


Java

JavaOne℠

ORACLE

# A Java™ Persistence API Mapping Magical Mystery Tour

Mike Keith

Oracle Corporation

michael.keith@oracle.com

# Agenda

- Introduction
- Basic Mappings
- Relationship Mappings
- Using Different Collection Types
- Embedded Mappings
- Derived Identifier Mappings
- Inheritance Mappings
- Summary

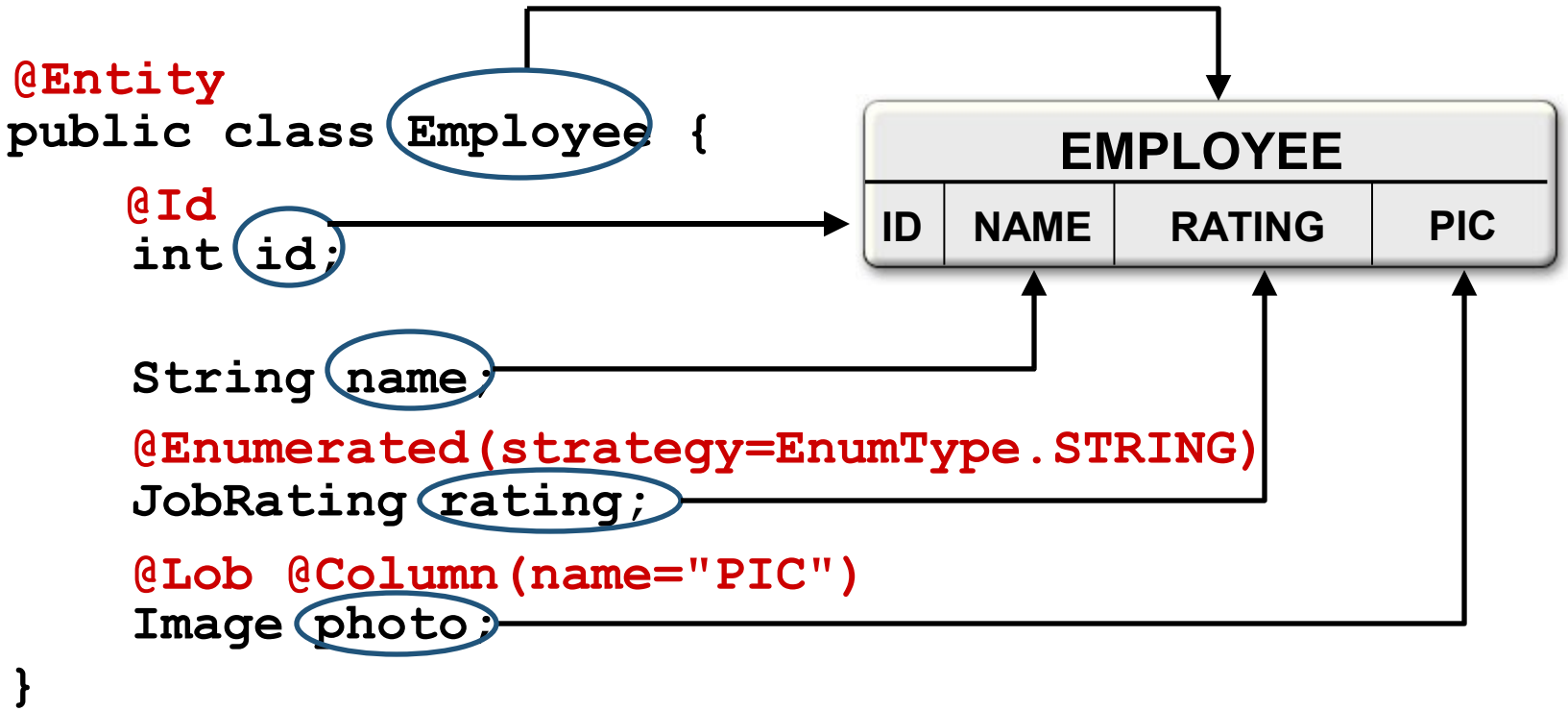
# Object-Relational Mapping

- Map state in Java objects to database columns
- Map relationships between entities to foreign keys
- Use defaulting rules whenever possible
- Mapping metadata as annotations and/or XML
  - Logical level metadata – describes object model
  - Physical level metadata – describes data model
- Facilitate different usage scenarios
  - Existing “legacy” schemas, or schema generation

# Basic Mappings

- Map attribute state to a single database column
- Default mapping when none is specified
- May optionally specify @Basic annotation
- Simple Java types (primitives, wrappers)
- Can override column defaults using @Column
- Additional annotations for specific types
  - @Enumerated – specify enum mapping
  - @Lob – treat state as BLOB/CLOB
  - @Temporal – indicate temporal granularity

# Basic Mappings

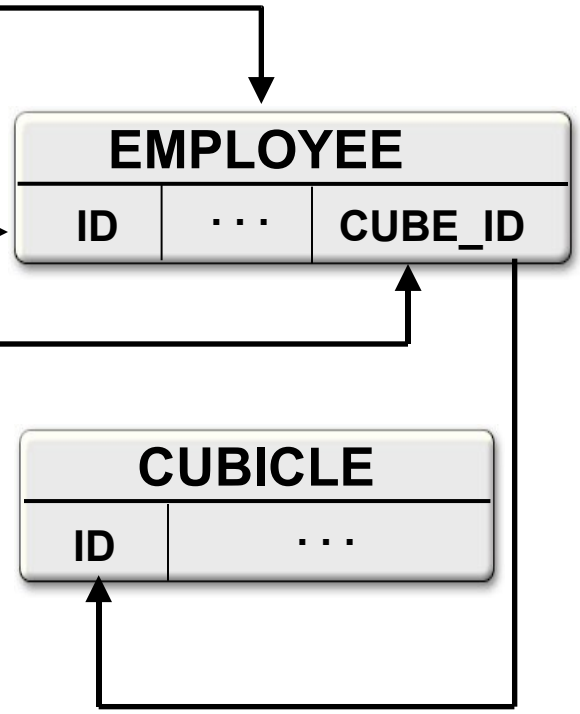


# Relationship Mappings

- Relationship – an entity refers to other entities
- Each relationship has direction – **source**, **target**
- Each side has cardinality – **one**, **many**
- Cardinalities of the source and target entities determine what type of relationship it is
- Common relationships:
  - ManyToOne, OneToOne – reference to one entity
  - OneToMany, ManyToMany – reference  a collection

