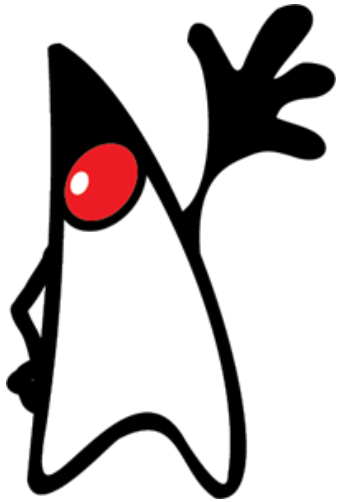
Inheritance



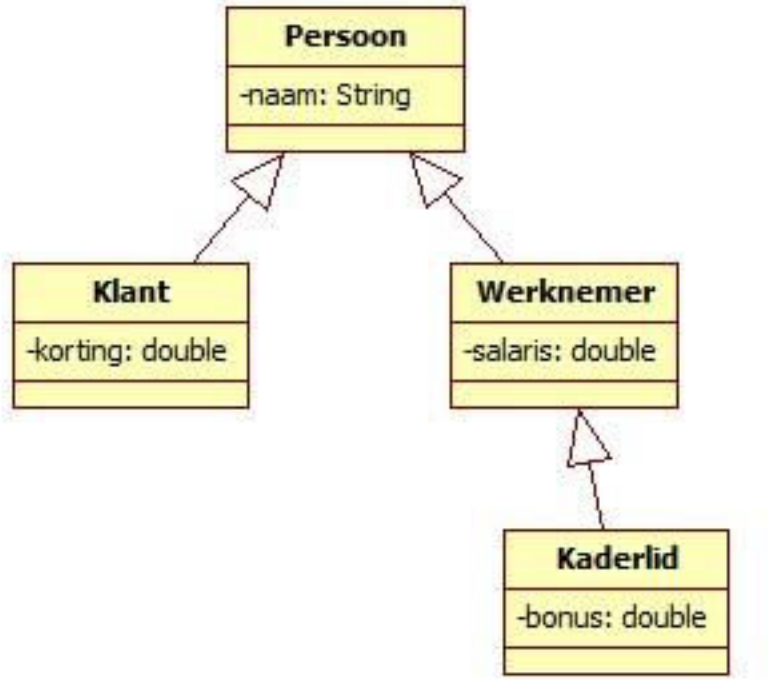
Contents

- Inheritance
  - Problem definition
  - Strategies

- Inheritance lesson examples with explanation
  - Implementing Inheritance: Start
  - Implementing Inheritance: SINGLE TABLE
  - Implementing Inheritance: JOINED
  - Implementing Inheritance: TABLE PER CLASS
  - Implementing Inheritance: Polymorphic Queries
  - Implementing Inheritance: @MappedSuperclass

Problem definition

- How do you deal with this class diagram? How do you implement this inheritance tree in JPA?



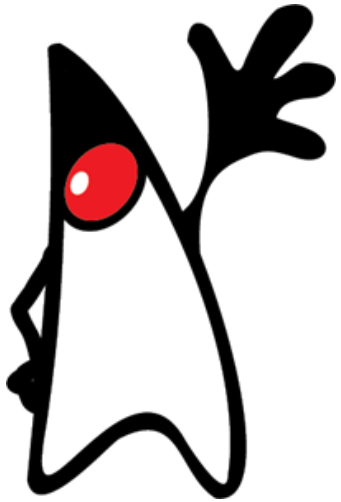
Strategies

- JPA foresees 3 inheritance strategies:
  1. One table for all classes in the hierarchy cfr. "modeling by supertype"
  2. A table per class with interrelationships
  3. A table per concrete class without interrelationships
     Cfr. "model by subtype" but also with a table for the supertype if it is not abstract
  4. 1 table per direct subclass if the superclass is not an Entity

  These are implemented with the annotation:

- @Inheritance with the attribute strategy with possibilities
  1. SINGLE_TABLE (default)
  2. JOINED
  3. TABLE_PER_CLASS
- @MappedSuperclass: if the Superclass is not an Entity

Implementing inheritance: START



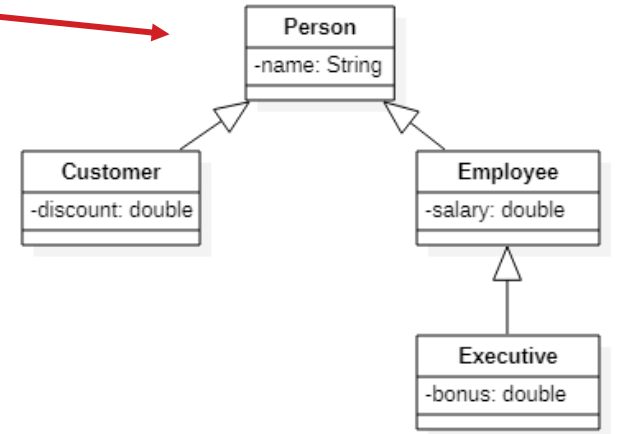
Content

- This presentation will guide you through the implementation of inheritance in a Spring Data JPA project.
- Follow these instructions carefully and you will learn what to do and how.
- The project that came with this item only serves in case you fail to make it yourself. It is best to use the (tele)coaching in that case...

