MONASH University

MONASH
INFORMATION
TECHNOLOGY

FIT5192 Lecture 7: Advanced Application Java Persistence

GROUP
OF EIGHT
AUSTRALIA

Last Lecture

- Examine how we can apply an MVC pattern approach to developing JSF web applications
- Review the approaches that we can take for validating data with the Java EE 7 platform
- Look at some examples including Ajax

This Lecture

- More advanced ORM
- Criteria API
- Container Managed Entity Manager

MONASH University

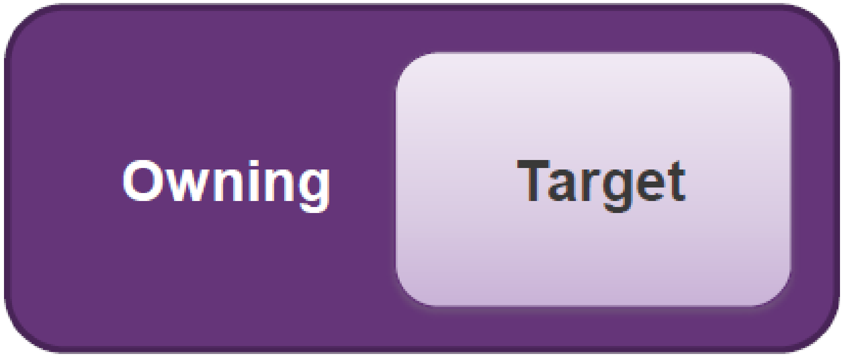
Object Relational Mapping (ORM)

Object-Relational Mapping

- Mapping data in object-oriented model to relational structure and vice versa.
- We only cover the most elementary mapping in the Intro to Java Persistence API
- This lecture, we will look at the mapping of:
  - Composition
  - Collection
  - Cardinality
  - Inheritance

Composition (1)

- A very common design in OO paradigm
- Two objects have a composition relationship, when one of them only exists as an intrinsic part of another.



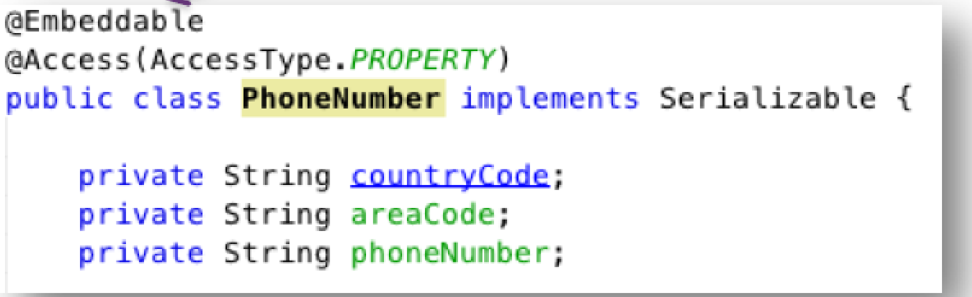
- That means, the lifetime of the target object depends on that of the owning object.

Composition (2)

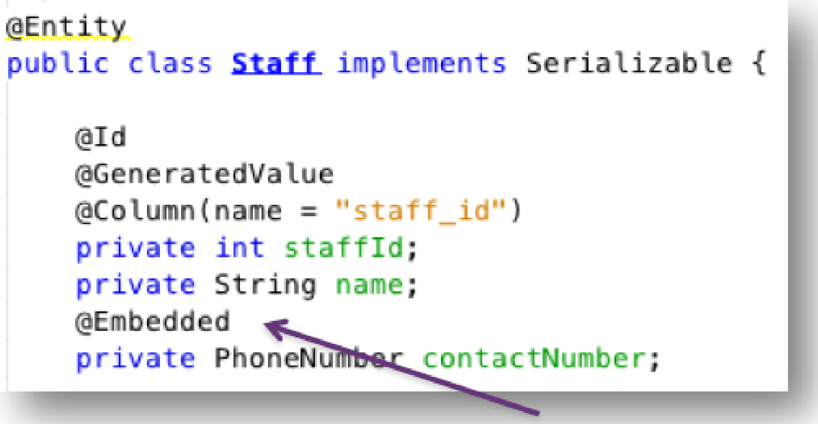
- When being mapped to a database, a target entity does not have its own persistent identity. It is stored as part of the owning entity and shares the identity of the owning entity.
- Mapped using
  - `@Embeddable`: on the target side
  - `@Embedded`: on the owning side

Composition Mapping

Indicates the objects of this class can be included as a target object

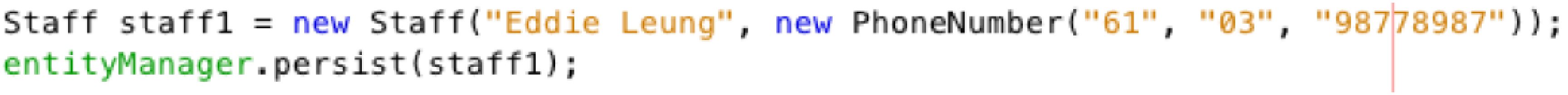


```
@Embeddable
@Access(AccessType.PROPERTY)
public class PhoneNumber implements Serializable {

    private String countryCode;
    private String areaCode;
    private String phoneNumber;
```



```
@Entity
public class Staff implements Serializable {

    @Id
    @GeneratedValue
    @Column(name = "staff_id")
    private int staffId;
    private String name;
    @Embedded
    private PhoneNumber contactNumber;
```

Mark the attribute
as a target object



```
Staff staff1 = new Staff("Eddie Leung", new PhoneNumber("61", "03", "98778987"));
entityManager.persist(staff1);
```

STAFF_ID	NAME	COUNTRY_CODE	AREA_CODE	PHONE_NUMBER
1	Eddie Leung	61	03	98778987

Access Type of an Embeddable Class

- By default, the access type of an embeddable class is determined by the access type of the owing entity class.
- If objects of an embeddable class is owned by multiple entity classes, problems may arise.

- As a result, explicitly specifying access type using `@Access` is strongly recommended.
  `@Access(` value=[FIELD,PROPERTY])

Collection

- A group of objects of:
  - basic types (i.e. non-entities) e.g. `List<Integer>`
  - Embeddable e.g. `Set<PhoneNumber>`
- Support data structures:
  - `java.util.Collection`
  - `java.util.Set`
  - `java.util.List`
- Mapped using `@ElementCollection`
- Customize settings using `@CollectionTable`
- Unless specified, the default table name is:

  Name of containing entity + "_" + attribute name (e.g. MOVIE_TAGS)

Map

- A group of objects that are stored as key-value.
- Since JPA 2.0, key and value can be of any types (e.g. basic types, embeddable objects, entities)
- Mapped using `@ElementCollection`
- Customize settings using `@CollectionTable`, `@MapKeyColumn` and `@Column`
- By default, the name of the key & value of a map is mapped to :
  - Key: The name of the referencing table + "S_KEY" (E.g. CHAPTERS_KEY)
  - Value: The name of the referencing table + "S" (E.g. CHAPTERS)

Relationship Mapping

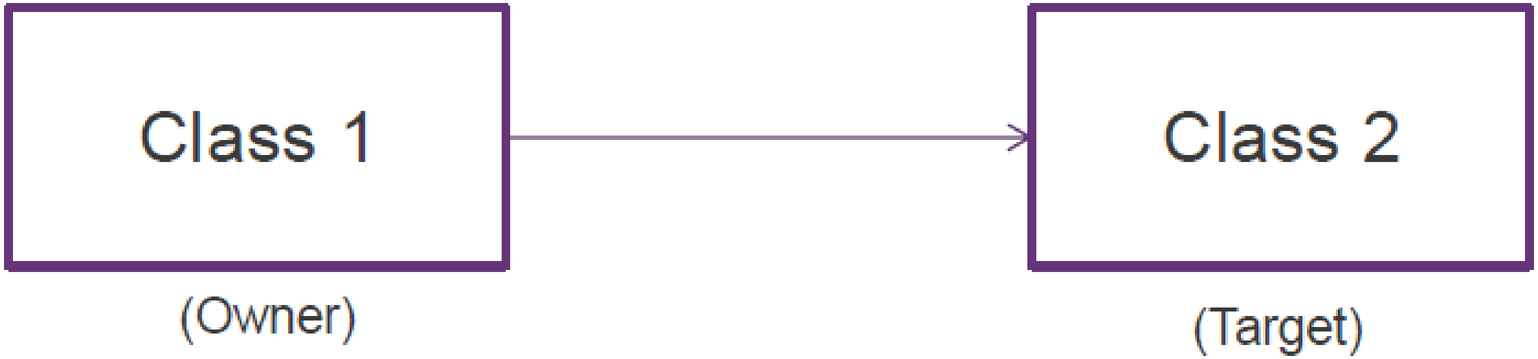
- Similar to records in relational database, objects often have relationships with each other.
- In ORM, we need to map the relationships in one to another

Relationship Directions

- Unlike relational database design, these relationships have directions.
- The direction of a relationship indicates whether object(s) on one side are “aware” of that on another.
- A relationship can be either ***unidirectional*** or ***bidirectional***.

