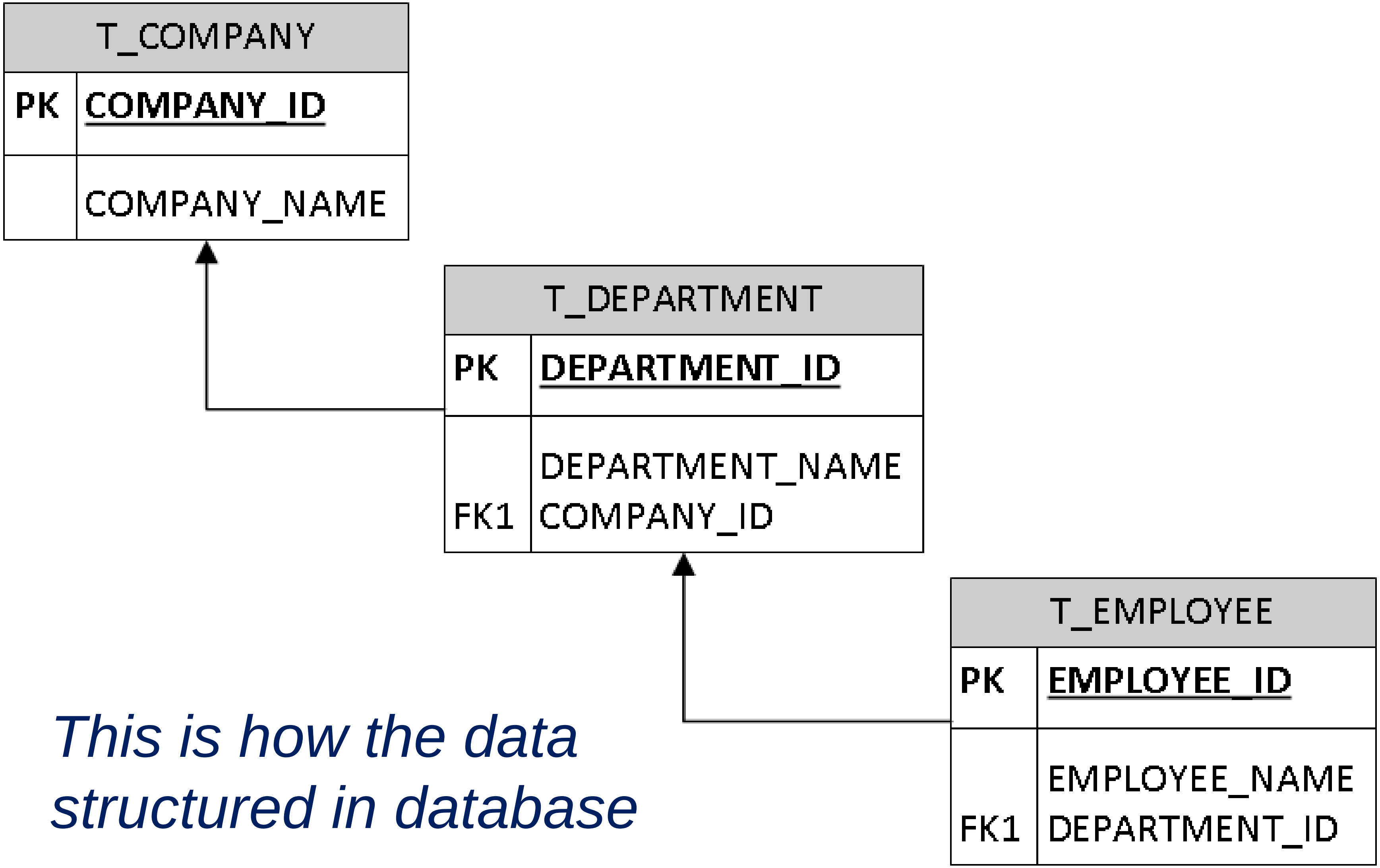
JVA-000
Java Persistence with Hibernate

Module 2
Entities

Objectives

- Understand the Entity term
- Learn how to define entity using JPA annotations
- Learn how to map entity to database objects

Database to POJO mapping using JPA



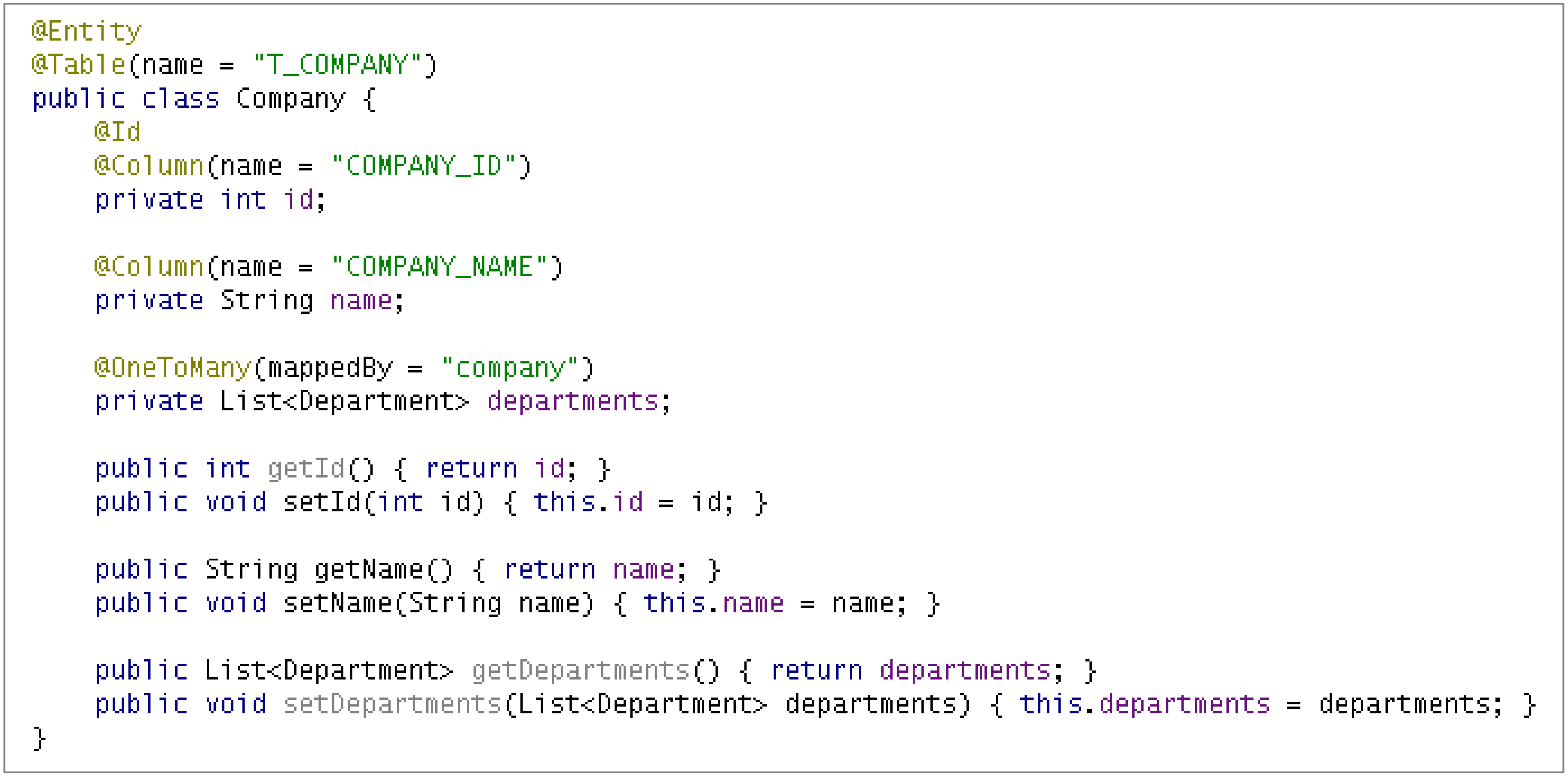
T_COMPANY	
PK	**COMPANY_ID**
	COMPANY_NAME

T_DEPARTMENT	
PK	**DEPARTMENT_ID**
 FK1	DEPARTMENT_NAME COMPANY_ID

T_EMPLOYEE	
PK	**EMPLOYEE_ID**
 FK1	EMPLOYEE_NAME DEPARTMENT_ID

This is how the data structured in database

Database to POJO mapping using JPA

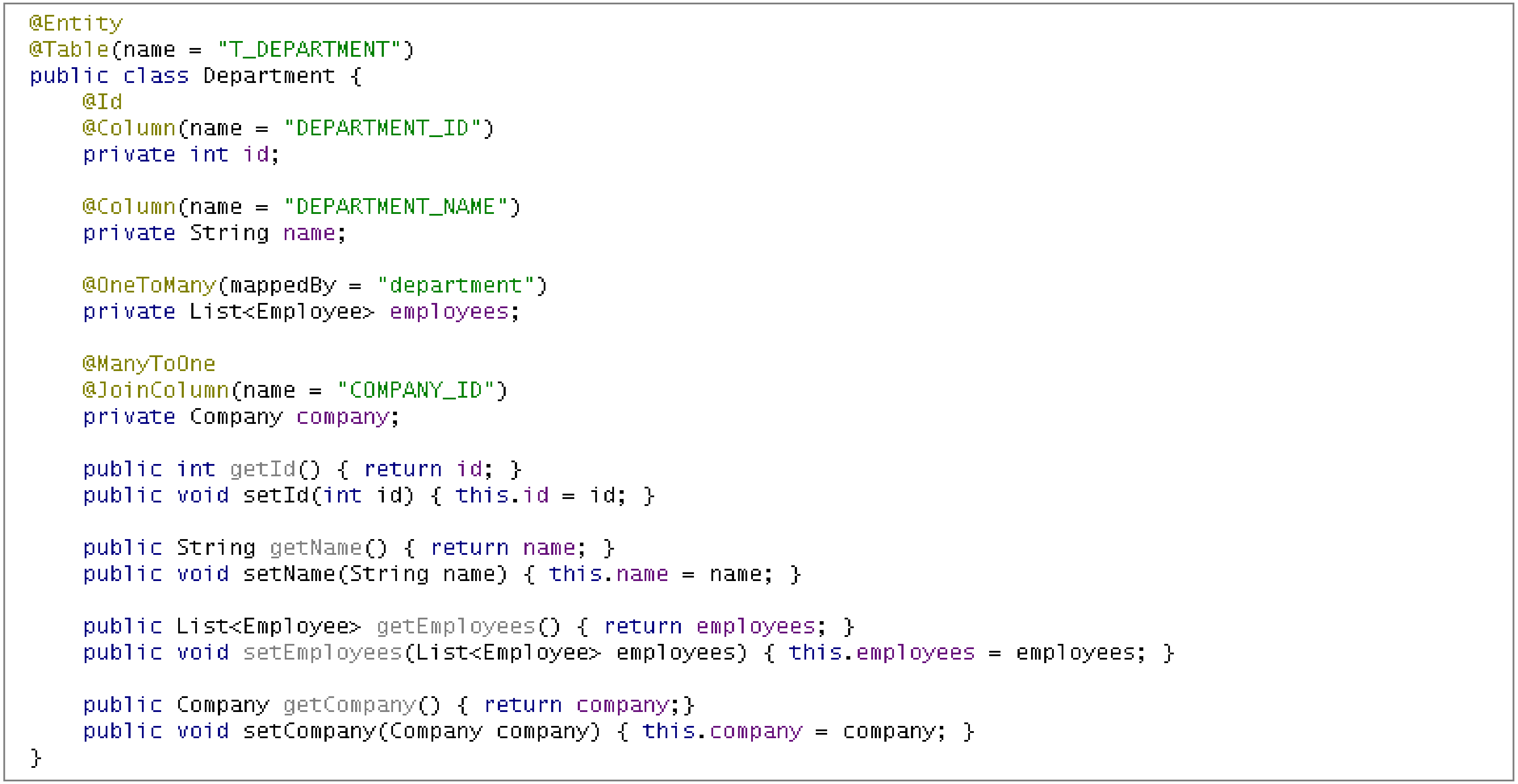


```
@Entity
@Table(name = "T_COMPANY")
public class Company {
    @Id
    @Column(name = "COMPANY_ID")
    private int id;

    @Column(name = "COMPANY_NAME")
    private String name;

    @OneToMany(mappedBy = "company")
    private List<Department> departments;

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public List<Department> getDepartments() { return departments; }
    public void setDepartments(List<Department> departments) { this.departments = departments; }
}
```

Class describing the Company entity in terms of JPA.

Maps to T_COMPANY table.

Database to POJO mapping using JPA

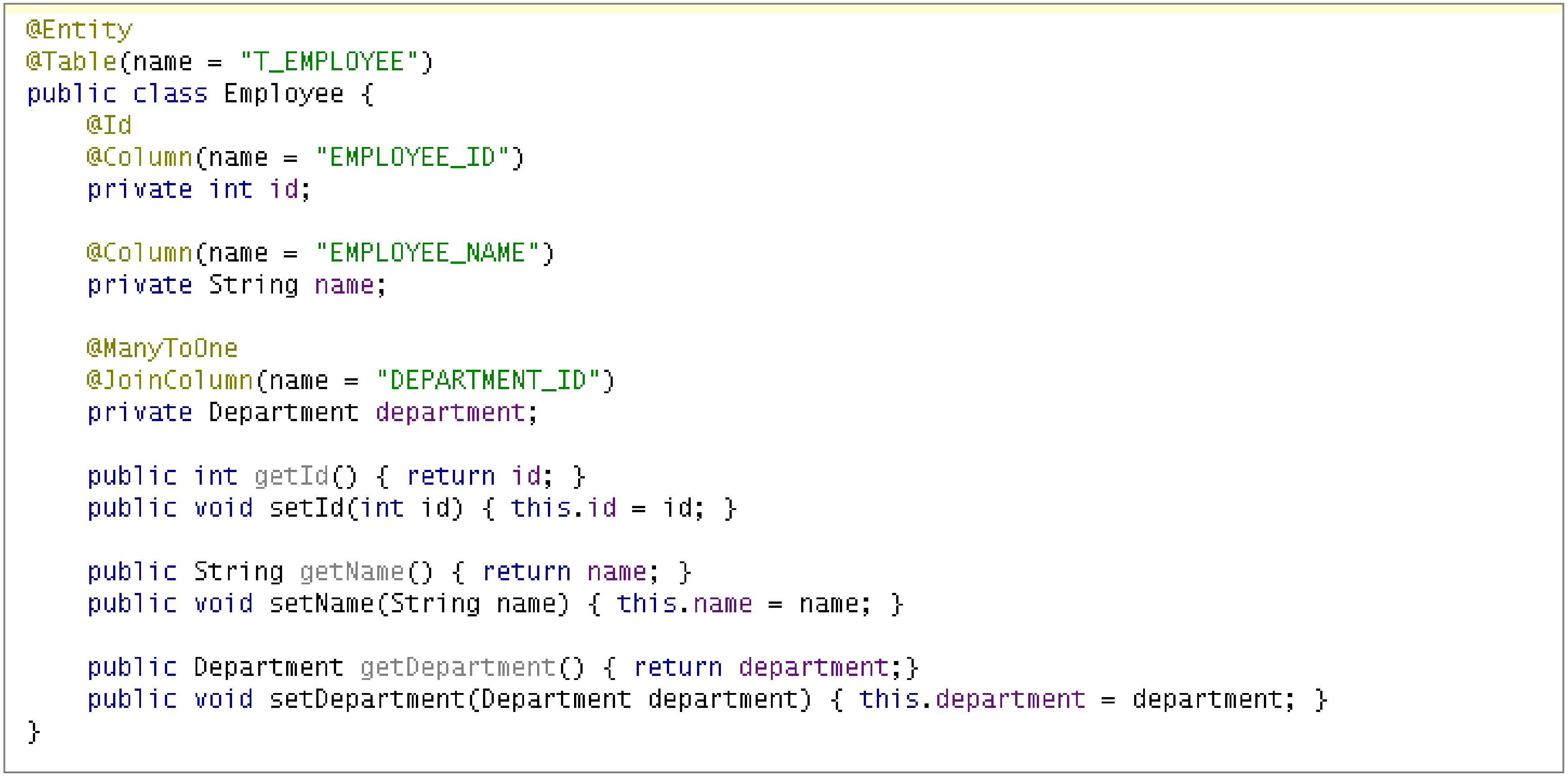


```java
@Entity
@Table(name = "T_DEPARTMENT")
public class Department {
    @Id
    @Column(name = "DEPARTMENT_ID")
    private int id;

    @Column(name = "DEPARTMENT_NAME")
    private String name;

    @OneToMany(mappedBy = "department")
    private List<Employee> employees;

    @ManyToOne
    @JoinColumn(name = "COMPANY_ID")
    private Company company;

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public List<Employee> getEmployees() { return employees; }
    public void setEmployees(List<Employee> employees) { this.employees = employees; }

    public Company getCompany() { return company;}
    public void setCompany(Company company) { this.company = company; }
}
```

Class describing the Department entity in terms of JPA.

