Day 24- Facilitation Guide

Index

For (1.5 hrs) ILT

During the previous session, we explored Embeddable Annotation and Hibernate Association.

We covered a variety of important concepts:

Embeddable Annotation in Hibernate:
The @Embeddable annotation in Hibernate is used to mark a class as an embeddable class. Embeddable classes represent components of an entity and are typically used to group related properties together. These classes do not have an identity of their own and are embedded within an entity's table. They are often used to simplify complex entity structures and improve code organization.

One-to-One Mapping in Hibernate:
One-to-One mapping in Hibernate is a relationship between two entities where each instance of one entity is associated with exactly one instance of another entity, and vice versa. This can be achieved using the @OneToOne annotation in Hibernate. One-to-One relationships can be unidirectional or bidirectional.

One-to-Many Mapping in Hibernate:
One-to-Many mapping in Hibernate represents a relationship where one entity is associated with multiple instances of another entity. This can be implemented using the @OneToMany annotation in Hibernate. In a one-to-many relationship, the owning entity (the "one" side) holds a collection or a set of the associated entities (the "many" side).

In this session, we will delve into Complex HQL queries,named query and inheritance in hibernate.

I. Introduction to HQL queries

Hibernate Query Language (HQL) is an easy to learn and powerful query language designed as an object-oriented extension to SQL that bridges the gap between the object-oriented systems and relational databases. The HQL syntax is very similar to the SQL syntax. It has a rich and powerful object-oriented query language available with the Hibernate ORM, which is a technique of mapping the objects to the databases. The data from object-oriented systems are mapped to relational databases with a SQL-based schema.
HQL can also be used to retrieve objects from database through O/R mapping by performing the following tasks:

- Apply restrictions to properties of objects
- Arrange the results returned by a query by using the order by clause
- Paginate the results
- Aggregate the records by using group by and having clauses
- Use Joins
- Create user-defined functions
- Execute subqueries

The Hibernate Query Language, designed as a "minimal" object-oriented extension to SQL provides an elegant bridge between Java classes and database tables. Hence it is mandatory to understand the needs of HQL in the current scenario.

Food for thought..

The trainer can ask the students the following question to engage them in a discussion:

Why do you believe HQL is preferred over SQL?

Features of HQL

Hibernate Query Language (HQL) comes with several features and capabilities that make it a valuable choice when working with Hibernate and relational databases. Here are some key features of HQL:

- Object-Oriented Approach: HQL is designed to work with Java objects and their properties rather than dealing with tables and columns directly, making it more intuitive for Java developers.

- Entity-Centric: HQL is centered around entity classes that map to database tables. These entity classes are annotated with Hibernate annotations to specify the mapping between the database schema and Java objects.

- SQL-Like Syntax: HQL uses a syntax that is reminiscent of SQL, allowing developers to write queries with a familiar feel. However, HQL uses entity class names and their properties instead of table and column names.

- Joins and Relationships: HQL supports navigating and querying relationships between entities, including one-to-one, one-to-many, and many-to-many associations. You can perform joins and retrieve related data easily.

- Parameterized Queries: HQL allows for parameterized queries, which enhances query flexibility and security. You can use placeholders for parameters and bind values to them dynamically.

- Named Queries: You can define named queries in Hibernate, which are preconfigured HQL queries stored in XML configuration files or as annotations. Named queries promote code reusability and maintainability.

- Aggregations: HQL supports various aggregate functions, such as SUM, AVG, MIN, MAX, and COUNT, enabling you to perform calculations on query results.

- Pagination: HQL provides features for result pagination, which is important for handling large data sets. You can limit the number of rows returned and specify an offset.

- Subqueries: HQL supports subqueries, allowing you to nest queries within queries, providing more advanced and complex query capabilities.

- Inheritance Queries: HQL can handle inheritance hierarchies in object models, making it possible to query polymorphic entities efficiently.

- Polymorphic Queries: You can perform polymorphic queries that retrieve data from subclasses as well as the superclass in an inheritance hierarchy.

- Support for Caching: Hibernate can cache query results, improving query performance and reducing database load.

- Integration with Hibernate Criteria API: HQL can be integrated with the Hibernate Criteria API, offering another way to create queries programmatically without writing HQL strings.

- Database Portability: HQL is database-agnostic. Hibernate generates appropriate SQL statements based on the underlying database, making it easier to switch between different database systems.

- Dynamic Generation: HQL queries can be generated dynamically in response to user input or application requirements, making your application more flexible.

- Performance Optimization: Hibernate's query execution plan and caching mechanisms help optimize query performance.

HQL Syntax
HQL is considered to be the most powerful query language designed as a minimal object-oriented extension to SQL. HQL queries are easy to understand and use persistent class and property names, instead of table and column names. HQL consists of the following elements.