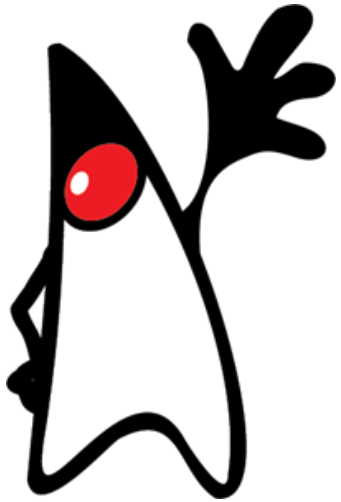
- Good luck!

Implementing the example: Inheritance

- Step 1: Create a web project
  - You can check this procedure again in the document on Canvas (How to Create a Project in IntelliJ.docx)
- Step 2: Create Entity "Person", "Customer", "Employee" and "Executive"
  - See presentation *3 JPA, keyword queries and JPQL* to know how to create an entity. Complete the entities with attributes as in this UML schema. Generate the no-arg constructor and getters and setters
  - Also encode the inheritance between the different entities
    - Tip: use "extends"
  - Complete / change the entities according to the chosen strategy
    - SINGLE_TABLE
    - JOINED
    - TABLE_PER_CLASS

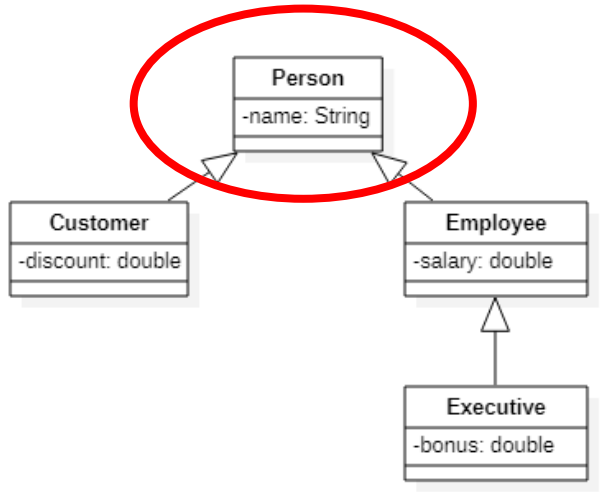


Inheritance strategy: SINGLE_TABLE



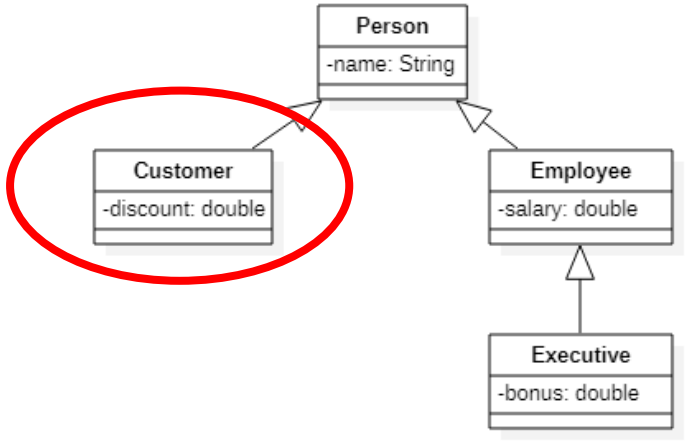
Change Person.java for SINGLE_TABLE implementation


```
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@Entity
public class Person {

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  private String name;
   ....
}
```



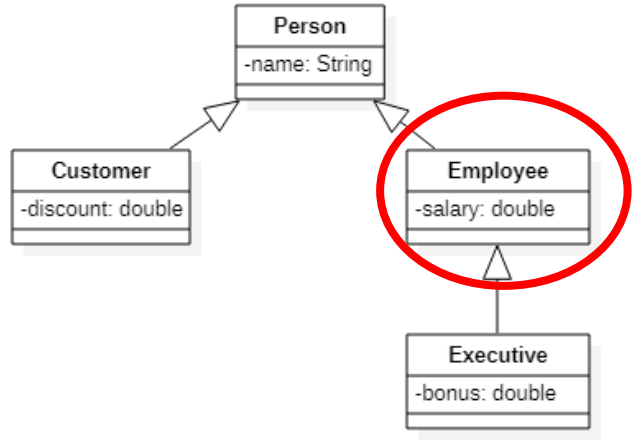
Change Customer.java for SINGLE_TABLE implementation

```
@Entity
public class Customer extends Person {
  private double discount;
  public Customer () {
  }
  public double getDiscount () {
    return discount;
  }
  public void setDiscount (double discount) {
    this. discount = discount;
  }
} //no attribute Id in this Entity because this is inherited from Person
```



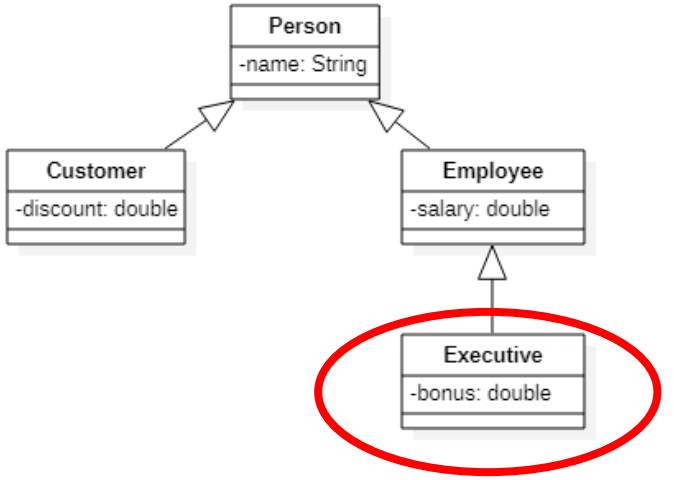
Change Employee.java for SINGLE_TABLE implementation

```
@Entity
public class Employee extends Person {
  private double salary;
  public Employee () {
  }
  public double getSalary() {
    return salary;
  }
  public void setSalary(double salary) {
    this. salary = salary;
  }
}//no attribute Id in this Entity because this is inherited from Person
```