Maps to T_DEPARTMENT table.

Database to POJO mapping using JPA



```
@Entity
@Table(name = "T_EMPLOYEE")
public class Employee {
    @Id
    @Column(name = "EMPLOYEE_ID")
    private int id;

    @Column(name = "EMPLOYEE_NAME")
    private String name;

    @ManyToOne
    @JoinColumn(name = "DEPARTMENT_ID")
    private Department department;

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public Department getDepartment() { return department;}
    public void setDepartment(Department department) { this.department = department; }
}
```

Class describing the Employee entity in terms of JPA.

Maps to T_EMPLOYEE table.

Entity

Entity is a lightweight persistent domain object

Entity class must:

- Be annotated with `@Entity` annotation
- Be not final top-level class *(enum or interface cannot be designated as entity)*
- Have public/protected no-arguments constructor (*but may have others*)

Entity class should implement `Serializable` interface *(if entity instance is to be passed by value as a detached object)*

Entity

Entity supports:

- Inheritance
- Polymorphic associations
- Polymorphic queries

Persistent state of entity is represented by instance variables, which may correspond to Java-Beans properties

Entity state is available to clients only through set/get or other business methods

Entity

Persistent state of entity is accessed by persistence provider runtime either:

- via JavaBeans style property accessors *(property access)*
- via instance variables *(field access)*

Instance **variables** must be **private**, **protected**, or **package visible**

Property access **methods** must be **public** or **protected**

Entity

Persistent field or property of entity may be of the following types:

- Java **primitive type** *(char, int, long, double)*
- Java **serializable types** *(including wrappers of the primitive types and user-defined types that implement the Serializable interface)*
- **Enums**
- **Entity types**
- **Embeddable types**
- **Collections** of **entity** types
- **Collections** of **basic** and **embeddable** types

Entity

Access methods signatures for single-valued property with name ***property*** of type T:

- `T getProperty()`
- `void setProperty(T t)`

Collection-valued fields and properties support:

- `java.util.Collection`
- `java.util.List`
- `java.util.Set`
- `java.util.Map`

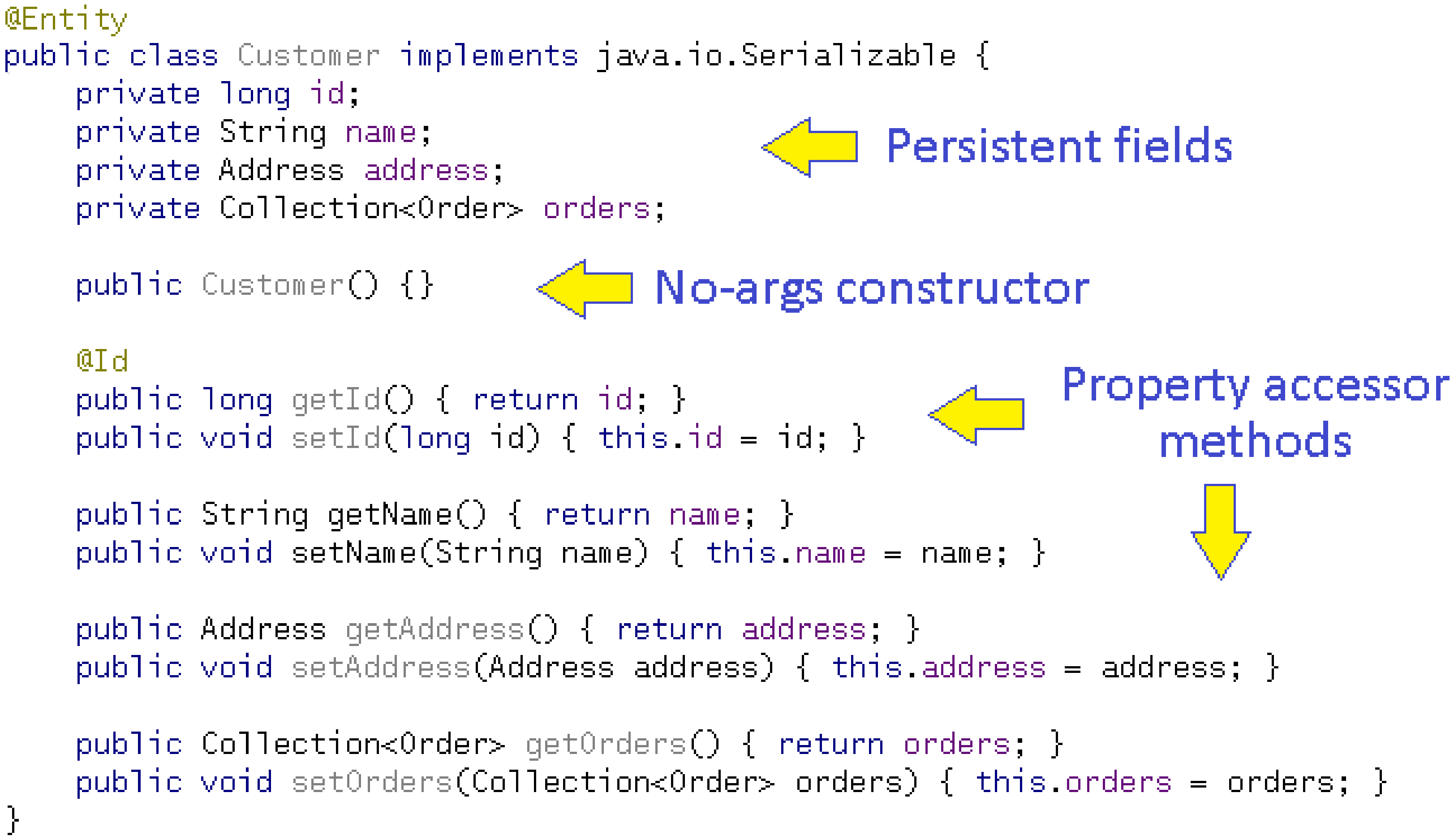
Entity

Property access + lazy fetching → state should be accessed only via accessor methods

Exception thrown by accessor method causes current transaction to be marked for rollback

Entity

Example of class definition for Customer entity that has several fields representing the state of the entity:



```
@Entity
public class Customer implements java.io.Serializable {
    private long id;
    private String name;
    private Address address;
    private Collection<Order> orders;

    public Customer() {}

    @Id
    public long getId() { return id; }
    public void setId(long id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public Address getAddress() { return address; }
    public void setAddress(Address address) { this.address = address; }

    public Collection<Order> getOrders() { return orders; }
    public void setOrders(Collection<Order> orders) { this.orders = orders; }
}
```

Persistent fields

No-args constructor

Property accessor methods

Entity Access Type

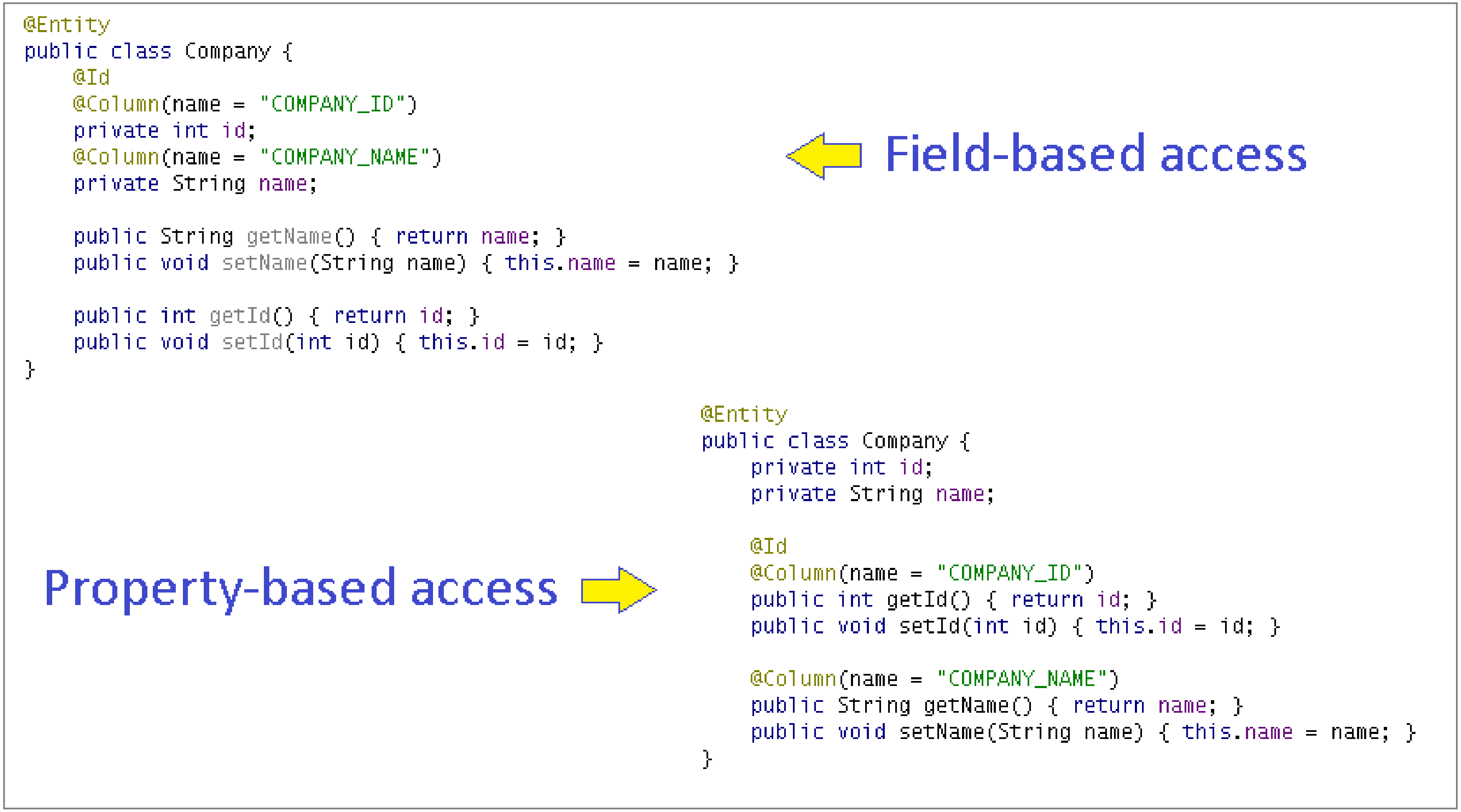
Access type is a **method** the persistence runtime uses **to access the persistent state** of the entity

Single access type (field or property) applies to an entity **by default**

Access type is **determined by placing** a mapping **annotations**:

- **Annotations** are on persistent **fields** then **field-based** access is used
- **Annotations** are on **properties** then **property-based** access is used

Entity Access Type



```
@Entity
public class Company {
    @Id
    @Column(name = "COMPANY_ID")
    private int id;
    @Column(name = "COMPANY_NAME")
    private String name;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
}
```

Field-based access

Property-based access

```
@Entity
public class Company {
    private int id;
    private String name;

    @Id
    @Column(name = "COMPANY_ID")
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    @Column(name = "COMPANY_NAME")
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

Company entity described with field and property based accesses to persistent state.

Entity Access Type

Behavior of applications that **mix the placement** of mapping annotations within entity (*without explicitly specifying the access type*) is **undefined**

Access type can be **explicitly defined** via **`@Access`** annotation

Entity Access Type

`@Access(AccessType.FIELD)`

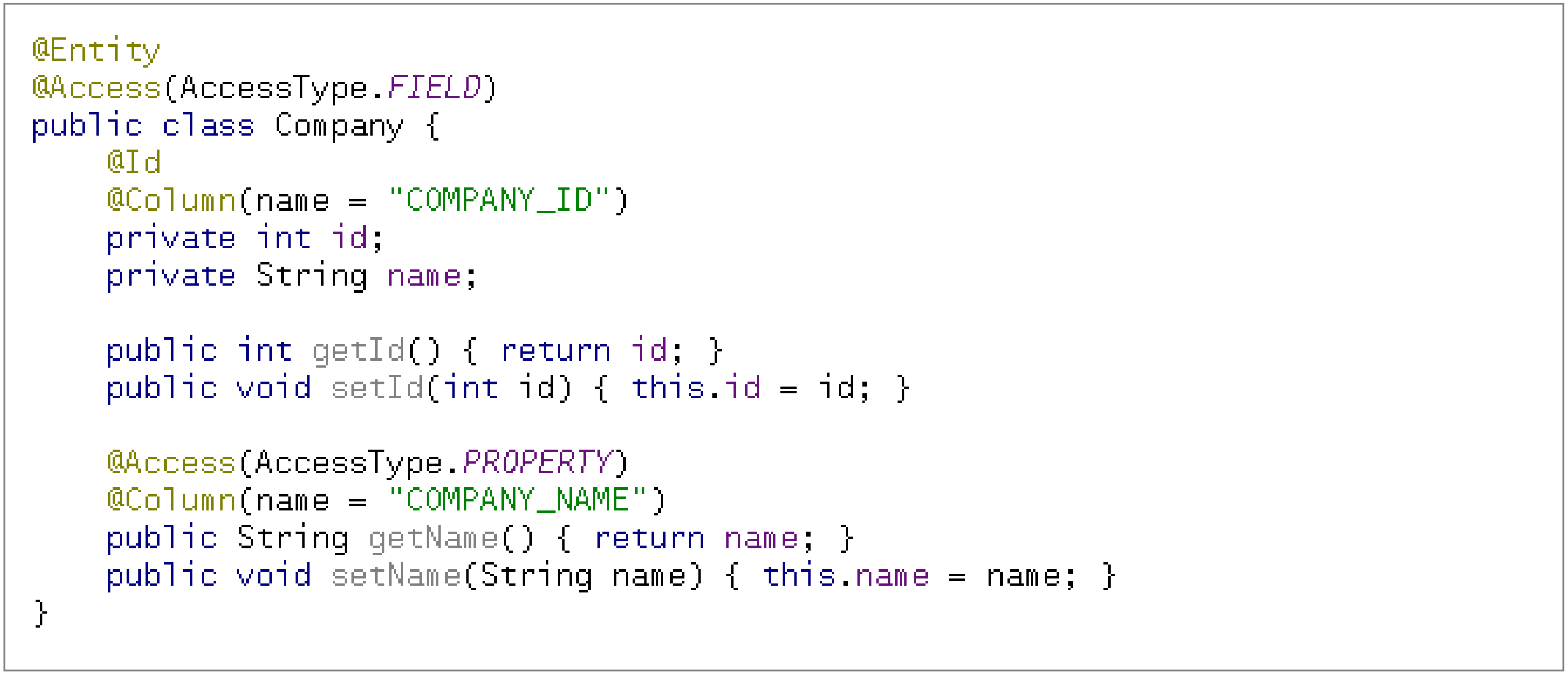
- Applied to an entity class defines access type default for this class
- Mapping annotations may be **placed on the instance variables** of that class
- Persistence runtime accesses **persistent state via the instance variables**
- It is possible to selectively designate individual attributes within the class for property access by specifying `@Access(AccessType.PROPERTY)` for needed property

Entity Access Type

`@Access(AccessType.PROPERTY)`

- Applied to an entity class defines access type default for this class
- Mapping annotations may be **placed on the properties** of that class
- Persistence provider runtime **accesses persistent state via the properties**
- It is possible to selectively designate individual attributes within the class for instance variable access by specifying `@Access(AccessType.FIELD)` for needed instance variable.

Entity Access Type



```
@Entity
@Access(AccessType.FIELD)
public class Company {
    @Id
    @Column(name = "COMPANY_ID")
    private int id;
    private String name;

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    @Access(AccessType.PROPERTY)
    @Column(name = "COMPANY_NAME")
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

Mixing access types in the entity class.

