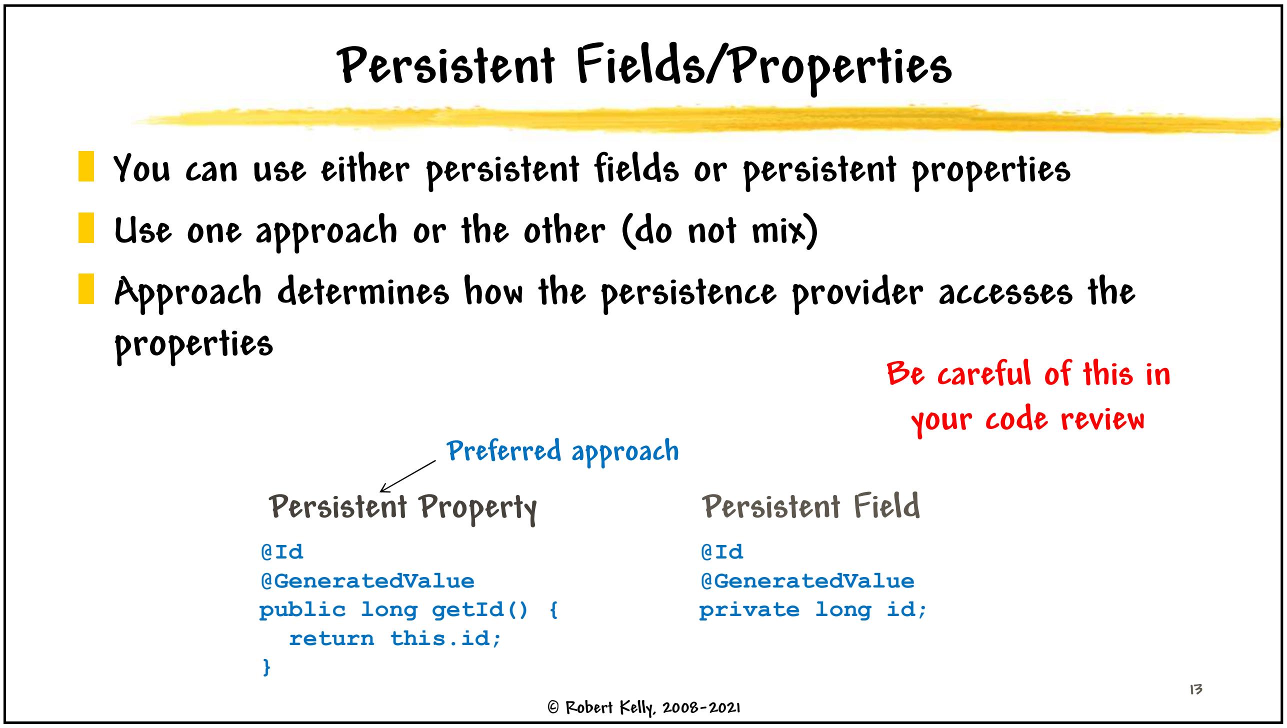
- Examples
  - Unit testing (JUnit)
  - Mapping classes to XML
  - Defining and using Web services
  - Specifying deployment information



Persistence

- You can define the structure of your DB in your Java code, using annotations
  - Entity – a domain object (typically implemented as a table in your DB)
  - Property – an object property (typically implemented as a column in a table)
  - Relationship – relationship between entities
- Properties in your objects can be
  - persistent (i.e. stored in DB)
  - non-persistent (i.e., transient)

An entity instance corresponds to a row in the DB entity table



Persistent Fields/Properties

- You can use either persistent fields or persistent properties
- Use one approach or the other (do not mix)
- Approach determines how the persistence provider accesses the properties

Be careful of this in your code review

Preferred approach

Persistent Property

```
@Id
@GeneratedValue
public long getId() {
  return this.id;
}
```

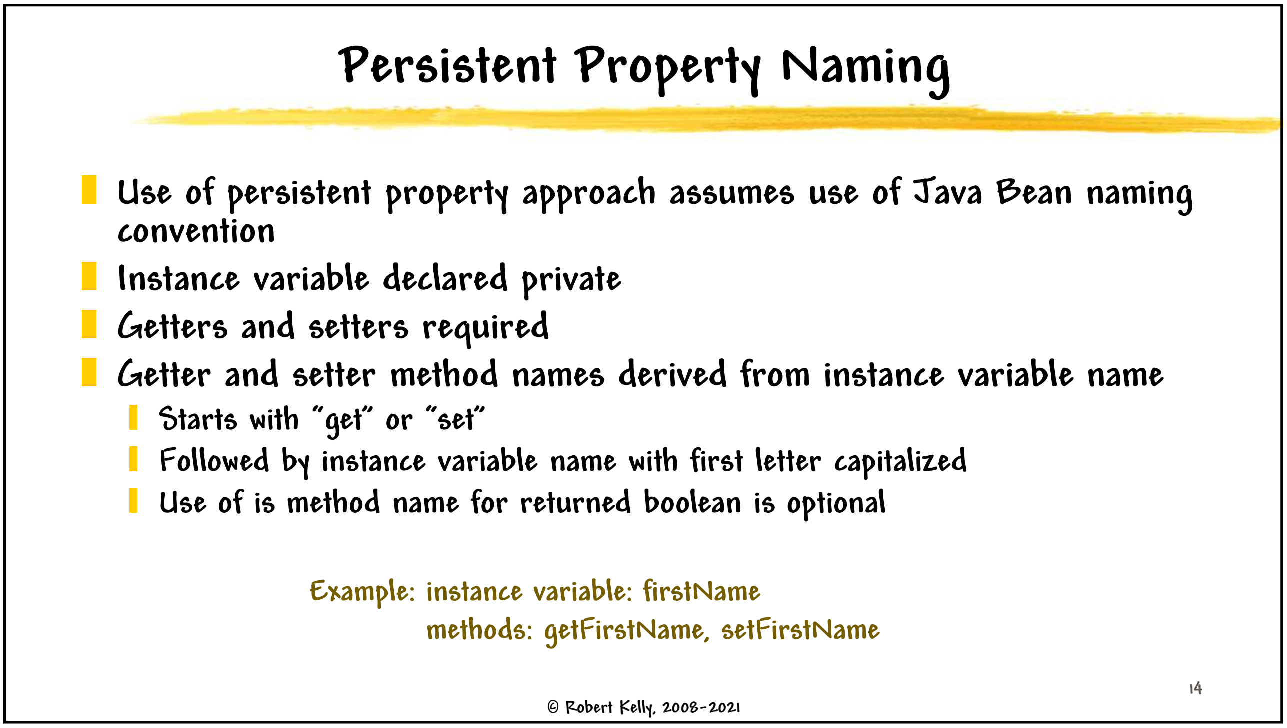
Persistent Field

```
@Id
@GeneratedValue
private long id;
```

© Robert Kelly, 2008-2021

13

13



Persistent Property Naming

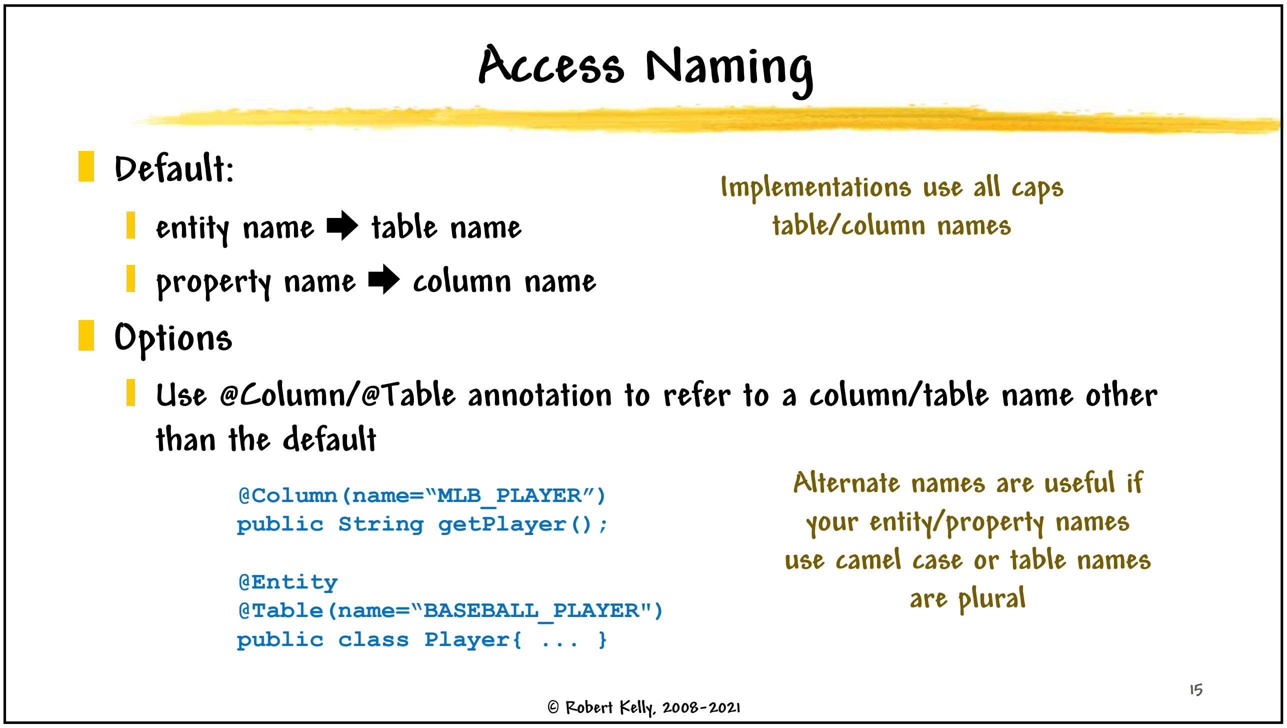
- Use of persistent property approach assumes use of Java Bean naming convention
- Instance variable declared private
- Getters and setters required
- Getter and setter method names derived from instance variable name
  - Starts with "get" or "set"
  - Followed by instance variable name with first letter capitalized
  - Use of is method name for returned boolean is optional

Example: instance variable: firstName
methods: getFirstName, setFirstName

© Robert Kelly, 2008-2021

14

14



Access Naming

- Default:
  - entity name ➡ table name
  - property name ➡ column name
- Options
  - Use @Column/@Table annotation to refer to a column/table name other than the default

Implementations use all caps table/column names

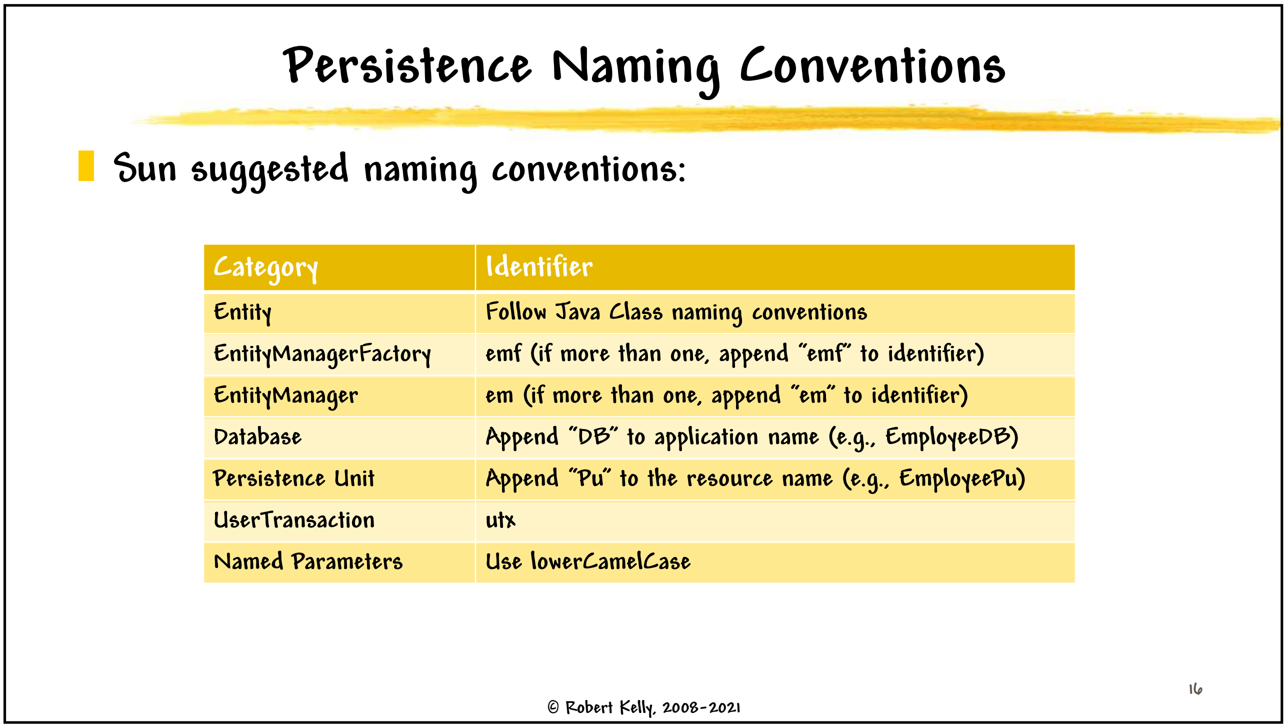
```
@Column(name="MLB_PLAYER")
public String getPlayer();

@Entity
@Table(name="BASEBALL_PLAYER")
public class Player{ ... }
```

Alternate names are useful if your entity/property names use camel case or table names are plural

© Robert Kelly, 2008-2021

15

15



Persistence Naming Conventions

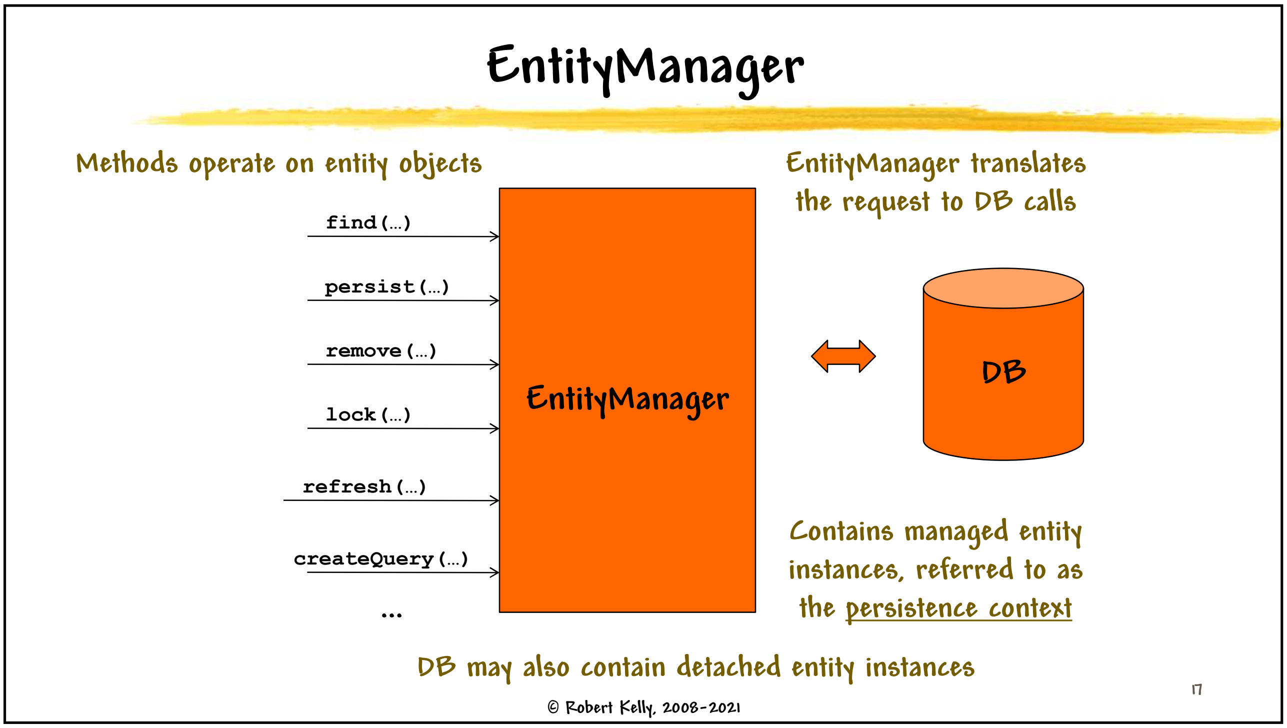
- Sun suggested naming conventions:

Category	Identifier
Entity	Follow Java Class naming conventions
EntityManagerFactory	emf (if more than one, append "emf" to identifier)
EntityManager	em (if more than one, append "em" to identifier)
Database	Append "DB" to application name (e.g., EmployeeDB)
Persistence Unit	Append "Pu" to the resource name (e.g., EmployeePu)
UserTransaction	utx
Named Parameters	Use lowerCamelCase

© Robert Kelly, 2008-2021

16

16



EntityManager

Methods operate on entity objects

find (...)

persist (...)

remove (...)

lock (...)

refresh (...)

createQuery (...)