Unidirectional Relationship

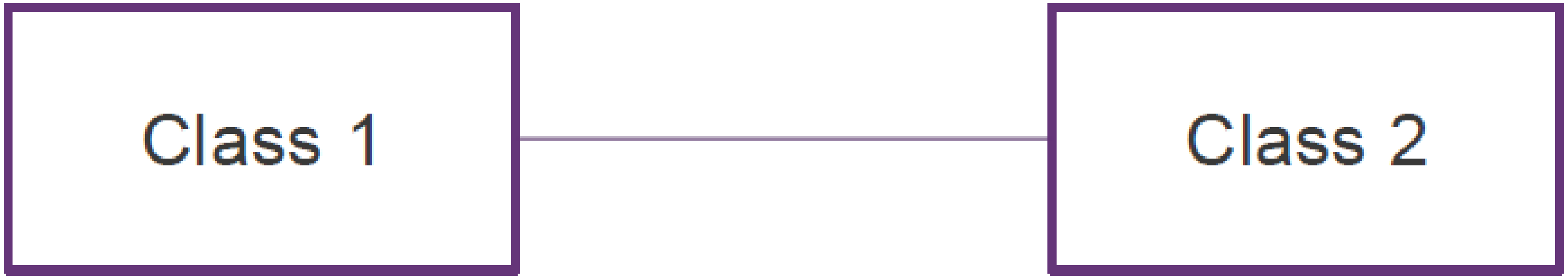
- Object(s) on one side are NOT aware of that on another.
- In UML, an arrow is used to indicate the orientation



- In Java, the direction is represented by the source class having an attribute of the target class e.g. Class1 having an attribute of type Class 2.

Bidirectional Relationship

- Object(s) on BOTH sides are "aware" of that on another.
- In UML, a line (with no arrow) is used to indicate the relationship.



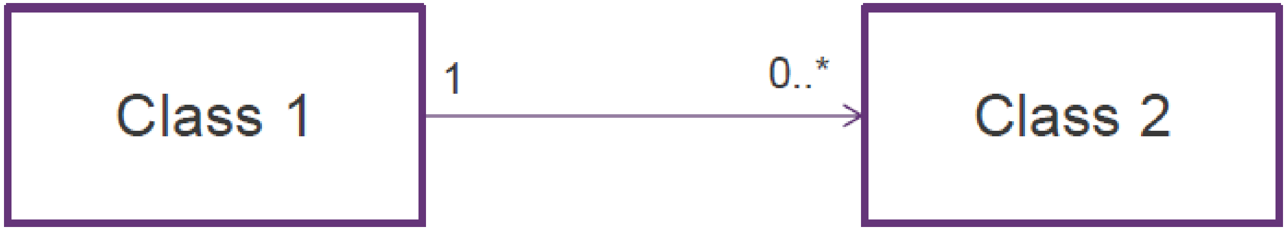
- In Java, the direction is represented by both classes having an attribute of each other e.g. Class1 has an attribute of type Class2 and Class2 has an attribute of type Class1

Cardinality (1)

- Similar to relational database design, object oriented data model has cardinality.
- Specify the minimum and maximum number of referring objects are involved in the relationship.
- In Java, the data structure used to store the attribute of each other indicates the cardinality

Cardinality (2)

UML Notation	Min. No. of Objects	Max. No. of Objects	Java Attribute
1	1	1	Single object
0..1	0	1	Single object (null accepted)
0..*	0	As many as needed	Dynamic data structure (e.g. List, Set, Map and etc.)
2..5	2	5	An array of size 5



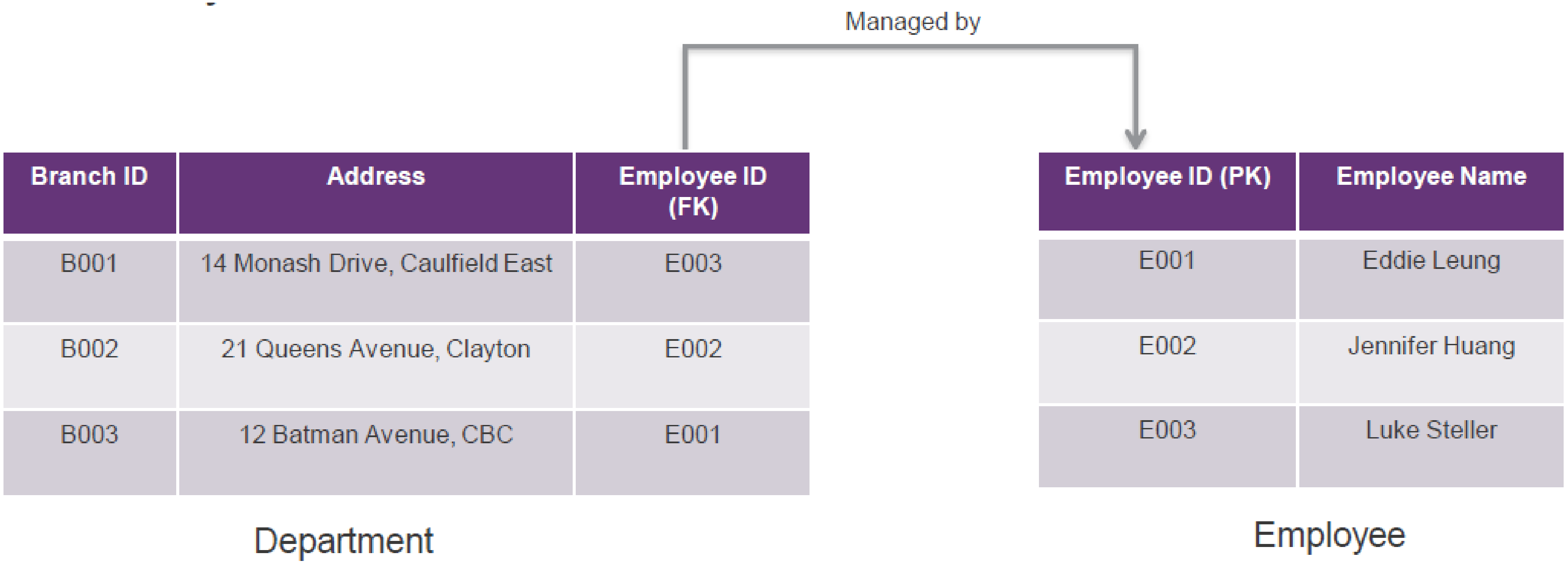
Cardinality (3)

Cardinality	Direction	Representation in Java
One-to-one	Unidirectional	Class1 has an Class2 object as attribute.
One-to-one	Bidirectional	Class1 has an Class2 object as attribute, and Class2 has an Class1 object as attribute.
One-to-many	Unidirectional	Class1 has a collection of Class2 objects as attribute.
One-to-many	Bidirectional	Class1 has a collection of Class2 objects as attribute, and Class2 has a single Class1 objects as attribute.
Many-to-one	Unidirectional	Class1 has a single Class2 objects as attribute
Many-to-one	Bidirectional	Class1 has a single Class2 objects as attribute, and Class2 has a collection of Class1 objects as attribute.
Many-to-many	Unidirectional	Class1 has a collection of Class2 objects as attribute.
Many-to-many	Bidirectional	Class1 has a collection of Class 2 objects as attribute, and Class2 has a collection of Class1 objects as attribute.

Links to Databases

Relationships in Relational Databases (1)

- Relationships among records are represented by either:
  - A foreign key (a join column): a column that refers to the primary key of another table.



Managed by

Branch ID	Address	Employee ID (FK)
B001	14 Monash Drive, Caulfield East	E003
B002	21 Queens Avenue, Clayton	E002
B003	12 Batman Avenue, CBC	E001

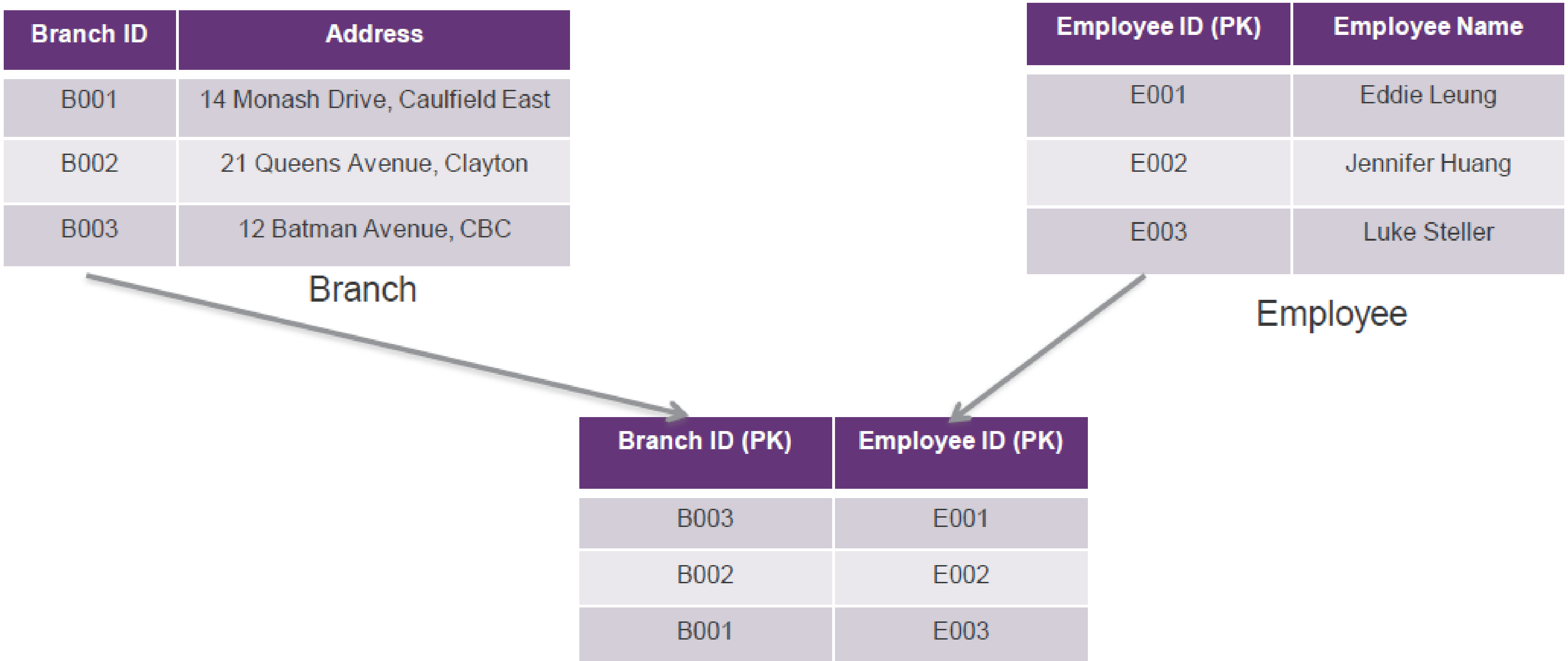
Department

Employee ID (PK)	Employee Name
E001	Eddie Leung
E002	Jennifer Huang
E003	Luke Steller