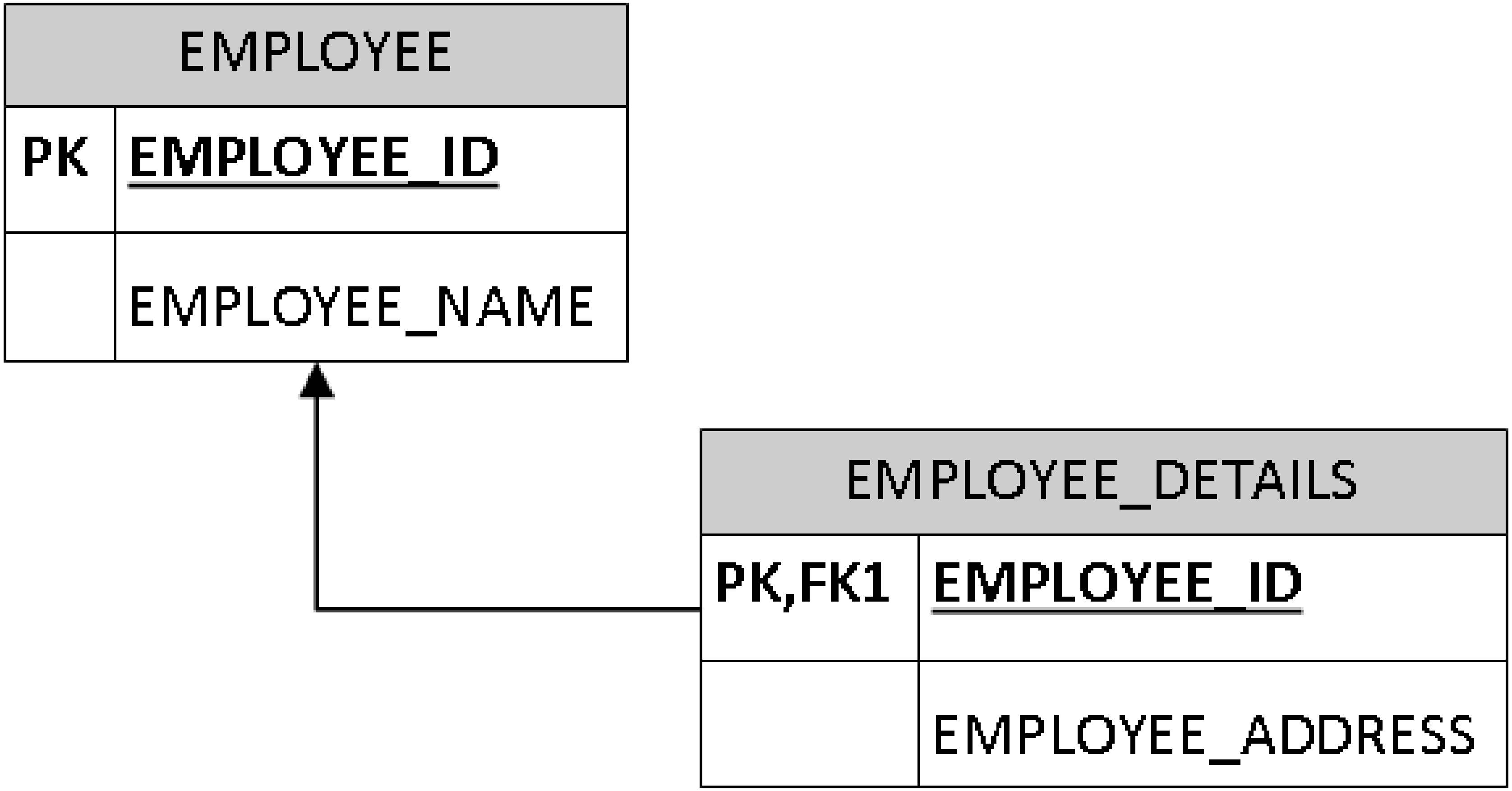
Mapping Database Objects

Naming of database objects is determined by the defaulting rules and the explicit names used in annotations

Annotations specifying the mapping of tables:

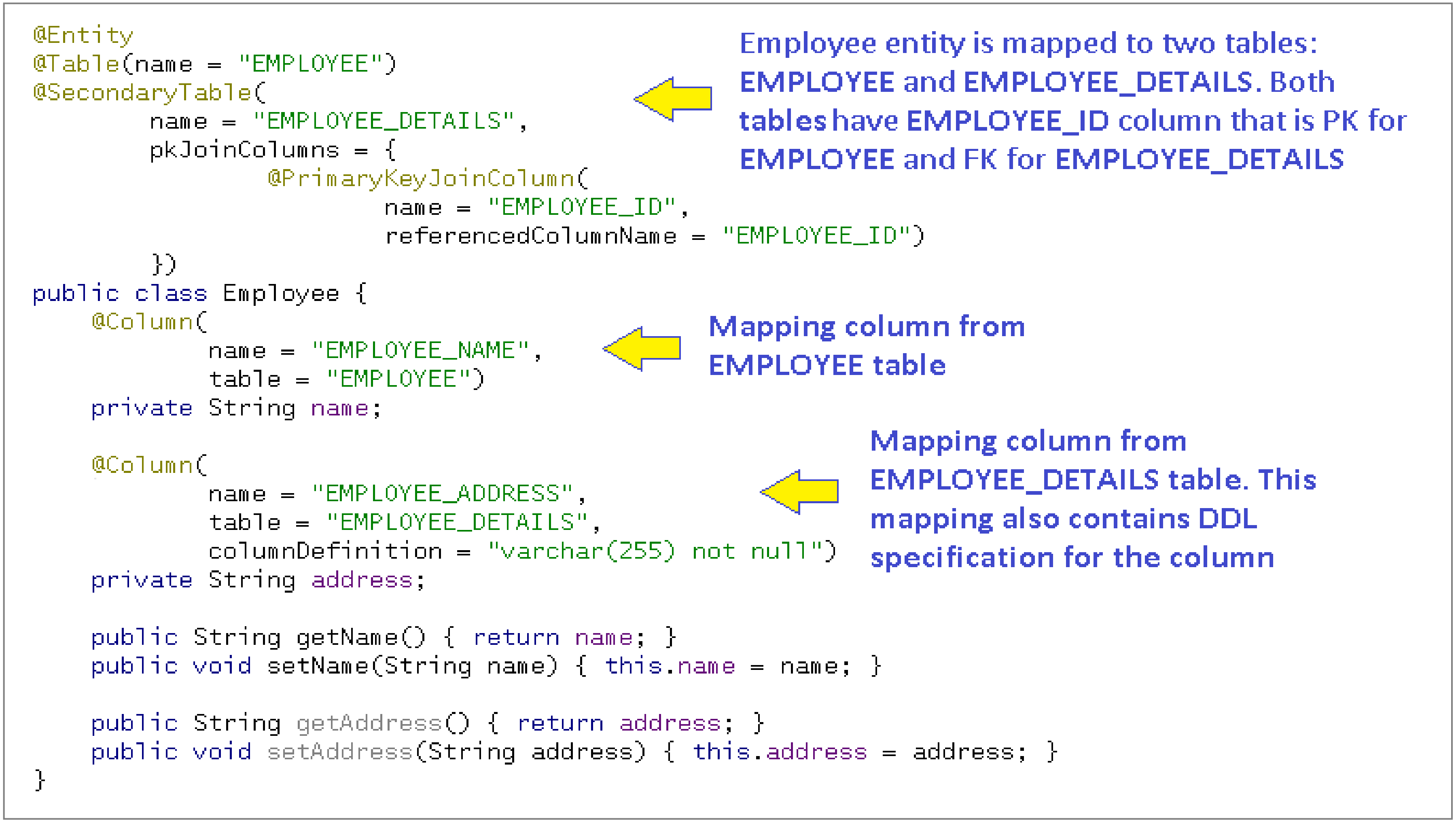
- `@Table`
- `@SecondaryTable`
- `@SecondaryTables`

Mapping Database Objects



Schema of the DB: Employee entity data are stored in tables EMPLOYEE and EMPLOYEE_DETAILS

Mapping Database Objects



```
@Entity
@Table(name = "EMPLOYEE")
@SecondaryTable(
        name = "EMPLOYEE_DETAILS",
        pkJoinColumns = {
                @PrimaryKeyJoinColumn(
                        name = "EMPLOYEE_ID",
                        referencedColumnName = "EMPLOYEE_ID")
        })
public class Employee {
    @Column(
            name = "EMPLOYEE_NAME",
            table = "EMPLOYEE")
    private String name;

    @Column(
            name = "EMPLOYEE_ADDRESS",
            table = "EMPLOYEE_DETAILS",
            columnDefinition = "varchar(255) not null")
    private String address;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public String getAddress() { return address; }
    public void setAddress(String address) { this.address = address; }
}
```

Employee entity is mapped to two tables: EMPLOYEE and EMPLOYEE_DETAILS. Both tables have EMPLOYEE_ID column that is PK for EMPLOYEE and FK for EMPLOYEE_DETAILS

Mapping column from EMPLOYEE table

Mapping column from EMPLOYEE_DETAILS table. This mapping also contains DDL specification for the column

Employee entity mapping on tables EMPLOYEE and EMPLOYEE_DETAILS

Mapping Database Objects

`@Table`

- Specifies the primary table for the annotated entity
- If not specified for an entity class, the default values apply

Parameters:

- **`name`** – Table name. Defaults to the entity name.
- **`catalog`** – Table catalog. Defaults to the default catalog.
- **`schema`** – Table schema. Defaults to the default schema.
- **`uniqueConstraints`** - Unique constraints that are to be placed on the table *(used if table generation is in effect)*.
- **`indexes`** - indexes for the table *(used if table generation is in effect)*.

Mapping Database Objects

`@SecondaryTable`

- Specifies a secondary table for the annotated entity class
- If not specified, it is assumed that all persistent fields or properties of the entity are mapped to the primary table

Parameters are the same as for `@Table` plus:

- **`pkJoinColumns`** – columns that are used to join with the primary table
- **`foreignKey`** - used to specify a foreign key constraint for the columns corresponding to the **`pkJoinColumns`** element *(used when table generation is in effect)*

Mapping Database Objects

Annotations specifying the mapping of table columns:

- `@Column`
- `@Lob`

Mapping Database Objects

`@Column`

- Specifies the mapped column for a persistent property/field
- If not specified, the default values apply

Parameters:

- **`name`** – Column name. Defaults to the property/field name.
- **`nullable`** – Whether the database column is nullable.
- **`insertable`** – Whether column is included in SQL INSERT statements generated by the persistence provider
- **`updatable`** – Whether column is included in SQL UPDATE statements generated by the persistence provider

Mapping Database Objects

Parameters (continue for `@Column`):

- **`table`** – The name of the table that contains the column. If absent the column is assumed to be in the primary table.
- **`length`** – The column length. (Applies only if a string-valued column is used.)
- **`unique`** – Whether the column is a unique key.
- **`columnDefinition`** – The SQL fragment that is used when generating the DDL for the column. Optional.

Practical work

Exercise 1:

Entity definition

Entity Primary Keys and Entity Identity

Every entity must have a primary key

Primary key corresponds to one or more fields or properties of the entity class:

Simple Primary Key

- Corresponds to a **single persistent field or property**
- **`@Id` annotation** is used to denote a simple primary key

Composite Primary Key

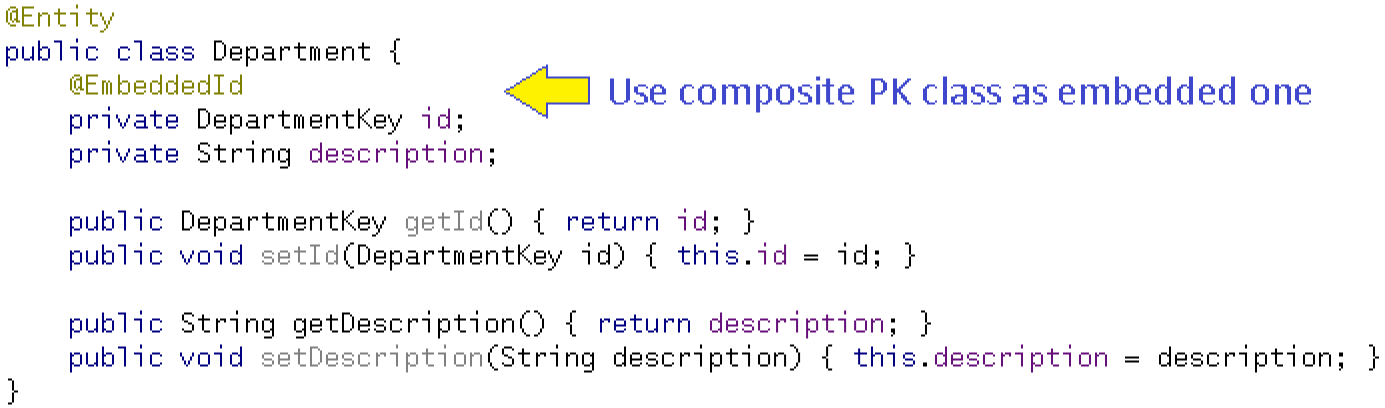
- Corresponds to either **a single persistent field or property** or **to a set of** such **fields or properties**
- **`@EmbeddedId` or `@IdClass` annotation** is used to denote a composite primary key

Entity Primary Keys and Entity Identity

Rules to apply for composite primary keys:

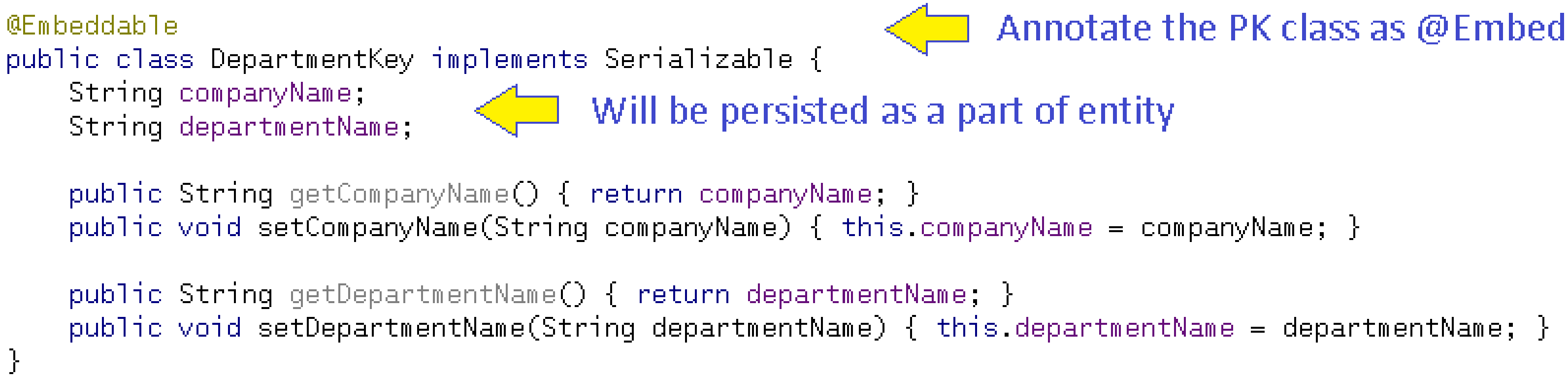
- **PK class** must be **public** and must **have public no-arg constructor**
- **Access type of PK** class is **determined** by the **access type of the entity** class
- PK class **must be serializable**
- PK class must **define `equals()` and `hashCode()` methods** (*semantics of value equality must be consistent with the database equality*)
- Can be represented as **embeddable class** or as **id class**
- If PK is represented as **id class** the **fields/properties** must **correspond** to **entity fields/properties** (names + types)

Entity Primary Keys and Entity Identity



```
@Entity
public class Department {
    @EmbeddedId
    private DepartmentKey id;
    private String description;

    public DepartmentKey getId() { return id; }
    public void setId(DepartmentKey id) { this.id = id; }

    public String getDescription() { return description; }
    public void setDescription(String description) { this.description = description; }
}
```

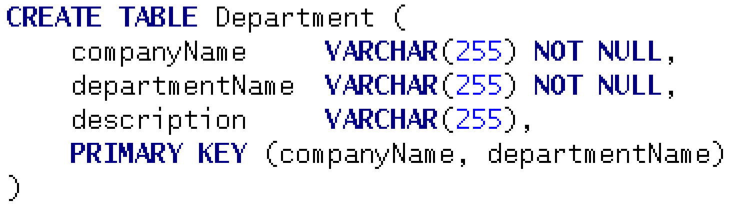
Use composite PK class as embedded one



```
@Embeddable
public class DepartmentKey implements Serializable {
    String companyName;
    String departmentName;

    public String getCompanyName() { return companyName; }
    public void setCompanyName(String companyName) { this.companyName = companyName; }

    public String getDepartmentName() { return departmentName; }
    public void setDepartmentName(String departmentName) { this.departmentName = departmentName; }
}
```

Annotate the PK class as @Embeddable

Will be persisted as a part of entity



```
CREATE TABLE Department (
    companyName      VARCHAR(255) NOT NULL,
    departmentName   VARCHAR(255) NOT NULL,
    description      VARCHAR(255),
    PRIMARY KEY (companyName, departmentName)
)
```

DDL statement generated for Departament entity

Entity Primary Keys and Entity Identity

```
@Entity
@IdClass(DepartmentKey.class)
public class Department {
    @Id
    private String companyName;
    private String departmentName;
    private String description;

    public String getCompanyName() { return companyName; }
    public void setCompanyName(String companyName) { this.companyName = companyName; }

    public String getDepartmentName() {return departmentName; }
    public void setDepartmentName(String departmentName) { this.departmentName = departmentName; }

    public String getDescription() { return description; }
    public void setDescription(String description) { this.description = description; }
}
```

@IdClass annotation is specified

companyName and departmentName fields must match PK fields

```
@Embeddable
public class DepartmentKey implements Serializable {
    String companyName;
    String departmentName;

    public String getCompanyName() { return companyName; }
    public void setCompanyName(String companyName) { this.companyName = companyName; }

    public String getDepartmentName() { return departmentName; }
    public void setDepartmentName(String departmentName) { this.departmentName = departmentName; }
}
```

Annotate the PK class as @Embeddable

Must match entity fields

```
CREATE TABLE Department (
    companyName     VARCHAR(255) NOT NULL,
    departmentName  VARCHAR(255) NOT NULL,
    description     VARCHAR(255),
    PRIMARY KEY (companyName, departmentName)
)
```

DDL statement generated for Departament entity

Entity Primary Keys and Entity Identity

JPA provides facilities for primary key generation

`@GeneratedValue` annotation:

- Specifies a generation strategy for the primary key value
- Used in conjunction with **`@Id`**
- Applied to persistent field or property
- Supported only for simple primary keys *(not for composite)*

Parameters:

- **`strategy`** – generation strategy to use *(optional)*
- **`generator`** – name of generator to use *(optional)*

Entity Primary Keys and Entity Identity

Supported primary key generation strategies:

- **Auto** (`strategy=GenerationType.AUTO`)

  Strategy by default. Indicates that the persistence provider should pick an appropriate strategy for the particular database.

- **Identity** (`strategy=GenerationType.IDENTITY`)

  Indicates that the persistence provider must assign primary keys for the entity using a database identity column.

- **Sequence** (`strategy=GenerationType.SEQUENCE`)

  Indicates that the persistence provider must assign primary keys for the entity using a database sequence.

- **Table** (`strategy=GenerationType.TABLE`)

  Indicates that the persistence provider must assign primary keys for the entity using an underlying database table to ensure uniqueness.

Entity Primary Keys and Entity Identity

`@TableGenerator`

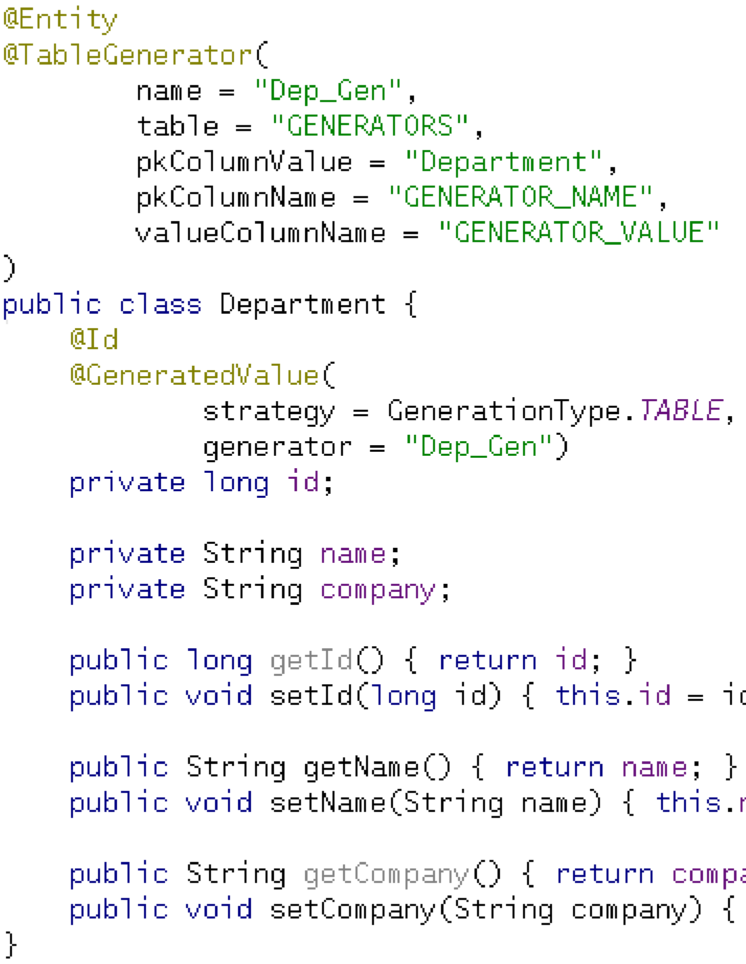
Defines generator for the Table strategy

Parameters:

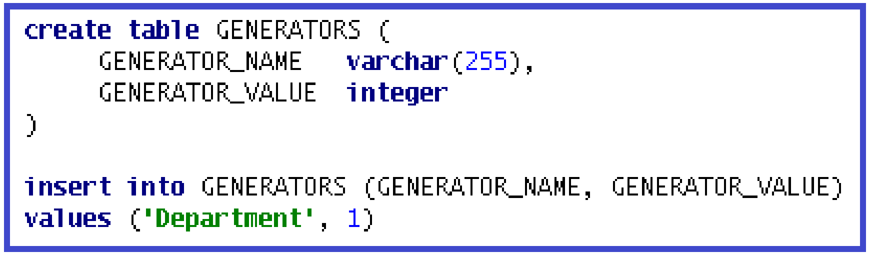
- **`name`** – a unique generator name that can be referenced by classes
- **`table`** – name of table that stores the generated id values
- **`catalog`** – catalog of the table
- **`schema`** – schema of the table
- **`pkColumnName`** – name of the primary key column in the table
- **`valueColumnName`** – column name that stores the last value generated
- **`pkColumnValue`** – primary key value in the generator table that distinguishes this set of generated values from others that may be stored in the table

Entity Primary Keys and Entity Identity

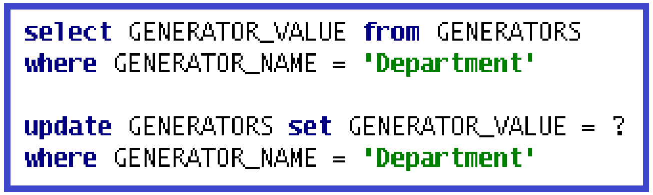
Example of using the Table generation strategy



```java
@Entity
@TableGenerator(
        name = "Dep_Gen",
        table = "GENERATORS",
        pkColumnValue = "Department",
        pkColumnName = "GENERATOR_NAME",
        valueColumnName = "GENERATOR_VALUE"
)
public class Department {
    @Id
    @GeneratedValue(
            strategy = GenerationType.TABLE,
            generator = "Dep_Gen")
    private long id;

    private String name;
    private String company;

    public long getId() { return id; }
    public void setId(long id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public String getCompany() { return company; }
    public void setCompany(String company) { this.company = company; }
}
```



```sql
create table GENERATORS (
     GENERATOR_NAME    varchar(255),
     GENERATOR_VALUE   integer
)

insert into GENERATORS (GENERATOR_NAME, GENERATOR_VALUE)
values ('Department', 1)
```



```sql
select GENERATOR_VALUE from GENERATORS
where GENERATOR_NAME = 'Department'

update GENERATORS set GENERATOR_VALUE = ?
where GENERATOR_NAME = 'Department'
```

Entity Primary Keys and Entity Identity

`@SequenceGenerator`

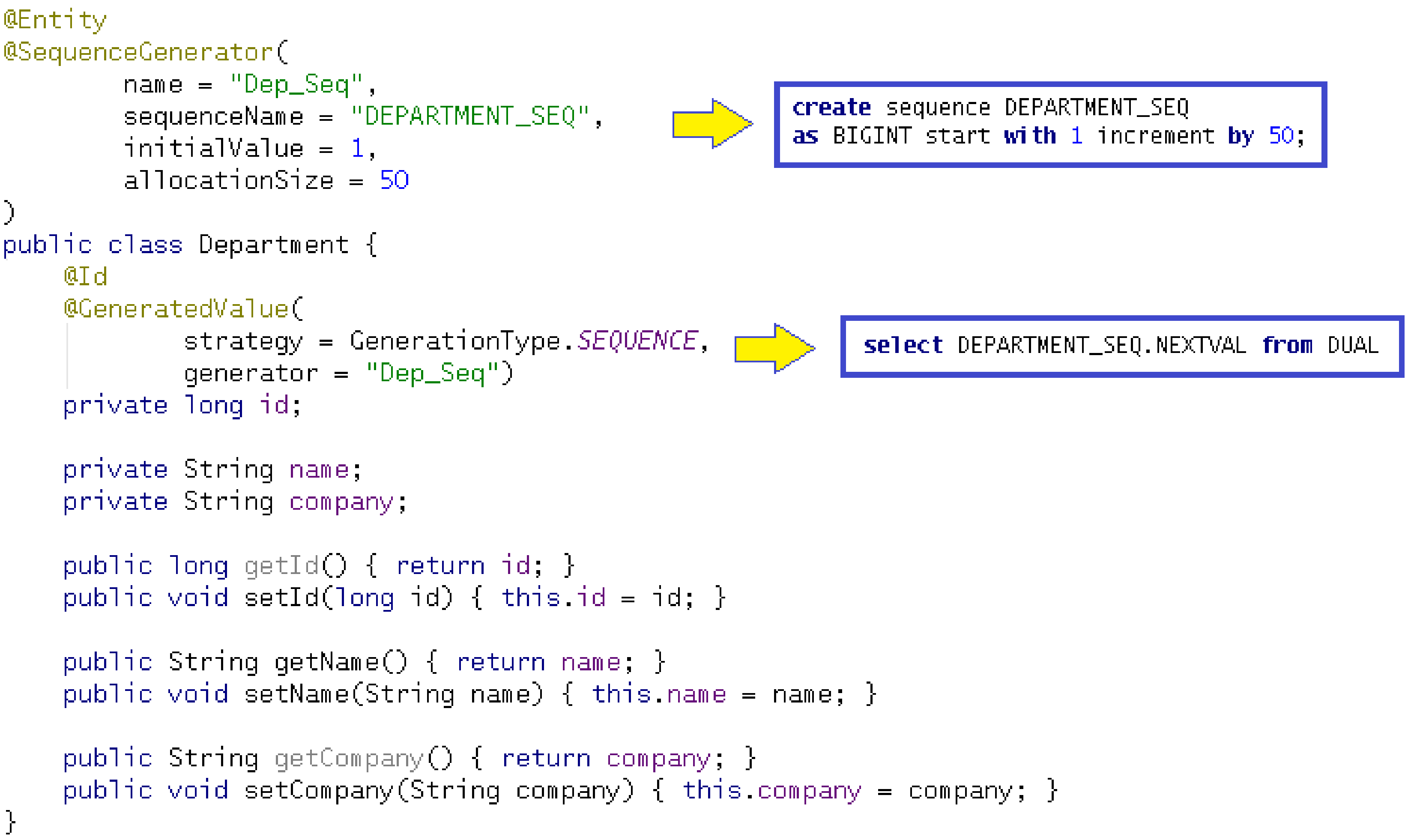
Defines generator for the Sequence strategy

Parameters:

- **`name`** – a unique generator name that can be referenced by classes
- **`sequenceName`** – name of the database sequence object from which to obtain primary key values
- **`catalog`** – catalog of the sequence generator
- **`schema`** – schema of the sequence generator
- **`initialValue`** – value from which the sequence object is to start generating
- **`allocationSize`** – amount to increment by when allocating sequence numbers from the sequence

Entity Primary Keys and Entity Identity

Example of using the Sequence generation strategy



```
@Entity
@SequenceGenerator(
        name = "Dep_Seq",
        sequenceName = "DEPARTMENT_SEQ",
        initialValue = 1,
        allocationSize = 50
)
public class Department {
    @Id
    @GeneratedValue(
            strategy = GenerationType.SEQUENCE,
            generator = "Dep_Seq")
    private long id;

    private String name;
    private String company;

    public long getId() { return id; }
    public void setId(long id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public String getCompany() { return company; }
    public void setCompany(String company) { this.company = company; }
}
```

```
create sequence DEPARTMENT_SEQ
as BIGINT start with 1 increment by 50;
```

```
select DEPARTMENT_SEQ.NEXTVAL from DUAL
```

Practical work

Exercise 2:

Identity of entity definition

Entity Relationships

JPA supports relationships for the entities:

- **One-to-Many**
- **Many-to-One**
- **Many-to-Many**
- **One-to-One**

Relationships can be:

- **Bidirectional** (has **owning** as well as **inverse** side)
- **Unidirectional** (has **owning** side only)

Entity Relationships

Owning side:

- Any relationship has an owning side
- Contains physical reference (foreign key)
- Drives the updates to relationship in a database

In **One-to-Many** and **Many-to-One** relationships, the **Many part** of the relationship is always **the Owning side**

The **inverse side** of a **bidirectional** relationship must **refer** to its **owning side**

Entity Relationships

JPA supports for relationships:

- Cascading operations

  Persist, Merge, Remove, Refresh, Detach, All

- Orphans removal

  Apply the remove operation to entities that have been removed from the relationship

- Lazy loading of related entities

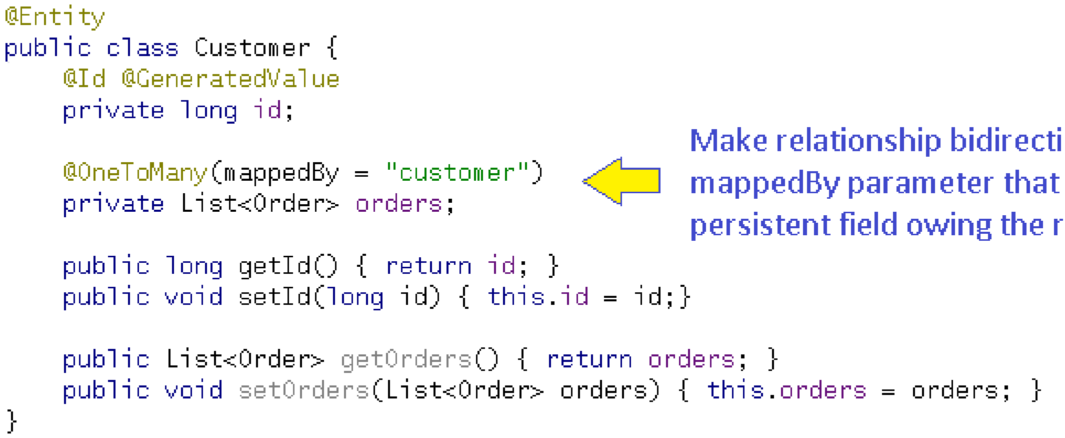
  ***Lazy** (load related entities when requested) and **Eager** (load related entities during loading of parent entity) modes are supported*

Entity Relationships

Let’s look on source code examples showing how relationships get defined…

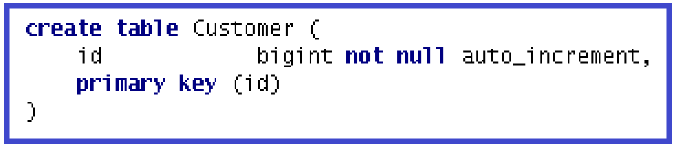
Entity Relationships

Bidirectional OneToMany/ManyToOne relationship:

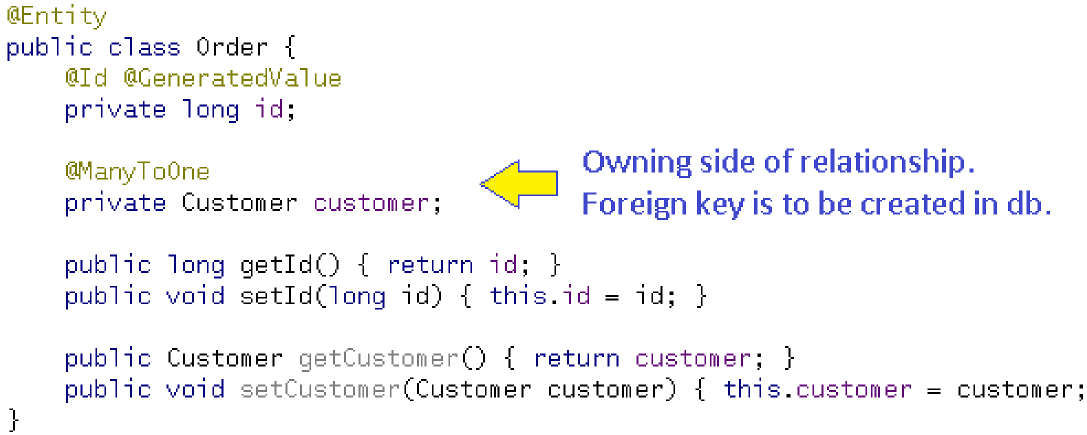


```
@Entity
public class Customer {
    @Id @GeneratedValue
    private long id;

    @OneToMany(mappedBy = "customer")
    private List<Order> orders;

    public long getId() { return id; }
    public void setId(long id) { this.id = id;}

    public List<Order> getOrders() { return orders; }
    public void setOrders(List<Order> orders) { this.orders = orders; }
}
```

Make relationship bidirectional by adding mappedBy parameter that reference to persistent field owing the relationship.

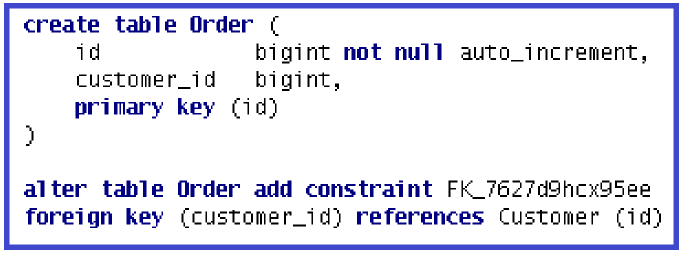


```
create table Customer (
    id              bigint not null auto_increment,
    primary key (id)
)
```



```
@Entity
public class Order {
    @Id @GeneratedValue
    private long id;

    @ManyToOne
    private Customer customer;

    public long getId() { return id; }
    public void setId(long id) { this.id = id; }

    public Customer getCustomer() { return customer; }
    public void setCustomer(Customer customer) { this.customer = customer; }
}
```

Owning side of relationship.
Foreign key is to be created in db.



```
create table Order (
    id              bigint not null auto_increment,
    customer_id     bigint,
    primary key (id)
)

alter table Order add constraint FK_7627d9hcx95ee
foreign key (customer_id) references Customer (id)
```

Entity Relationships

Bidirectional ManyToMany relationship:

```
@Entity
public class Product {
    @Id @GeneratedValue
    private long id;

    @ManyToMany(mappedBy = "products")
    private List<Order> orders;

    public long getId() { return id;}
    public void setId(long id) { this.id = id; }

    public List<Order> getOrders() { return orders; }
    public void setOrders(List<Order> orders) { this.orders = orders; }
}
```

Make relationship bidirectional by adding mappedBy parameter that reference to persistent field owing the relationship.

```
@Entity
public class Order {
    @Id @GeneratedValue
    private long id;

    @ManyToMany
    private List<Product> products;

    public long getId() { return id;}
    public void setId(long id) { this.id = id; }

    public List<Product> getProducts() { return products; }
    public void setProducts(List<Product> products) { this.products = products; }
}
```

Owning side of relationship.

```
create table Product (
    id    bigint not null auto_increment,
    primary key (id)
)

create table Order (
    id    bigint not null auto_increment,
    primary key (id)
)

create table Order_Product (
    orders_id      bigint not null,
    products_id    bigint not null
)

alter table Order_Product add constraint FK_4
foreign key(products_id) references Product(id)

alter table Order_Product add constraint FK_5
foreign key(orders_id) references ProductOrder(id)
```

Entity Relationships

Annotations that are used to define entities relationships:

- `@OneToMany` – defines one-to-many relationship
- `@ManyToOne` – defines many-to-one relationship
- `@OneToOne` – defines one-to-one relationship
- `@ManyToMany` – defines many-to-many relationship

Entity Relationships

Relationship annotations parameters:

- `targetEntity` – entity class that is the target of the association
- `cascade` – operations that must be cascaded to the target of the association
- `fetch` – whether the association should be lazily loaded or must be eagerly fetched
- `optional` – whether the association is optional *(causes inner join or outer join is to be used)*
- `mappedBy` – name of field that owns the relationship
- `orphanRemoval` – whether to apply the remove operation to entities that have been removed from the relationship

Entity Relationships

Annotations, useful for relationship definition:

- `@JoinTable`
- `@JoinColumn`
- `@JoinColumns`

Entity Relationships

`@JoinTable` **–** specifies the cross-reference table for the mapping of relationship

Must be **specified on the owning** side of relationship

Parameters:

- **`name`** – name of the cross-reference table
- **`joinColumns`** – foreign key columns (in the cross-reference table) which reference the table of the entity **that owns** the relationship
- **`inverseJoinColumns`** – foreign key columns (in the cross-reference table) which reference the table of the entity **that does not own** the relationship

Entity Relationships

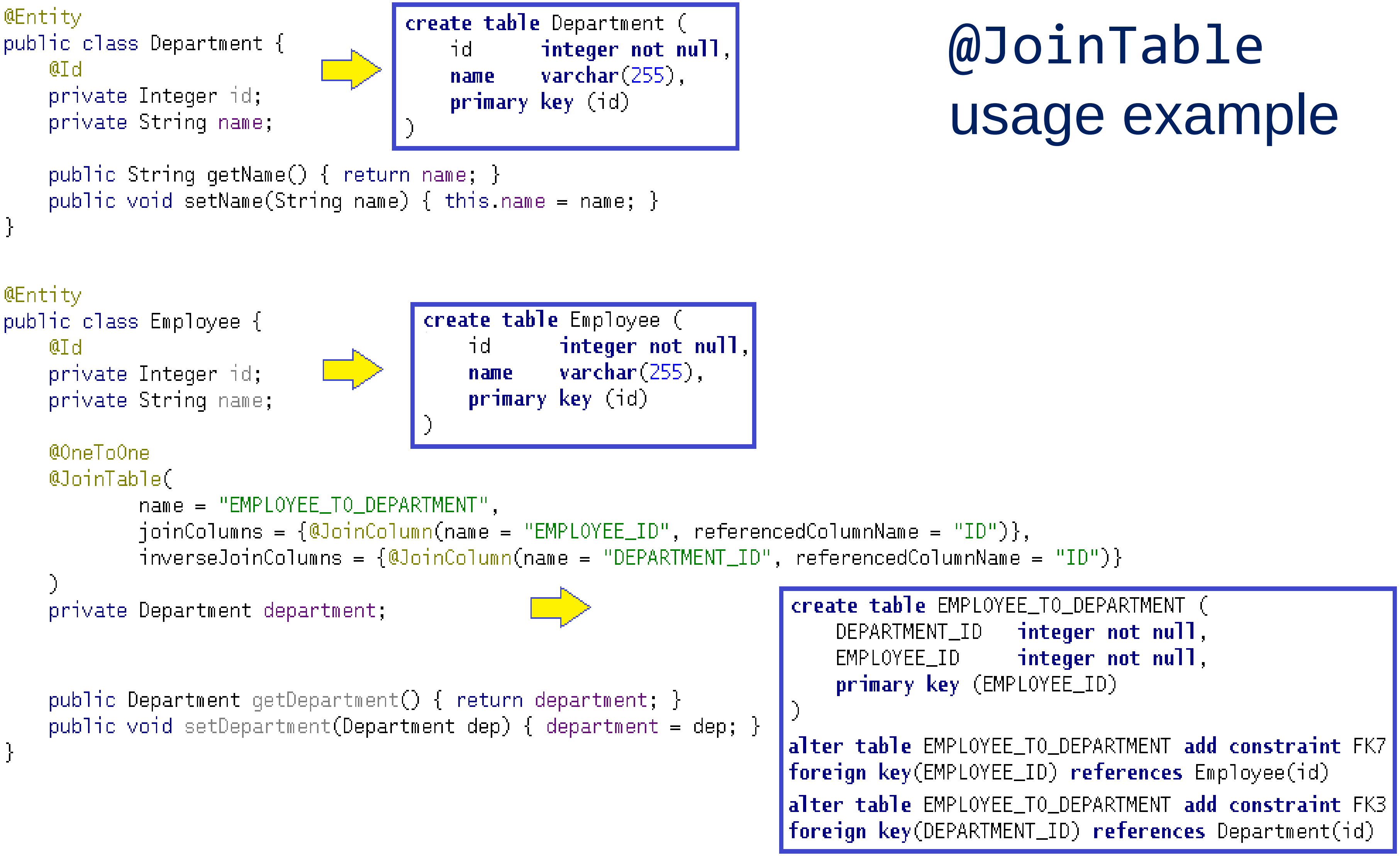
`@JoinColumn –` specifies a column for joining an entity association

Parameters:

- **`name`** – name of the foreign key
- **`referencedColumnName`** – name of the column referenced by this foreign key column
- **`nullable`** - whether the foreign key column is nullable *(inner or outer join)*
- **`insertable`** – whether to include into INSERT statements
- **`updatable`** – whether to include into UPDATE statements

Entity Relationships

@JoinTable usage example



```
@Entity
public class Department {
    @Id
    private Integer id;
    private String name;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

```
create table Department (
    id       integer not null,
    name     varchar(255),
    primary key (id)
)
```

```
@Entity
public class Employee {
    @Id
    private Integer id;
    private String name;

    @OneToOne
    @JoinTable(
            name = "EMPLOYEE_TO_DEPARTMENT",
            joinColumns = {@JoinColumn(name = "EMPLOYEE_ID", referencedColumnName = "ID")},
            inverseJoinColumns = {@JoinColumn(name = "DEPARTMENT_ID", referencedColumnName = "ID")}
    )
    private Department department;


    public Department getDepartment() { return department; }
    public void setDepartment(Department dep) { department = dep; }
}
```

```
create table Employee (
    id       integer not null,
    name     varchar(255),
    primary key (id)
)
```