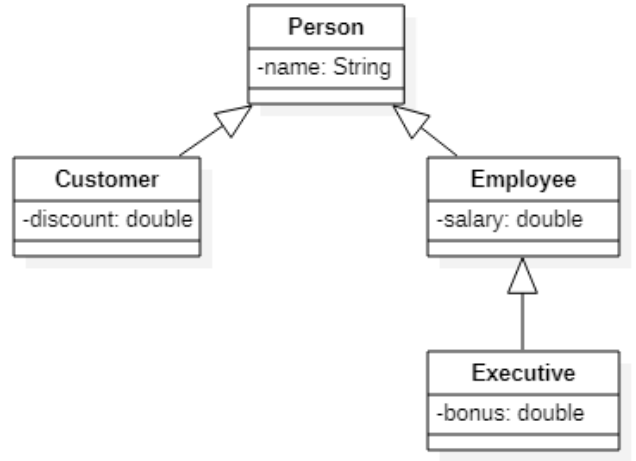
Change Executive .java for SINGLE_TABLE implementation

```
@Entity
public class Executive extends Employee {
  private double bonus;
  public Executive () {

  }
  public double getBonus() {
    return bonus;
  }
  public void setBonus(double bonus) {
    this.bonus = bonus;
  }
} //no attribute Id in this Entity because this is inherited from Person
```



Implementing the example : Inheritance

- Step 3: Create repository for the entity "Person"
  - See presentation *3 JPA, keyword queries and JPQL* to know how to do this
  - Via polymorphism (declare attribute as Person but instantiate as one of the subclasses) it is sufficient to create only PersonRepository (Customer Repository, EmployeeRepository and ExecutiveRepository are NOT necessary)
- Step 4: Create PersonController
  - See presentation *0a MVC with Spring and Thymeleaf* to know h this
  - => PersonController
    - has an association with Person Repository
    - contains the necessary methods to create the different objects and redirect to the different pages



Step 4: Create PersonController

- Now create the PersonController:
  - See presentation *0a MVC with Spring and Thymeleaf* to know how to do this
  - Make an association with PersonRepository
  - Create the following methods:
    - Method to show the **addPerson** page where you can enter the details (name) of a new Person
    - Method to, after forwarding the data of the **addPerson** page, create a new Person object with this data, save this person in the database and then show the index page
    - Method to show the **addCustomer** page where you can enter the details (name, discount) of a new Customer
    - Method to, after forwarding the data of the **addCustomer** page, create a new Customer object with this data, save this customer in the database and then show the index page
    - Method to show the **addEmployee** page where you can enter the details (name, salary) of a new Employee
    - Method to, after forwarding the data of the **addEmployee** page, create a new Employee object with this data, save this employee in the database and then show the index page
    - Method to show the **addExecutive** page where you can enter the details (name, salary, bonus) of a new executive
    - Method to, after forwarding the data from the **addExecutive** page, create a new Executive object with this data, save this executive in the database and then show the index page

Step 4: Create PersonController: code


```
@Controller
public class PersonController {
    private PersonRepository personRepository;

    public PersonController(PersonRepository personRepository) {
        this.personRepository = personRepository;
    }

    @RequestMapping("/")
    public String index() {
        return "index";
    }

    @RequestMapping("/addPerson")
    public String addPerson() {
        return "addPerson";
    }

    @RequestMapping("/processAddPerson")
    public String processAddPerson(Model model, HttpServletRequest request) {
        String name = request.getParameter("name");
        PERSON person = new Person();
        PERSON.setName(name);
        PERSONRepository.save(person);
        return "index";
    }

    @RequestMapping("/addCustomer")
    public String addCustomer() {
        return "addCustomer";
    }

    @RequestMapping("/processAddCustomer")
    public String processAddCustomer(Model model, HttpServletRequest request) {
        String name = request.getParameter("name");
        double discount = Double.parseDouble(request.getParameter("discount"));
        Customer customer = new Customer();
        customer.setName(name);
        customer.setDiscount(discount);
        personRepository.save(customer);
        return "index";
    }
```