Employee

Relationships in Relational Databases (2)

A junction table: a table that is usually used with many-to-many relationship to store the keys on each table and their common attributes.



Branch

Branch ID	Address
B001	14 Monash Drive, Caulfield East
B002	21 Queens Avenue, Clayton
B003	12 Batman Avenue, CBC

Employee

Employee ID (PK)	Employee Name
E001	Eddie Leung
E002	Jennifer Huang
E003	Luke Steller

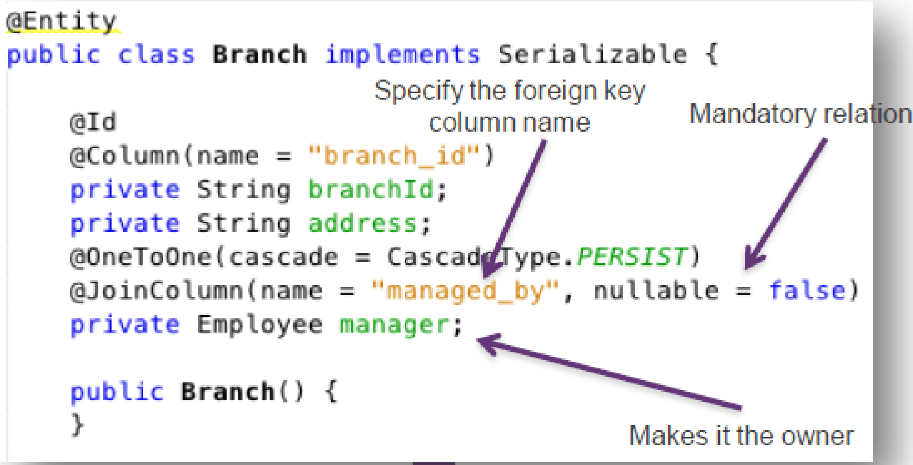
Branch ID (PK)	Employee ID (PK)
B003	E001
B002	E002
B001	E003

Mapping Unidirectional Relationship to Database (1)

- In unidirectional relationship, only one of the two classes has an attribute of the other side.
- The class that has an attribute is the *owner* of the relationship and the other one is the *inverse* owner of the relationship.
- When mapping to database, the owner table will contain a foreign key referring to the inverse owner table.
- In Java, the owner of a relationship contains the mapping annotation and is able to customize the mapping of the relationship.

Mapping Unidirectional Relationship to Database (2)

Example: A branch is managed by an employee. (`example4.unidirectional.*`)

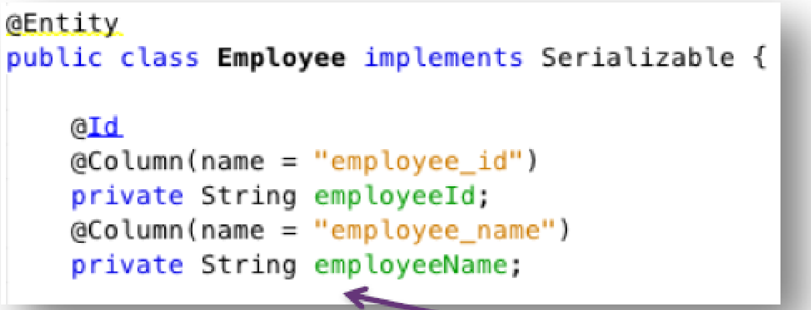


```
@Entity
public class Branch implements Serializable {

    @Id
    @Column(name = "branch_id")
    private String branchId;
    private String address;
    @OneToOne(cascade = CascadeType.PERSIST)
    @JoinColumn(name = "managed_by", nullable = false)
    private Employee manager;

    public Branch() {
    }
```

Specify the foreign key column name

Mandatory relationship

Makes it the owner



```
@Entity
public class Employee implements Serializable {

    @Id
    @Column(name = "employee_id")
    private String employeeId;
    @Column(name = "employee_name")
    private String employeeName;
```

The absence of branch object makes this a unidirectional relationship

BRANCH_ID	ADDRESS	MANAGED_BY
B001	14 Monash Drive, Caulfield East	E003
B002	21 Queens Avenue, Clayton	E002
B003	12 Batman Avenue, CBC	E001

BRANCH (Owner)

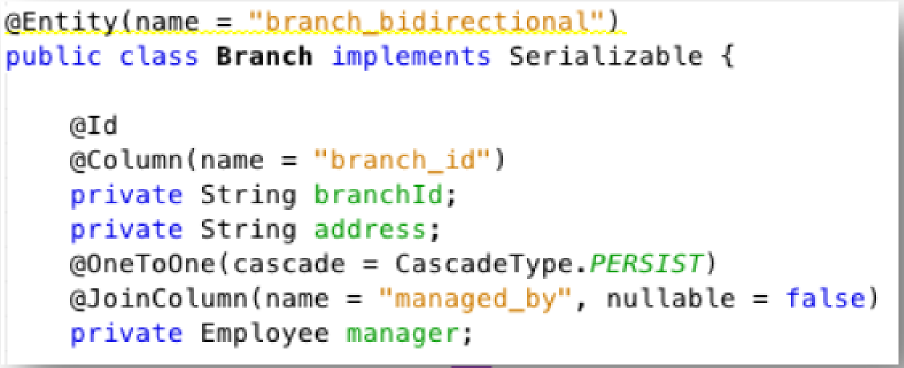
EMPLOYEE_ID	EMPLOYEE_NAME
E001	Eddie Leung
E002	Jennifer Huang
E003	Luke Steller

EMPLOYEE (Inverse)

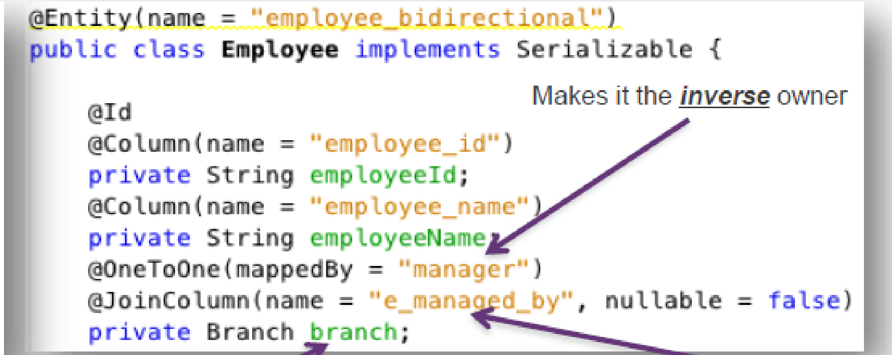
Managed by

Mapping Bidirectional Relationship to Database (1)

Example: A branch is managed by an employee, and an employee can only manage one branch (`example5.bidirectional.*`)



```java
@Entity(name = "branch_bidirectional")
public class Branch implements Serializable {

    @Id
    @Column(name = "branch_id")
    private String branchId;
    private String address;
    @OneToOne(cascade = CascadeType.PERSIST)
    @JoinColumn(name = "managed_by", nullable = false)
    private Employee manager;
```



```java
@Entity(name = "employee_bidirectional")
public class Employee implements Serializable {

    @Id
    @Column(name = "employee_id")
    private String employeeId;
    @Column(name = "employee_name")
    private String employeeName;
    @OneToOne(mappedBy = "manager")
    @JoinColumn(name = "e_managed_by", nullable = false)
    private Branch branch;
```

Makes it the ***inverse*** owner

Ignored as it is not the owner

Make this a bidirectional relationship

BRANCH_ID	ADDRESS	MANAGED_BY (FK)
B001	14 Monash Drive, Caulfield East	E003
B002	21 Queens Avenue, Clayton	E002
B003	12 Batman Avenue, CBC	E001

BRANCH (Owner)