```
create table EMPLOYEE_TO_DEPARTMENT (
    DEPARTMENT_ID   integer not null,
    EMPLOYEE_ID     integer not null,
    primary key (EMPLOYEE_ID)
)
alter table EMPLOYEE_TO_DEPARTMENT add constraint FK7
foreign key(EMPLOYEE_ID) references Employee(id)
alter table EMPLOYEE_TO_DEPARTMENT add constraint FK3
foreign key(DEPARTMENT_ID) references Department(id)
```

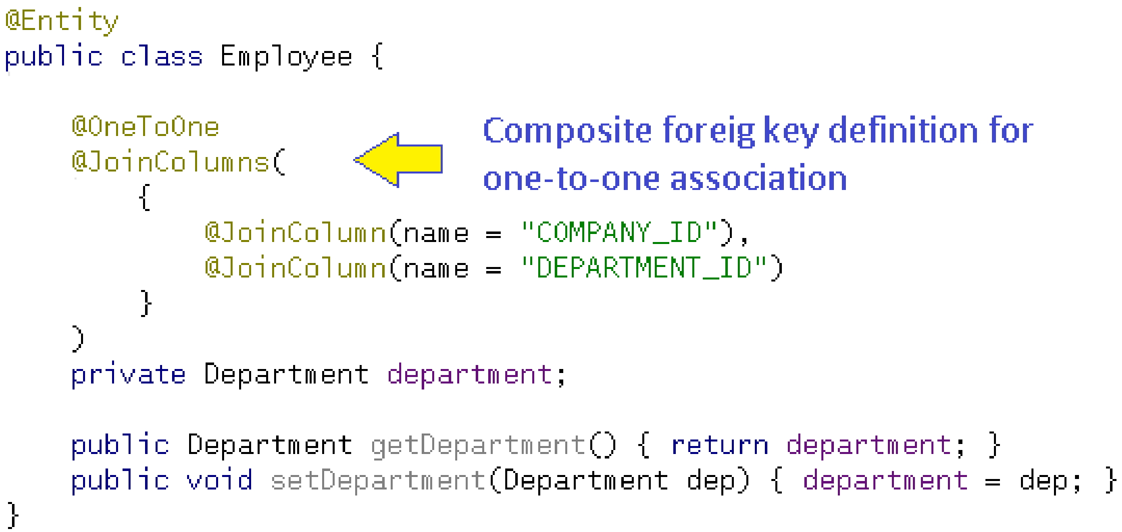
Entity Relationships

`@JoinColumns` – defines the mapping for composite foreign keys (grouping `@JoinColumn` annotations)

Parameters:

- **`value`** – arrays of `@JoinColumn` defining composite foreign key

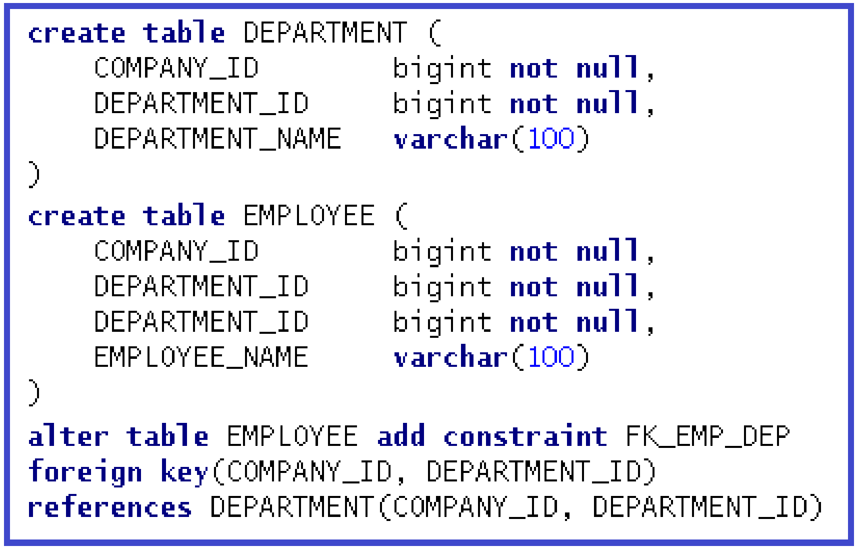
Entity Relationships



```
@Entity
public class Employee {

    @OneToOne
    @JoinColumns(
        {
            @JoinColumn(name = "COMPANY_ID"),
            @JoinColumn(name = "DEPARTMENT_ID")
        }
    )
    private Department department;

    public Department getDepartment() { return department; }
    public void setDepartment(Department dep) { department = dep; }
}
```

Composite foreig key definition for one-to-one association

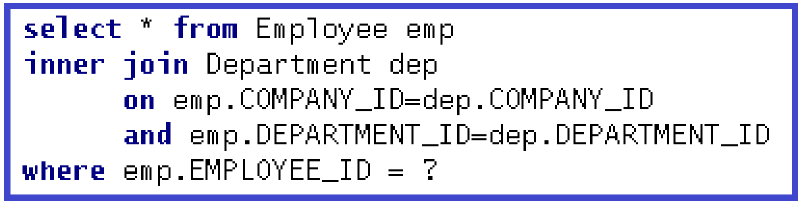
@JoinColumns usage example



```
create table DEPARTMENT (
    COMPANY_ID          bigint not null,
    DEPARTMENT_ID       bigint not null,
    DEPARTMENT_NAME     varchar(100)
)
create table EMPLOYEE (
    COMPANY_ID          bigint not null,
    DEPARTMENT_ID       bigint not null,
    DEPARTMENT_ID       bigint not null,
    EMPLOYEE_NAME       varchar(100)
)
alter table EMPLOYEE add constraint FK_EMP_DEP
foreign key(COMPANY_ID, DEPARTMENT_ID)
references DEPARTMENT(COMPANY_ID, DEPARTMENT_ID)
```

DDL satement for db objects

DML statement Hibernate generates to load the data from db



```
select * from Employee emp
inner join Department dep
      on emp.COMPANY_ID=dep.COMPANY_ID
      and emp.DEPARTMENT_ID=dep.DEPARTMENT_ID
where emp.EMPLOYEE_ID = ?
```

Embeddable Classes

Embeddable classes:

- Fine-grained classes representing entity state
- Do not have persistent identity of their own
- Exist only as part of the state of the entity to which they belong
- Cannot be shared across persistent entities (*attempting to share has undefined semantics*)

Entity may have collections of embeddables as well as single-valued embeddable attributes

Embeddable Classes

Embeddable classes follow the same rules as entity except annotating as `@Entity`

Embeddables classes must be annotated as `@Embeddable`

Embeddable class may contain relationship to entity or collection of entities

Since instances of embeddable classes themselves have no persistent identity, the relationship ***from referenced entity*** *is to the* ***entity that contains embeddable instance*** *and not to the embeddable itself.*

Embeddable Classes

Customization of embeddable classes mapping can be done with help of:

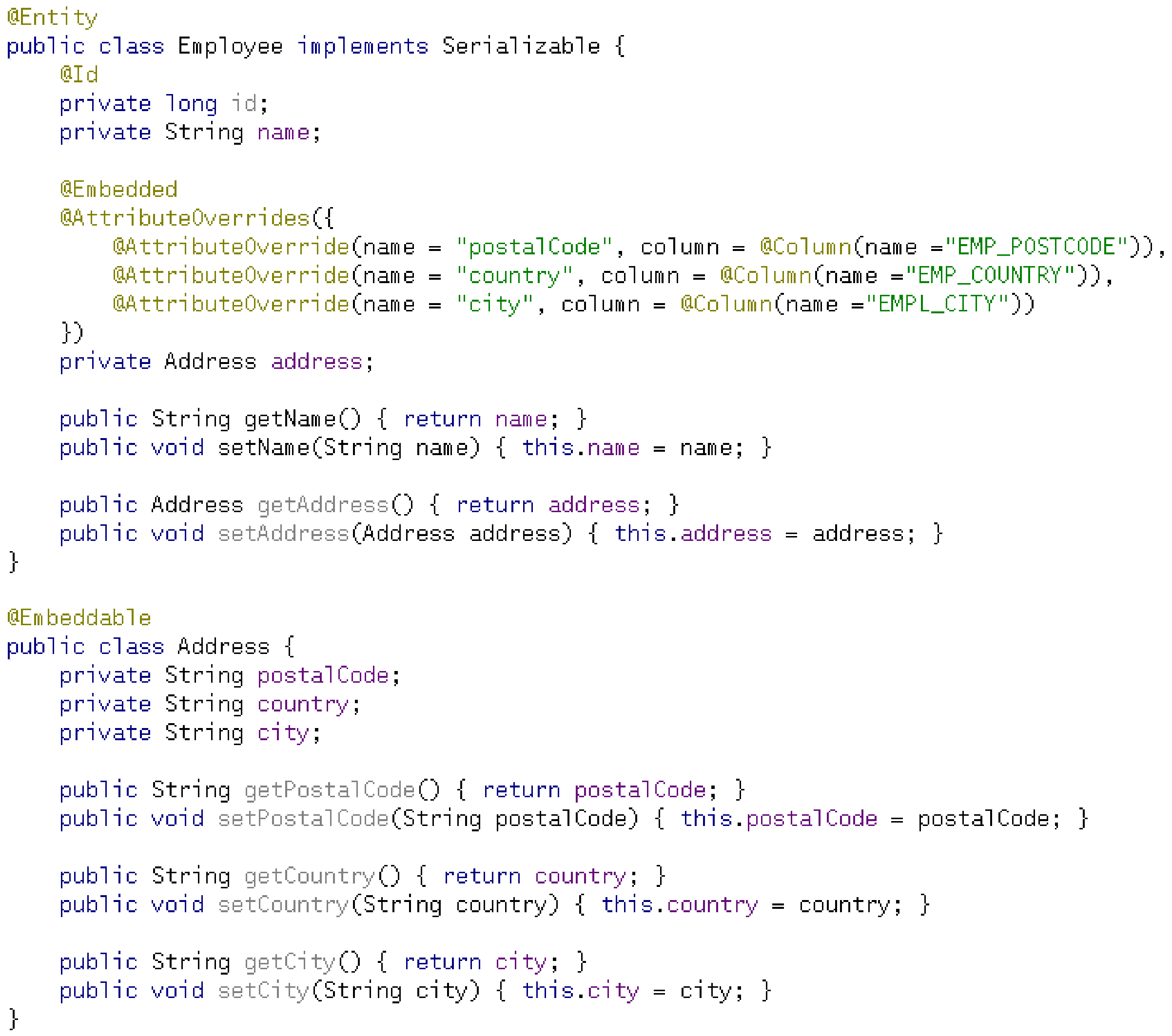
- **`@AttributeOverride`** – overrides mapping for particular field or property of embeddable class

  `name` – name of field/property to override the mapping

  `column` – database column name

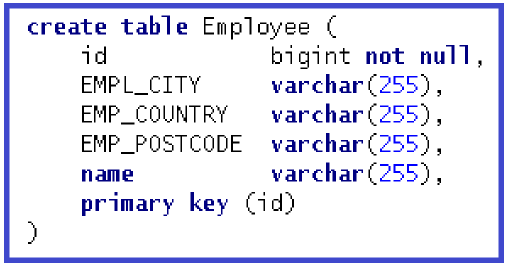
- **`@AttributeOverrides`** - overrides mappings of multiple properties or fields

Embeddable Classes



```java
@Entity
public class Employee implements Serializable {
    @Id
    private long id;
    private String name;

    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name = "postalCode", column = @Column(name ="EMP_POSTCODE")),
        @AttributeOverride(name = "country", column = @Column(name ="EMP_COUNTRY")),
        @AttributeOverride(name = "city", column = @Column(name ="EMPL_CITY"))
    })
    private Address address;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public Address getAddress() { return address; }
    public void setAddress(Address address) { this.address = address; }
}

@Embeddable
public class Address {
    private String postalCode;
    private String country;
    private String city;

    public String getPostalCode() { return postalCode; }
    public void setPostalCode(String postalCode) { this.postalCode = postalCode; }

    public String getCountry() { return country; }
    public void setCountry(String country) { this.country = country; }

    public String getCity() { return city; }
    public void setCity(String city) { this.city = city; }
}
```

Embeddable class usage samples



```sql
create table Employee (
    id            bigint not null,
    EMPL_CITY     varchar(255),
    EMP_COUNTRY   varchar(255),
    EMP_POSTCODE  varchar(255),
    name          varchar(255),
    primary key (id)
)
```

Collections of Embeddable Classes

JPA 2.0 supports having collections of basic types or embeddable classes for the entity *(similar to One-to-Many relation for entities)*

Useful annotations:

- **`@ElementCollection`** - defines collection of instances of a basic type or embeddable class
- **`@CollectionTable`** - specifies the table that is used for the mapping of collections of basic or embeddable types

Supported collections: all Java collection types (Collection, List, Set, Map)

Collections of Embeddable Classes

```java
@Embeddable
public class Project {
    private String name;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}

@Entity
public class Employee implements Serializable {
    @Id
    private long id;
    private String name;

    @ElementCollection
    @CollectionTable(
            name = "Employee_Projects",
            joinColumns = {
                    @JoinColumn(name = "employee_id", referencedColumnName = "id")
            })
    private List<Project> projects;

    public long getId() { return id; }
    public void setId(long id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public List<Project> getProjects() { return projects; }
    public void setProjects(List<Project> projects) { this.projects = projects; }
}
```

```sql
create table Employee (
    id      bigint not null,
    name    varchar(255),
    primary key (id)
)
create table Employee_Projects (
    employee_id   bigint not null,
    name varchar(255)
)

alter table Employee_Projects add constraint FK_1
foreign key (employee_id) references Employee (id)
```

Collection of embeddable classes example

Collections of Embeddable Classes

JPA 2.0 also supports mapping embeddable classes and basic types to Map:

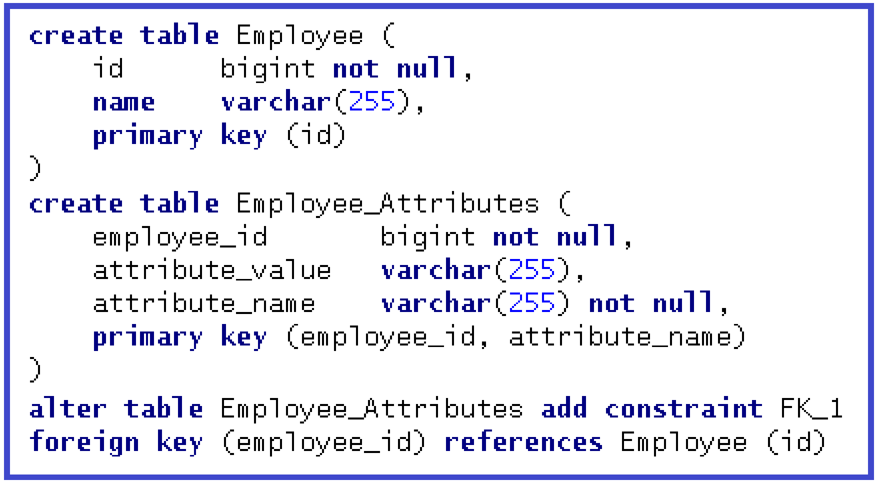
- Key is **basic type**, value is **embeddable** class
- Key is **embeddable** class, value is **basic type**
- Key and value **both are basic types**

Annotations:

`@MapKeyColumn` is used to specify column name for map key (if key is basic type)

`@Column` is used to specify column for map value (if value is basic type)

Collections of Embeddable Classes



```
create table Employee (
    id      bigint not null,
    name    varchar(255),
    primary key (id)
)
create table Employee_Attributes (
    employee_id        bigint not null,
    attribute_value    varchar(255),
    attribute_name     varchar(255) not null,
    primary key (employee_id, attribute_name)
)
alter table Employee_Attributes add constraint FK_1
foreign key (employee_id) references Employee (id)
```

```
@Entity
public class Employee implements Serializable {
    @Id
    private long id;
    private String name;

    @{...}
    private List<Project> projects;