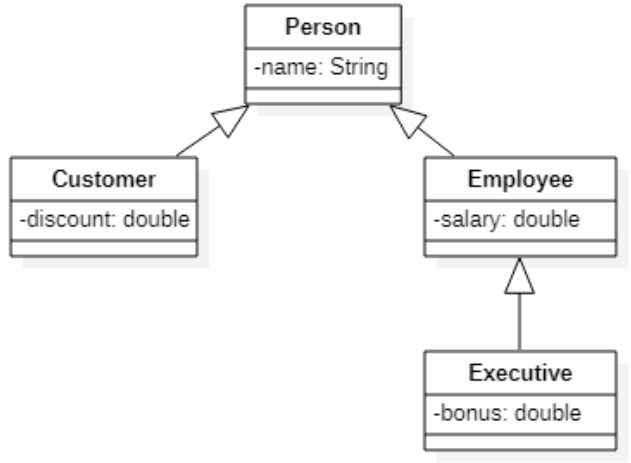
Step 4: Create PersonController: code (continuation)


```
@RequestMapping("/addEmployee")
    public String addEmployee () {
        return "addEmployee";
    }

    @RequestMapping("/processAddEmployee")
    public String processAddEmployee (Model model, HttpServletRequest request) {
        String NAME = request.getParameter("name");
        double salary =  Double.parseDouble(request.getParameter("salary"));
        EMPLOYEE employee = new Employee ();
        EMPLOYEE. setName(name);
        EMPLOYEE.setSalary(salary);
        PERSONRepository.save(employee);
        return "index";
    }
```

```
@RequestMapping("/addExecutive")
    public String addExecutive () {
        return "addExecutive";
    }

    @RequestMapping("/processAddExecutive")
    public String processAddExecutive (Model model, HttpServletRequest request) {
        String name = request.getParameter("name");
        double salary = Double.parseDouble(request.getParameter("salary"));
        double bonus = Double.parseDouble(request.getParameter("bonus"));
        Executive executive = new Executive();
        executive.setName (name);
        executive.setSalary (salary);
        executive.setBonus(bonus);
        personRepository.save(executive);
        return "index";
    }
}
```

Implementing the example: Inheritance

- Step 5: create user interface:
  - index.html
  - addPerson.html
  - addCustomer.html
  - addEmployee.html
  - addExecutive.html



Step 5: Create user interface index.html

Now we are left with creating a user interface. Create a new Thymeleaf HTML file index.html so that new people/customers/employees/executives can be registered:

```
<body>

<p></p><a href="/addPerson">Add a person </a></p>
<p><a href="/addCustomer">Add a customer</a></p>
<p><a href="/addEmployee">Add an employee</a></p>
<p><a href="/addExecutive">Add an executive</a></p>

</body>
```

Step 5: Create user interface addPerson.html

Now the pages addPerson.html, addCustomer.html, addEmployee.html and addExecutive.html still need to be added. We start with the body of addPerson.html:

```
<body>
<h1>Create Person</h1>

<form action="/processAddPerson" method="post">
    <p>
        <label for="name">Name:</label>
        <input type="text" name="name" id="name">
    </p>
    <p>
        <input type="submit" value="Save" name="save">
    </p>
</form>
</body>
```

Step 5: Create user interface addCustomer.html

We work out the body of addCustomer.html in a similar way:

```
<body>
<h1>Create Customer</h1>

<form action="/processAddCustomer" method="post">
    <p>
        <label for="name">Name:</label>
        <input type="text" name="name" id="name">
    </p>
    <p>
        <label for="discount">Discount:</label>
        <input type="text" name="discount" id="discount">
    </p>
    <p>
        <input type="submit" value="Save" name="save">
    </p>
</form>
</body>
```

Step 5: Create user interface addEmployee.html

We work out the body of addEmployee.html in a similar way:

```
<body>
<h1>Create Employee</h1>

<form action="/processAddEmployee" method="post">
    <p>
        <label for="name">Name:</label>
        <input type="text" name="name" id="name">
    </p>
    <p>
        <label for="salary">Salary:</label>
        <input type="text" name="salary" id="salary">
    </p>
    <p>
        <input type="submit" value="Save" name="save">
    </p>
</form>
</body>
```

Step 5: Create user interface addExecutive.html

We work out the body of addExecutive.html in a similar way:

```
<body>
<h1>Add a new executive</h1>

<form action="/processAddExecutive" method="post">
    <p>
        <label for="name">Name:</label>
        <input type="text" name="name" id="name">
    </p>
    <p>
        <label for="salary">Salary:</label>
        <input type="text" name="salary" id="salary">
    </p>
    <p>
        <label for="bonus">Bonus:</label>
        <input type="text" name="bonus" id="bonus">
    </p>
    <p>
        <input type="submit" value="Save" name="save">
    </p>
</form>
</body>
```

Testing SINGLE_TABLE inheritance strategy

We run the web app and add the following Customer, Employee and Executive:

Add Customer

Name: Boermans Kitchens

Discount: 6.5

Save

Add Employee

Name: Joke Wens

Salary: 1568.2

Save

Add executive

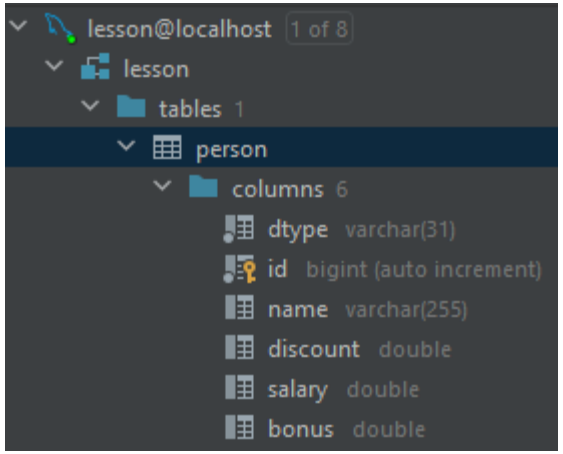
Name: Karel Oppers

Salary: 5400

Bonus: 8520.4

Save

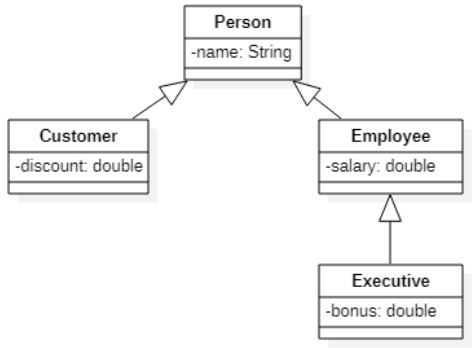
Result database SINGLE_TABLE inheritance strategy