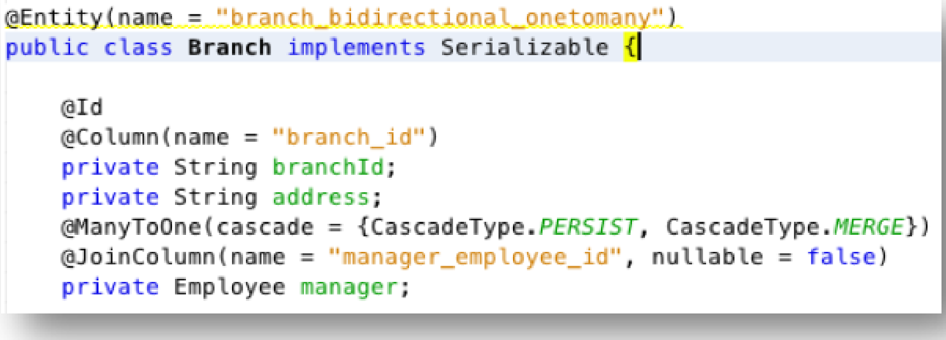
EMPLOYEE_ID (PK)	Employee Name
E001	Eddie Leung
E002	Jennifer Huang
E003	Luke Steller

Managed by

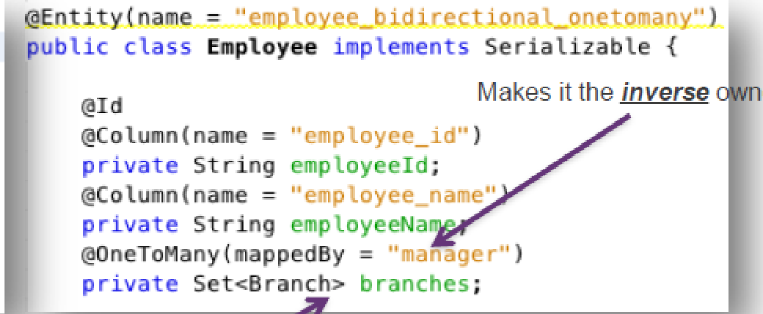
EMPLOYEE (Inverse)

Mapping Bidirectional Relationship to Database (2)

Example: A branch is managed by an employee, and an employee may manage zero or more branches (`example6.bidirectional.onetomany.*`)



```java
@Entity(name = "branch_bidirectional_onetomany")
public class Branch implements Serializable {

    @Id
    @Column(name = "branch_id")
    private String branchId;
    private String address;
    @ManyToOne(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    @JoinColumn(name = "manager_employee_id", nullable = false)
    private Employee manager;
```



```java
@Entity(name = "employee_bidirectional_onetomany")
public class Employee implements Serializable {

    @Id
    @Column(name = "employee_id")
    private String employeeId;
    @Column(name = "employee_name")
    private String employeeName;
    @OneToMany(mappedBy = "manager")
    private Set<Branch> branches;
```

Makes it the ***inverse*** owner

Make this a bidirectional relationship

BRANCH_ID	ADDRESS	MANAGER_EMPLOYEE_ID (FK)
B001	14 Monash Drive, Caulfield East	E002
B002	21 Queens Avenue, Clayton	E001
B003	12 Batman Avenue, CBC	E001

BRANCH (Owner)

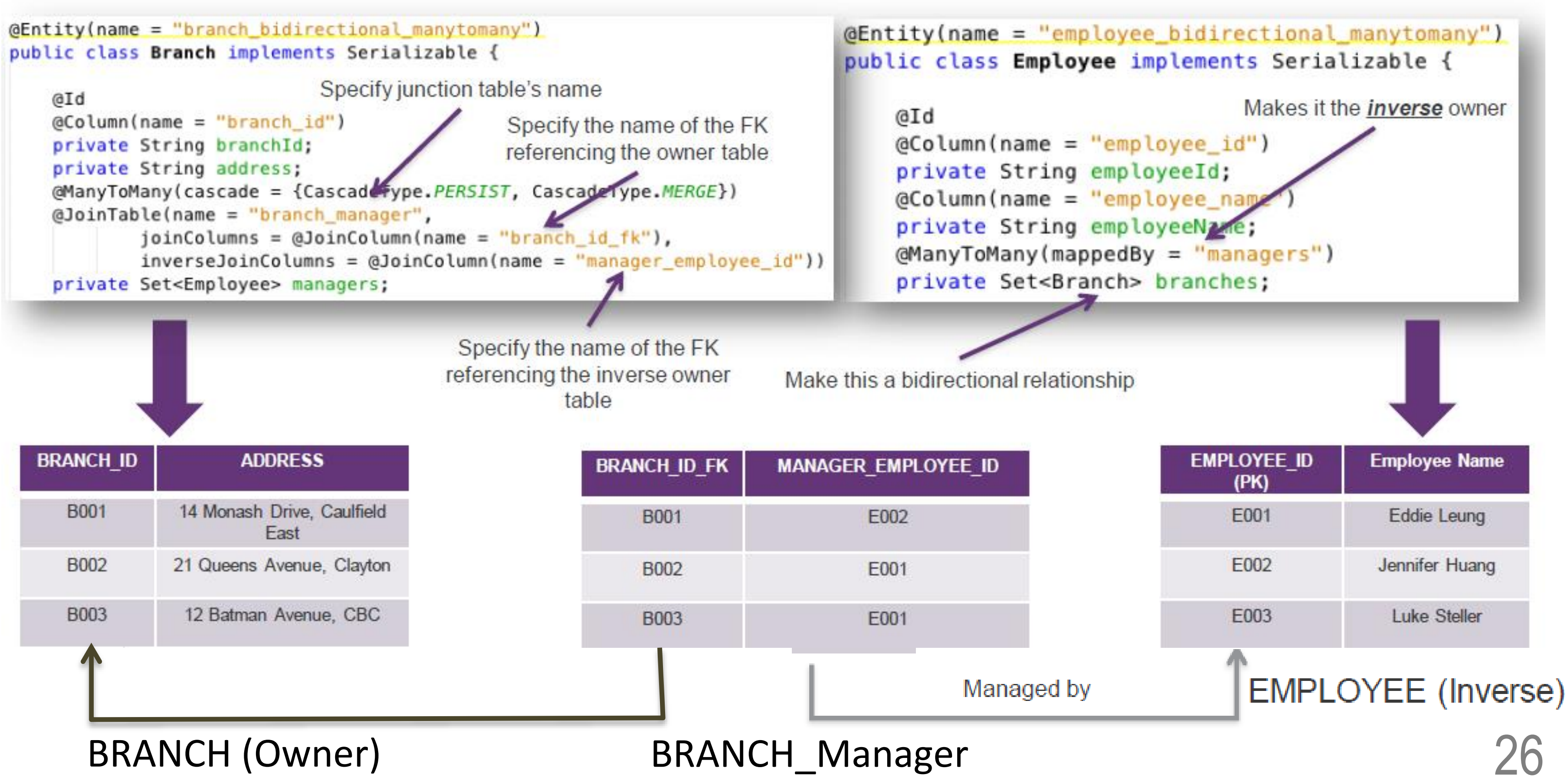
Managed by

EMPLOYEE_ID (PK)	Employee Name
E001	Eddie Leung
E002	Jennifer Huang
E003	Luke Steller

EMPLOYEE (Inverse)

Mapping Bidirectional Relationship to Database (3)

Example: A branch is managed one or more employees, and an employee may manage zero or more branches (`example7.bidirectional.manytomany.*`)



```java
@Entity(name = "branch_bidirectional_manytomany")
public class Branch implements Serializable {

    @Id
    @Column(name = "branch_id")
    private String branchId;
    private String address;
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    @JoinTable(name = "branch_manager",
            joinColumns = @JoinColumn(name = "branch_id_fk"),
            inverseJoinColumns = @JoinColumn(name = "manager_employee_id"))
    private Set<Employee> managers;
```

Specify junction table's name

Specify the name of the FK referencing the owner table

Specify the name of the FK referencing the inverse owner table

```java
@Entity(name = "employee_bidirectional_manytomany")
public class Employee implements Serializable {

    @Id
    @Column(name = "employee_id")
    private String employeeId;
    @Column(name = "employee_name")
    private String employeeName;
    @ManyToMany(mappedBy = "managers")
    private Set<Branch> branches;
```

Makes it the ***inverse*** owner

Make this a bidirectional relationship

BRANCH_ID	ADDRESS
B001	14 Monash Drive, Caulfield East
B002	21 Queens Avenue, Clayton
B003	12 Batman Avenue, CBC

BRANCH_ID_FK	MANAGER_EMPLOYEE_ID
B001	E002
B002	E001
B003	E001

EMPLOYEE_ID (PK)	Employee Name
E001	Eddie Leung
E002	Jennifer Huang
E003	Luke Steller

Managed by

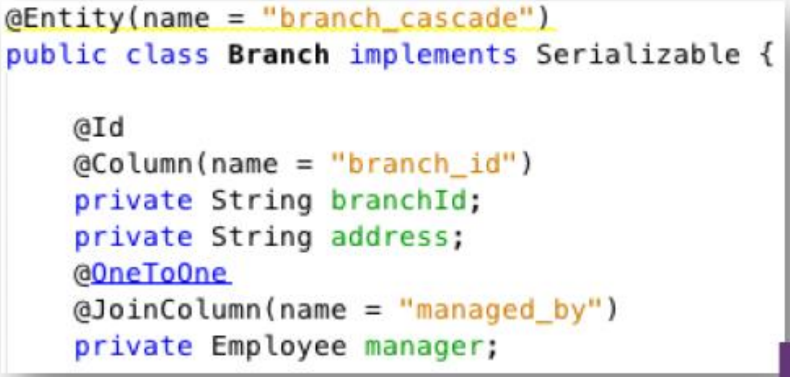
BRANCH (Owner)