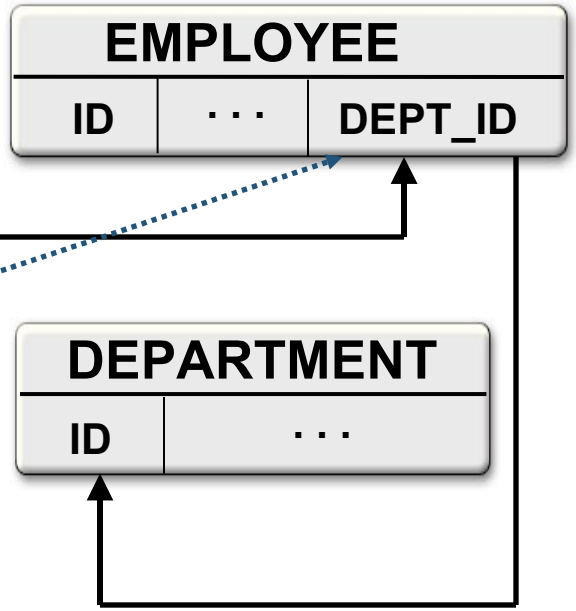
# OneToOne Mapping

```
@Entity
public class Employee {
    @Id
    String id;
    ...
    @OneToOne
    Cubicle cube;
}
```



# ManyToOne Mapping

```
@Entity
public class Employee {
    @Id
    String id;
    ...
    @ManyToOne
    @JoinColumn(name="DEPT_ID")
    Department department;
}
```

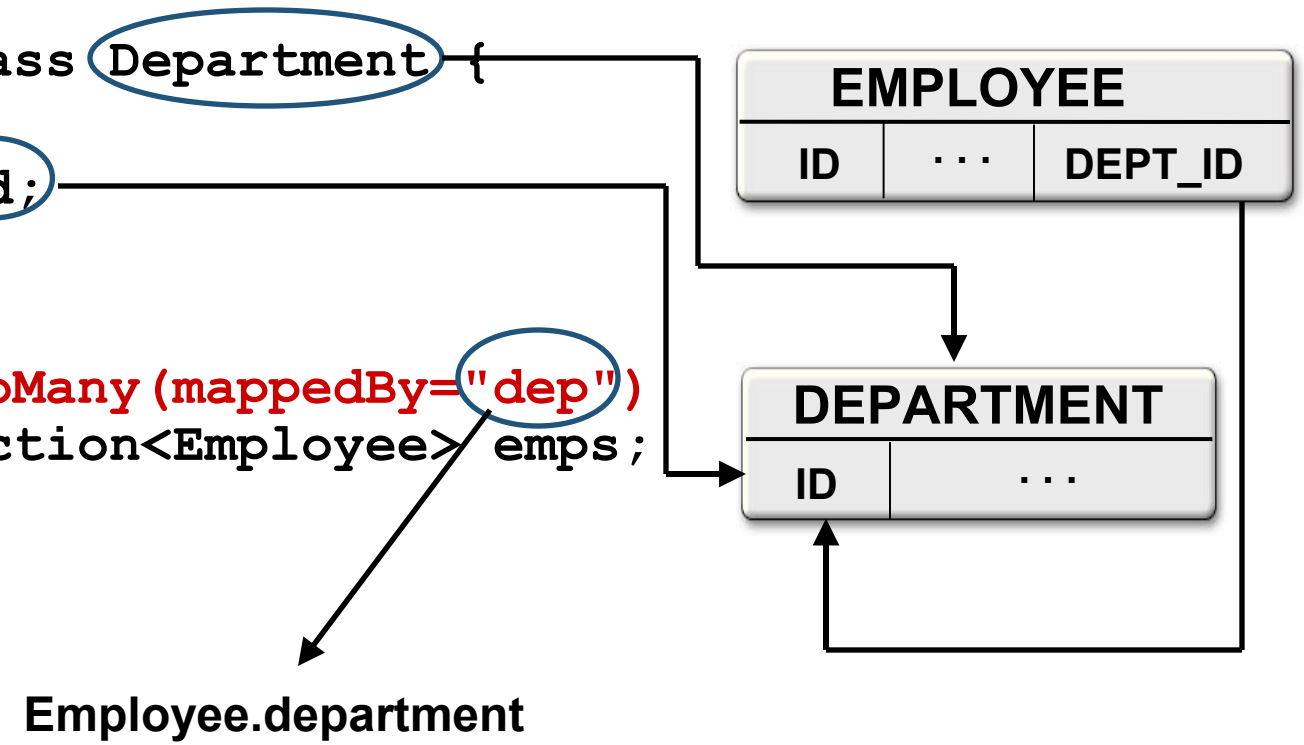


# Bidirectional Relationships

- Each side has a relationship to the other side
- Both sides managed by a single foreign key
- Owning side
  - Owns/maps physical level metadata (foreign key)
- Inverse (non-owning) side
  - Points to attribute in owning side for mapping
- Must be properly managed by the client in memory
  - Relationship consistency, caching