...

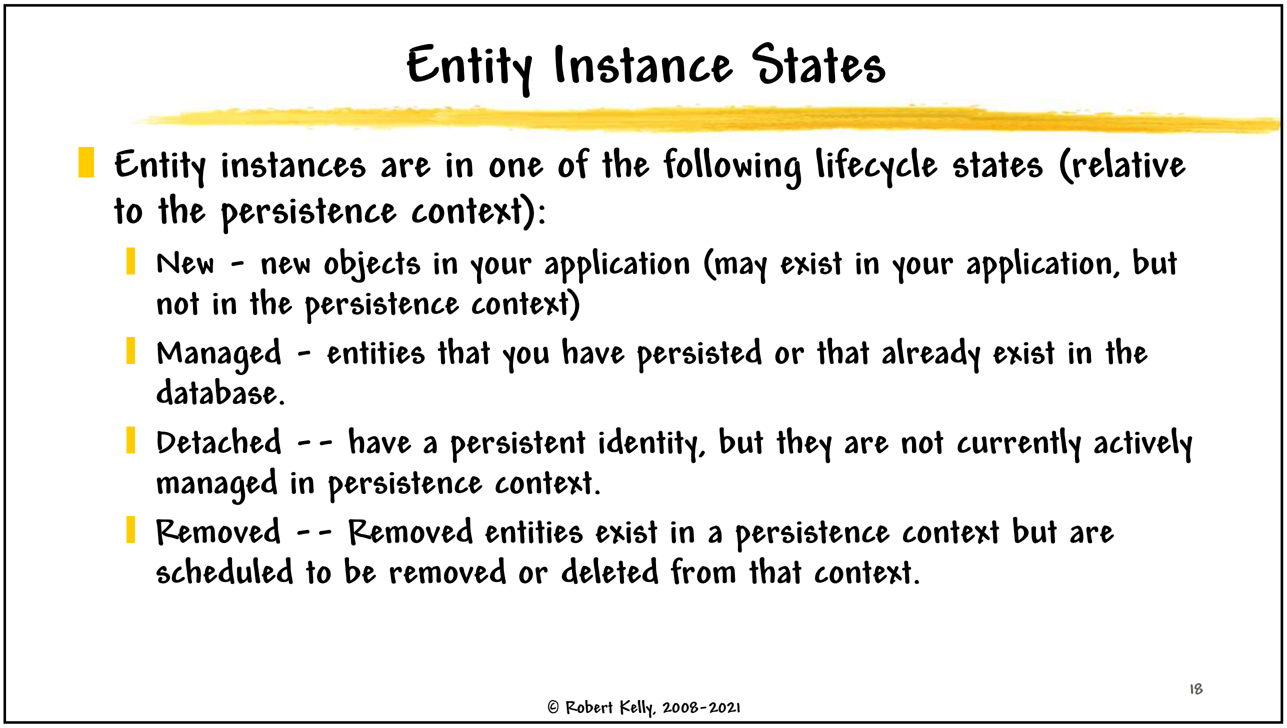
EntityManager

EntityManager translates the request to DB calls

DB

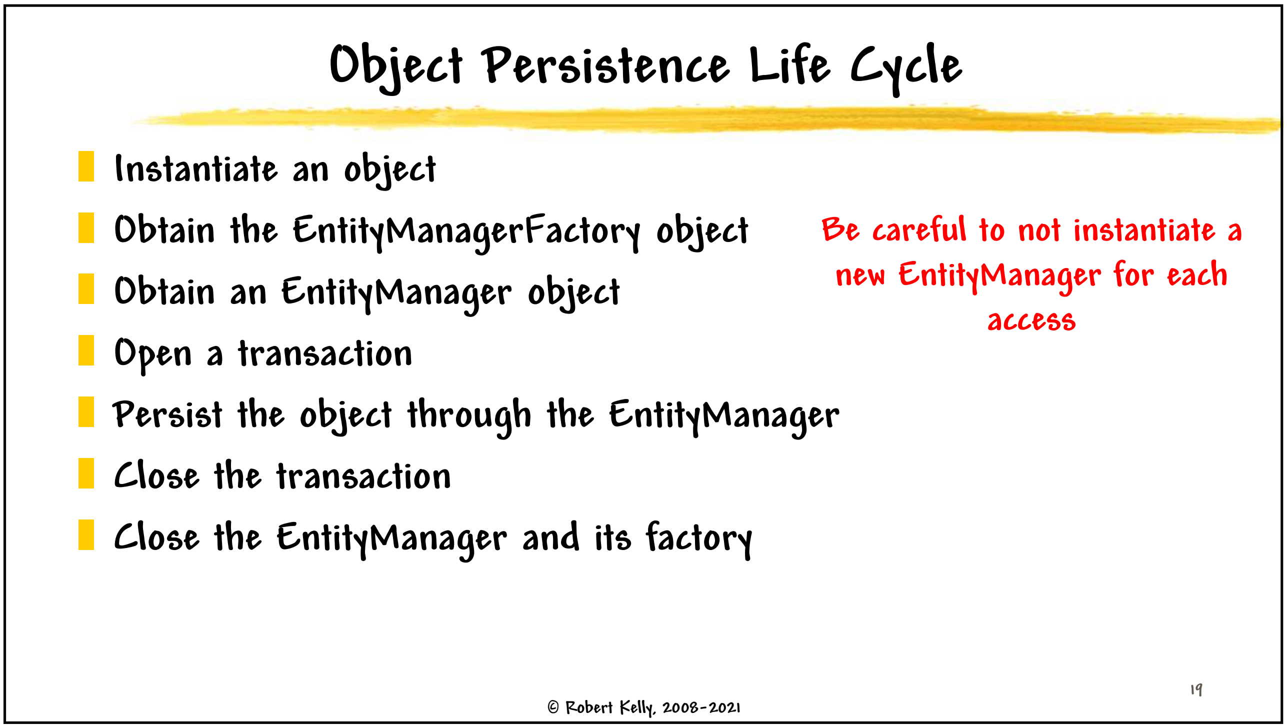
Contains managed entity instances, referred to as the persistence context

DB may also contain detached entity instances



Entity Instance States

- Entity instances are in one of the following lifecycle states (relative to the persistence context):
  - New - new objects in your application (may exist in your application, but not in the persistence context)
  - Managed - entities that you have persisted or that already exist in the database.
  - Detached -- have a persistent identity, but they are not currently actively managed in persistence context.
  - Removed -- Removed entities exist in a persistence context but are scheduled to be removed or deleted from that context.



Object Persistence Life Cycle

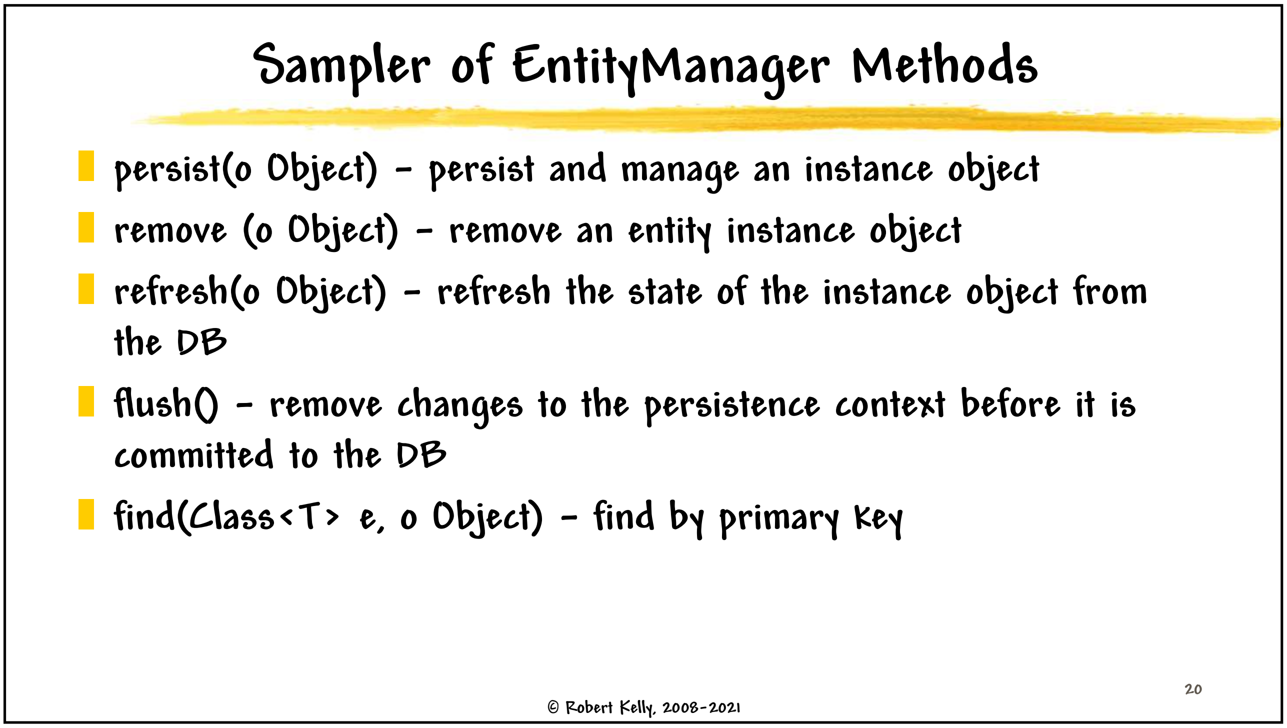
- Instantiate an object
- Obtain the EntityManagerFactory object
- Obtain an EntityManager object
- Open a transaction
- Persist the object through the EntityManager
- Close the transaction
- Close the EntityManager and its factory

Be careful to not instantiate a new EntityManager for each access

© Robert Kelly, 2008-2021

19

19



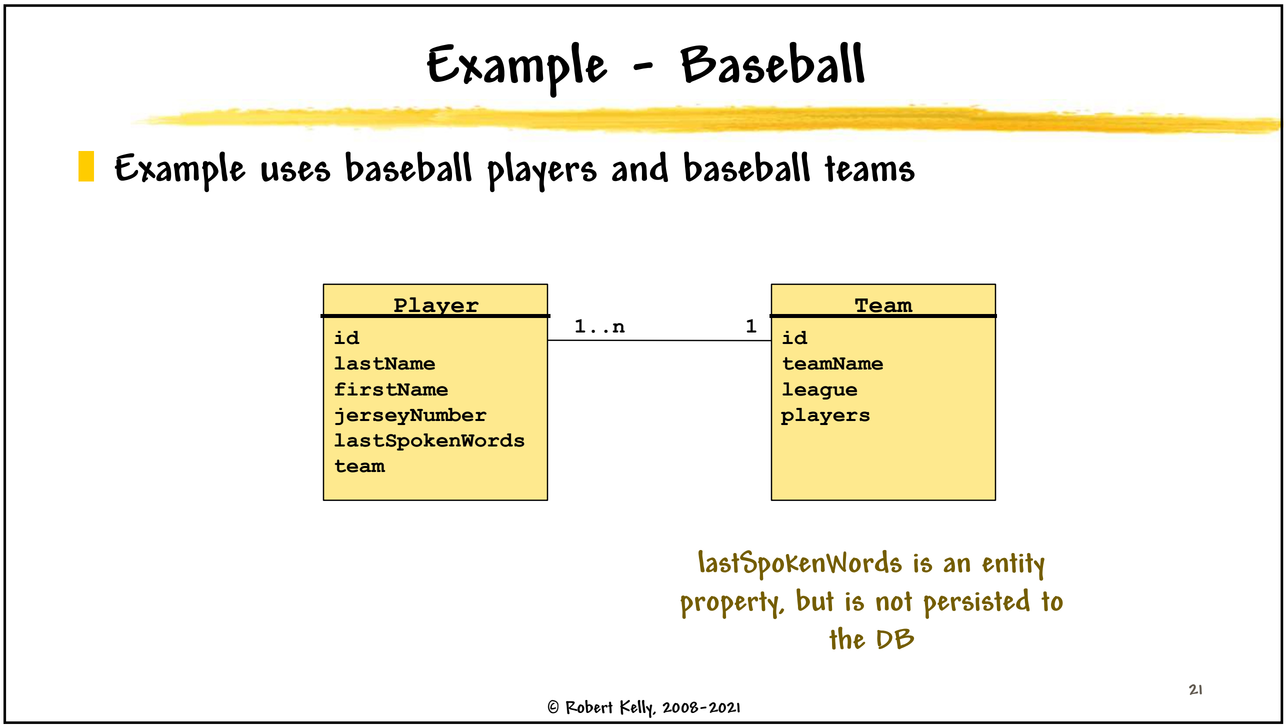
Sampler of EntityManager Methods

- persist(o Object) – persist and manage an instance object
- remove (o Object) – remove an entity instance object
- refresh(o Object) – refresh the state of the instance object from the DB
- flush() – remove changes to the persistence context before it is committed to the DB
- find(Class<T> e, o Object) – find by primary key

© Robert Kelly, 2008-2021

20

20



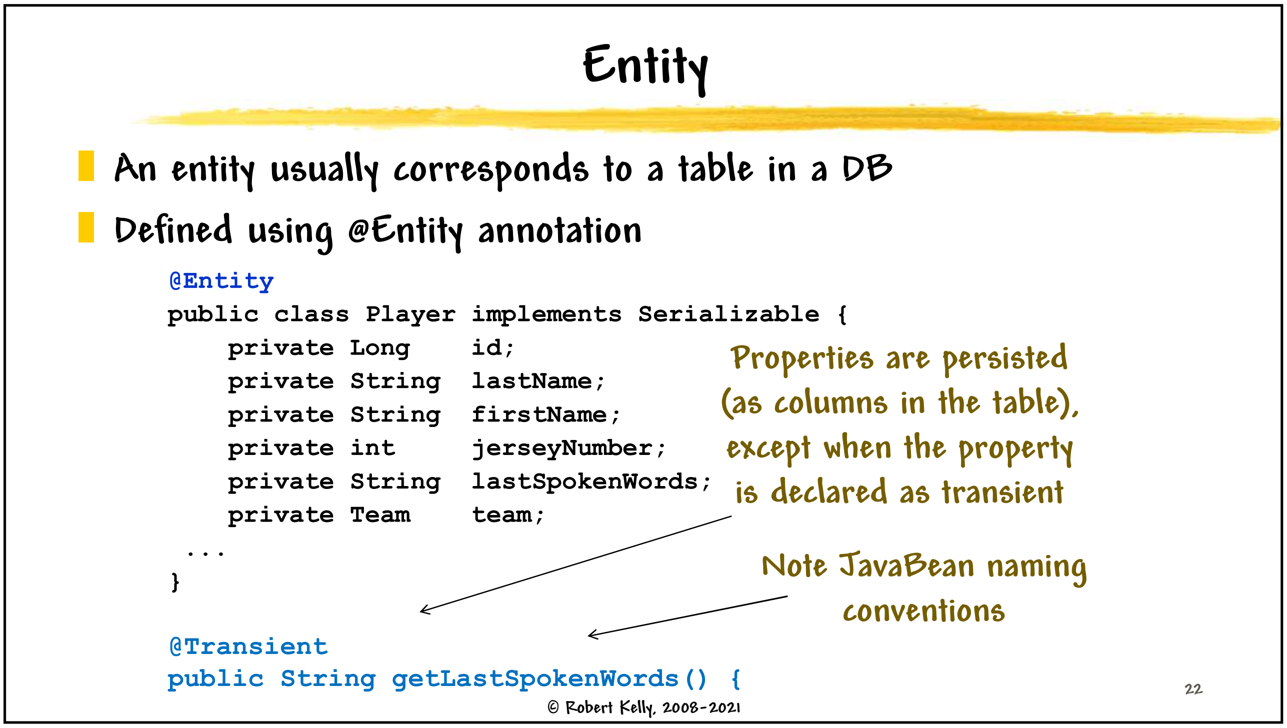
Example - Baseball

- Example uses baseball players and baseball teams

Player	Team
id lastName firstName jerseyNumber lastSpokenWords team	id teamName league players

lastSpokenWords is an entity property, but is not persisted to the DB

21



Entity

- An entity usually corresponds to a table in a DB
- Defined using @Entity annotation

```
@Entity
public class Player implements Serializable {
     private Long    id;
     private String  lastName;
     private String  firstName;
     private int     jerseyNumber;
     private String  lastSpokenWords;
     private Team    team;
 ...
}

@Transient
public String getLastSpokenWords() {
```