BRANCH_Manager

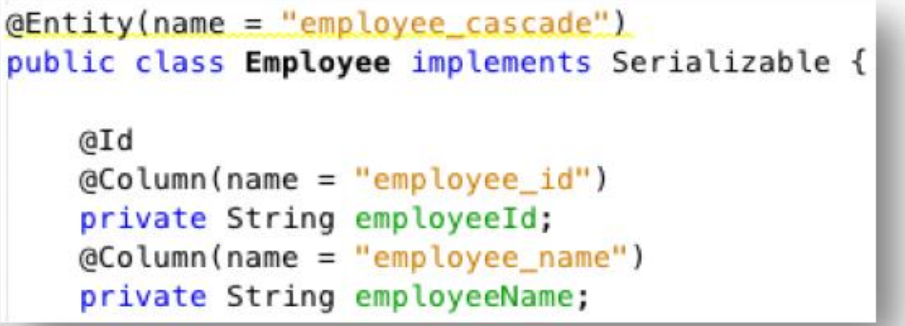
EMPLOYEE (Inverse)

Cascade (1)

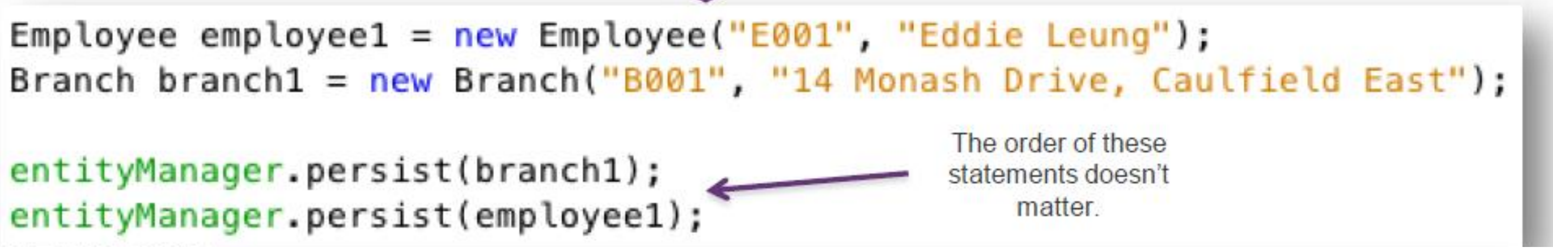
- Allow data persistence functions to be propagated to related entities.
- By default, the cascade element is empty, which means no persistence functions are propagated to related entities.
- Example:



```
@Entity(name = "branch_cascade")
public class Branch implements Serializable {

    @Id
    @Column(name = "branch_id")
    private String branchId;
    private String address;
    @OneToOne
    @JoinColumn(name = "managed_by")
    private Employee manager;
```



```
@Entity(name = "employee_cascade")
public class Employee implements Serializable {

    @Id
    @Column(name = "employee_id")
    private String employeeId;
    @Column(name = "employee_name")
    private String employeeName;
```

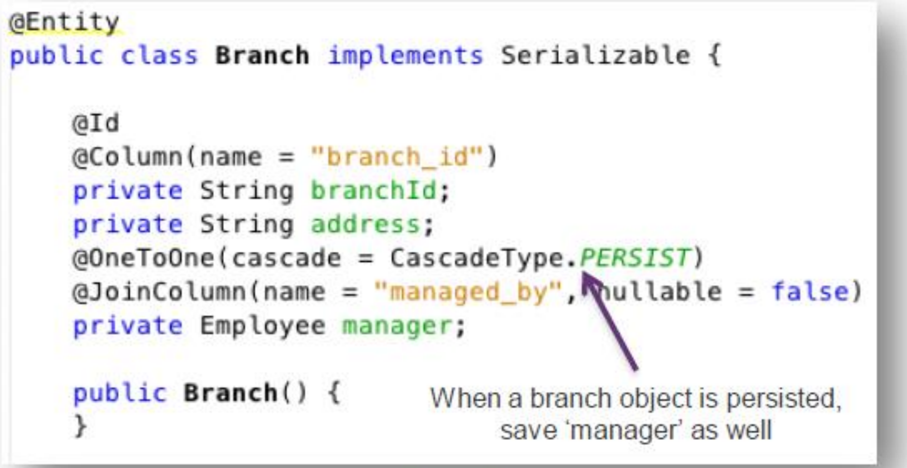


```
Employee employee1 = new Employee("E001", "Eddie Leung");
Branch branch1 = new Branch("B001", "14 Monash Drive, Caulfield East");

entityManager.persist(branch1);
entityManager.persist(employee1);
```

The order of these statements doesn't matter.

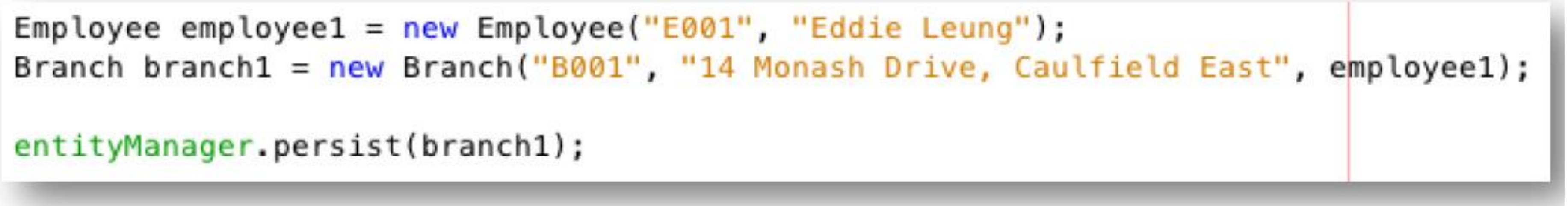
Cascade (2)

- Can be enabled by specifying in the ‘cascade’ element in each of the cardinality annotations. E.g.:



```
@Entity
public class Branch implements Serializable {

    @Id
    @Column(name = "branch_id")
    private String branchId;
    private String address;
    @OneToOne(cascade = CascadeType.PERSIST)
    @JoinColumn(name = "managed_by", nullable = false)
    private Employee manager;

    public Branch() {
    }
```

When a branch object is persisted, save ‘manager’ as well



```
Employee employee1 = new Employee("E001", "Eddie Leung");
Branch branch1 = new Branch("B001", "14 Monash Drive, Caulfield East", employee1);

entityManager.persist(branch1);
```

Cascade (3)

- There are 5 possible cascade types, including:

Cascade Type	Effects
`CascadeType.PERSIST`	Propagate EntityManager.persist operation to related entities
`CascadeType.MERGE`	Propagate EntityManager.merge operation to related entities
`CascadeType.REFRESH`	Propagate EntityManager.refresh operation to related entities
`CascadeType.REMOVE`	Propagate EntityManager.remove operation to related entities
`CascadeType.ALL`	Propagate all EntityManager operations to related entities

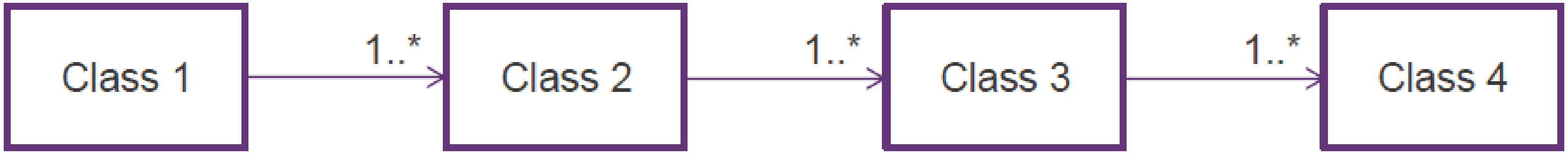
Entity Fetching Modes

Entity Fetching Modes

Determines how related entities are loaded by `EntityManager`.

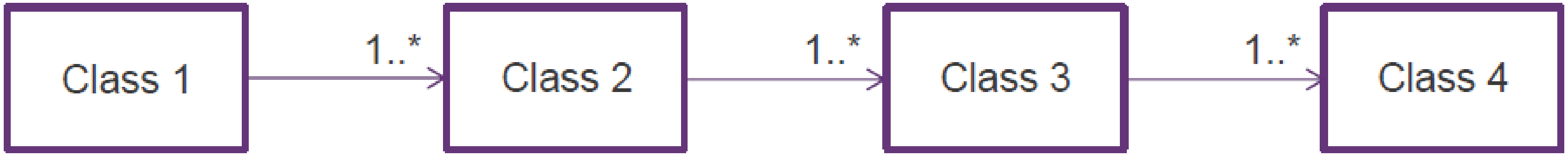
There are two modes:

Eager loading: related entities are loaded together when the owning entity is initially retrieved from database.



getClass1();

Lazy loading: related entities are ONLY loaded when it is being accessed.



getClass1().getClass2().getClass3().getClass4();

Pros and Cons of Fetching Mode

- ***Eager loading***
  - Can potentially have a large memory footprint
  - Minimum database access