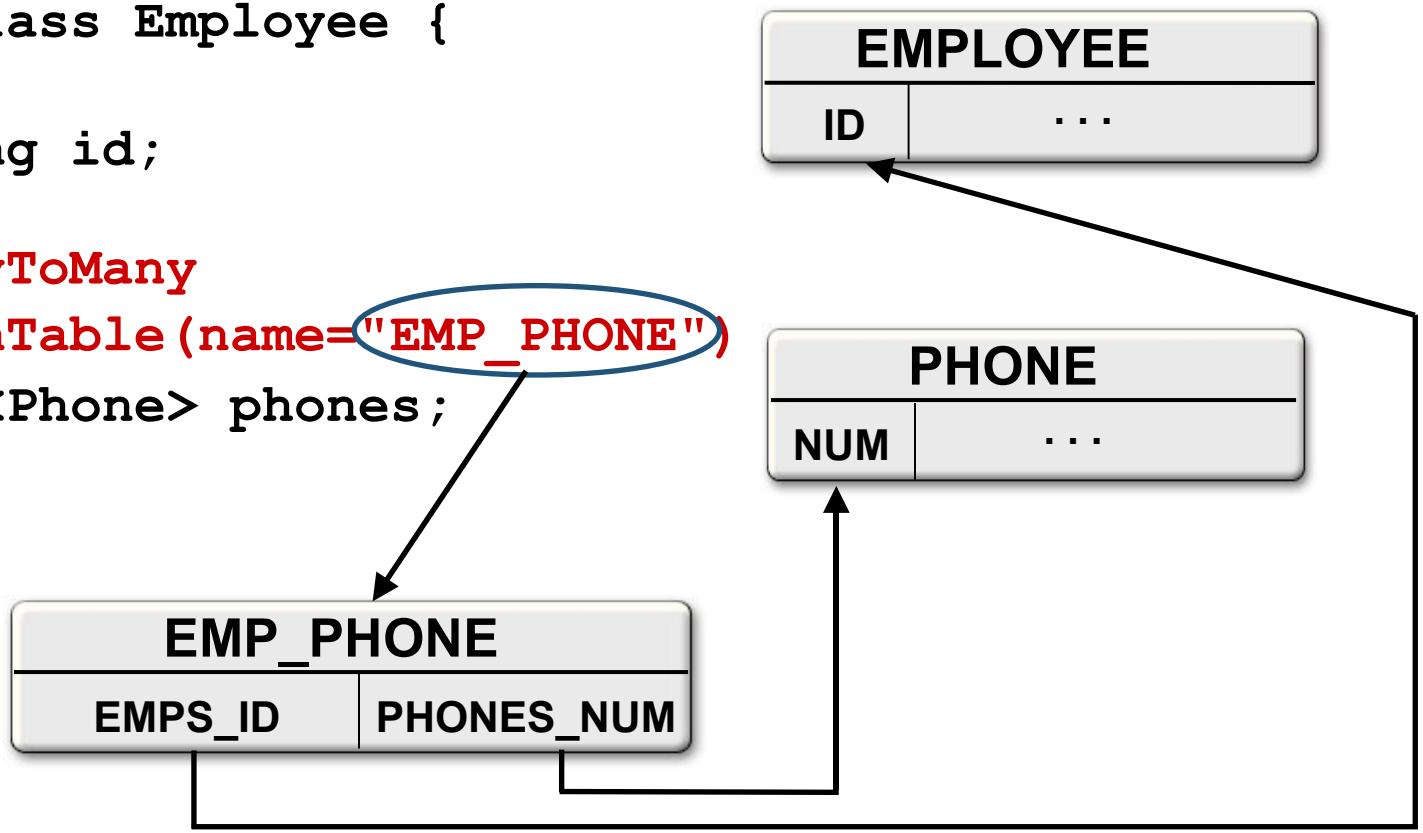
# OneToMany Mapping

```
@Entity
public class Department {
    @Id
    int id;

    ...
    @OneToMany(mappedBy="dep")
    Collection<Employee> emps;
}
```



# ManyToMany Mapping

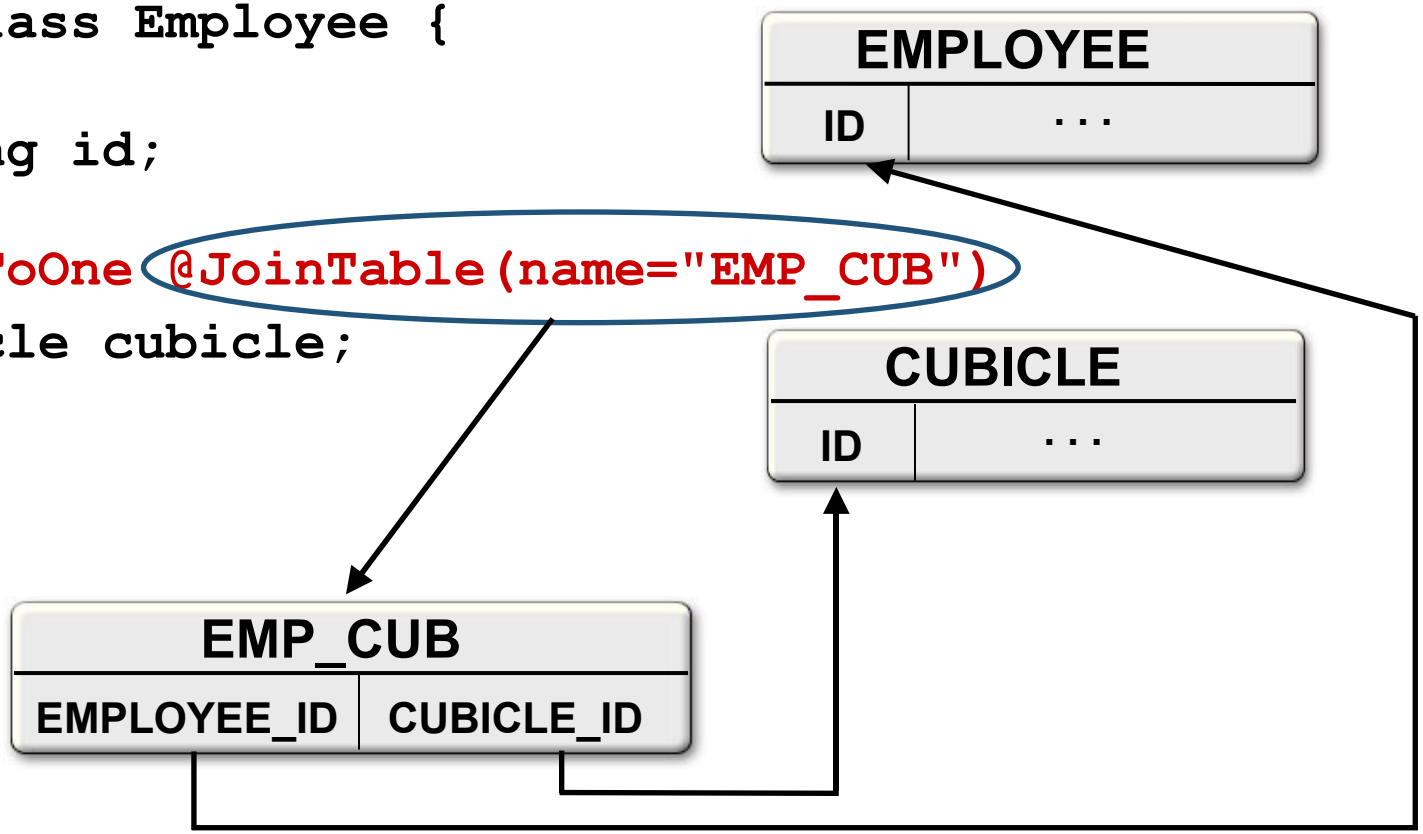
```
@Entity
public class Employee {
    @Id
    String id;
    ...
    @ManyToMany
    @JoinTable(name="EMP_PHONE")
    List<Phone> phones;
}
```