    @ElementCollection
    @MapKeyColumn(name = "attribute_name")
    @Column(name = "attribute_value")
    @CollectionTable(
        name = "Employee_Attributes",
        joinColumns = {@JoinColumn(name = "employee_id", referencedColumnName = "id")})
    private Map<String, String> attributes;

    public long getId() { return id; }
    public void setId(long id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public List<Project> getProjects() { return projects; }
    public void setProjects(List<Project> projects) { this.projects = projects; }

    public Map<String, String> getAttributes() { return attributes;}
    public void setAttributes(Map<String, String> attributes) { this.attributes = attributes; }
}
```

Mapping to Map collection example

Exercise 3:

Entities relations definition

Entity Inheritance: Hierarchy definition

An entity may inherit from another entity class

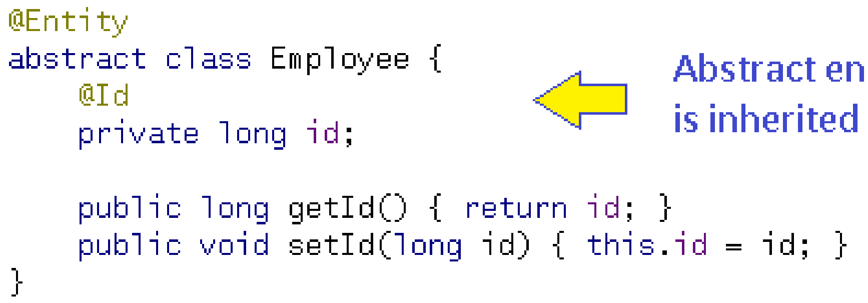
An abstract class can be specified as entity *(but cannot be directly instantiated)*

An abstract entity class:

- Annotated with the `@Entity` annotation
- Mapped as an entity
- Can be the target of queries

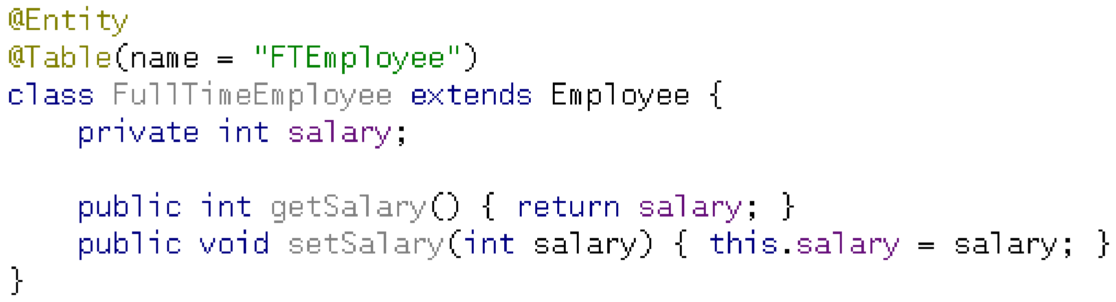
JPA supports polymorphic associations and queries for an entities

Entity Inheritance: Hierarchy definition

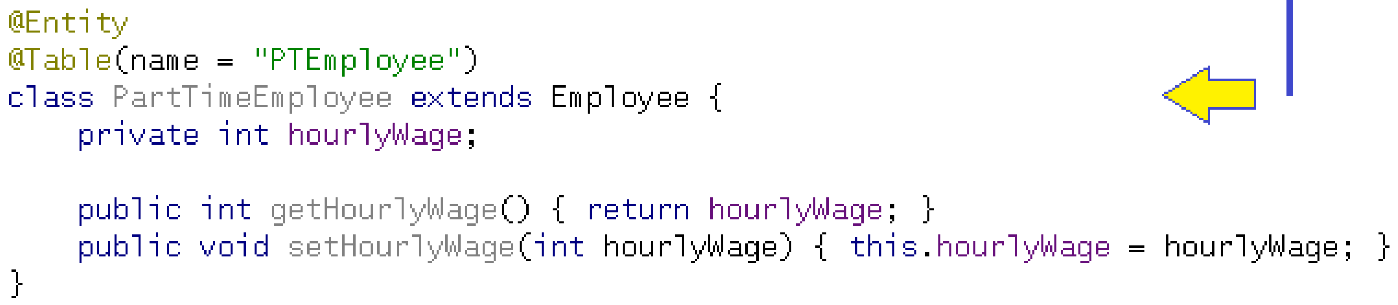


```
@Entity
abstract class Employee {
    @Id
    private long id;

    public long getId() { return id; }
    public void setId(long id) { this.id = id; }
}
```

Abstract entity class defining persistance state that is inherited by its subclasses



```
@Entity
@Table(name = "FTEmployee")
class FullTimeEmployee extends Employee {
    private int salary;

    public int getSalary() { return salary; }
    public void setSalary(int salary) { this.salary = salary; }
}
```



```
@Entity
@Table(name = "PTEmployee")
class PartTimeEmployee extends Employee {
    private int hourlyWage;

    public int getHourlyWage() { return hourlyWage; }
    public void setHourlyWage(int hourlyWage) { this.hourlyWage = hourlyWage; }
}
```

Concrete entity classes extending abstract entity

Example of abstract entity class extension

Entity Inheritance: Hierarchy definition

An entity may inherit from a superclass that provides persistent entity state and mapping information, but which is not itself an entity

The purpose of such a mapped superclass is to define state and mapping information that is common to multiple entity classes

Mapped superclass:

- Not queryable
- Relationships defined by a mapped superclass must be unidirectional

Entity Inheritance: Hierarchy definition

`@MappedSuperclass` annotation is used to specify class as mapped superclass

`@AttributeOverride` and `@AssociationOverride` can be used to override mapping for concrete class

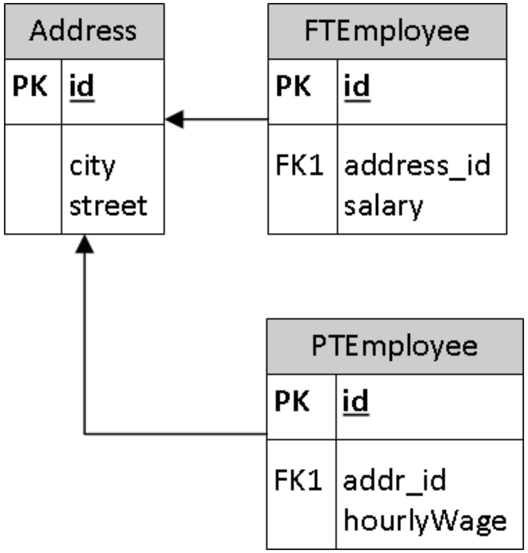
Entity Inheritance: Hierarchy definition

```
@MappedSuperclass
class Employee {
    @Id
    private long id;
    @ManyToOne
    @JoinColumn(name = "address_id")
    private Address address;

    public Address getAddress() { return address; }
    public void setAddress(Address address) { this.address = address; }
}
```

Mapped superclass is an template for an entities. Doesn't have its own persistance.

```
@Entity
@Table(name = "FTEmployee")
class FullTimeEmployee extends Employee {
    private int salary;
    public int getSalary() { return salary; }
    public void setSalary(int salary) { this.salary = salary; }
}
```

```
@Entity
@Table(name = "PTEmployee")
@AssociationOverride(name = "address", joinColumns = @JoinColumn(name="addr_id"))
class PartTimeEmployee extends Employee {
    private int hourlyWage;
    public int getHourlyWage() { return hourlyWage; }
    public void setHourlyWage(int hourlyWage) { this.hourlyWage = hourlyWage; }
}
```



Address	
PK	id
	city street

FTEmployee	
PK	id
FK1	address_id salary

PTEmployee	
PK	id
FK1	addr_id hourlyWage

@MappedSuperclass is an just template and doesn't have its own persistence

Entity Inheritance: Hierarchy definition

An entity can have a non-entity superclass, which may be either a concrete or abstract class

- Used for inheritance of behavior only
- State of a non-entity superclass is not persistent
- Any annotations on such superclass are ignored

Entity Inheritance: Mapping

There are three basic strategies that are used when mapping a class or class hierarchy to a relational database:

- Single table per class hierarchy
- Joined subclass strategy *(in which fields that are specific to a subclass are mapped to a separate table than the fields that are common to the parent class, and a join is performed to instantiate the subclass)*
- Table per concrete entity class

Entity Inheritance: Mapping

Single Table per Class Hierarchy Strategy:

- All the classes in a hierarchy are mapped to a single table
- The table has a column that serves as a “discriminator column” *(whose value identifies the specific subclass)*

Benefits:

- Provides good support for polymorphic relationships

Drawbacks:

- Requires that the columns that correspond to state specific to the subclasses be nullable

Entity Inheritance: Mapping

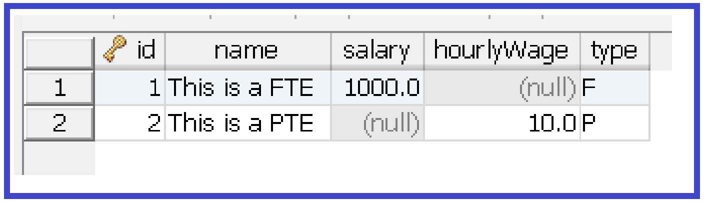
```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "type")
public abstract class Employee {
    @Id @GeneratedValue
    private long id;
    private String name;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

Define the mapping strategy and discriminator column

```
create table Employee (
    id          bigint auto_increment,
    type        varchar(31) not null,
    name        varchar(255),
    salary      double precision,
    hourlyWage  double precision,
    primary key (id)
)
```

```
@Entity
@DiscriminatorValue("F")
public class FullTimeEmployee extends Employee {
    private double salary;

    public double getSalary() { return salary; }
    public void setSalary(double salary) { this.salary = salary; }
}
```

Defined discriminator value for FTE class



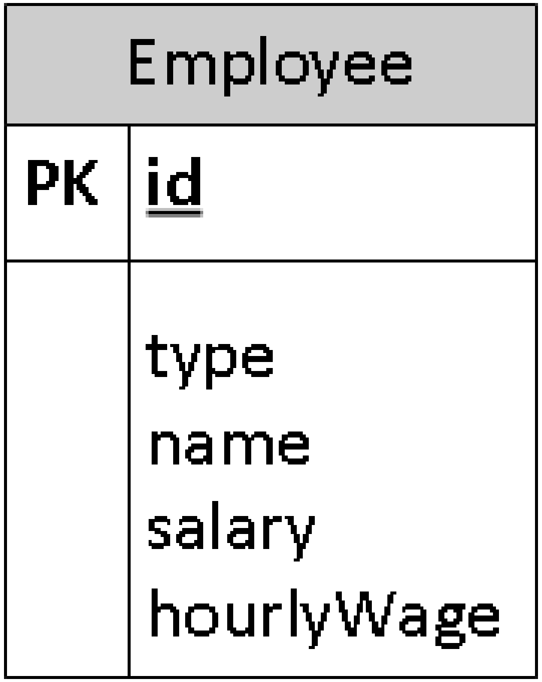
	id	name	salary	hourlyWage	type
1	1	This is a FTE	1000.0	(null)	F
2	2	This is a PTE	(null)	10.0	P

```
@Entity
@DiscriminatorValue("P")
public class PartTimeEmployee extends Employee {
    private double hourlyWage;

    public double getHourlyWage() { return hourlyWage; }
    public void setHourlyWage(double hourlyWage) { this.hourlyWage = hourlyWage; }
}
```

Defined discriminator value for PTE class

Mapping Inheritance - Single Table per Class Hierarchy

Entity Inheritance: Mapping



Employee	
PK	id
	type name salary hourlyWage

Mapping Inheritance - Single Table per Class Hierarchy

Entity Inheritance: Mapping

`@Inheritance`

- Defines the inheritance strategy to be used for an entity class hierarchy
- It is specified on the entity class that is the root of the entity class hierarchy
- Default strategy is `InheritanceType.SINGLE_TABLE`

Entity Inheritance: Mapping

`@DiscriminatorColumn`

- Specifies the discriminator column for the mapping `SINGLE_TABLE` and `JOINED` strategies
- Discriminator column is only specified in the root of an entity class hierarchy
- If the annotation is missing the name of the discriminator column defaults to `DTYPE` and discriminator type to `STRING`

Parameters:

- **`name`** - column name to be used for the discriminator
- **`discriminatorType`** - type column to use as discriminator
- **`length`** - column length for String-based discriminator types

Entity Inheritance: Mapping

`@DiscriminatorValue`

- Specifies the value of the discriminator column for entities of the given type
- Can only be specified on a concrete entity class
- If the annotation is not specified and discriminator column is used, a provider-specific function will be used to generate a value *(class name in Hibernate)*

Entity Inheritance: Mapping

Joined Subclass Strategy:

- Root of the class hierarchy is represented by a single table
- Each subclass is represented by a separate table that contains fields that are specific to this subclass
- The primary key column of the subclass table serves as foreign key to the primary key of the superclass table

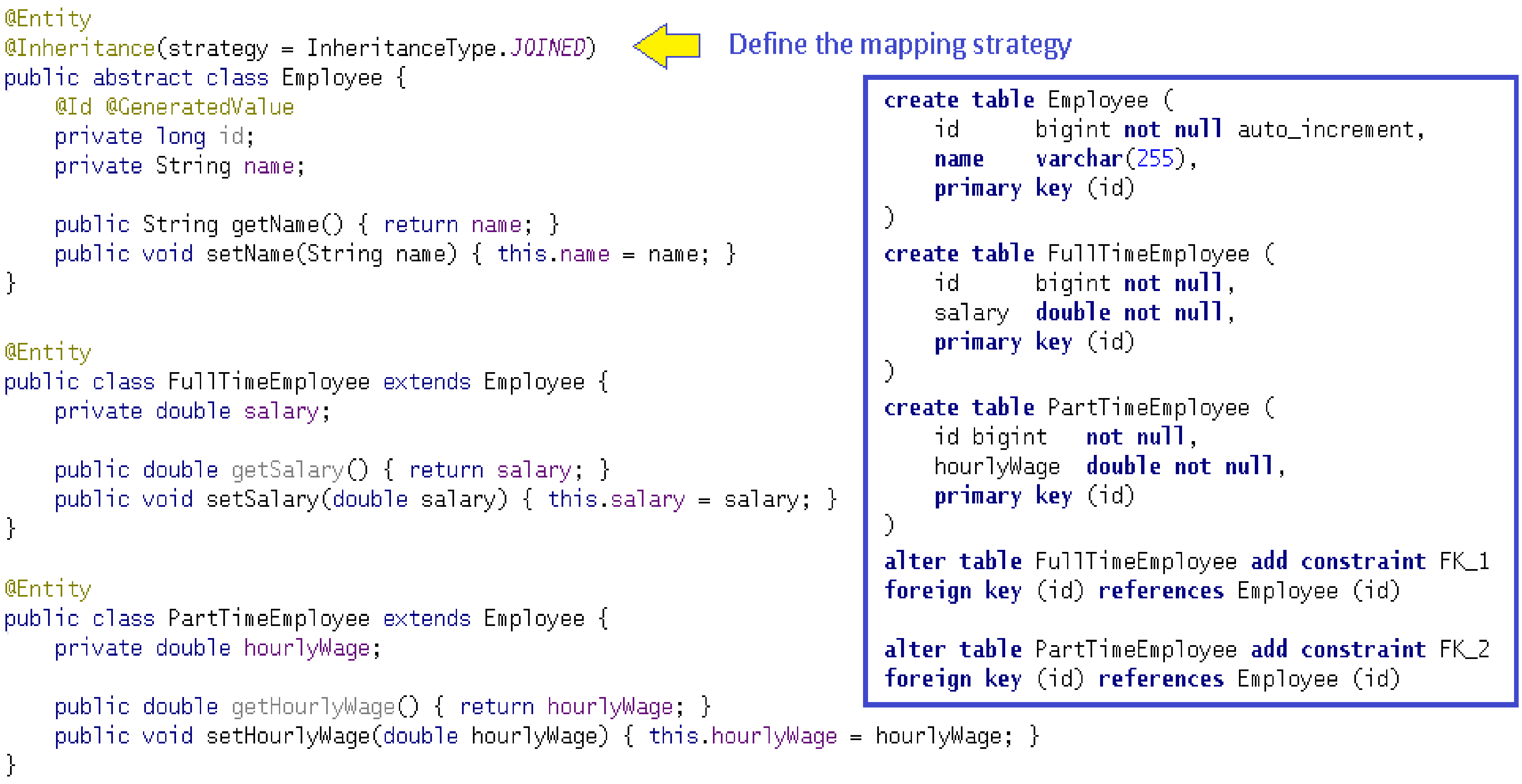
Benefits:

- Support for polymorphic relationships between entities

Drawbacks:

- Requires one or more join operations to be performed to instantiate instances of a subclass *(deeper hierarchy → more joins → bad performance)*