- Clauses (FROM, Select, Where, Order by, Group by)
- Associations and joins (Inner join, Left outer join, Right outer join, Full join)
- Aggregate functions (avg, sum, min, max, count, etc.)
- Expressions (Mathematical operators, Binary comparison, String concatenation, SQL scalar function, etc.)
- Subqueries (any, all, some, in)

Note: you can use SQL statements directly with Hibernate using Native SQL, but I would recommend to use HQL whenever possible to avoid database portability hassles, and to take advantage of Hibernate's SQL generation and caching strategies.

Let's explore HQL through a hands-on demonstration.

1. **FROM Clause**

You will use FROM clause if you want to load complete persistent objects into memory.

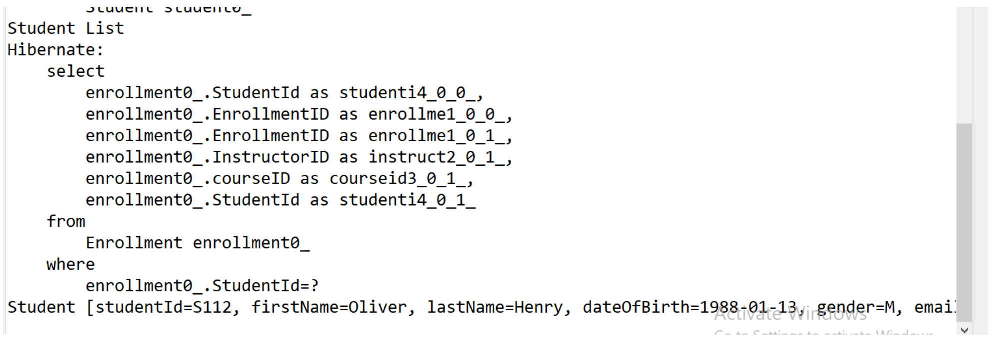
```
//From Clause
String hql="from Student";
```

```
Query q=session.createQuery(hql);
```

Complete Example:
We already have a student database; now, let's retrieve information from the Student table

```
package com.sms;
import java.time.LocalDate;
import java.util.List;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.query.Query;
public class HqlOperation {
	public static void main(String[] args) {
		// Obtain a Hibernate SessionFactory
		SessionFactory factory = HibernateUtil.getSessionFactory();
		// Open a new session
		Session session = factory.openSession();

		// Begin a transaction
		Transaction transaction = session.beginTransaction();
		//From Clause
		String hql="from Student";
		Query q=session.createQuery(hql);
		List<Student> students=q.getResultList();

		System.out.println("Student List");
		for(Student s:students)
		{
			System.out.println(s);
		}

	}
}
```

Output:



```
        Student student0_
Student List
Hibernate:
    select
        enrollment0_.StudentId as studenti4_0_0_,
        enrollment0_.EnrollmentID as enrollme1_0_0_,
        enrollment0_.EnrollmentID as enrollme1_0_1_,
        enrollment0_.InstructorID as instruct2_0_1_,
        enrollment0_.courseID as courseid3_0_1_,
        enrollment0_.StudentId as studenti4_0_1_
    from
        Enrollment enrollment0_
    where
        enrollment0_.StudentId=?
Student [studentId=S112, firstName=Oliver, lastName=Henry, dateOfBirth=1988-01-13, gender=M, emai
```

Here's a breakdown of what's happening in the code:

- **String hql = "from Student";:** This line defines an HQL query as a string. The query is "from Student," which means it's fetching all records from the "Student" entity (table) in the database. This query corresponds to a simple SELECT statement in SQL.

- **Query q = session.createQuery(hql);:** This line creates a Hibernate Query object by passing the HQL string to createQuery() method. The session object represents a Hibernate session, and it's used to interact with the database.

- **List<Student> students = q.getResultList();:** This line executes the HQL query and retrieves the result set. The getResultList() method returns a list of Student objects, representing the rows retrieved from the "Student" table.

The code then proceeds to print the list of students using a for loop. It iterates through the students list and prints each student object.