Properties are persisted (as columns in the table), except when the property is declared as transient

Note JavaBean naming conventions

22

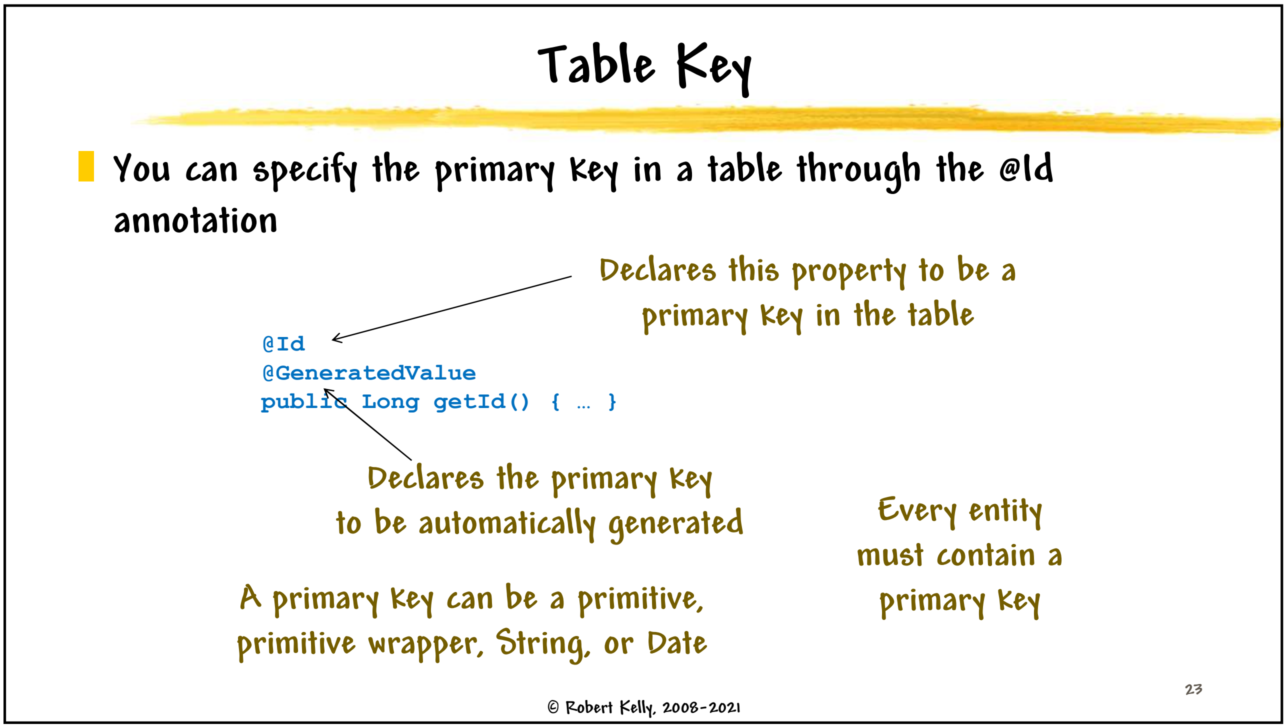


Table Key

- You can specify the primary key in a table through the @Id annotation

```
@Id
@GeneratedValue
public Long getId() { … }
```

Declares this property to be a primary key in the table

Declares the primary key to be automatically generated

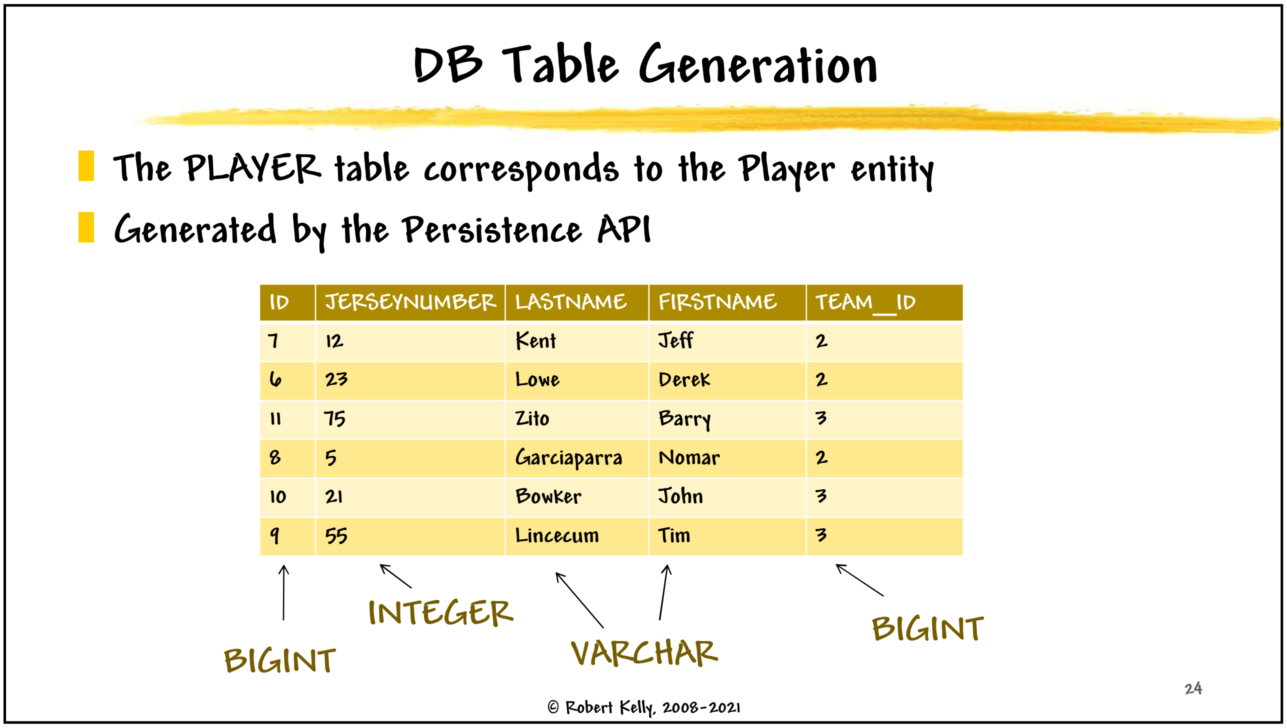
Every entity must contain a primary key

A primary key can be a primitive, primitive wrapper, String, or Date

© Robert Kelly, 2008-2021

23

23



DB Table Generation

- The PLAYER table corresponds to the Player entity
- Generated by the Persistence API

ID	JERSEYNUMBER	LASTNAME	FIRSTNAME	TEAM_ID
7	12	Kent	Jeff	2
6	23	Lowe	Derek	2
11	75	Zito	Barry	3
8	5	Garciaparra	Nomar	2
10	21	Bowker	John	3
9	55	Lincecum	Tim	3

BIGINT

INTEGER

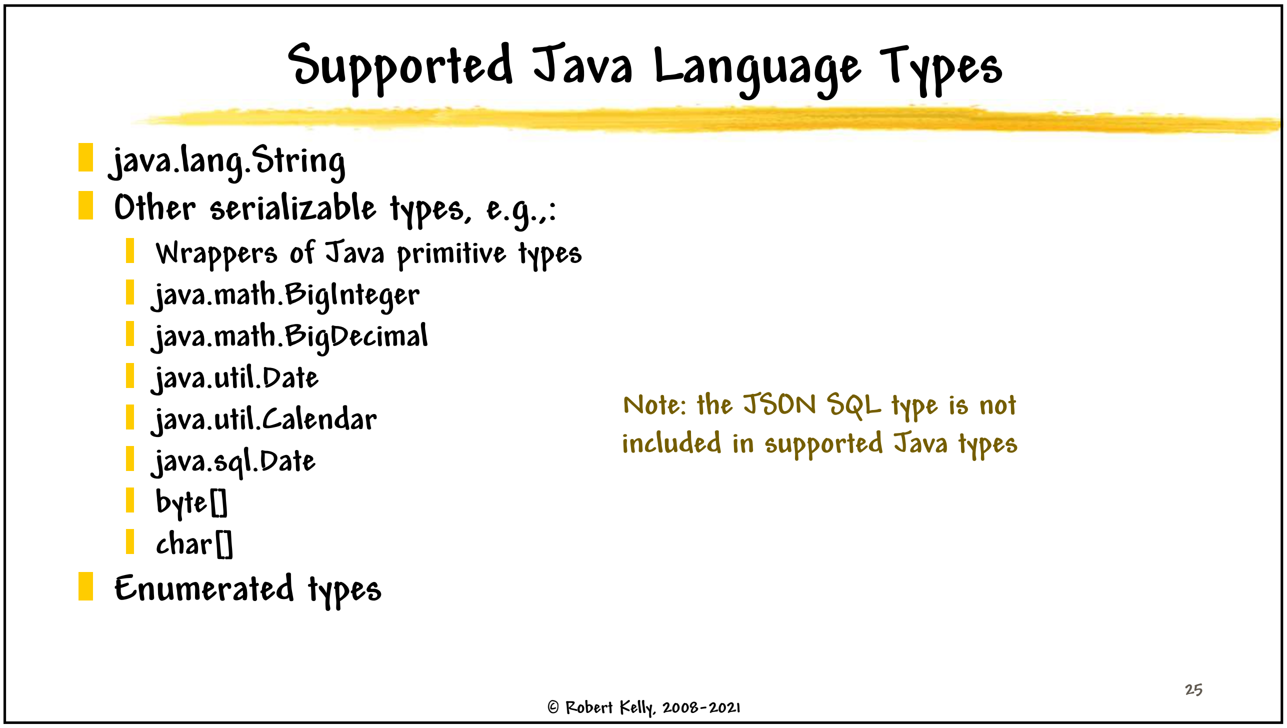
VARCHAR

BIGINT

© Robert Kelly, 2008-2021

24

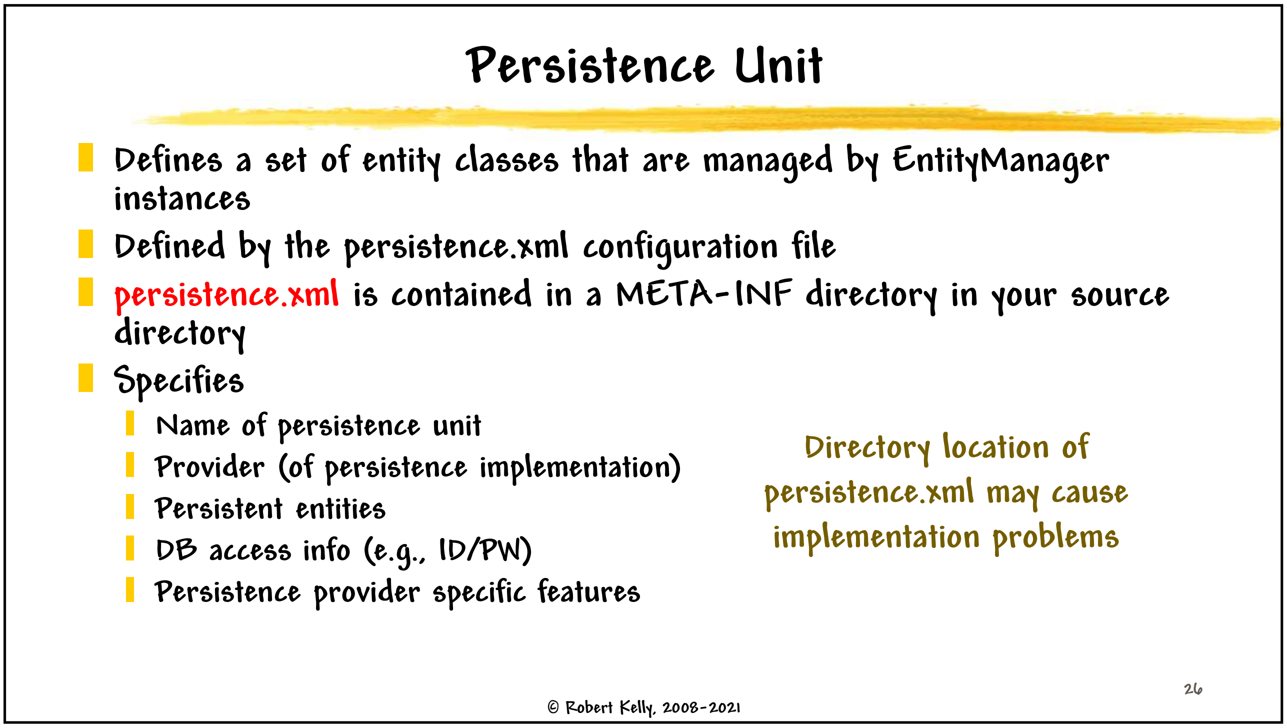
24



Supported Java Language Types

- java.lang.String
- Other serializable types, e.g.,:
  - Wrappers of Java primitive types
  - java.math.BigInteger
  - java.math.BigDecimal
  - java.util.Date
  - java.util.Calendar
  - java.sql.Date
  - byte[]
  - char[]
- Enumerated types

Note: the JSON SQL type is not included in supported Java types



Persistence Unit

- Defines a set of entity classes that are managed by EntityManager instances
- Defined by the persistence.xml configuration file
- persistence.xml is contained in a META-INF directory in your source directory
- Specifies
  - Name of persistence unit
  - Provider (of persistence implementation)
  - Persistent entities
  - DB access info (e.g., ID/PW)
  - Persistence provider specific features

Directory location of persistence.xml may cause implementation problems

Persistence Provider

- Many choices for persistence provider (each will provide a set of jar files that implement the JPA)
  - EclipseLink- simple set-up
  - Hibernate - market leader

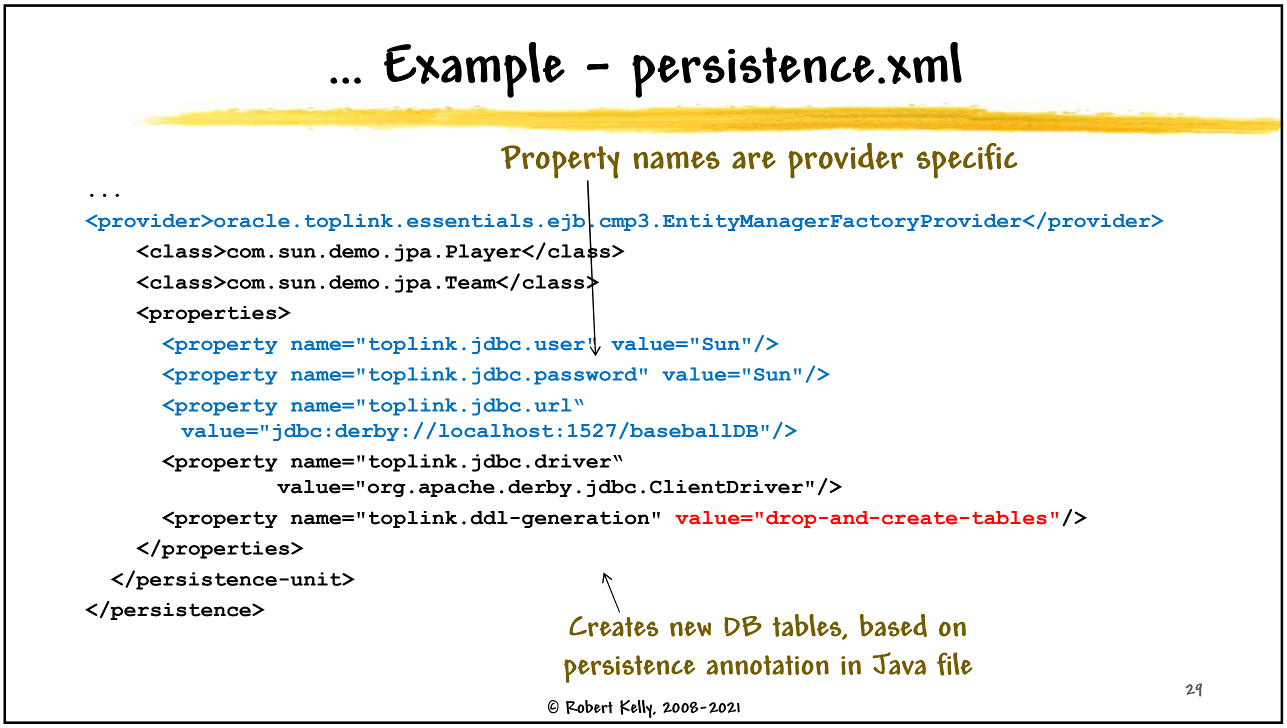
Example - persistence.xml ...

```
EntityManagerFactory emf =
      Persistence.createEntityManagerFactory("leaguePu");
EntityManager em = emf.createEntityManager();
```