# OneToOne Mapping

## Join Table

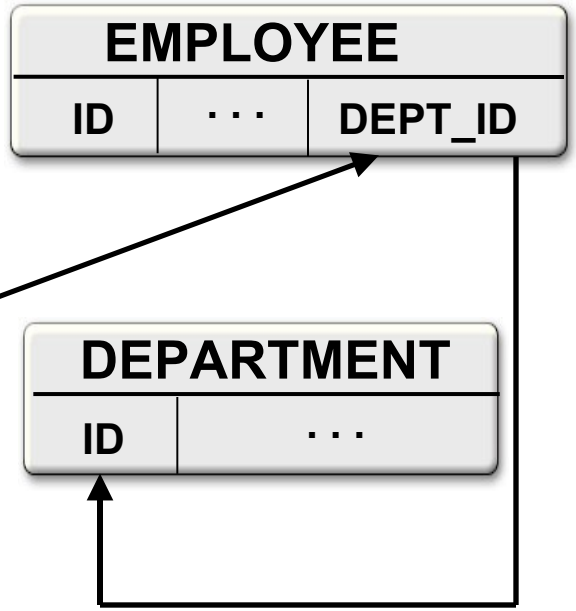
```
@Entity
public class Employee {
    @Id
    String id;
    ...
    @OneToOne @JoinTable(name="EMP_CUB")
    Cubicle cubicle;
}
```



# Unidirectional OneToMany Mapping

## Target Foreign Key

```
@Entity
public class Department {
    @Id
    int id;

    ...
    @OneToMany
    @JoinColumn(name="DEPT_ID")
    Collection<Employee> emps;
}
```



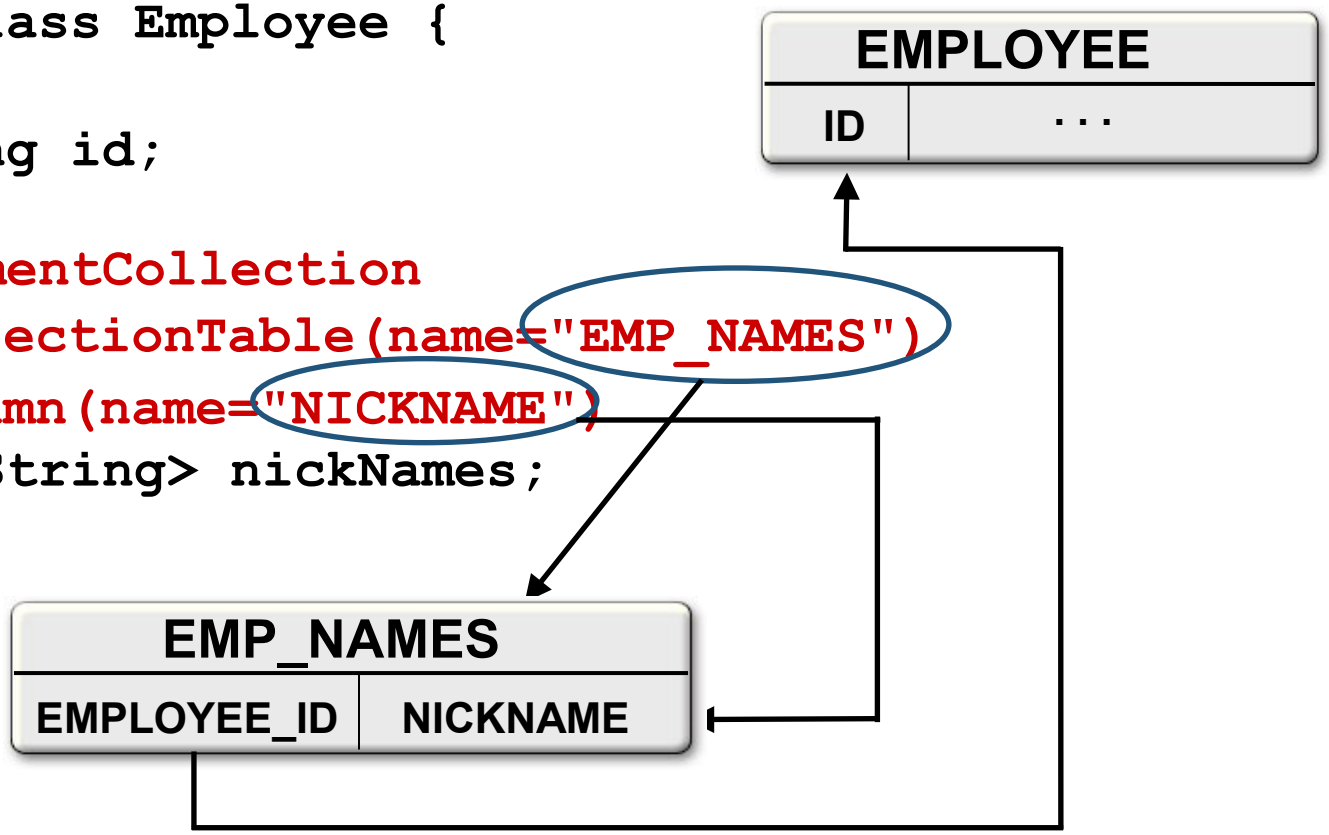
# Using Collections

- Can use one of defined Collection types:
  - Collection, Set, List, Map
- Collection of entities (OneToMany, ManyToMany)
  - Foreign keys stored in target table or join table
- Collection of simple objects or embedded types
  - Foreign keys and values stored in collection table
- Lists may have element/entity order persisted
- Maps may be keyed on basic/embeddable/entity
- Additional metadata may need to be mapped (depending upon the type of Collection used)