2. SELECT Clause

The SELECT clause provides more control over the result set then the form clause. If you want to obtain a few properties of objects instead of the complete object, use the SELECT clause. Following is the simple syntax of using SELECT clause to get just FirstName field of the Student object −

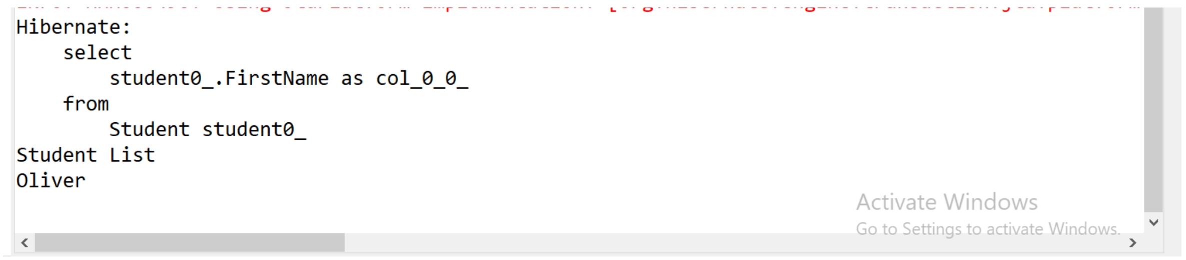
```
//Select Clause
String hql="Select s.firstName from Student s";
Query q=session.createQuery(hql);
List<String> students=q.getResultList();
```

Note: s.firstName is a property of Student object rather than a field of the Student table.

Complete Example:

```
package com.sms;
import java.time.LocalDate;
import java.util.List;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.query.Query;
public class HqlOperation {
	public static void main(String[] args) {
		// Obtain a Hibernate SessionFactory
		SessionFactory factory = HibernateUtil.getSessionFactory();
		// Open a new session
		Session session = factory.openSession();
		// Begin a transaction
		Transaction transaction = session.beginTransaction();
		// From Clause
		String hql = "Select s.firstName from Student s";
		Query q = session.createQuery(hql);
		List<String> students = q.getResultList();
		System.out.println("Student List");
		for (String s : students) {
			System.out.println(s);
		}
	}
}
```

Output:



```
Hibernate:
    select
        student0_.FirstName as col_0_0_
    from
        Student student0_
Student List
Oliver
```

Question: Explain the primary purpose of using HQL in Hibernate.

II. Executing some complex queries using HQL

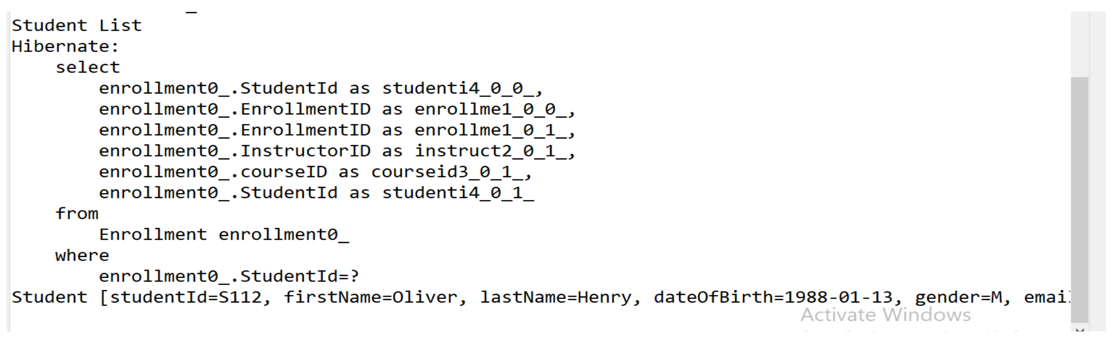
WHERE Clause

If you want to narrow the specific objects that are returned from storage, you use the WHERE clause.

Complete Example:

```
package com.sms;
import java.util.List;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.query.Query;
public class HqlOperation {
        public static void main(String[] args) {
                // Obtain a Hibernate SessionFactory
                SessionFactory factory = HibernateUtil.getSessionFactory();
                // Open a new session
                Session session = factory.openSession();
                // Begin a transaction
                Transaction transaction = session.beginTransaction();
                // Where Clause
                String hql = "from Student s where s.studentId=:i";
                Query q = session.createQuery(hql);
                q.setParameter("i", "S112");
                List<Student> students = q.getResultList();
                System.out.println("Student List");
                for (Student s : students) {
                        System.out.println(s);
                }


        }
}
```

Output:



```
Student List
Hibernate:
    select
        enrollment0_.StudentId as studenti4_0_0_,
        enrollment0_.EnrollmentID as enrollme1_0_0_,
        enrollment0_.EnrollmentID as enrollme1_0_1_,
        enrollment0_.InstructorID as instruct2_0_1_,
        enrollment0_.courseID as courseid3_0_1_,
        enrollment0_.StudentId as studenti4_0_1_
    from
        Enrollment enrollment0_
    where
        enrollment0_.StudentId=?
Student [studentId=S112, firstName=Oliver, lastName=Henry, dateOfBirth=1988-01-13, gender=M, emai
```