Entity Inheritance: Mapping



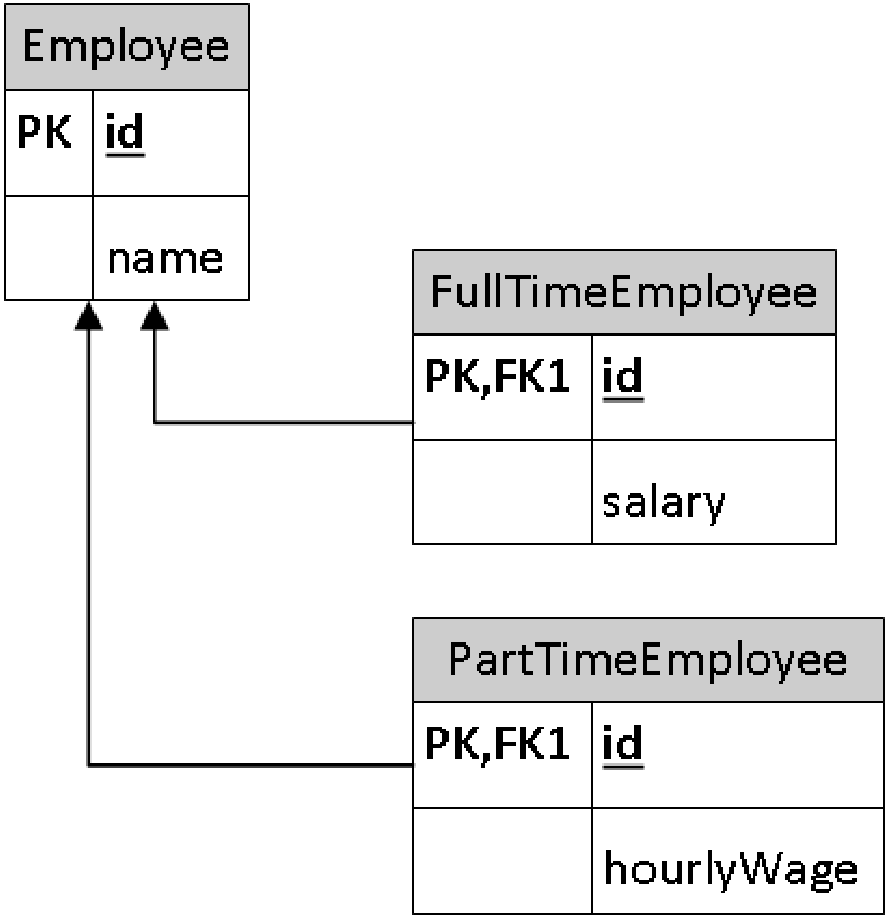
```java
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Employee {
    @Id @GeneratedValue
    private long id;
    private String name;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}

@Entity
public class FullTimeEmployee extends Employee {
    private double salary;

    public double getSalary() { return salary; }
    public void setSalary(double salary) { this.salary = salary; }
}

@Entity
public class PartTimeEmployee extends Employee {
    private double hourlyWage;

    public double getHourlyWage() { return hourlyWage; }
    public void setHourlyWage(double hourlyWage) { this.hourlyWage = hourlyWage; }
}
```

Define the mapping strategy

```sql
create table Employee (
    id       bigint not null auto_increment,
    name     varchar(255),
    primary key (id)
)
create table FullTimeEmployee (
    id       bigint not null,
    salary   double not null,
    primary key (id)
)
create table PartTimeEmployee (
    id bigint    not null,
    hourlyWage   double not null,
    primary key (id)
)
alter table FullTimeEmployee add constraint FK_1
foreign key (id) references Employee (id)

alter table PartTimeEmployee add constraint FK_2
foreign key (id) references Employee (id)
```

Mapping Inheritance - Joined Subclass Strategy

Entity Inheritance: Mapping



Employee	
PK	**id**
	name

FullTimeEmployee	
PK,FK1	**id**
	salary

PartTimeEmployee	
PK,FK1	**id**
	hourlyWage

Mapping Inheritance - Joined Subclass Strategy

Entity Inheritance: Mapping

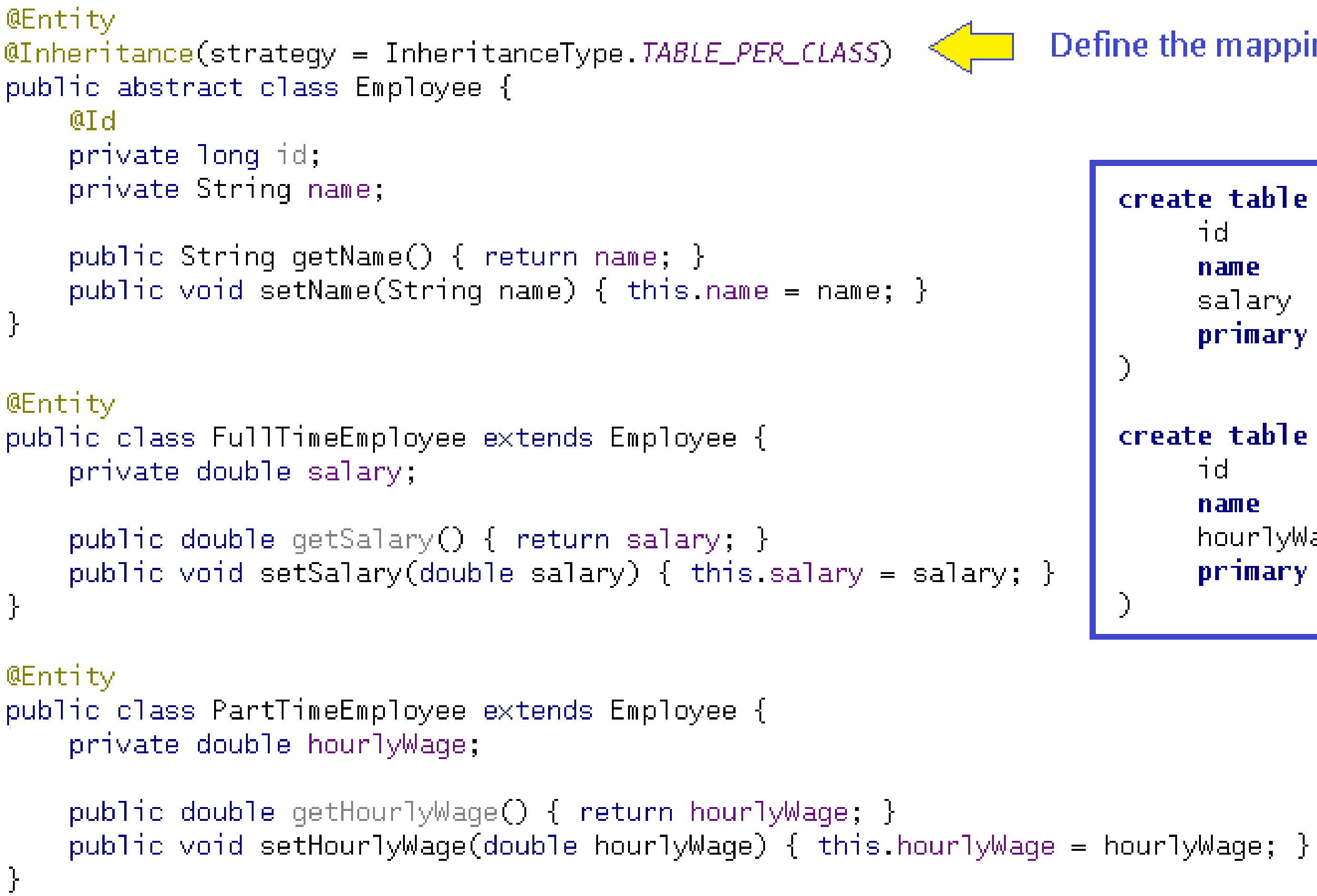
Table per Concrete Class Strategy:

- Each class is mapped to a separate table
- All properties of the class *(including inherited properties)* are mapped to columns of the table for the class

Drawbacks:

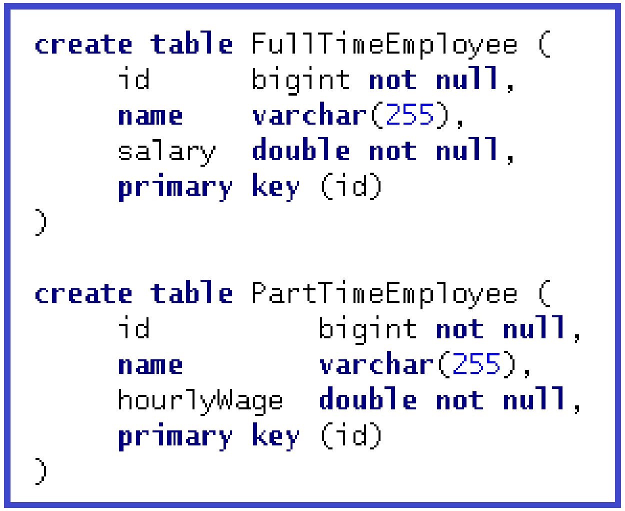
- Provides poor support for polymorphic relationships
- Typically requires SQL UNION for queries that are intended to range over the class hierarchy
- Possible problems with using ID generation strategies

Entity Inheritance: Mapping



```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Employee {
    @Id
    private long id;
    private String name;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}

@Entity
public class FullTimeEmployee extends Employee {
    private double salary;

    public double getSalary() { return salary; }
    public void setSalary(double salary) { this.salary = salary; }
}

@Entity
public class PartTimeEmployee extends Employee {
    private double hourlyWage;

    public double getHourlyWage() { return hourlyWage; }
    public void setHourlyWage(double hourlyWage) { this.hourlyWage = hourlyWage; }
}
```

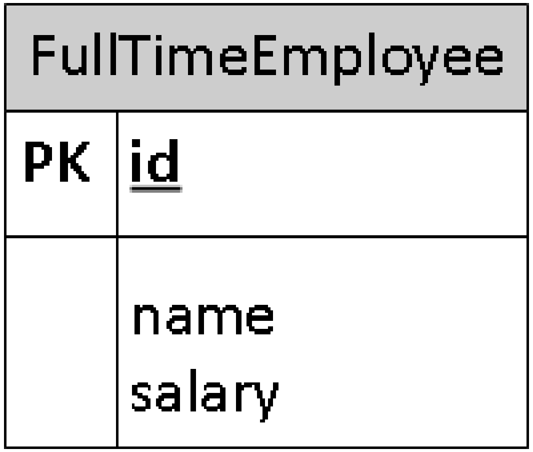
Define the mapping strategy



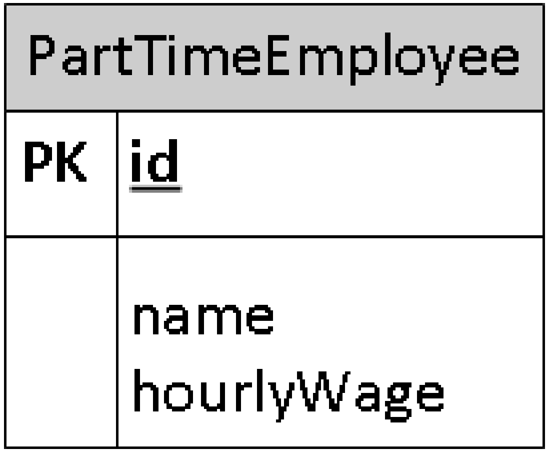
```
create table FullTimeEmployee (
     id        bigint not null,
     name      varchar(255),
     salary    double not null,
     primary key (id)
)

create table PartTimeEmployee (
     id            bigint not null,
     name          varchar(255),
     hourlyWage    double not null,
     primary key (id)
)
```

Mapping Inheritance - Table per Concrete Class Strategy

Entity Inheritance: Mapping



FullTimeEmployee	
PK	**id**
	name salary



PartTimeEmployee	
PK	**id**
	name hourlyWage

Mapping Inheritance - Table per Concrete Class Strategy

Practical work

Exercise 4:

Entities class hierarchy

Conversion

A common problem in storing values to the database is that the value desired in Java differs from the value used in the database (ex. Boolean to 0/1 or Yes/No)

JPA 2.1 provides conversion service:

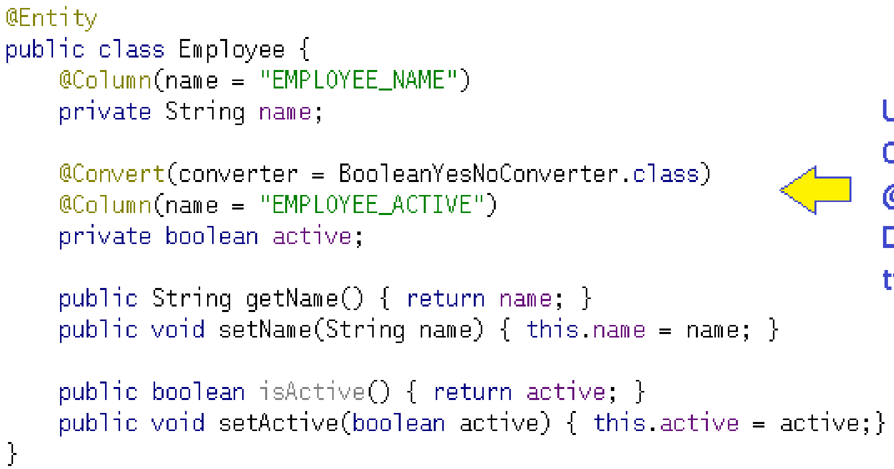
- Annotations **`@Converter`** and **`@Convert`**

  Are used to specify the conversion of field or property.

- Interface **`javax.persistence.AttributeConverter`**

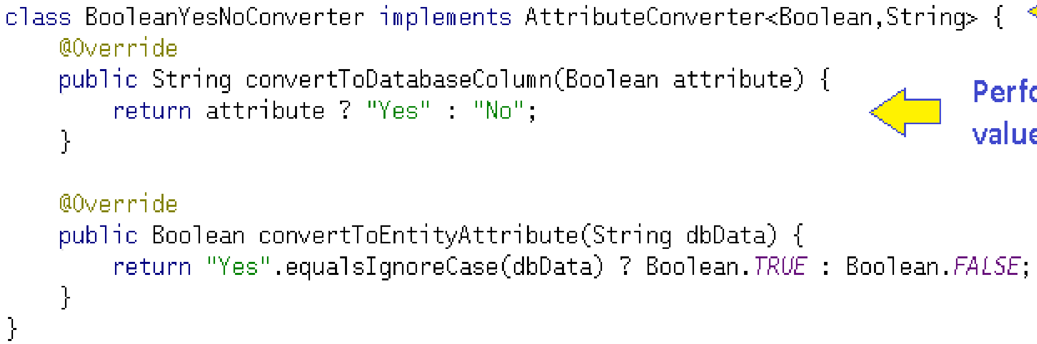
  Class that implements this interface can be used to convert entity attribute state into database column representation and back again.

Conversion



```
@Entity
public class Employee {
    @Column(name = "EMPLOYEE_NAME")
    private String name;

    @Convert(converter = BooleanYesNoConverter.class)
    @Column(name = "EMPLOYEE_ACTIVE")
    private boolean active;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public boolean isActive() { return active; }
    public void setActive(boolean active) { this.active = active;}
}
```

Use conversion for entity field. Conversion class is specified with @Convert annotation. Database column will have VARCHAR type in this case (not boolen)



```
class BooleanYesNoConverter implements AttributeConverter<Boolean,String> {
    @Override
    public String convertToDatabaseColumn(Boolean attribute) {
        return attribute ? "Yes" : "No";
    }

    @Override
    public Boolean convertToEntityAttribute(String dbData) {
        return "Yes".equalsIgnoreCase(dbData) ? Boolean.TRUE : Boolean.FALSE;
    }
}
```

Class implementing logic of entity attributes state conversion. AttributeConverter is parametrized with source and target types

Performs conversion from entity attribute value to database column value

Performs conversion from database column value to entity attribute value

Using conversion service provided by JPA 2.1

Thank you for your attention!

Questions?