The EntityManager object is instantiated with a reference to the persistence-unit name in persistence.xml

A persistence unit is identified by its name

```
<?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="leaguePu"
   transaction-type="RESOURCE_LOCAL">
...
```



… Example – persistence.xml

Property names are provider specific

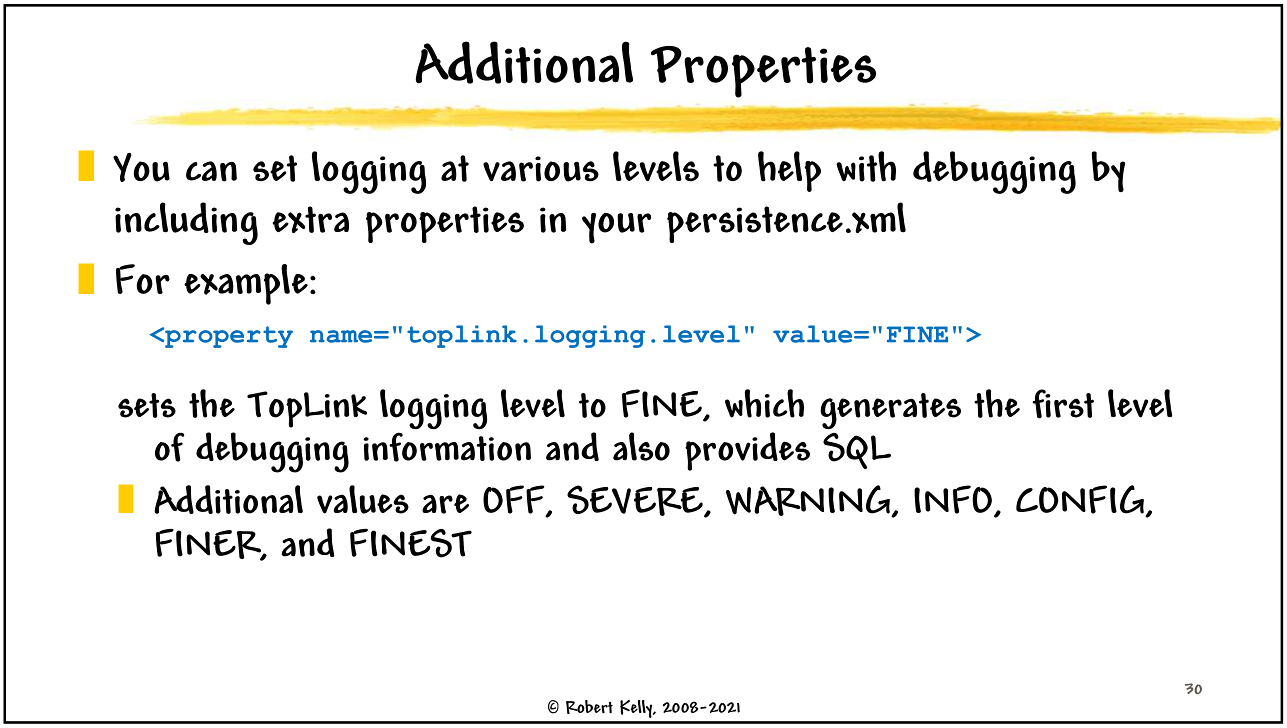
```
...
<provider>oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider</provider>
    <class>com.sun.demo.jpa.Player</class>
    <class>com.sun.demo.jpa.Team</class>
    <properties>
      <property name="toplink.jdbc.user" value="Sun"/>
      <property name="toplink.jdbc.password" value="Sun"/>
      <property name="toplink.jdbc.url"
        value="jdbc:derby://localhost:1527/baseballDB"/>
      <property name="toplink.jdbc.driver"
               value="org.apache.derby.jdbc.ClientDriver"/>
      <property name="toplink.ddl-generation" value="drop-and-create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

Creates new DB tables, based on persistence annotation in Java file

29

© Robert Kelly, 2008-2021

29



Additional Properties

- You can set logging at various levels to help with debugging by including extra properties in your persistence.xml
- For example:

```
<property name="toplink.logging.level" value="FINE">
```

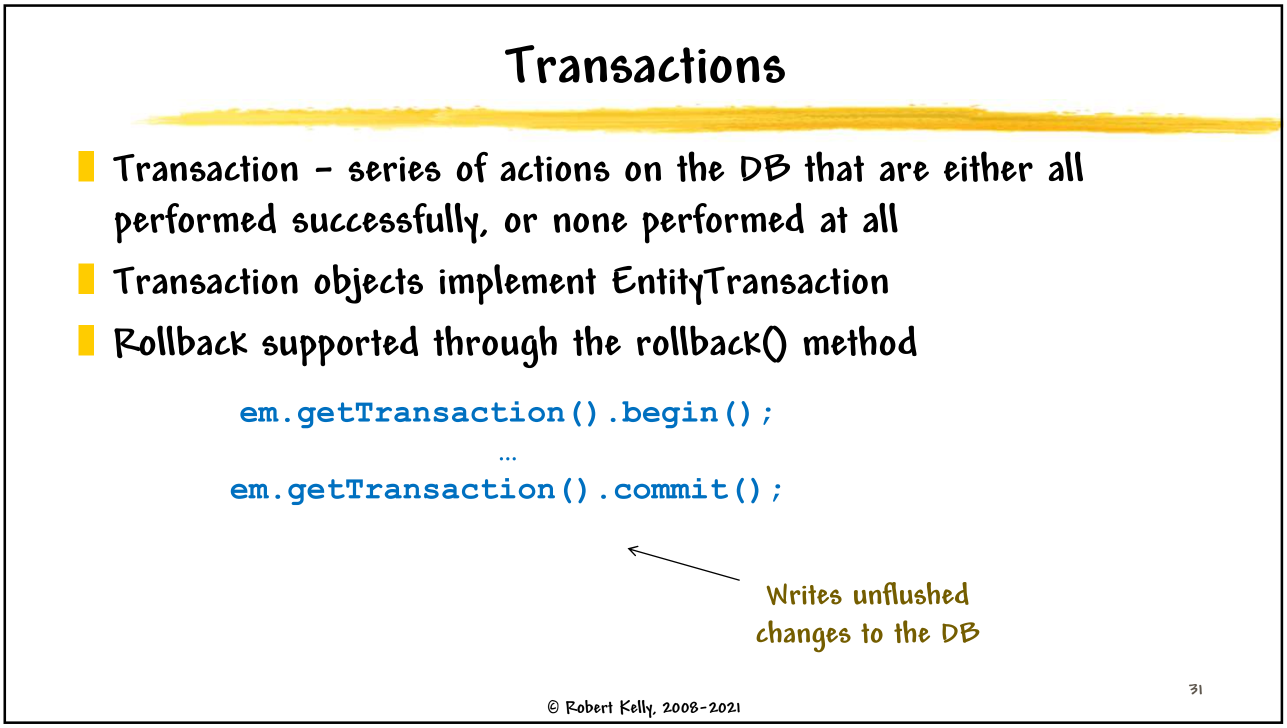
sets the TopLink logging level to FINE, which generates the first level of debugging information and also provides SQL

- Additional values are OFF, SEVERE, WARNING, INFO, CONFIG, FINER, and FINEST

30

© Robert Kelly, 2008-2021

30



Transactions

- Transaction – series of actions on the DB that are either all performed successfully, or none performed at all
- Transaction objects implement EntityTransaction
- Rollback supported through the rollback() method

```
em.getTransaction().begin();
                 …
em.getTransaction().commit();
```

Writes unflushed changes to the DB

© Robert Kelly, 2008-2021

31

31

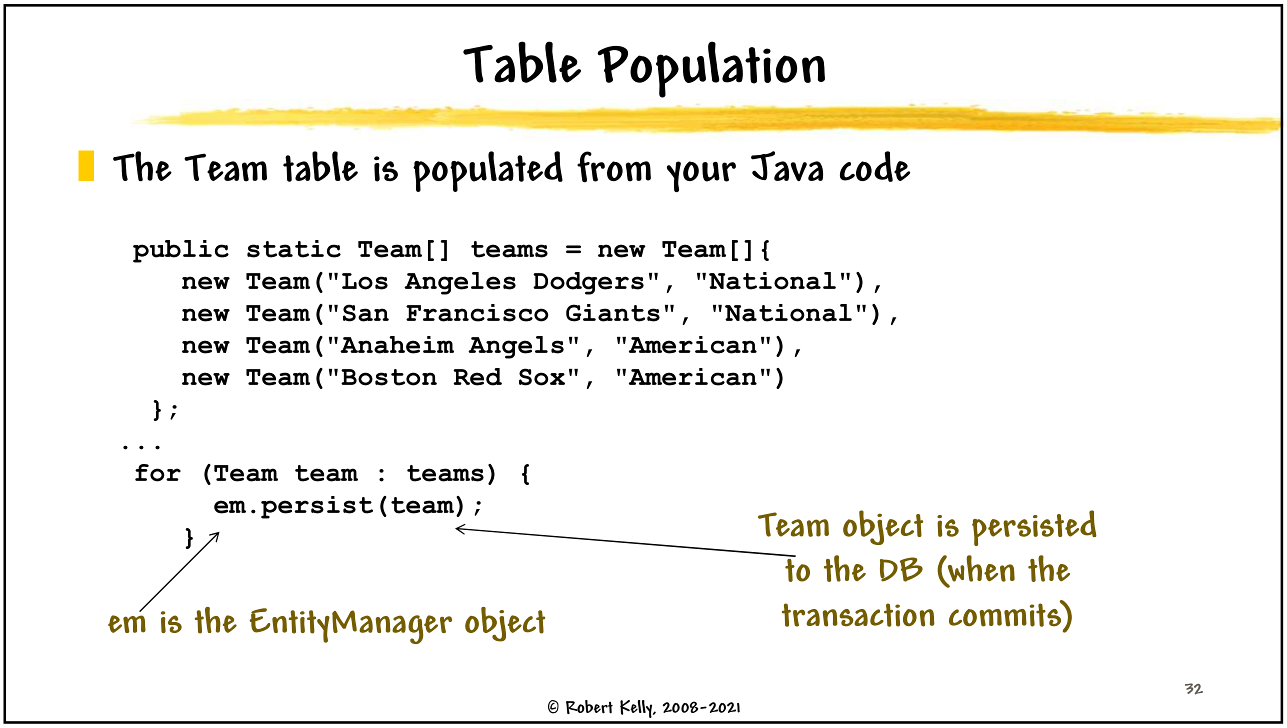


Table Population

- The Team table is populated from your Java code

```
public static Team[] teams = new Team[]{
   new Team("Los Angeles Dodgers", "National"),
   new Team("San Francisco Giants", "National"),
   new Team("Anaheim Angels", "American"),
   new Team("Boston Red Sox", "American")
 };
...
for (Team team : teams) {
     em.persist(team);
   }
```

Team object is persisted to the DB (when the transaction commits)

em is the EntityManager object

© Robert Kelly, 2008-2021

32

32

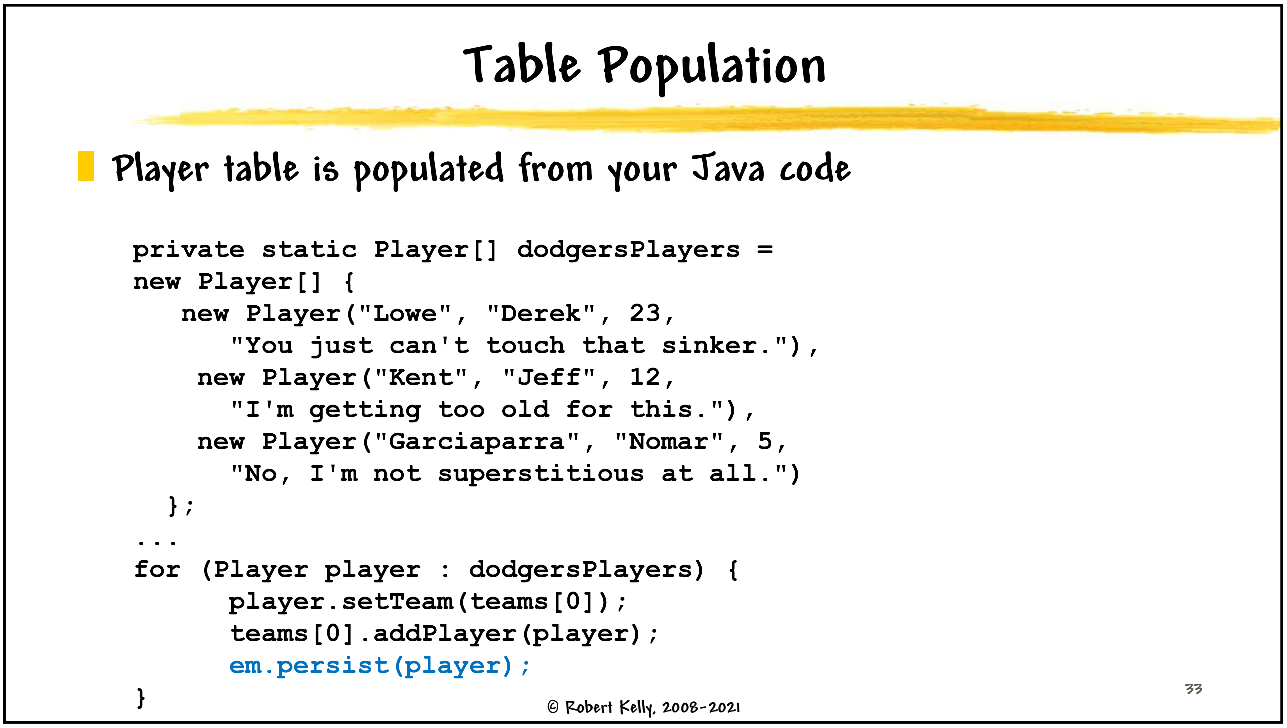


Table Population

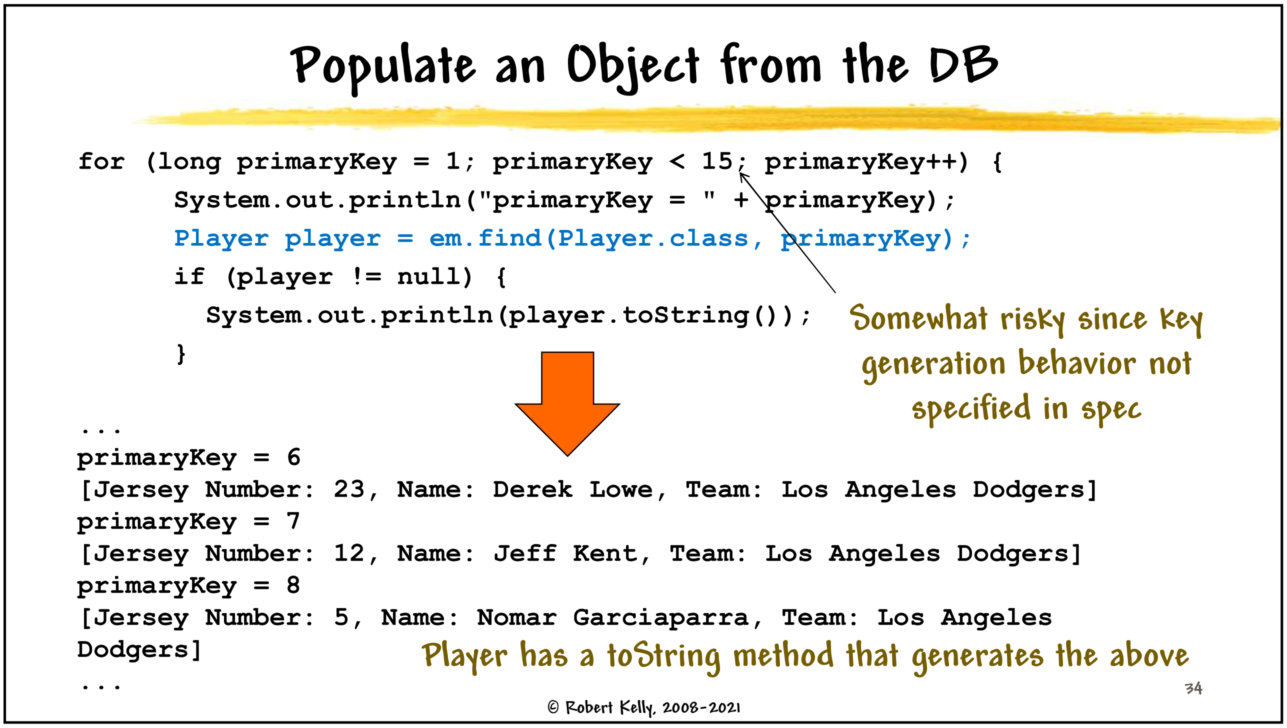
- Player table is populated from your Java code