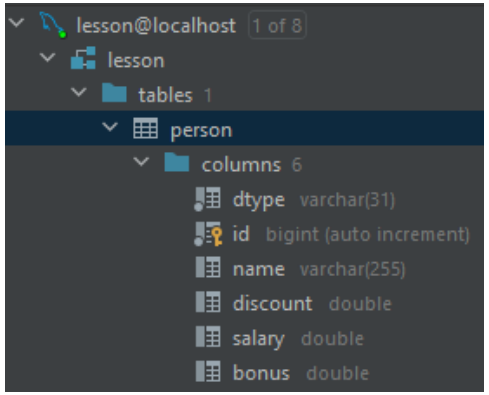
	dtype	id	name	discount	salary	bonus
1	Customer	1	Boermans Kitchens	6.5	<null>	<null>
2	Employee	2	Joke Wens	<null>	1588.2	<null>
3	Executive	3	Karel Oppers	<null>	5400	8520.4

Explanation Strategy SINGLE_TABLE

```java
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@Entity
public class Person {
    2 usages
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;
    2 usages
    private String name;
```



- SINGLE_TABLE is the default inheritance strategy. So if you do not indicate a strategy, by default the SINGLE_TABLE strategy will be used. The PERSON table will contain the following columns:
- **DTYPE**: het discriminatortype waarmee je kan herkennen welk soort object het is.
- **ID**
- **NAME**
- **DISCOUNT**
- **SALARY**
- **BONUS**



The columns correspond to all attributes of the superclass and all attributes of the subclasses + the column DTYPE.

	dtype	id	name	discount	salary	bonus
1	Customer	1	Boermans Kitchens	6.5	<null>	<null>
2	Employee	2	Joke Wens	<null>	1588.2	<null>
3	Executive	3	Karel Oppers	<null>	5400	8520.4

Strategy SINGLE_TABLE additional annotations

- In addition to @**Inheritance**, additional annotations are possible to indicate how the objects in the database will be stored.

```
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="PERSONTYPE", discriminatorType = DiscriminatorType.STRING)
@DiscriminatorValue("Person")
@Entity
public class Person {
```

- With the **name** attribute of the annotation **@DiscriminatorColumn** we can specify which name the column in which the discriminator type is tracked should be given (in this example PERSONTYPE).
- Via the **discriminatorType** attribute, we can indicate how this type should be tracked (STRING, INTEGER or CHAR). (In this example STRING)
- Through this discriminator column we can find out to which subclass the person in the given row belongs. The value that ends up in this column can be indicated with the annotation **@DiscriminatorValue**. You can also add this last annotation to each subclass

	persontype	id	name	discount	salary	bonus
1	Customer	1	Boermans Kitchens	6.5	<null>	<null>

- **ATTENTION:** this discriminator column is only needed at database level and may NOT be included as an attribute in the corresponding class. In this case, we do not create an attribute persontype in the person class. Every person object "knows" to which subclass it belongs. How you can request this we explain on slide 47.

Implementation SINGLE_TABLE additional annotations

Summary:

- Additional Annotations in super class

```
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn (name = "PERSONTYPE", discriminatorType = DiscriminatorType.STRING)
@DiscriminatorValue("PERSON")
```

- Additional Annotations in subclass

```
@DiscriminatorValue("Klant")
```

- NOTE: These annotations are **not mandatory**
  - SINGLE_TABLE is the default strategy
  - When no DiscriminatorColumn is indicated
    => discriminator column is named Dtype
  - When no DicriminatorValue is indicated
    => discriminatorvalue = class name

Strategy SINGLE_TABLE

- Advantages:
  - The inheritance strategy SINGLE_TABLE gives the best performance because:
    - No joins are needed to retrieve a person
    - When writing a person object to the database, only 1 insert or update statement is required
- Disadvantages:
  - The table can become very wide with many null values
  - This strategy presents problems in expanding the inheritance hierarchy.
    - You can't just add columns to an existing table
      - => more difficult to add subclasses (with attributes) to the hierarchy
      - Adding an attribute to 1 subclass affects the table where ALL persons are stored
      - If you make such a change, you will have to remove the table from the database and have it regenerated!

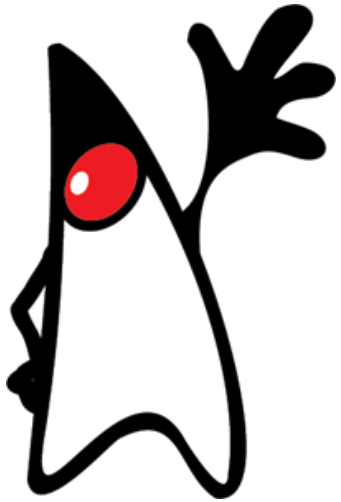
Strategy SINGLE_TABLE

- **Conclusion**:

  - The inheritance strategy SINGLE_TABLE is a **good choice** if:
    - Objects within the hierarchy mainly differ in behavior (and not in properties)
    - The classes within the hierarchy are stable (little chance of changes)

  - The inheritance strategy SINGLE_TABLE is a **bad choice** if:
    - Objects within the hierarchy have very different properties
    - It is very likely that changes will be made to the hierarchy in the future.