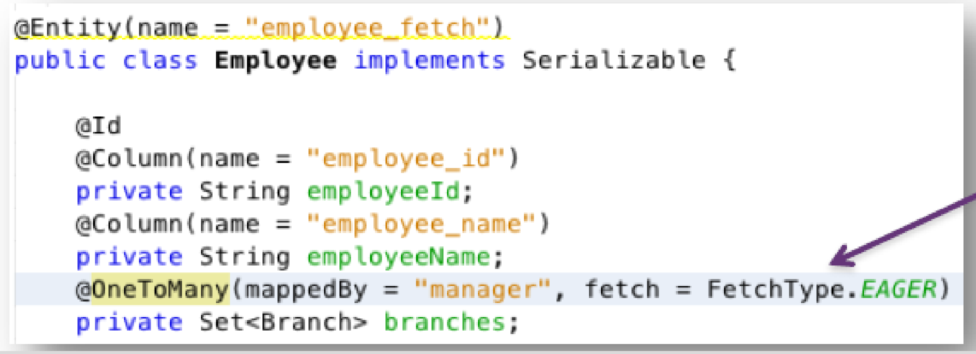
- ***Lazy loading***
  - Can reduce memory footprint
  - Might introduce performance issues due to more database access

Default Fetching Mode

- Since JPA adopts a configuration-by-exceptions approach, each cardinality has a default fetch mode as listed below:

Annotation	Default Fetch Mode	Fetch Type
@OneToOne	Eager	FetchType.EAGER
@ManyToOne	Eager	FetchType.EAGER
@OneToMany	Lazy	FetchType.LAZY
@ManyToMany	Lazy	FetchType.LAZY

- Can be changed in the ‘fetch’ element in each of the cardinality annotations. E.g.:



```
@Entity(name = "employee_fetch")
public class Employee implements Serializable {

    @Id
    @Column(name = "employee_id")
    private String employeeId;
    @Column(name = "employee_name")
    private String employeeName;
    @OneToMany(mappedBy = "manager", fetch = FetchType.EAGER)
    private Set<Branch> branches;
```

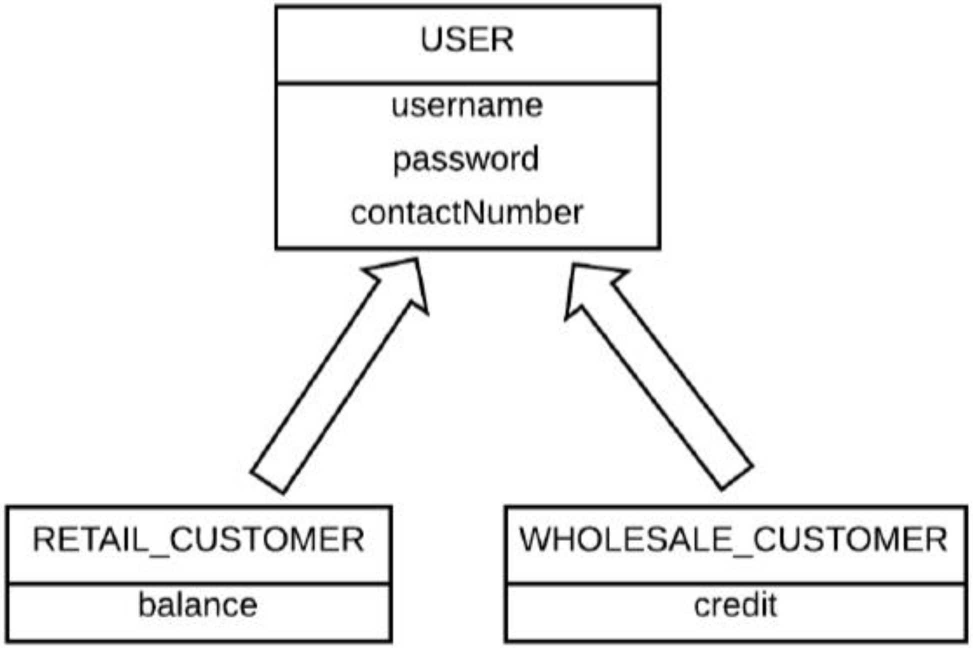
Change to eager loading

Mapping Inheritance Hierarchy to Relational Database

- The most severe case of object-relational mismatch.
- Unlike cardinality, there is no direct equivalent in relational database design.
- Three main strategies:
  - Single table
  - Joined tables
  - Table per class

Scenario

- Let’s consider an e-commerce scenario where we have 3 classes:



Single Table Strategy (1)

- Default inheritance mapping strategy for EJB 3.
- All classes in the inheritance hierarchy are mapped to a single table.
- The table will contain a superset of all data stored in the class hierarchy.
- Objects from different classes are identified using a special column called ***discriminator*** column.
- The discriminator column contains a value unique to the object type in a given row.

Single Table Strategy (2)

Discriminator Column

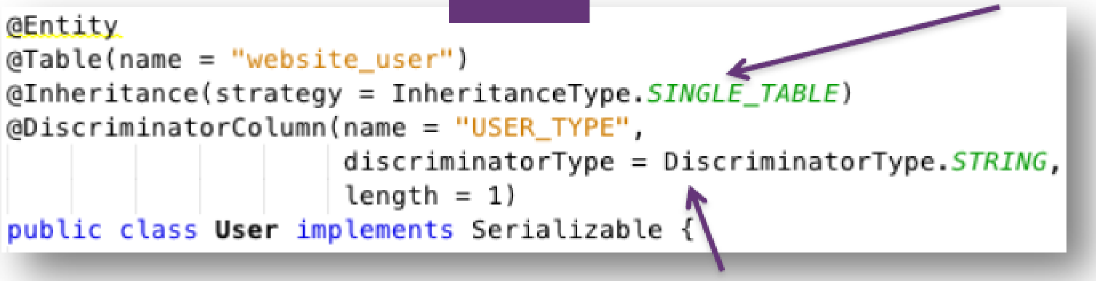
Attribute specific to RetailCustomer

Attribute specific to WholesaleCustomer

USERNAME	PASSWORD	CONTACT_NO	USER_TYPE	BALANCE	OWING
retail1	1234	0423889789	R	$300	NULL
wholesale1	5678	0411322786	W	NULL	$2500
wholesale2	8876	0432765787	W	NULL	$4300
retail2	9876	0402998997	R	$75	NULL

Attributes that are common to all users

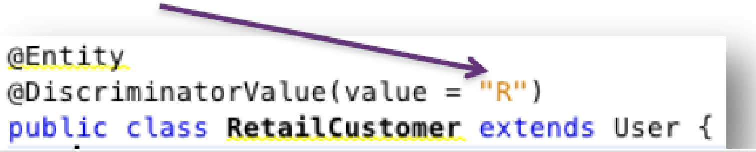
Specify single table strategy is used



```
@Entity
@Table(name = "website_user")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "USER_TYPE",
                     discriminatorType = DiscriminatorType.STRING,
                     length = 1)
public class User implements Serializable {
```

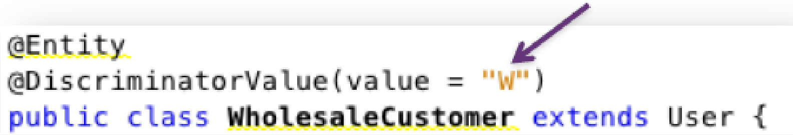
Specify discriminator details, including column, type and length of the value

Unique value that represents RetailCustomer entity



```
@Entity
@DiscriminatorValue(value = "R")
public class RetailCustomer extends User {
```

Unique value that represents WholesaleCustomer entity



```
@Entity
@DiscriminatorValue(value = "W")
public class WholesaleCustomer extends User {
```

Single Table Strategy (3)