```
private static Player[] dodgersPlayers =
new Player[] {
   new Player("Lowe", "Derek", 23,
      "You just can't touch that sinker."),
    new Player("Kent", "Jeff", 12,
      "I'm getting too old for this."),
    new Player("Garciaparra", "Nomar", 5,
      "No, I'm not superstitious at all.")
  };
...
for (Player player : dodgersPlayers) {
      player.setTeam(teams[0]);
      teams[0].addPlayer(player);
      em.persist(player);
}
```

© Robert Kelly, 2008-2021

33

33



Populate an Object from the DB

```
for (long primaryKey = 1; primaryKey < 15; primaryKey++) {
      System.out.println("primaryKey = " + primaryKey);
      Player player = em.find(Player.class, primaryKey);
      if (player != null) {
        System.out.println(player.toString());
      }
```

Somewhat risky since key generation behavior not specified in spec

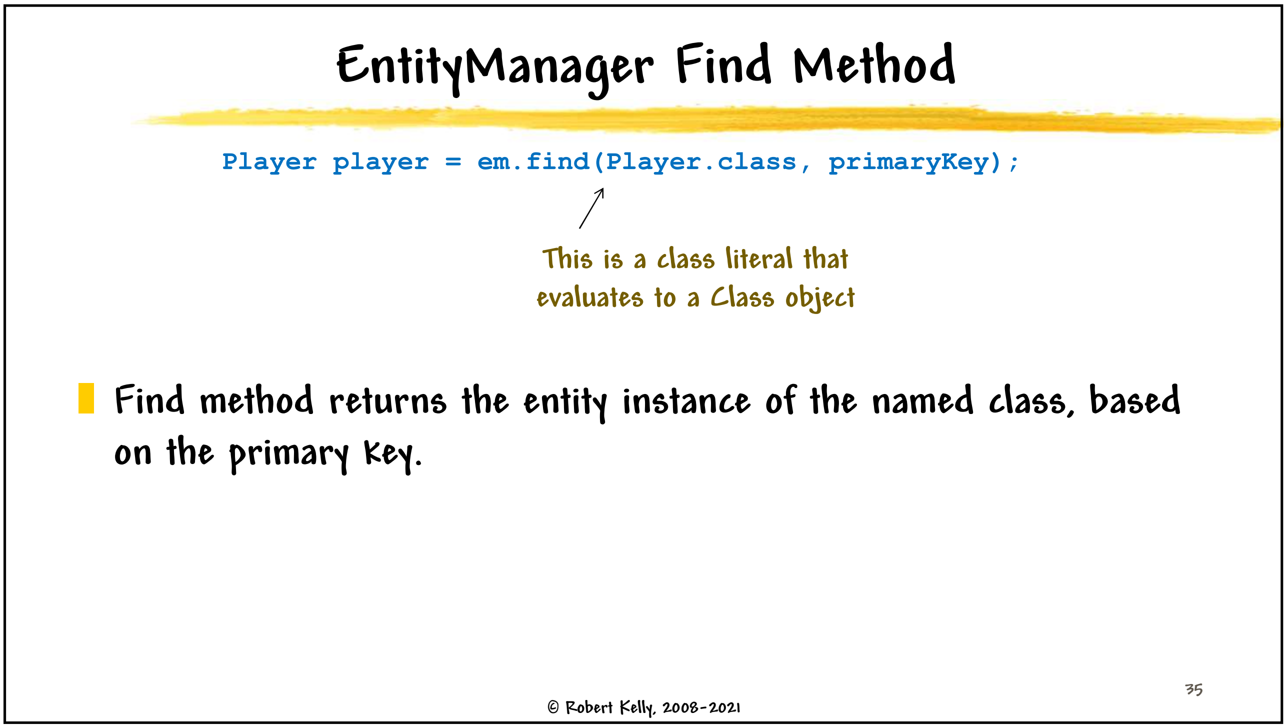
```
...
primaryKey = 6
[Jersey Number: 23, Name: Derek Lowe, Team: Los Angeles Dodgers]
primaryKey = 7
[Jersey Number: 12, Name: Jeff Kent, Team: Los Angeles Dodgers]
primaryKey = 8
[Jersey Number: 5, Name: Nomar Garciaparra, Team: Los Angeles
Dodgers]
...
```

Player has a toString method that generates the above

© Robert Kelly, 2008-2021

34

34



EntityManager Find Method

```
Player player = em.find(Player.class, primaryKey);
```

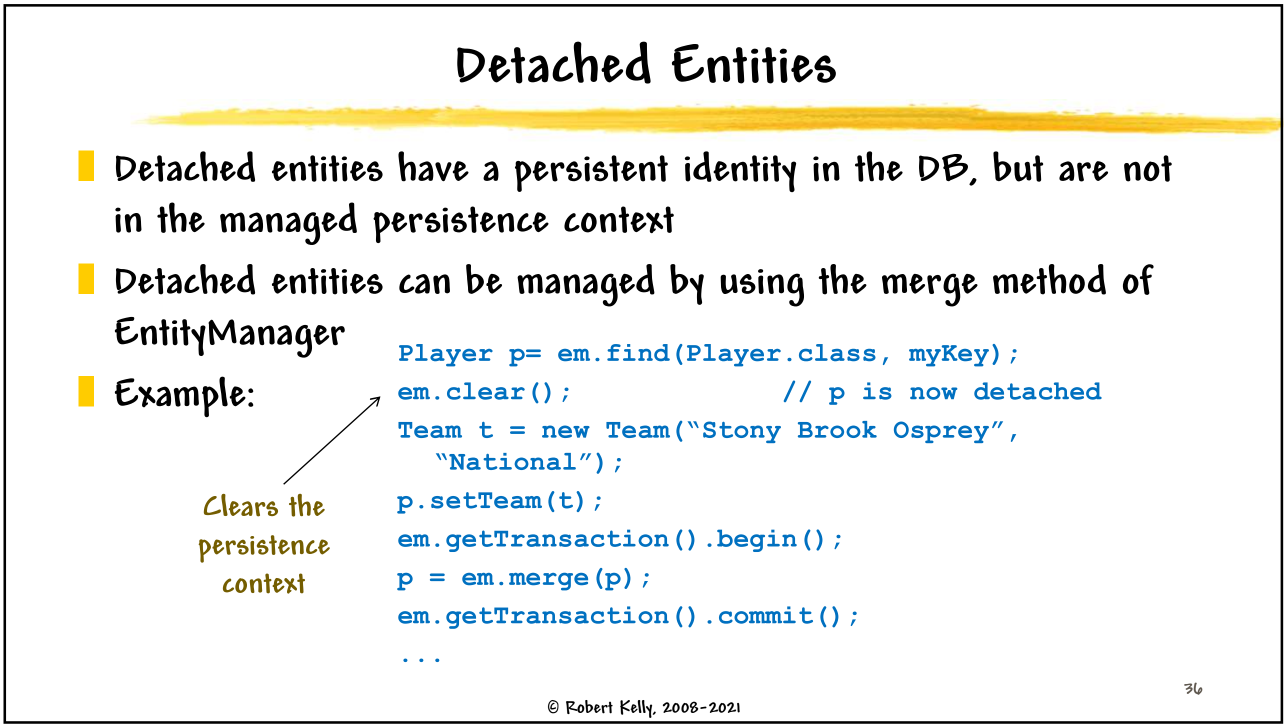
This is a class literal that evaluates to a Class object

- Find method returns the entity instance of the named class, based on the primary key.

© Robert Kelly, 2008-2021

35

35



Detached Entities

- Detached entities have a persistent identity in the DB, but are not in the managed persistence context
- Detached entities can be managed by using the merge method of EntityManager
- Example:

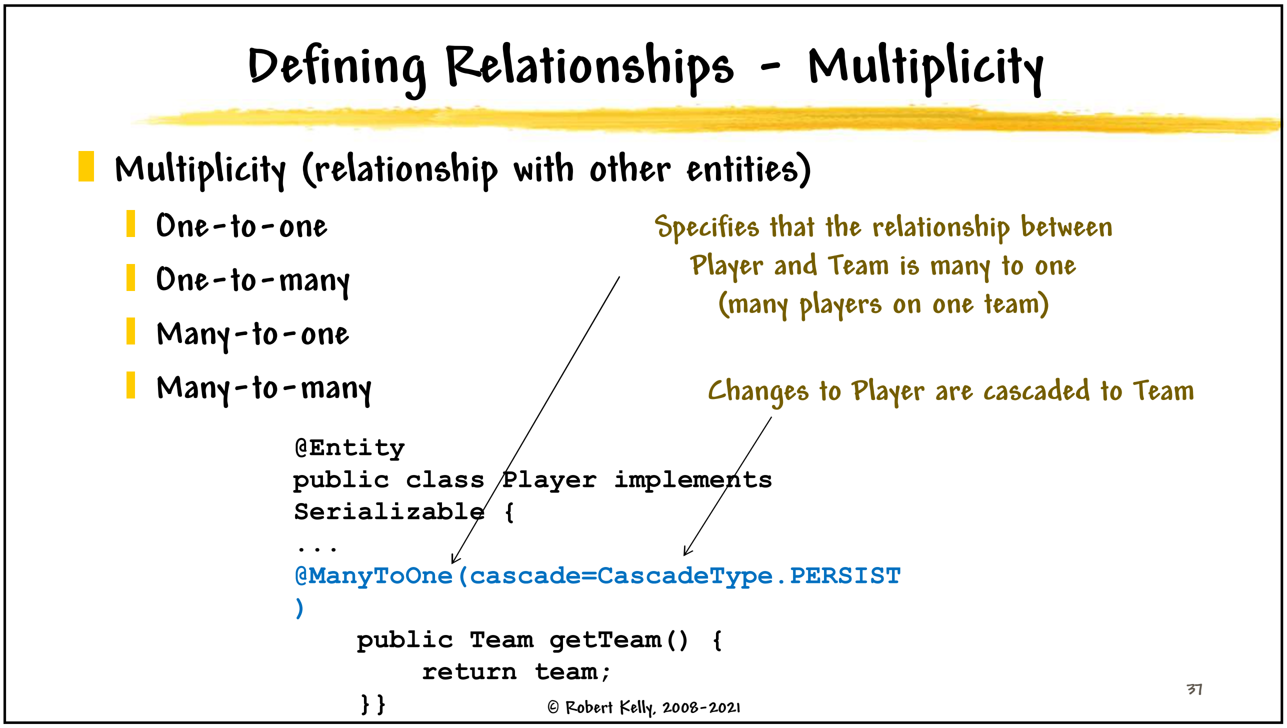
```
Player p= em.find(Player.class, myKey);
em.clear();                // p is now detached
Team t = new Team("Stony Brook Osprey",
   "National");
p.setTeam(t);
em.getTransaction().begin();
p = em.merge(p);
em.getTransaction().commit();
...
```

Clears the persistence context

© Robert Kelly, 2008-2021

36

36



Defining Relationships - Multiplicity

- Multiplicity (relationship with other entities)
  - One-to-one
  - One-to-many
  - Many-to-one
  - Many-to-many

Specifies that the relationship between Player and Team is many to one (many players on one team)

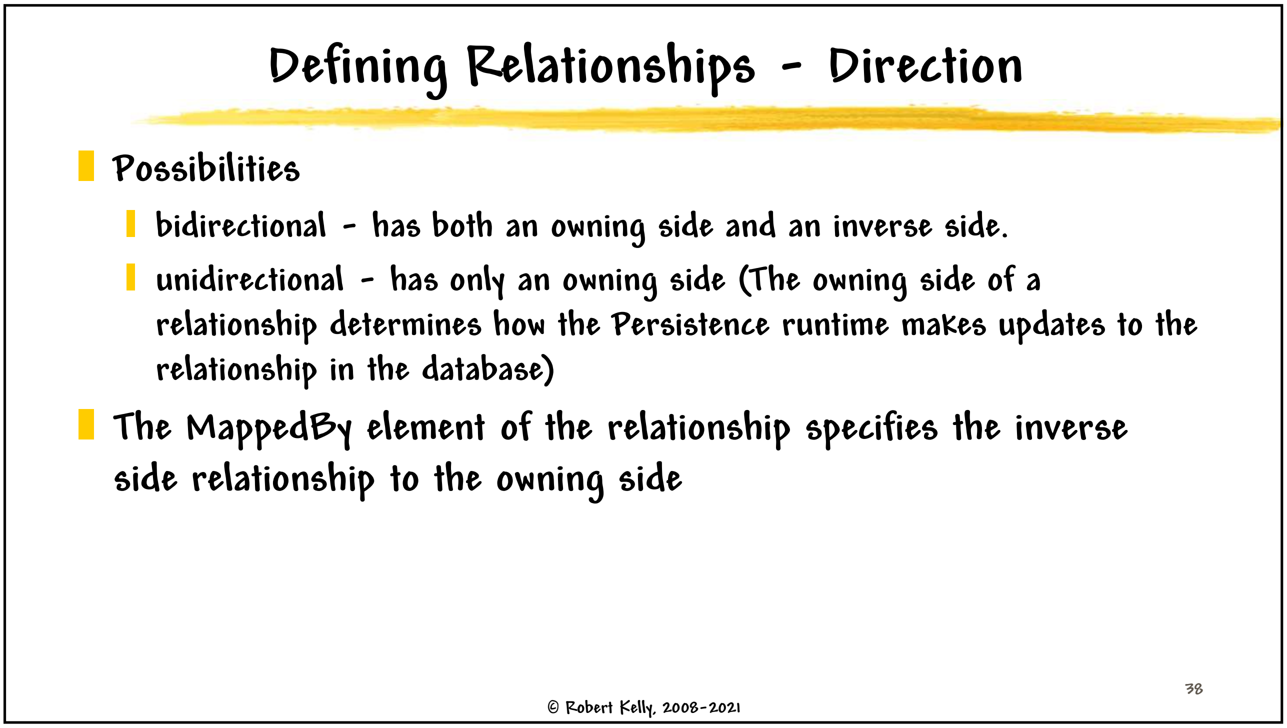
Changes to Player are cascaded to Team