# Element Collection Mapping

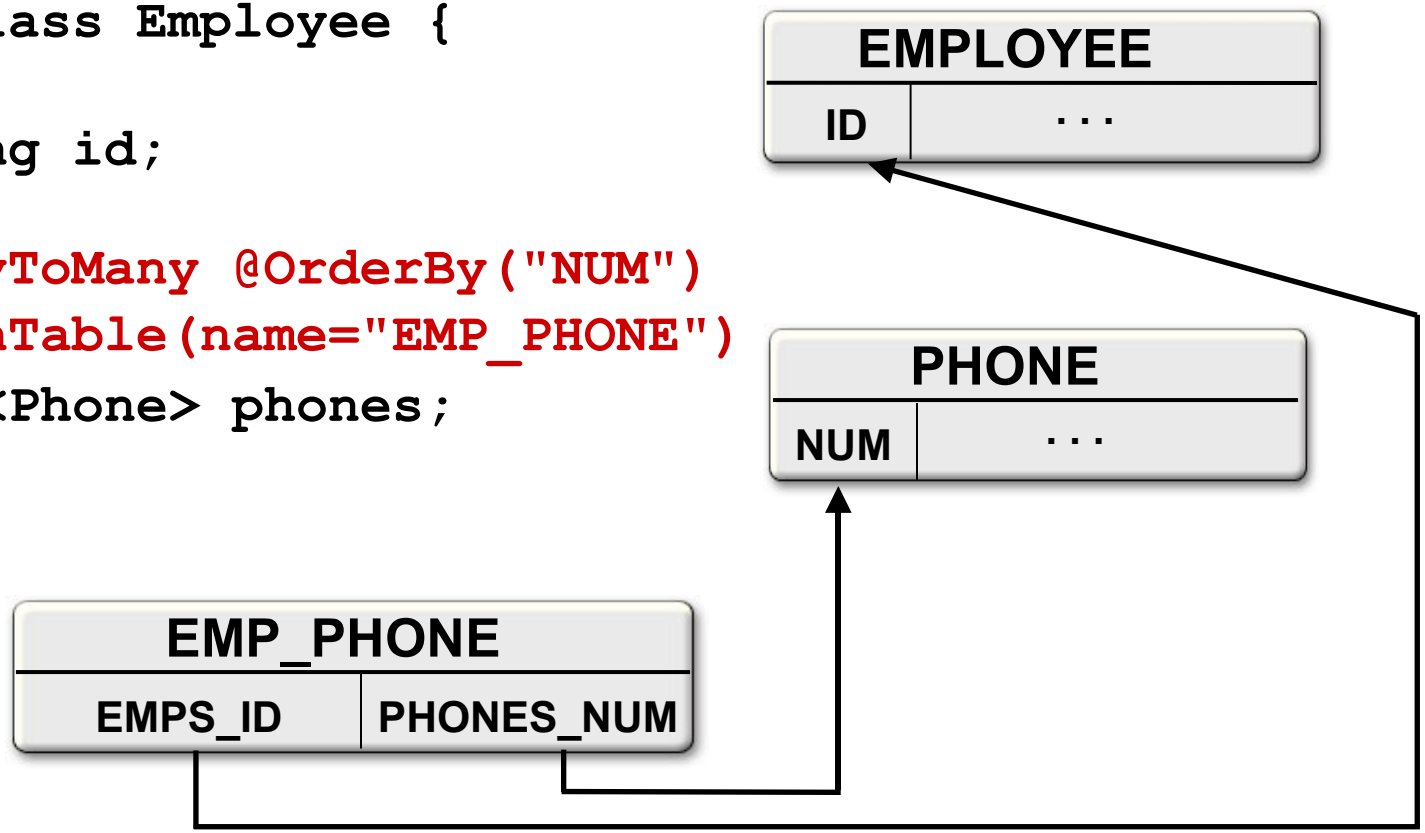
## Basic Objects

```
@Entity
public class Employee {
    @Id
    String id;
    ...
    @ElementCollection
    @CollectionTable(name="EMP_NAMES")
    @Column(name="NICKNAME")
    Set<String> nickNames;
}
```



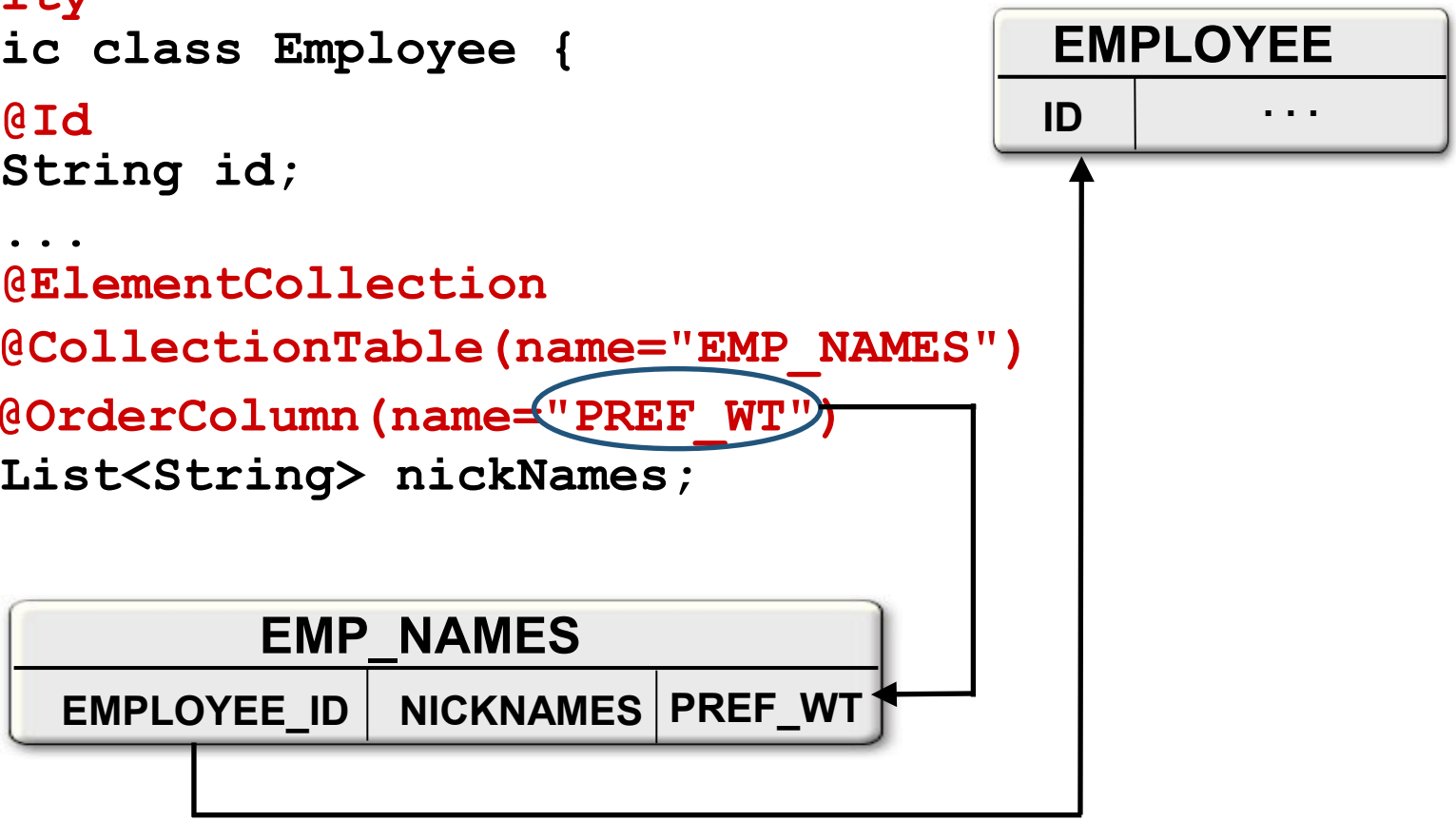
# Using a List
## Relationships

```
@Entity
public class Employee {
    @Id
    String id;
    ...
    @ManyToMany @OrderBy("NUM")
    @JoinTable(name="EMP_PHONE")
    List<Phone> phones;
}
```