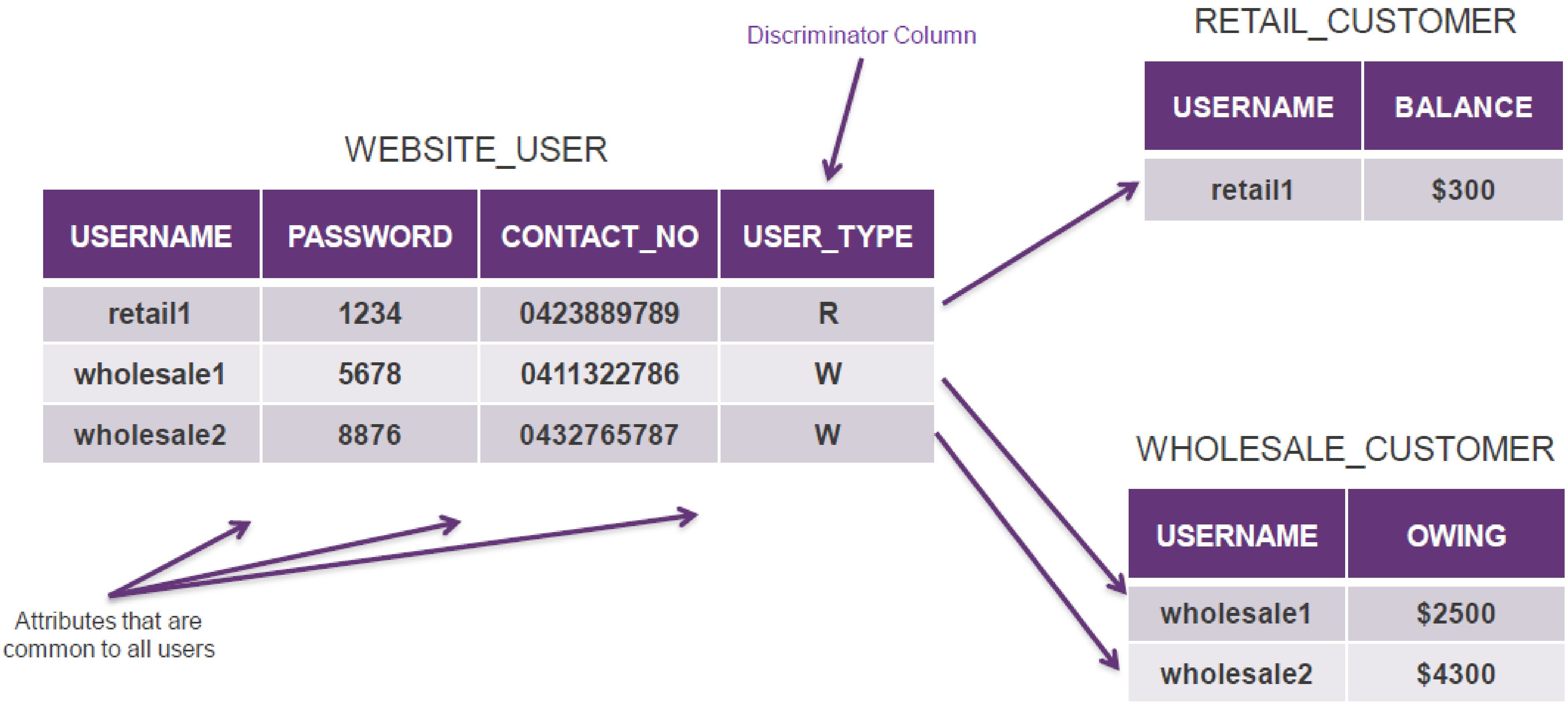
- Advantage:
  - Easy to use

- Disadvantage:
  - Since columns are not specific to a particular type of entity, a large number of NULL values are created.
  - For the same reason, the ability to enforce data integrity constraints is limited. E.g. enforcing the column ‘OWING’ as not null is not possible

Joined-Tables Strategy (1)

- Uses one-to-one relationships to model inheritance.
- Create separate tables for each entity in the inheritance hierarchy.
- Relating direct descendants in the hierarchy with one-to-one relationship

Joined-Tables Strategy (2)



Discriminator Column

WEBSITE_USER

USERNAME	PASSWORD	CONTACT_NO	USER_TYPE
retail1	1234	0423889789	R
wholesale1	5678	0411322786	W
wholesale2	8876	0432765787	W

Attributes that are common to all users

RETAIL_CUSTOMER

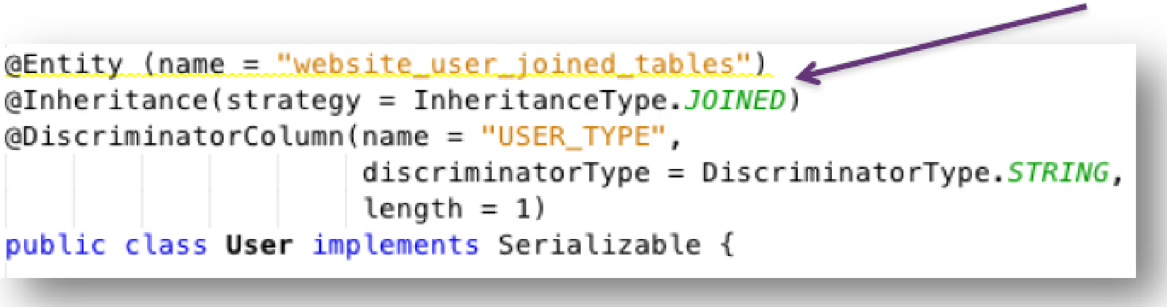
USERNAME	BALANCE
retail1	$300

WHOLESALE_CUSTOMER

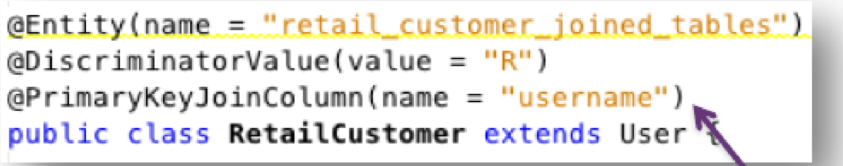
USERNAME	OWING
wholesale1	$2500
wholesale2	$4300

Joined-Tables Strategy (3)

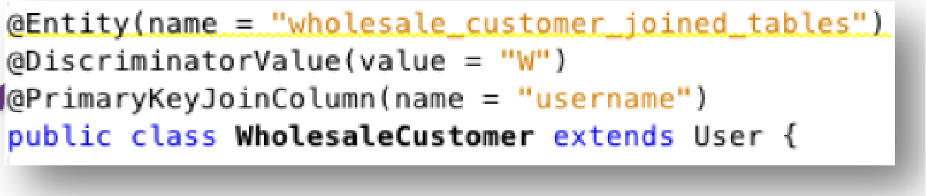
Specify joined tables strategy is used



```
@Entity (name = "website_user_joined_tables")
@Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn(name = "USER_TYPE",
                     discriminatorType = DiscriminatorType.STRING,
                     length = 1)
public class User implements Serializable {
```



```
@Entity(name = "retail_customer_joined_tables")
@DiscriminatorValue(value = "R")
@PrimaryKeyJoinColumn(name = "username")
public class RetailCustomer extends User {
```



```
@Entity(name = "wholesale_customer_joined_tables")
@DiscriminatorValue(value = "W")
@PrimaryKeyJoinColumn(name = "username")
public class WholesaleCustomer extends User {
```

Customize the name of the
primary key column

Joined-Tables Strategy (4)

- From a design perspective, many consider it as the best choice for mapping inheritance to relational database.
- From a performance perspective, it is an inferior choice compared to single table strategy because it requires the joining of multiple tables for loading entities from the subclasses.
- The deeper the inheritance hierarchy, the more severe the impact it has on performance.

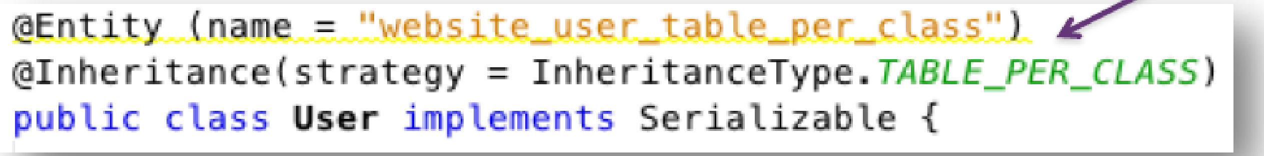
Table-Per-Class Strategy (1)

- Simplest inheritance mapping strategy.
- With this strategy, both parent and child classes are stored in their own table and no relationship exists between any of the tables.

Table-Per-Class Strategy (2)

WEBSITE_USER

USERNAME	PASSWORD	CONTACT_NO

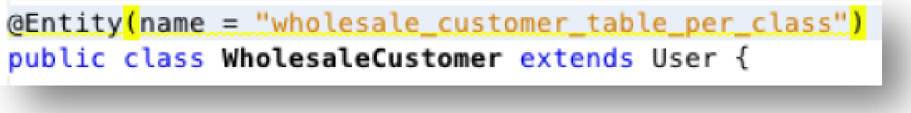


```
@Entity (name = "website_user_table_per_class")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class User implements Serializable {
```

Specify table per class strategy is used

WHOLESALE_CUSTOMER

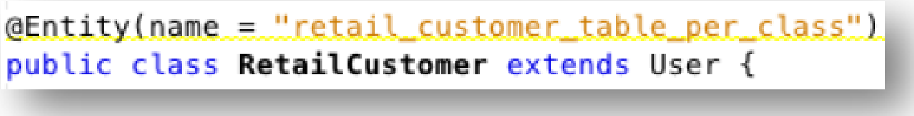
USERNAME	PASSWORD	CONTACT_NO	OWING
wholesale1	5678	0411322786	$2500
wholesale2	8876	0432765787	$4300



```
@Entity(name = "wholesale_customer_table_per_class")
public class WholesaleCustomer extends User {
```

RETAIL_CUSTOMER

USERNAME	PASSWORD	CONTACT_NO	BALANCE
retail1	1234	0423889789	$300



```
@Entity(name = "retail_customer_table_per_class")
public class RetailCustomer extends User {
```

Table-Per-Class Strategy (3)

- Worst from both a relational and OO standpoint.
- Does not have good support for polymorphic relations or queries as each subclass is mapped to its own table.
- E.g. When you want to retrieve entities over persistence provider, it must use SQL UNION to query all tables or retrieve each entity with separate SQL for each subclass in hierarchy.

Criteria API

Problems of JPQL

```
Query query = this.entityManager.createQuery(“SELECT c
FROM Company c WHERE c.name = :cname”);
query.setParameter(“:cname”, name);

return query.getResultList();
```

Problems:

– Easy to make syntax mistakes.

– Difficult to manipulate.

– Errors are discovered at runtime

Criteria API

- Allows developer to write queries using an object-oriented approach.
- Queries written using Criteria API are type-safe.
- Most of the errors are discovered at compile time.
- Support everything that JPQL can do.

Steps to Use Criteria API (1)

1. Obtain a CriteriaBuilder object. CriteriaBuilder is an interface that plays the role of a factory for all individual parts of a query.

```
CriteriaBuilder builder =
entityManager.getCriteriaBuilder();
```

2. Create a type-safe criteria query that stores the information about the tasks the query tries to achieve with a specified result type.

```
CriteriaQuery cQuery =
builder.createQuery(Employee.class);
```

This query returns entity/entities of type Employee

Steps to Use Criteria API (2)

3. Obtain a query root, which specifies the domain objects on which the query is evaluated.

```
Root<Employee> e = cQuery.from(Employee.class);
```

This query will evaluate Employee entity

4. Specify what would be returned as the result of the query.

```
cQuery.select(e);
```

Return the domain object specified in the root, Employee in this case. You can also specify to return the values of certain attributes here.