```
@Entity
public class Player implements
Serializable {
...
@ManyToOne(cascade=CascadeType.PERSIST
)
    public Team getTeam() {
        return team;
    }}
```

© Robert Kelly, 2008-2021

37

37



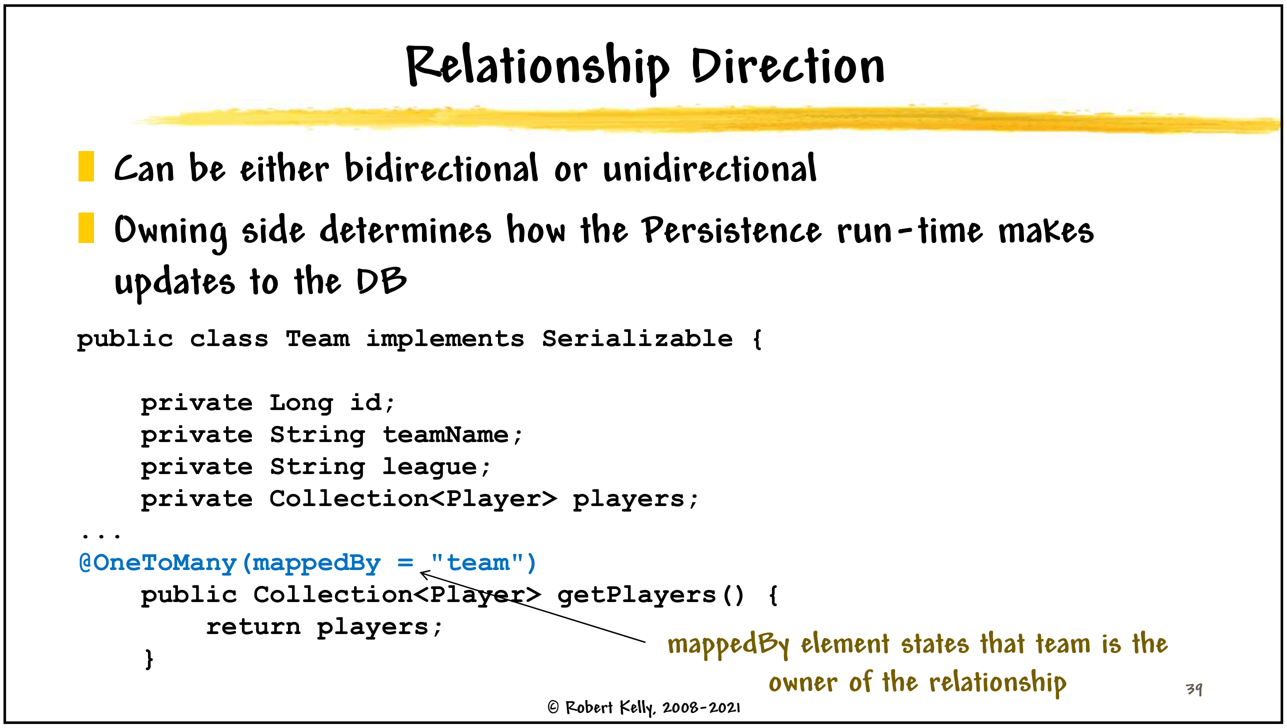
Defining Relationships - Direction

- Possibilities
  - bidirectional - has both an owning side and an inverse side.
  - unidirectional - has only an owning side (The owning side of a relationship determines how the Persistence runtime makes updates to the relationship in the database)
- The MappedBy element of the relationship specifies the inverse side relationship to the owning side

© Robert Kelly, 2008-2021

38

38



Relationship Direction

- Can be either bidirectional or unidirectional
- Owning side determines how the Persistence run-time makes updates to the DB

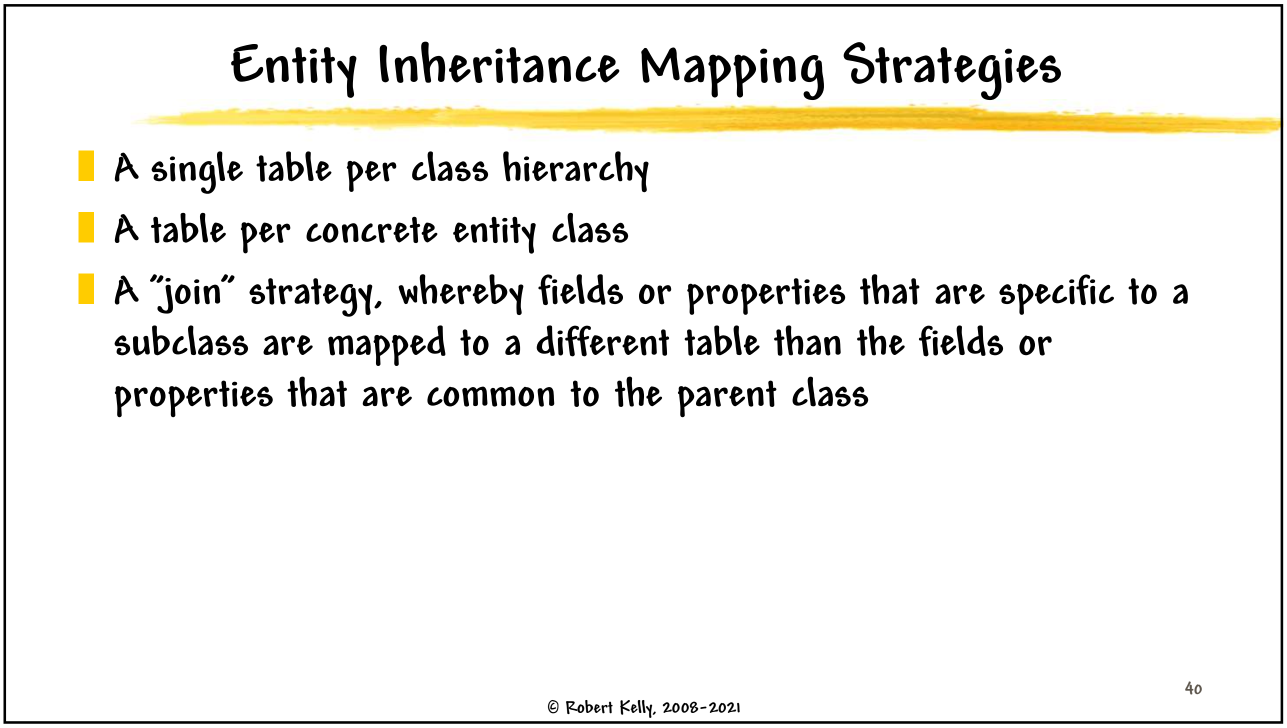
```
public class Team implements Serializable {

    private Long id;
    private String teamName;
    private String league;
    private Collection<Player> players;
...
@OneToMany(mappedBy = "team")
    public Collection<Player> getPlayers() {
        return players;
    }
```

mappedBy element states that team is the owner of the relationship

© Robert Kelly, 2008-2021

39

39



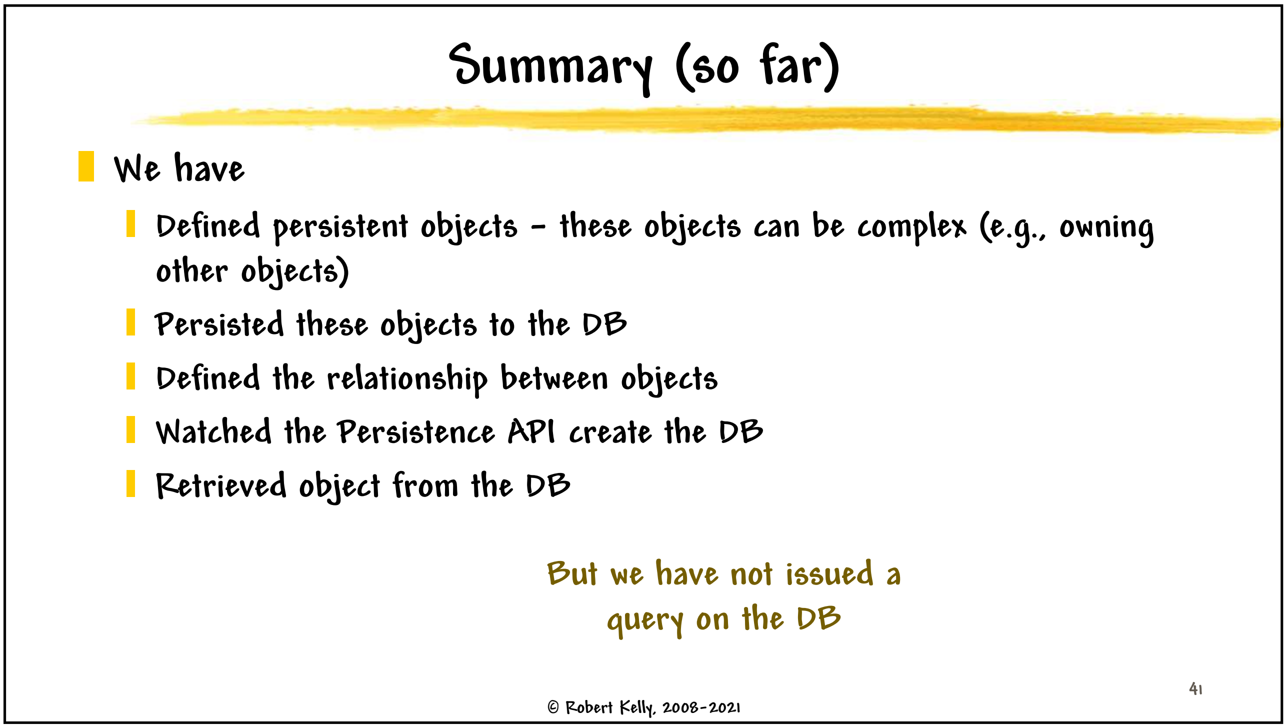
Entity Inheritance Mapping Strategies

- A single table per class hierarchy
- A table per concrete entity class
- A "join" strategy, whereby fields or properties that are specific to a subclass are mapped to a different table than the fields or properties that are common to the parent class

© Robert Kelly, 2008-2021

40

40



Summary (so far)

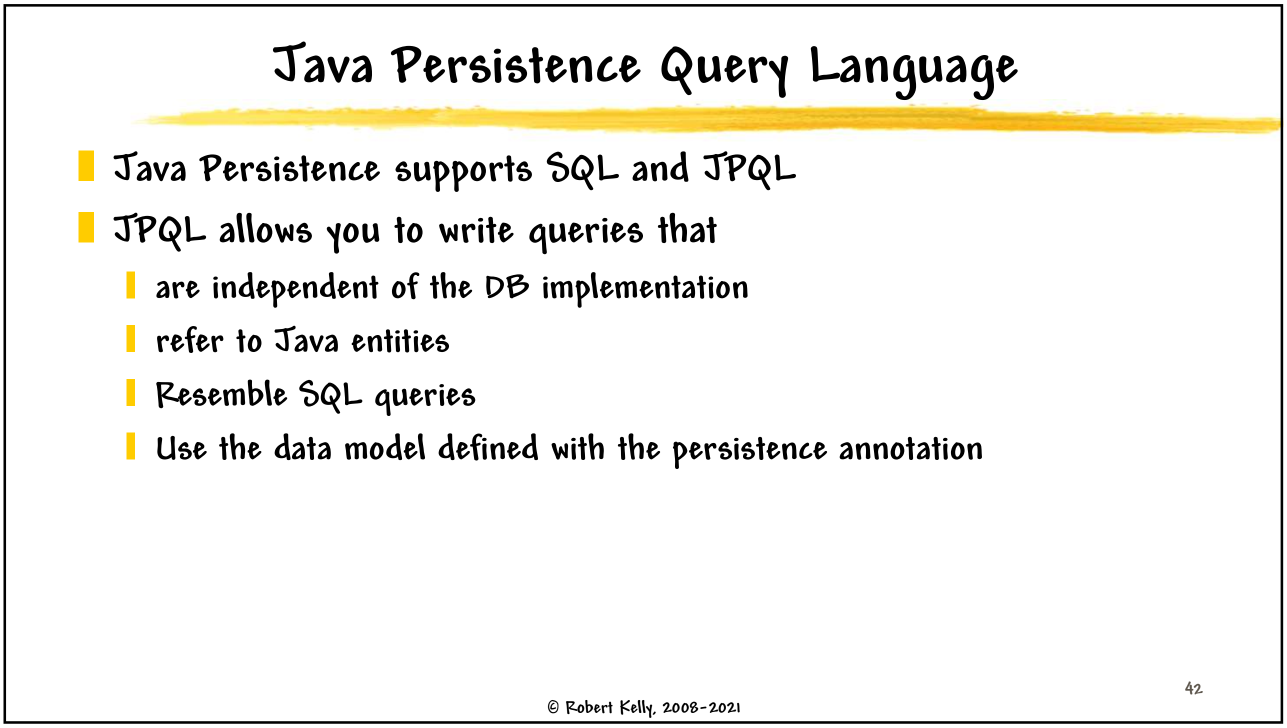
- We have
  - Defined persistent objects – these objects can be complex (e.g., owning other objects)
  - Persisted these objects to the DB
  - Defined the relationship between objects
  - Watched the Persistence API create the DB
  - Retrieved object from the DB

But we have not issued a query on the DB

© Robert Kelly, 2008-2021

41

41



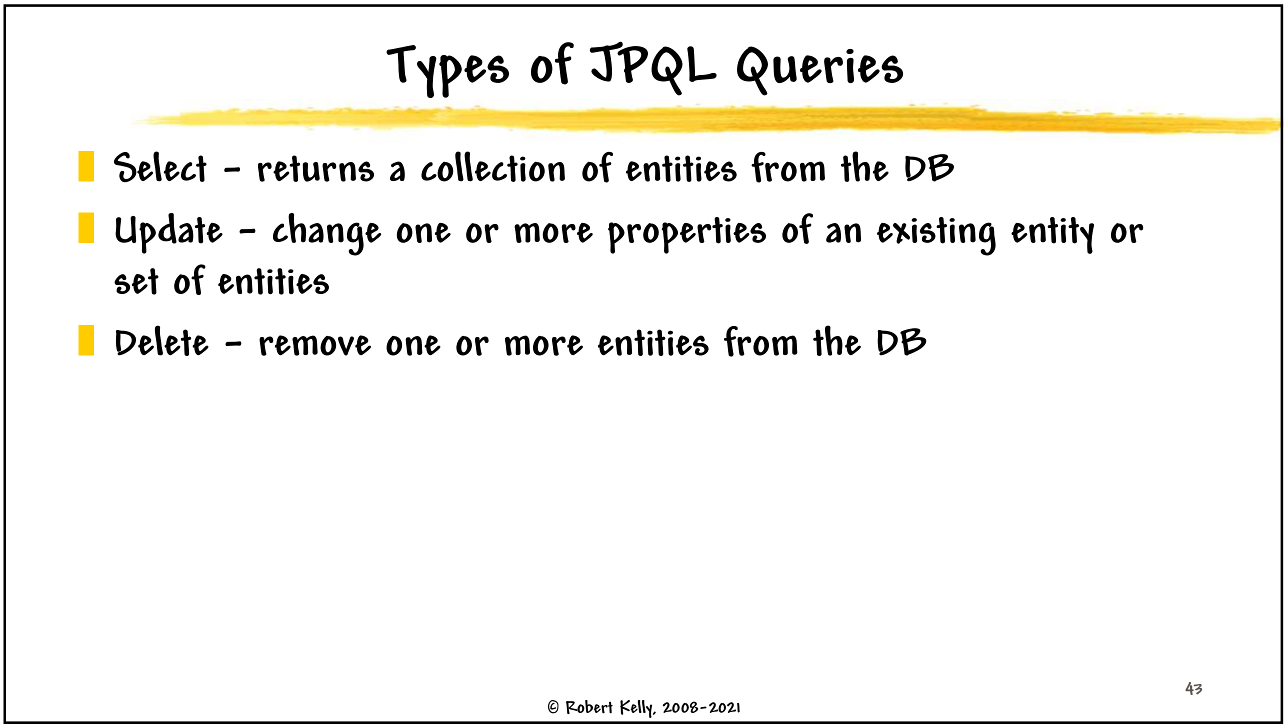
Java Persistence Query Language

- Java Persistence supports SQL and JPQL
- JPQL allows you to write queries that
  - are independent of the DB implementation
  - refer to Java entities
  - Resemble SQL queries
  - Use the data model defined with the persistence annotation

© Robert Kelly, 2008-2021

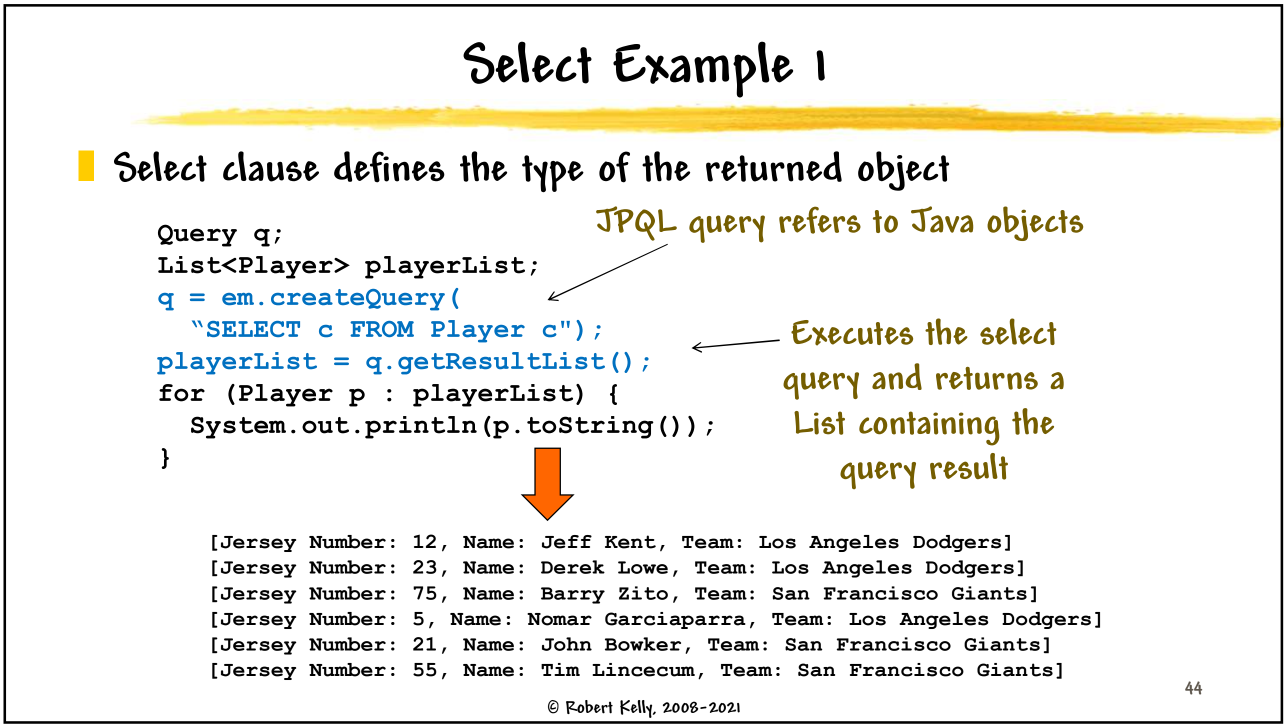
42

42



Types of JPQL Queries

- Select – returns a collection of entities from the DB
- Update – change one or more properties of an existing entity or set of entities
- Delete – remove one or more entities from the DB



Select Example 1

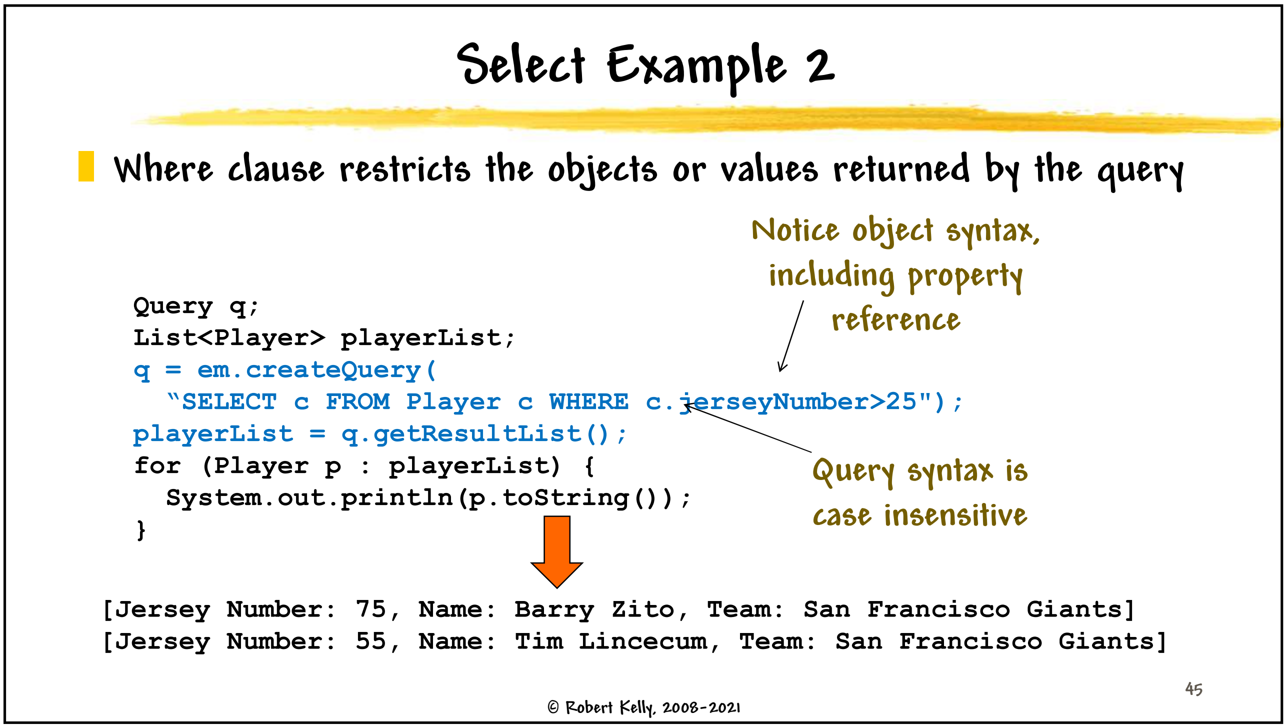
- Select clause defines the type of the returned object