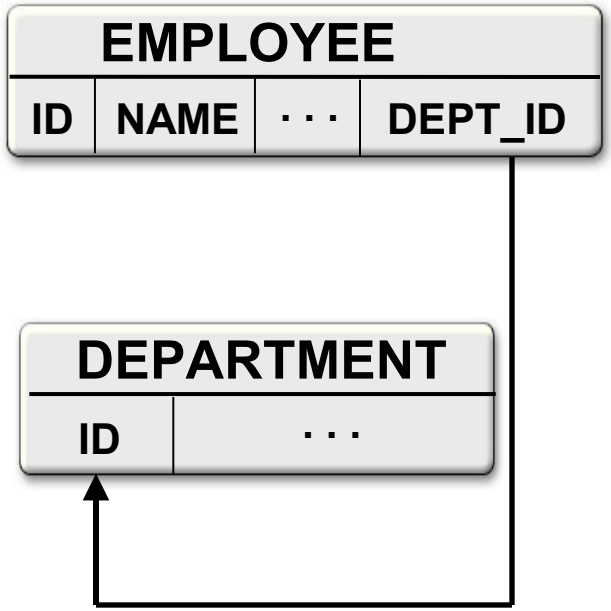
# Using a List
## Persistent Ordering

```
@Entity
public class Employee {
    @Id
    String id;
    ...
    @ElementCollection
    @CollectionTable(name="EMP_NAMES")
    @OrderColumn(name="PREF_WT")
    List<String> nickNames;
}
```



# Using a Map
## Relationships

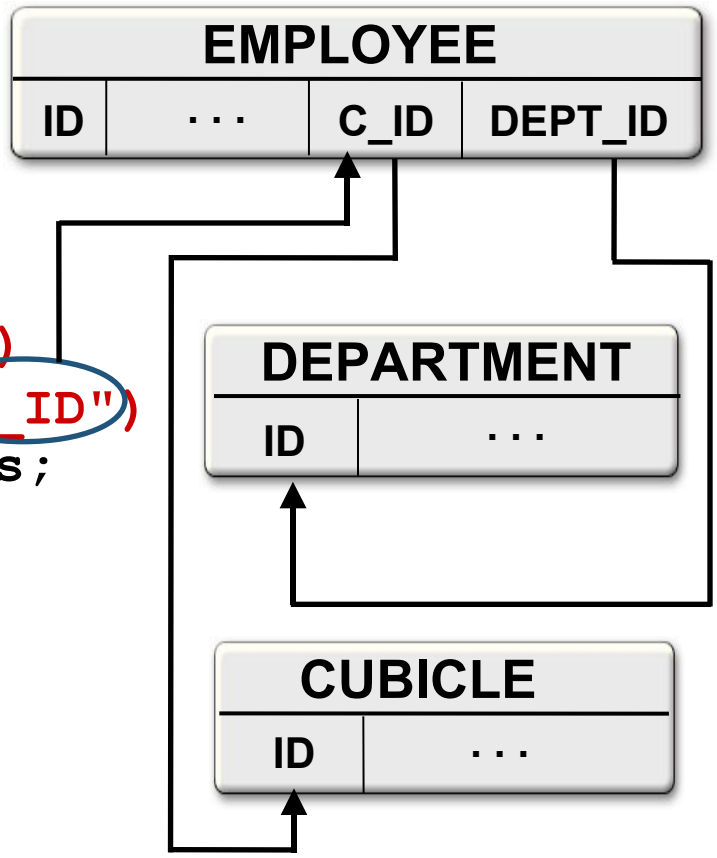
```
@Entity
public class Department {
     @Id
     int id;

     ...
     @OneToMany(mappedBy="dep")
     @MapKey(name="name")
     Map<String, Employee> emps;
}
```




Employee.name

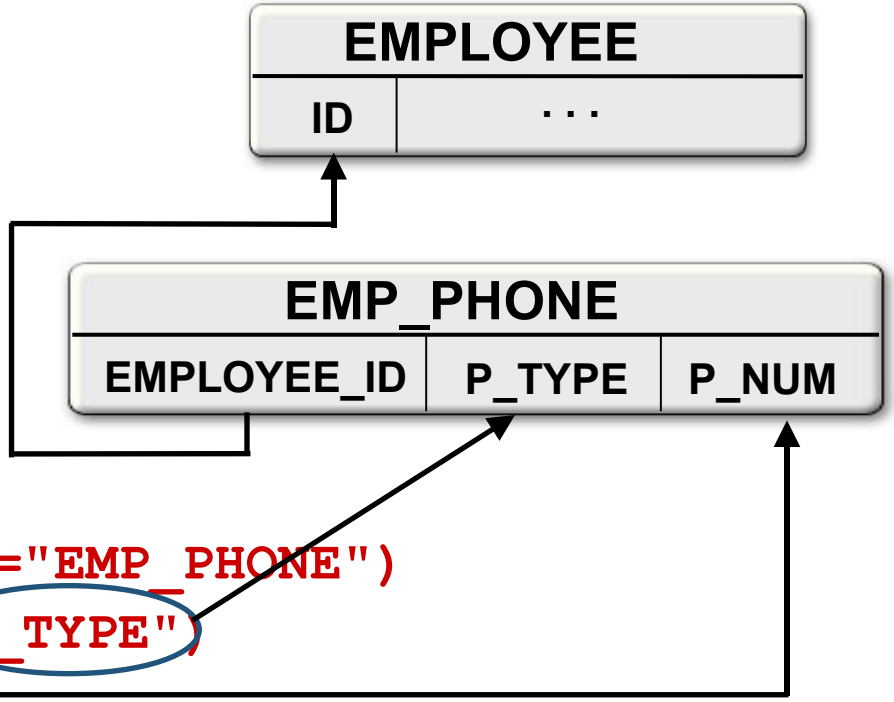
# Using a Map
## Entity Keys

```
@Entity
public class Department {
     @Id
     int id;

     ...
     @OneToMany(mappedBy="dep")
     @MapKeyJoinColumn(name="C_ID")
     Map<Cubicle,Employee> emps;
}
```