Implementing inheritance: JOINED



Implementing the example : Inheritance

- The first example is ready!
- We now make a second example where we apply the inheritance strategy **JOINED** instead of **SINGLE TABLE**
- **Copy** the project you just created. In the following slides you will continue to work on this copy

Modify Person.java for inheritance strategy JOINED

We start from a copy of the example made to demonstrate the joined inheritance strategy.
To this end, we take the following steps:

- **Remove** the newly created tables from the database
- Adjust the **annotations in the Person** class as follows:

```
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@Entity
public class Person{
```

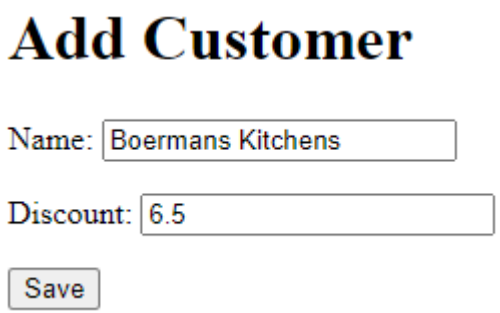
becomes

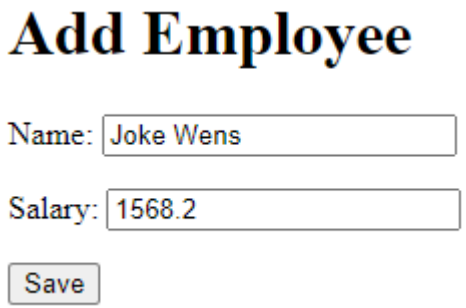
```
@Inheritance(strategy = InheritanceType.JOINED)
@Entity
public class Person{
```

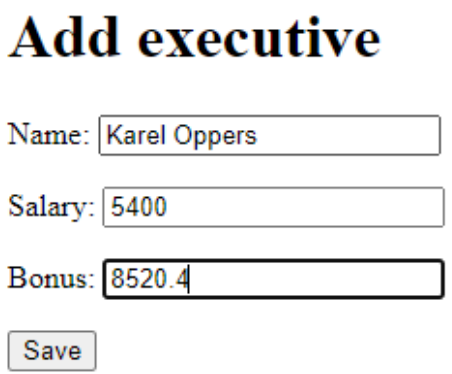
In the subclasses, we don't have to change anything.

Testing inheritance strategy JOINED

We are running the web app again and add the following Customer, Employee and Executive:



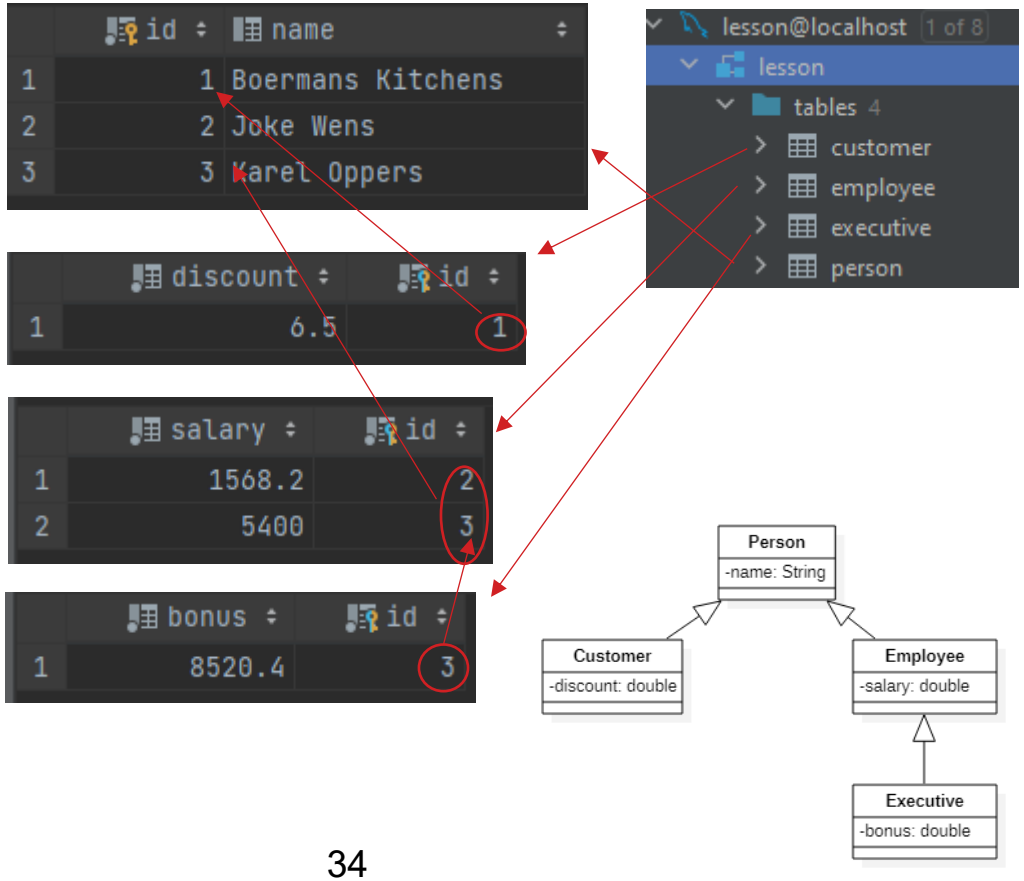




Strategy JOINED: result database

This inheritance strategy results in a table per class with the following characteristics:

- The table PERSON contains the common attributes:
  - ID
  - NAME
- Table CUSTOMER contains
  - ID (foreign key to ID of superclass PERSON)
  - Discount
- Table EMPLOYEE contains
  - ID (foreign key to ID of superclass PERSON)
  - SALARY
- Table EXECUTIVE contains
  - ID (foreign key to ID of superclass EMPLOYEE)
  - BONUS



Implementation JOINED

Summary:

- Strategy JOINED must be declared mandatory for superclass.
  No discriminator type and value needed (but allowed)

```
@Inheritance(strategy = InheritanceType.JOINED)
@Entity
public class PERSON {
. . .
}
```

- With the JOINED strategy, no Additional Annotations are required in the subclass (unless you want to specify a custom tableNAME)

```
@Entity
public class Kaderlid extends EMPLOYEE {
  . . .
}
```

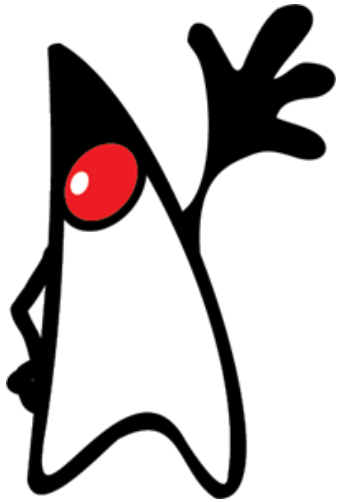
Strategy JOINED

- Advantages:
  - The resulting tables do not contain redundant data
  - When you want to add a subclass, this has no impact on the already existing tables

- Disadvantages:
  - Worse performance
    - A child join is always needed to request information from a subclass
    - Multiple INSERT or UPDATE statements are required to store an object of a subclass in the database

- Conclusion:
  - Good choice if:
    - Subclasses have many unique attributes
    - It is very likely that changes will still be made to the inheritance hierarchy.

Implementing Inheritance: TABLE PER CLASS



Implementing the example : Inheritance

- The second example is now also ready!
- We are now making a third and final example where we apply the **TABLE PER CLASS** inheritance strategy.
- **Copy** the project you just created. In the following slides you will continue to work on this copy

Change PERSON.java for inheritance strategy TABLE_PER_CLASS

We start from a copy of the example made to demonstrate the inheritance strategy TABLE PER CLASS. To this end, we take the following steps:

- **Remove** the newly created tables from the database
- **Adjust** the annotations in the PERSON class as follows:

```
@Inheritance(strategy = InheritanceType.JOINED)
@Entity
public class PERSON{
      @Id
      @GeneratedValue(strategy = GenerationType.AUTO)
      private long id;
```

becomes

```
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
@Entity
public class PERSON{
@Id
@GeneratedValue(strategy = GenerationType.TABLE)
   private long id;
```

So we also adjust the autonumbering generation because GenerationType.AUTO cannot be combined with the Table Per Class inheritance strategy and Mysql.

In the subclasses we don't have to change anything

Testing TABLE_PER_CLASS inheritance strategy

We run the web app again and add the following Customer, Employee and Executive:

Add Customer

Name: Boermans Kitchens

Discount: 6.5

Save

Add Employee

Name: Joke Wens

Salary: 1568.2

Save

Add executive

Name: Karel Oppers

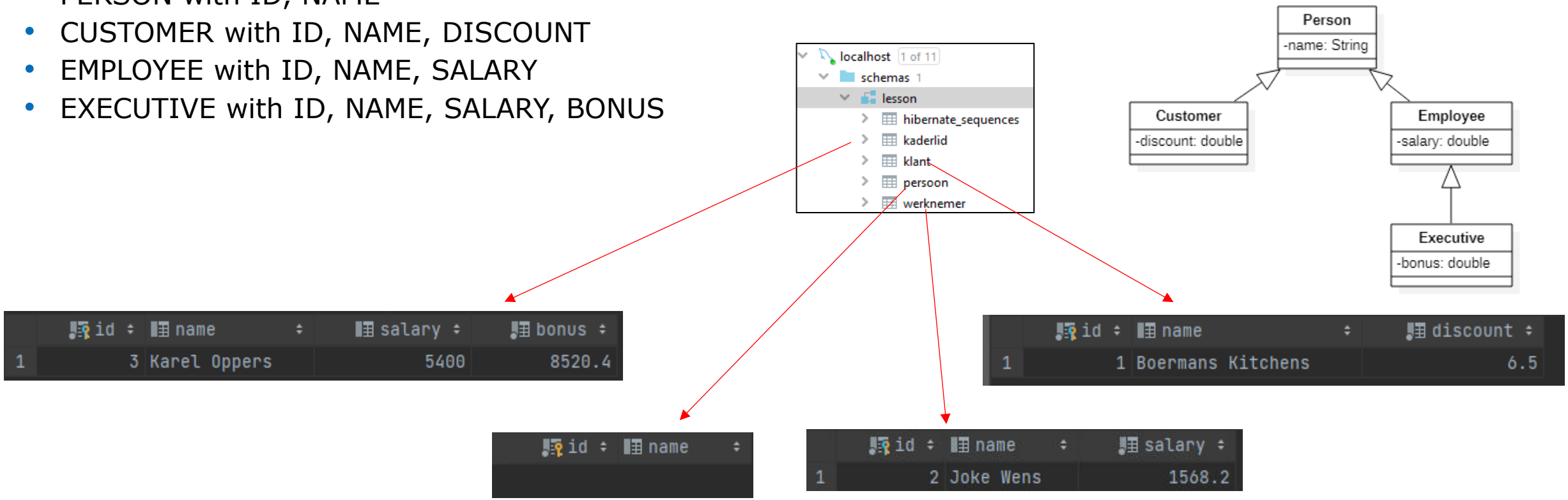
Salary: 5400

Bonus: 8520.4

Save

Result database inheritance strategy TABLE_PER_CLASS

- This inheritance strategy results in a table per class with a column for each attribute. In this example, the following tables:
  - PERSON with ID, NAME
  - CUSTOMER with ID, NAME, DISCOUNT
  - EMPLOYEE with ID, NAME, SALARY
  - EXECUTIVE with ID, NAME, SALARY, BONUS



Implementation TABLE_PER_CLASS

SUMMARY:

- Strategy TABLE_PER_CLASS must be indicated to the superclass.
  You also have to adjust the strategy of the autonumbering to GenerationType.TABLE if you work with MySQL

```
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
@Entity
public class PERSON{
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private long id;
```

- This strategy does not require additional annotations in the subclass

```
@Entity
public class EXECUTIVE extends EMPLOYEE{
  . . .
}
```

Strategy TABLE_PER_CLASS

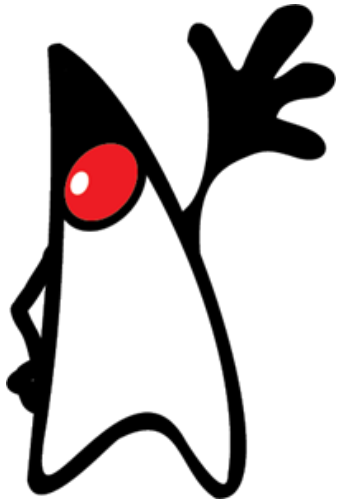
- Advantages:
  - Better performance:
    - no joins needed to query an object of a particular class
    - In case of change only 1 insert or update statement is needed

- Disadvantages:
  - Many tables
  - Sql "UNION ALL" needed to request all persons

- Conclusion:
  - Good choice if:
    - It is very likely that changes will still be made to the inheritance hierarchy.

- This strategy is usually unnecessary and is only supported by Hibernate

Implementing Inheritance: POLYMORPHIC QUERIES



Polymorphic queries

- A query returns all instances that meet a condition, including instances of subclasses:
  select x from EMPLOYEE x where x.salary > 3500 returns EMPLOYEES but also all EXECUTIVES who meet this salary condition.
- PERSONRepository.findAll() returns all Persons, including all CUSTOMERS, EMPLOYEES and EXECUTIVES
-

Polymorphic queries

- Work this out in one of the 3 inheritance projects you have made for this (single table, joined or table per class). In the examples on canvas you can find this implementation in the "single table" project.
- We now want to show all persons on the homepage. We first need to make sure that we have access to a list of all people and for this purpose, in PERSONController, adjust ANY method that refers to the index page so that it receives a list of people:

```
    (…)
    List<PERSON> list = PERSONRepository.findAll();
    model.addAttribute("personlist", list);
    return "index";
}
```

- In index.html we add the following code to the body:

```
<ol>
  <li th:each="PERSON : ${personlist}"><span th:text="${PERSON.getNAME()}"/> </li>
</ol>
```

Polymorphic queries

- We now want to further refine this list on the index page:
  - With an EMPLOYEE we want to mention his SALARY
  - With a CUSTOMER his DISCOUNT
  - For an EXECUTIVE his BONUS
- If you want to find out to which class an object belongs, you can check that by calling **.class.simpleName** on that object

Polymorphic queries

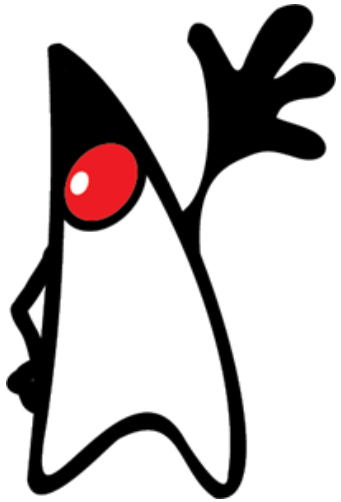
We adjust the code in index.html as follows:

```
<ol>
    <li th:each="PERSON : ${personlist}"><span th:text="${PERSON.getNAME()}"/>
        <span th:if="${PERSON.class.simpleName == 'Customer'}"
            th:text="'gets ' + ${PERSON.getDiscount()}  + ' Discount'" />
        <span th:if="${PERSON.class.simpleName == 'Executive'}"
            th:text="'has a bonus of ' + ${PERSON.getBonus()}" />
        <span th:if="${PERSON.class.simpleName == 'Employee'}"
            th:text="'earns ' + ${PERSON.getSalary()}" />
    </li>
</ol>
```

Result:

1. Karel Oppers has a bonus of 8520.4
2. Boermans Kitchens gets 6.5 discount
3. Joke Wens earns 1568.2

@MappedSuperclass



@MappedSuperclass

- What if the superclass of the inheritance tree is not an Entity (and therefore should not be tracked in the database)?
- In that case, use the annotation @MappedSuperclass

```
@MappedSuperclass
public class Person {
```

- This corresponds to an inheritance strategy SINGLE_TABLE applied to each of the subclasses of the PERSON class.