```
Query q;
List<Player> playerList;
q = em.createQuery(
  "SELECT c FROM Player c");
playerList = q.getResultList();
for (Player p : playerList) {
  System.out.println(p.toString());
}
```

JPQL query refers to Java objects

Executes the select query and returns a List containing the query result

```
[Jersey Number: 12, Name: Jeff Kent, Team: Los Angeles Dodgers]
[Jersey Number: 23, Name: Derek Lowe, Team: Los Angeles Dodgers]
[Jersey Number: 75, Name: Barry Zito, Team: San Francisco Giants]
[Jersey Number: 5, Name: Nomar Garciaparra, Team: Los Angeles Dodgers]
[Jersey Number: 21, Name: John Bowker, Team: San Francisco Giants]
[Jersey Number: 55, Name: Tim Lincecum, Team: San Francisco Giants]
```



Select Example 2

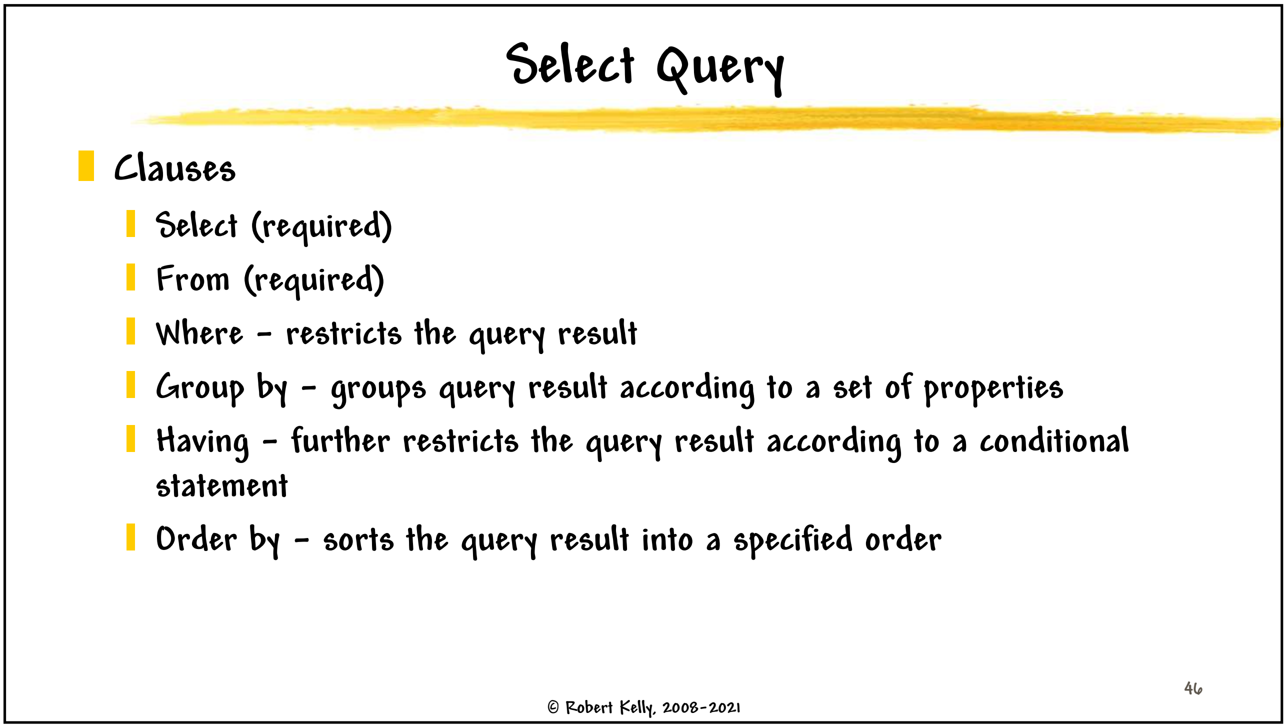
- Where clause restricts the objects or values returned by the query

```
Query q;
List<Player> playerList;
q = em.createQuery(
  “SELECT c FROM Player c WHERE c.jerseyNumber>25");
playerList = q.getResultList();
for (Player p : playerList) {
  System.out.println(p.toString());
}
```

Notice object syntax, including property reference

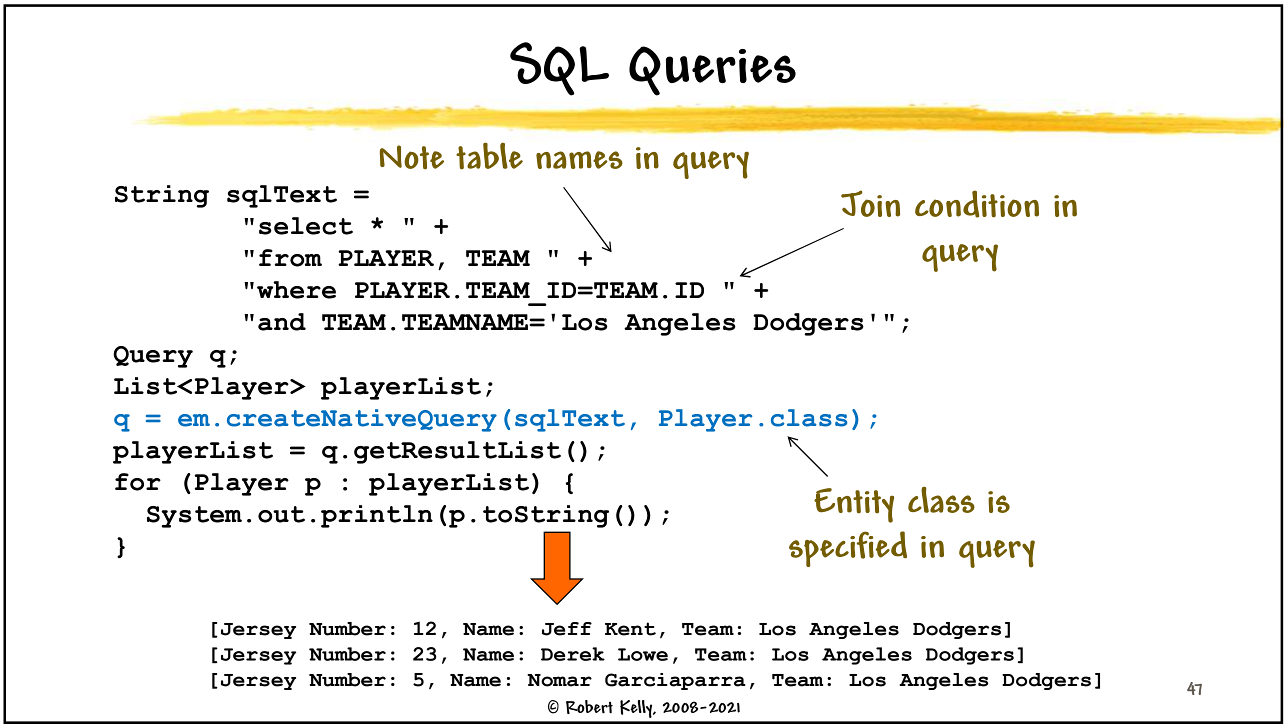
Query syntax is case insensitive

```
[Jersey Number: 75, Name: Barry Zito, Team: San Francisco Giants]
[Jersey Number: 55, Name: Tim Lincecum, Team: San Francisco Giants]
```



Select Query

- Clauses
  - Select (required)
  - From (required)
  - Where – restricts the query result
  - Group by – groups query result according to a set of properties
  - Having – further restricts the query result according to a conditional statement
  - Order by – sorts the query result into a specified order



SQL Queries

Note table names in query

Join condition in query

```
String sqlText =
        "select * " +
        "from PLAYER, TEAM " +
        "where PLAYER.TEAM_ID=TEAM.ID " +
        "and TEAM.TEAMNAME='Los Angeles Dodgers'";
Query q;
List<Player> playerList;
q = em.createNativeQuery(sqlText, Player.class);
playerList = q.getResultList();
for (Player p : playerList) {
  System.out.println(p.toString());
}
```

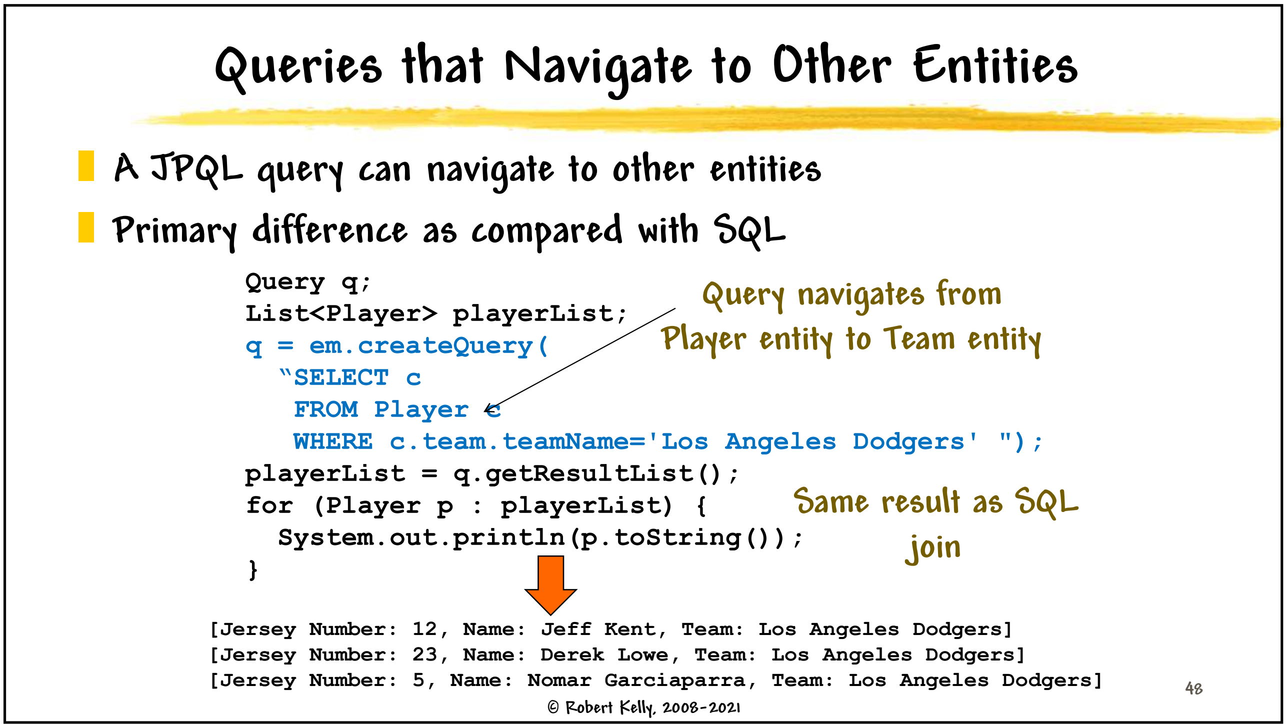
Entity class is specified in query

```
[Jersey Number: 12, Name: Jeff Kent, Team: Los Angeles Dodgers]
[Jersey Number: 23, Name: Derek Lowe, Team: Los Angeles Dodgers]
[Jersey Number: 5, Name: Nomar Garciaparra, Team: Los Angeles Dodgers]
```

47

© Robert Kelly, 2008-2021

47



Queries that Navigate to Other Entities

- A JPQL query can navigate to other entities
- Primary difference as compared with SQL