Steps to Use Criteria API (3)

5. Construct the criteria for filtering entity instance as needed (optional).

```
Predicate predicate =
builder.equal(e.get(“employeeName”).as(String.class),
               “Eddie Leung”);
```

Return only the employees whose name is equal to ‘Eddie Leung’

6. Store the criteria in the criteria query.

```
cQuery.where(predicate);
```

Steps to Use Criteria API (4)

7. Create a typed query, which is a type-safe query for EntityManager

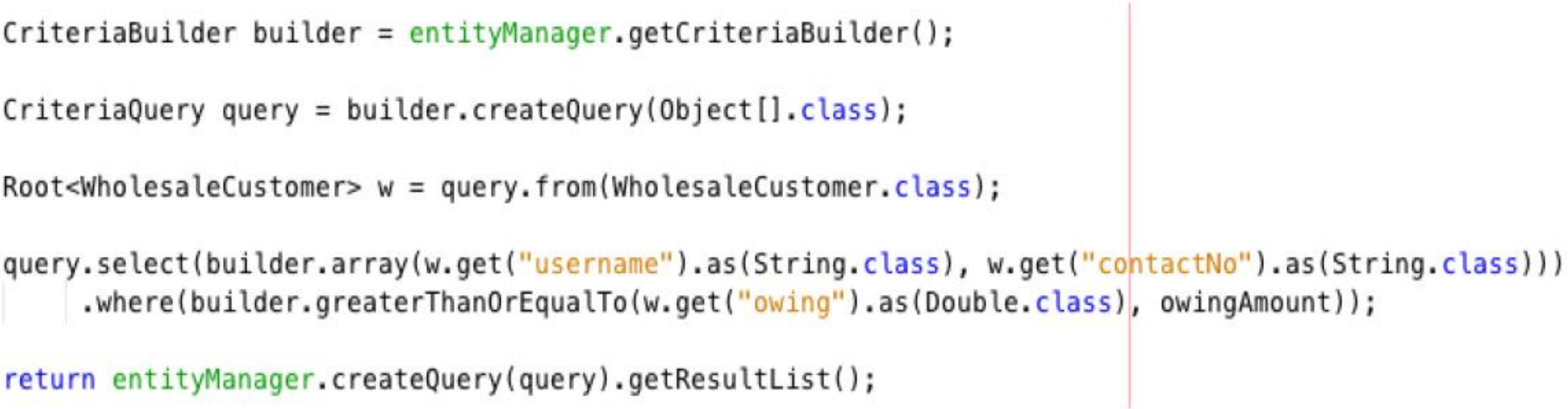
```
TypedQuery tQuery = entityManager.createQuery(cQuery);
```

8. Query the underlying database, and return the results to the caller.

```
return tQuery.getResultList();
```

Sample Queries

- Return the full name and phone number of the wholesale customers who owes more than a particular amount of money.



```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery query = builder.createQuery(Object[].class);

Root<WholesaleCustomer> w = query.from(WholesaleCustomer.class);

query.select(builder.array(w.get("username").as(String.class), w.get("contactNo").as(String.class)))
        .where(builder.greaterThanOrEqualTo(w.get("owing").as(Double.class), owingAmount));

return entityManager.createQuery(query).getResultList();
```

MONASH University

Container versus Application Managed Persistence

Container Managed Entity Manager

- Container is responsible for creating and closing the entity manager.
- Transaction management is handled by container.
- It is obtained by using `@PersistenceContext`

Container Managed VS Application Managed (1)

Container Managed

- Created and injected to an application by container
- Container is responsible for creating and closing the entity manager.
- Transaction management is handled by container.

Application Managed

- Created and instantiated by developer
- Developer is responsible for creating and closing the entity manager.
- Transaction management is handled by developer.

Container Managed VS Application Managed (2)

- Obtaining an EntityManager
  - Application managed EntityManager
    Via **EntityManagerFactory**

```
entityManagerFactory  =
Persistence.createEntityManagerFactory(“Lecture7ExamplePU”);
entityManager = entityManagerFactory.createEntityManager();
```

  - Container managed EntityManager
    It is obtained by using **@PersistenceContext**

Ask container to inject an entity manager

Specify the name of the persistence unit

```
@PersistenceContext (unitName = “Lecture7ExamplePU”);
Private EntityManager entityManager;
```

Container Managed VS Application Managed (3)

- Persisting an entity
  - Application managed EntityManager

```
public void addBranch(Branch branch) throws Exception {
    EntityTransaction transaction = entityManager.getTransaction();
    try {
        transaction.begin();
        entityManager.persist(branch);
        transaction.commit();
    } catch (Exception ex) {
        transaction.rollback();
    }
}
```

  - Container managed EntityManager

```
public void addBranch(Branch branch) throws Exception {
    entityManager.persist(branch);
}
```

Container Managed VS Application Managed (4)

- Updating an entity
  - Application managed EntityManager

```
public void editBranch(Branch branch) throws Exception {
    EntityTransaction transaction = entityManager.getTransaction();
    try {
        transaction.begin();
        entityManager.merge(branch);
        transaction.commit();
    } catch (Exception ex) {
        transaction.rollback();
    }
}
```

  - Container managed EntityManager

```
public void editBranch(Branch branch) throws Exception {
    entityManager.merge(branch);
}
```

Container Managed VS Application Managed (5)

- Removing an entity
  - Application managed EntityManager

```java
public Branch removeBranch(Branch branch) throws Exception {
    EntityTransaction transaction = entityManager.getTransaction();
    try {
        transaction.begin();
        if (branch != null) { entityManager.remove(branch); }
        transaction.commit();
    } catch (Exception ex) {
        transaction.rollback();
    }
}
```

  - Container managed EntityManager

```java
public void removeBranch(Branch branch) throws Exception {
    if (branch != null) { entityManager.remove(branch); }
}
```

Container Managed VS Application Managed (6)

- Retrieving an entity by its ID
  - Application managed EntityManager

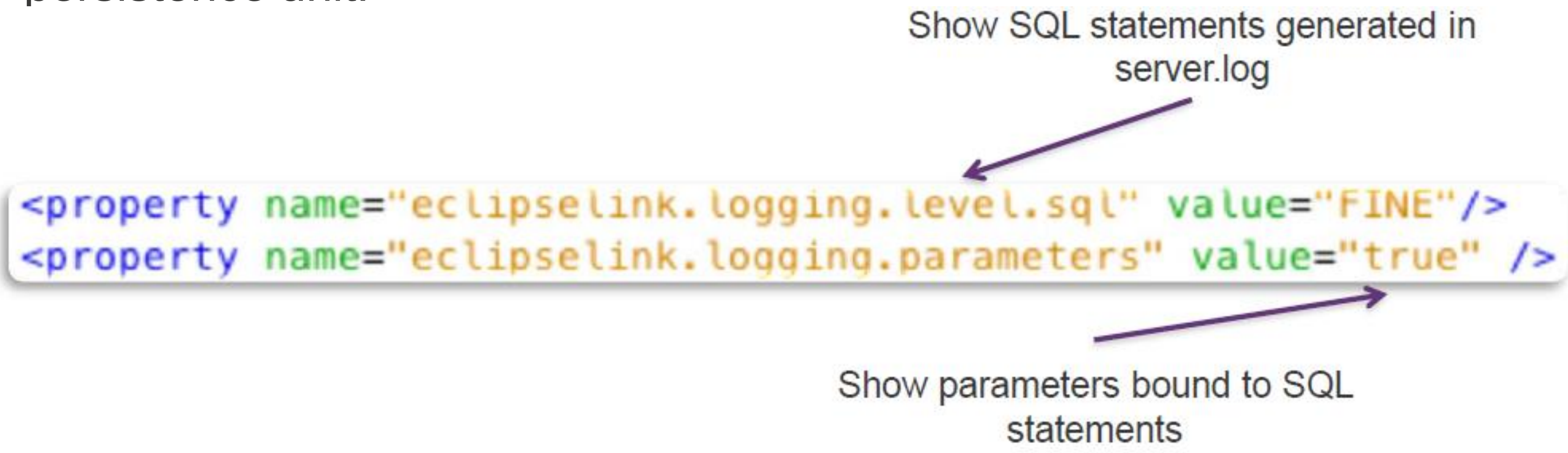
```
public void searchBranchbyId(int id) throws Exception {
    return entityManager.find(Branch.class, id);
}
```

  - Container managed EntityManager

```
public void searchBranchbyId(int id) throws Exception {
    return entityManager.find(Branch.class, id);
}
```

Debugging

- One of the primary goals of JPA is to spare Java programmers from SQL.
- It transfers persistence operation requests created in Java and generate the necessary SQL statements.
- The actual SQL generated is provider dependent hence it is difficult to know what exactly happens behind the scene.
- However, sometimes it is important to know what the SQL in order to debug.
- Can show the SQL generated by adding the two properties below to the persistence unit:



Show SQL statements generated in server.log

```
<property name="eclipselink.logging.level.sql" value="FINE"/>
<property name="eclipselink.logging.parameters" value="true" />
```

Show parameters bound to SQL statements

Summary

- More advanced ORM
- Criteria API
- Container Managed Entity Manager

Next Lecture

- Introduction to Java Enterprise Beans

See you in the Studio !

Readings

- ***Recommended:*** Chapter 6: Managing Persistence Objects in *Beginning Java EE 7, Antonio Goncalves*