# Using a Map
## Simple Keys and Values

```
@Entity
public class Employee {
    @Id
    String id;
    ...

    @ElementCollection
    @CollectionTable(name="EMP_PHONE")
    @MapKeyColumn(name="P_TYPE")
    @Column(name="P_NUM")
    Map<String,String> contactNumbers;
}
```



# Embeddable Objects

- Subgroup of state stored in separate Java object
- Indicated using @Embedded annotation
- May embed basic attributes, embeddables, relationships, or element collections
- Embeddable type may be reused across entities
- Embedded state is mapped in embeddable object
- Mappings can be overridden using @AttributeOverride and @AssociationOverride

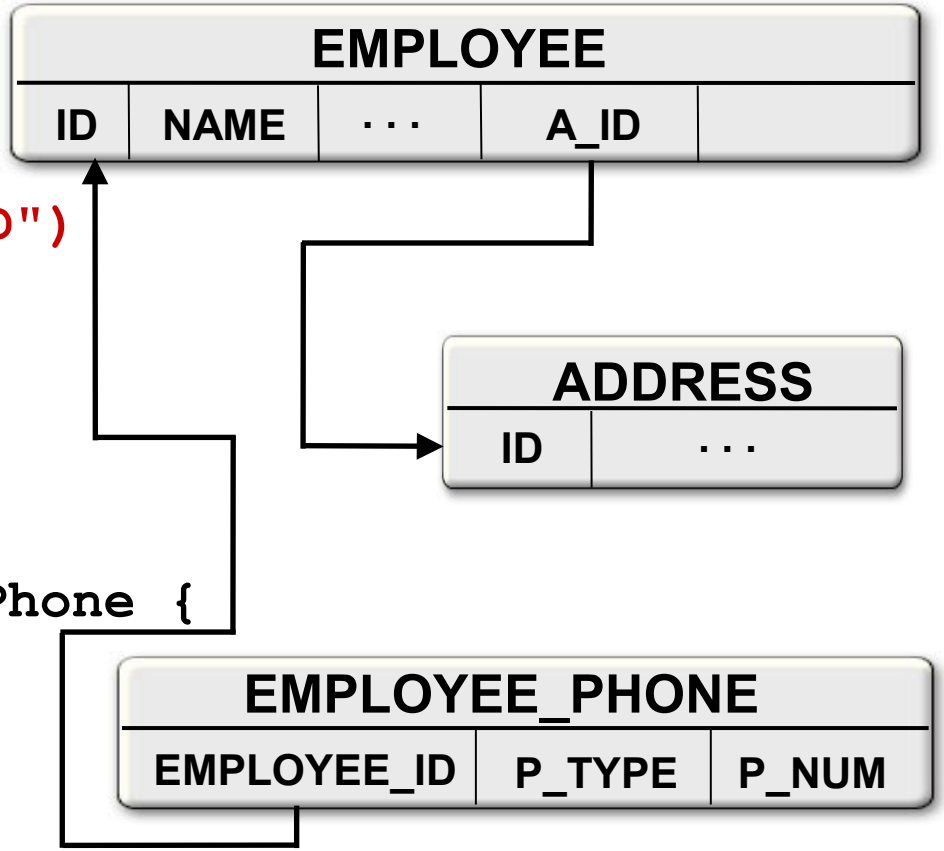
# Embedded Mapping

```
@Entity
public class Employee {
    @Id
    String id;
    ...
    @Embedded
    @AttributeOverride
      (name="birthDate",column=@Column(name="BDAY"))
    EmployeeInfo info;
}

@Embeddable public class EmployeeInfo {
    String name;
    Integer age;
    @Temporal(DATE) Date birthDate;
}
```

# Embedded Mapping

## Nested Relationships and Collections

```
@Embeddable
public class EmployeeInfo {
     String name;
     ...
     @ManyToOne
     @JoinColumn(name="A_ID")
     Address addr;

     @ElementCollection
     List<Phone> phones;
}
@Embeddable public class Phone {
     String p_type;
     String p_num;
}
```



# Identifiers

- Identifiers uniquely distinguish entity instances
- Compound identifier – id with multiple attributes
  - Need PK class to encapsulate id attributes
  - Two flavors:
    - @IdClass – All id attributes are mapped in entity class
    - @EmbeddedId – Id attributes are mapped in embeddable
- Derived identifier – id depends on related entity
  - Multitude of scenarios depending upon flavors used
  - Avoid duplication of id state and mapping info

# Compound Identifiers
## Id Class

```
@Entity @IdClass(EmployeePK.class)
public class Employee {
    @Id String name;
    @Id @Temporal(DATE)
    Date birthDate;
    ...
}

public class EmployeePK {
    String name;
    Date birthDate;

    // … equals(), hashCode() methods, etc.
}
```

# Compound Identifiers
## Embedded Id

```
@Entity
public class Employee {
    @EmbeddedId EmployeePK id;
    ...
}

@Embeddable
public class EmployeePK {
    String name;
    @Temporal(DATE)
    Date birthDate;

    // … equals(), hashCode() methods, etc.
}
```

# Derived Identifiers

## Old Way

```
@Entity @IdClass(EmployeePK.class)
public class Employee {
    @Id String name;
    @Id @Column(name="C_ID",
                insertable=false, updatable=false)
    int cube_id;

    @OneToOne
    @JoinColumn(name="C_ID")
    Cubicle cube;
    ...
}
public class EmployeePK {
    String name;
    int cube_id;
}
```

Duplicate column mapping

Additional unnecessary state

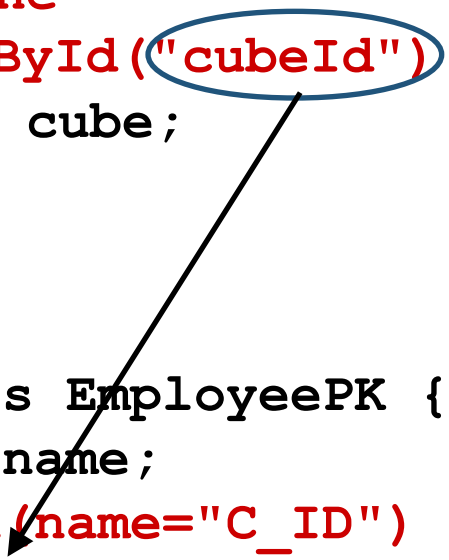
# Derived Identifiers

## Id Class

```
@Entity @IdClass(EmployeePK.class)
public class Employee {
    @Id String name;

    @OneToOne @Id
    @JoinColumn(name="C_ID")
    Cubicle cube;
    ...
}

public class EmployeePK {
    String name;
    int cube;
}
```