```
Query q;
List<Player> playerList;
q = em.createQuery(
  “SELECT c
   FROM Player c
   WHERE c.team.teamName='Los Angeles Dodgers' ");
playerList = q.getResultList();
for (Player p : playerList) {
  System.out.println(p.toString());
}
```

Query navigates from Player entity to Team entity

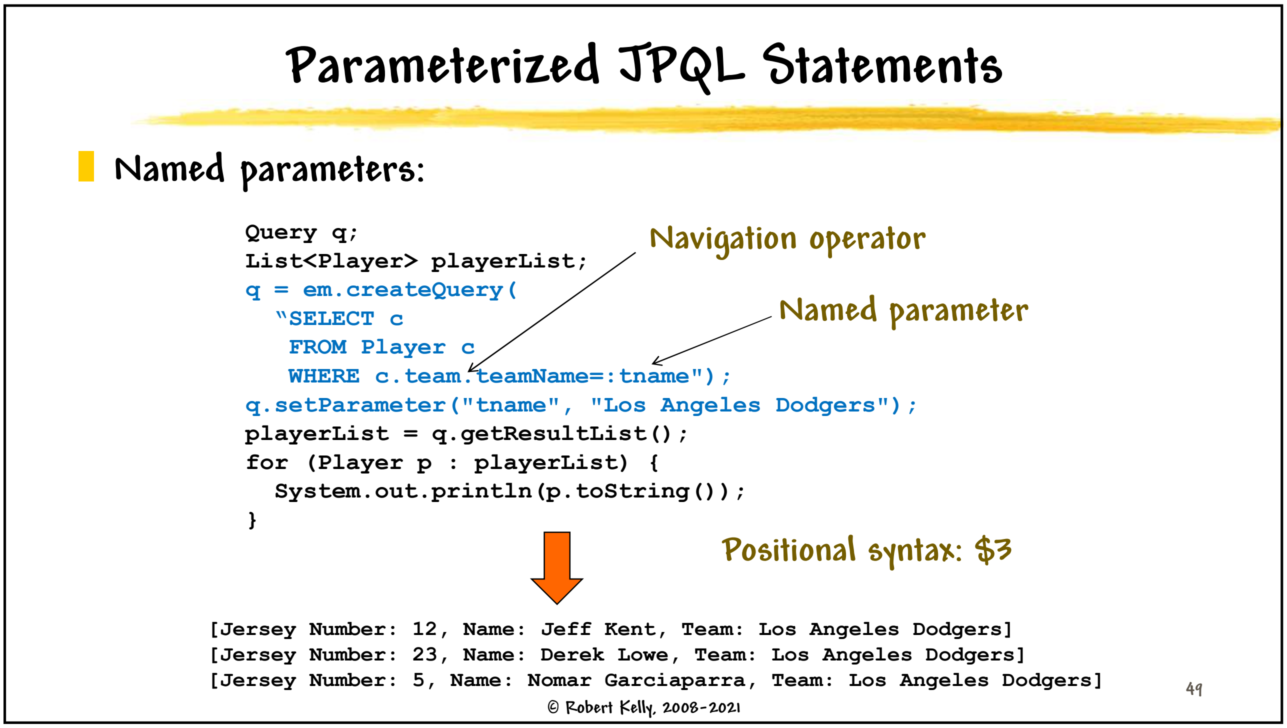
Same result as SQL join

```
[Jersey Number: 12, Name: Jeff Kent, Team: Los Angeles Dodgers]
[Jersey Number: 23, Name: Derek Lowe, Team: Los Angeles Dodgers]
[Jersey Number: 5, Name: Nomar Garciaparra, Team: Los Angeles Dodgers]
```

48

© Robert Kelly, 2008-2021

48



Parameterized JPQL Statements

- Named parameters:

```
Query q;
List<Player> playerList;
q = em.createQuery(
  "SELECT c
   FROM Player c
   WHERE c.team.teamName=:tname");
q.setParameter("tname", "Los Angeles Dodgers");
playerList = q.getResultList();
for (Player p : playerList) {
  System.out.println(p.toString());
}
```

Navigation operator

Named parameter

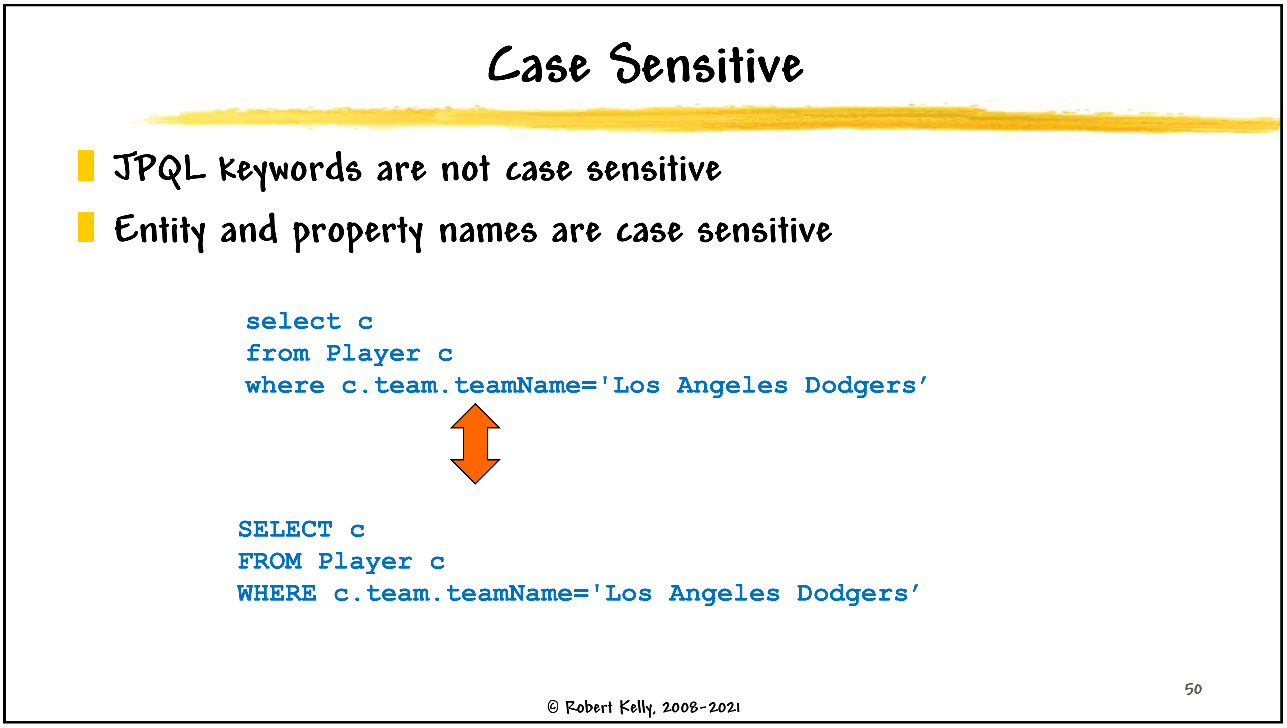
Positional syntax: $3

```
[Jersey Number: 12, Name: Jeff Kent, Team: Los Angeles Dodgers]
[Jersey Number: 23, Name: Derek Lowe, Team: Los Angeles Dodgers]
[Jersey Number: 5, Name: Nomar Garciaparra, Team: Los Angeles Dodgers]
```

© Robert Kelly, 2008-2021

49

49



Case Sensitive

- JPQL keywords are not case sensitive
- Entity and property names are case sensitive

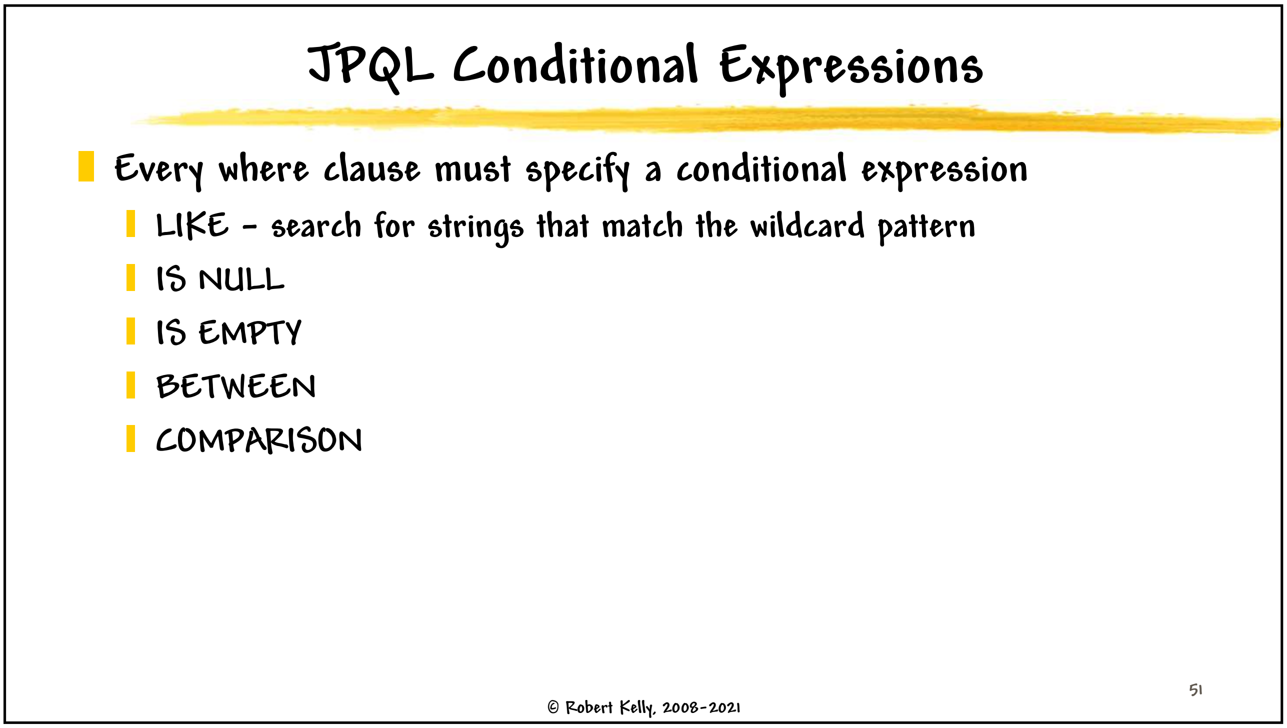
```
select c
from Player c
where c.team.teamName='Los Angeles Dodgers'
```

```
SELECT c
FROM Player c
WHERE c.team.teamName='Los Angeles Dodgers'
```

© Robert Kelly, 2008-2021

50

50

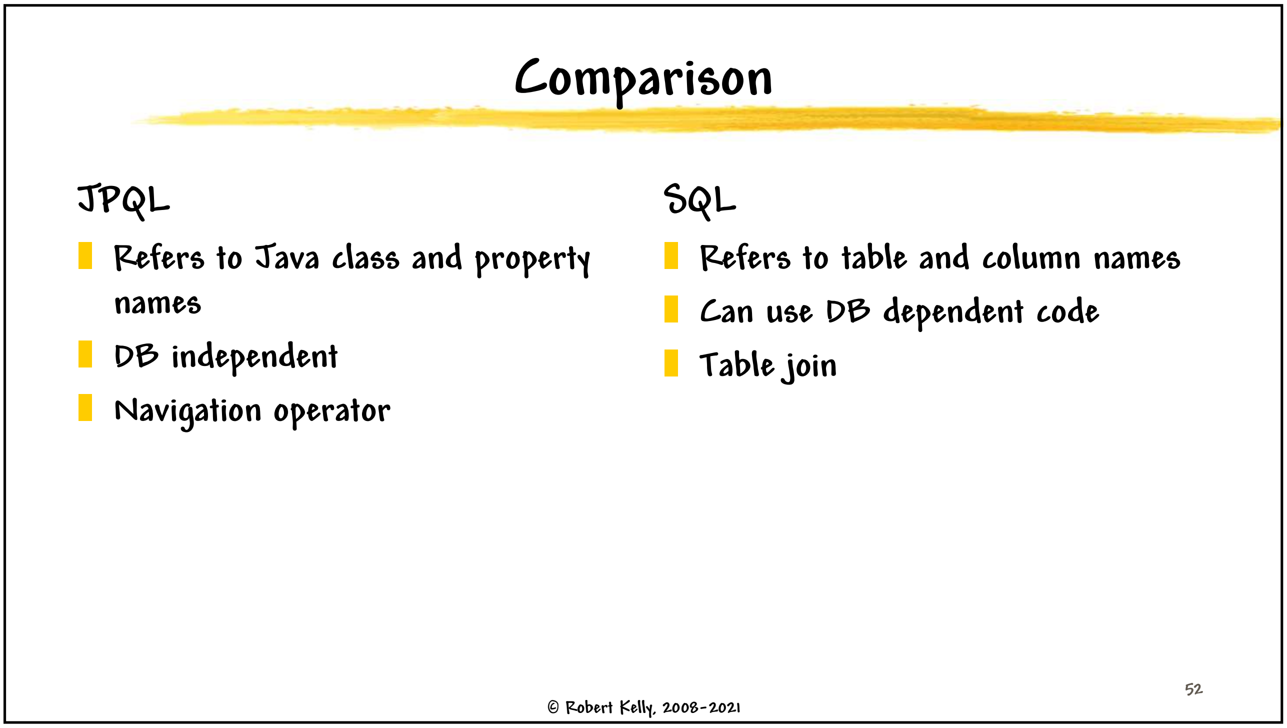


JPQL Conditional Expressions

- Every where clause must specify a conditional expression
  - LIKE - search for strings that match the wildcard pattern
  - IS NULL
  - IS EMPTY
  - BETWEEN
  - COMPARISON

51

51



Comparison

JPQL

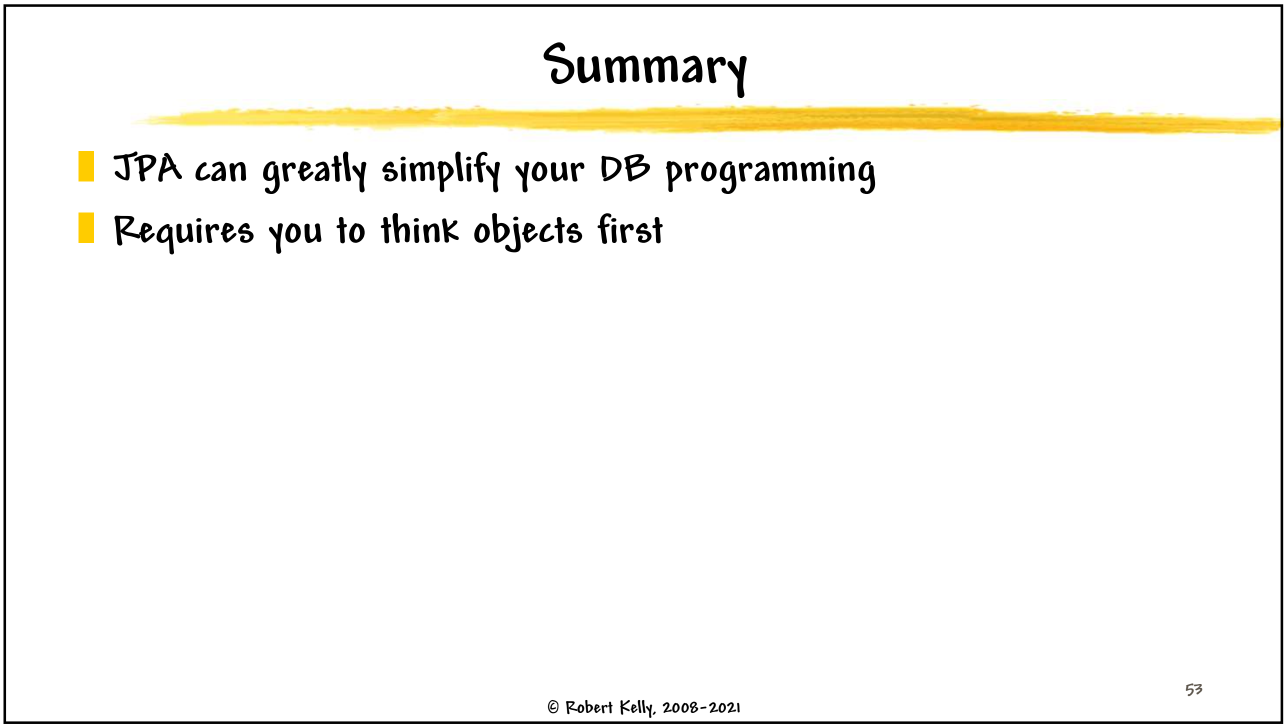
- Refers to Java class and property names
- DB independent
- Navigation operator

SQL

- Refers to table and column names
- Can use DB dependent code
- Table join

52

52



Summary

- JPA can greatly simplify your DB programming
- Requires you to think objects first

© Robert Kelly, 2008-2021

53

53