# Derived Identifiers
## Embedded Id

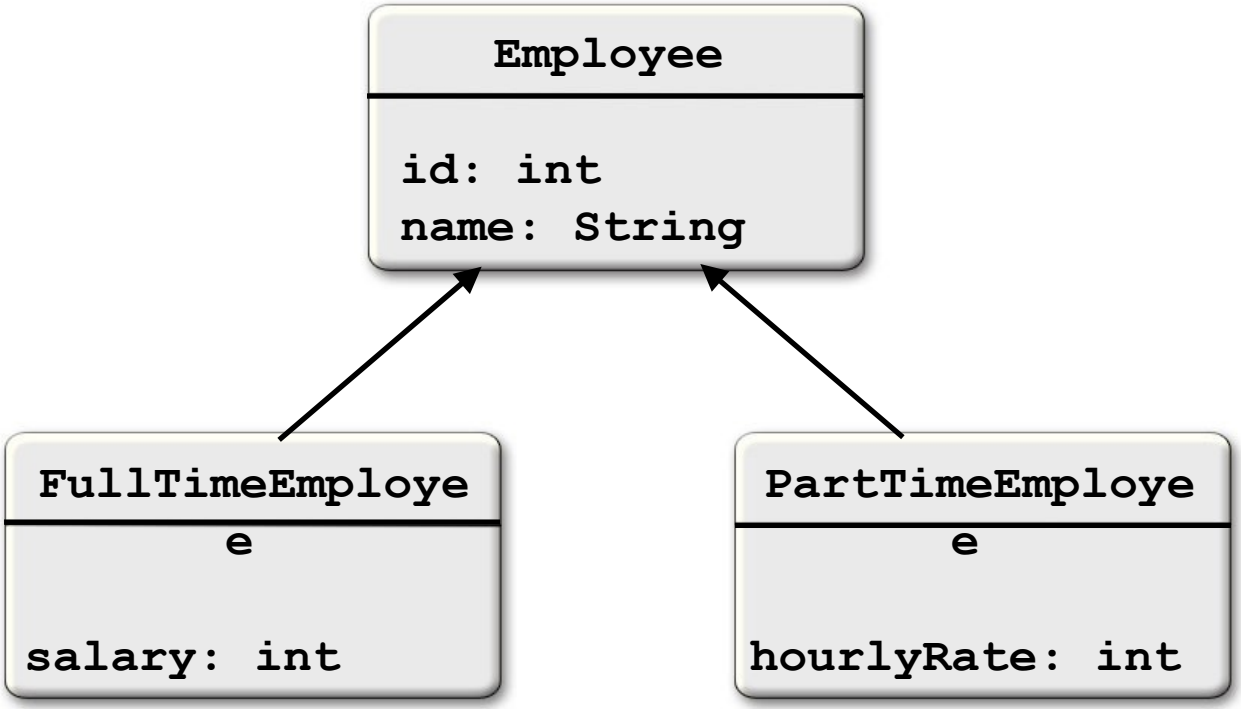
```
@Entity
public class Employee {
    @EmbeddedId EmployeePK id;

    @OneToOne
    @MappedById("cubeId")
    Cubicle cube;
    ...
}
@Embeddable
public class EmployeePK {
    String name;
    @Column(name="C_ID")
    int cubeId;
}
```



# Inheritance

- Inheritance an important part of OO programming
- Entity classes can extend:
  - Entity classes (concrete or abstract)
  - Mapped superclasses (concrete or abstract)
  - Non-managed classes (concrete or abstract)
- Three strategies for mapping inheritance:
  1) Single Table
  2) Joined
  3) Table per Class (optional)

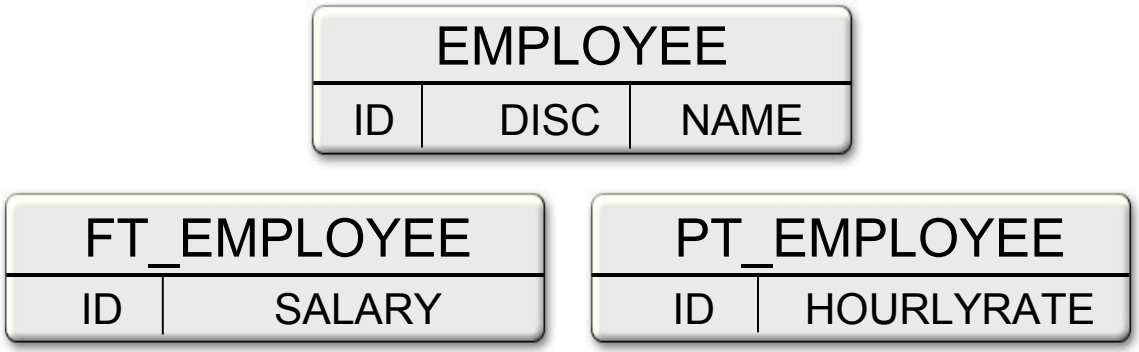
# Inheritance
## Object Model



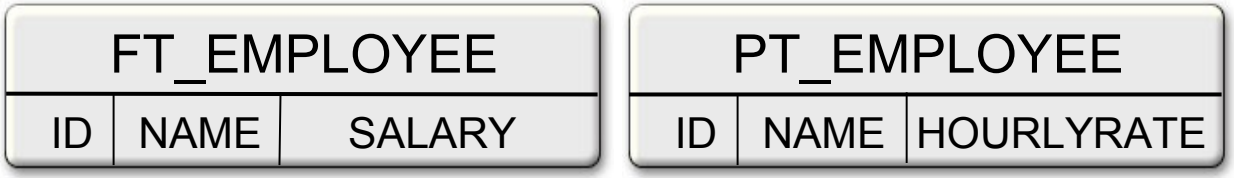
# Data Models

- Single table:


- Joined:




- Table per Class:



# Summary

- Anti-Mallory principle:

> **Just because it's there**
> **doesn't mean you should be using it.**

- Many more "corner-case" mappings
- Mapping to a "legacy" database just got easier
- Existing JPA 1.0 mappings still the most useful
- Lots more to see besides mappings!


# Thank You

Mike Keith
michael.keith@oracle.com