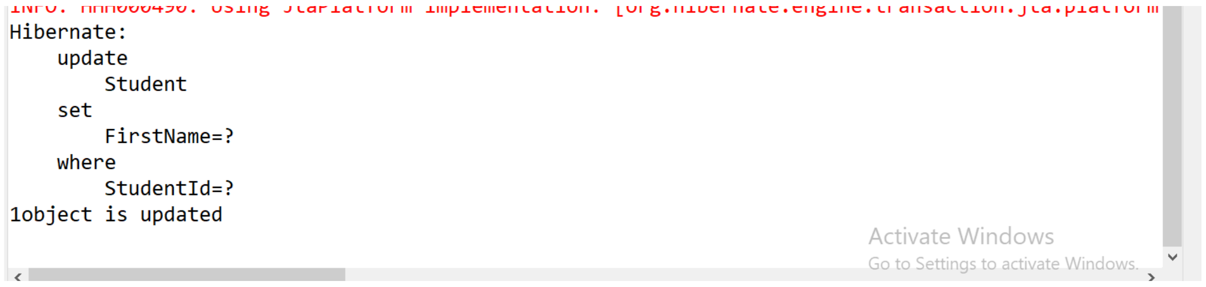
UPDATE Clause

Bulk updates are new to HQL with Hibernate 3, and delete work differently in Hibernate 3 than they did in Hibernate 2. The Query interface now contains a method called executeUpdate() for executing HQL UPDATE or DELETE statements.

The UPDATE clause can be used to update one or more properties of an object.

```
package com.sms;
import java.util.List;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.query.Query;
public class HqlOperation {
        public static void main(String[] args) {
                // Obtain a Hibernate SessionFactory
                SessionFactory factory = HibernateUtil.getSessionFactory();
                // Open a new session
                Session session = factory.openSession();
                // Begin a transaction
                Transaction transaction = session.beginTransaction();
                // Update Clause
                String hql = "update Student set firstName=:n where studentId=:i";
                Query q = session.createQuery(hql);
                q.setParameter("n", "Steve");
                q.setParameter("i", "S112");
                int row=q.executeUpdate();
                System.out.println(row+ " object is updated");
                transaction.commit();
        }
}
```

Output:



```
INFO: HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.jta.platform
Hibernate: 
    update
        Student 
    set
        FirstName=? 
    where
        StudentId=?
1object is updated
```

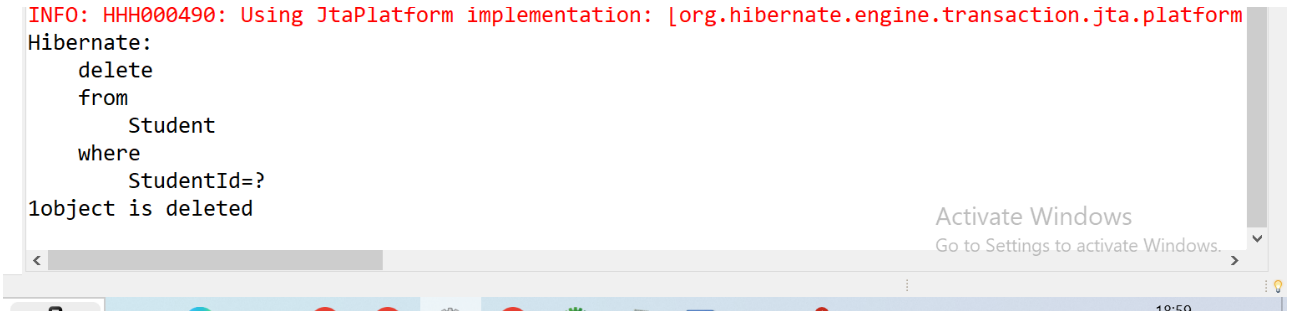
DELETE Clause

The DELETE clause can be used to delete one or more objects.

Complete Example:

```
package com.sms;
import java.util.List;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.query.Query;
public class HqlOperation {
        public static void main(String[] args) {
                // Obtain a Hibernate SessionFactory
                SessionFactory factory = HibernateUtil.getSessionFactory();
                // Open a new session
                Session session = factory.openSession();
                // Begin a transaction
                Transaction transaction = session.beginTransaction();
                // Delete Clause
                Query q2=session.createQuery("delete from Student where studentId=:i");
                q2.setParameter("i", "S112");
                int row=q2.executeUpdate();
                System.out.println(row+ "object is deleted");
                transaction.commit();
        }
}
```

Output:



```
INFO: HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.jta.platform
Hibernate:
    delete
    from
        Student
    where
        StudentId=?
1object is deleted
```

Food for thought..

The trainer can ask the students the following question to engage them in a discussion:

Why do you think a named query is required?

III. Named Queries

Named queries in Hibernate Query Language (HQL) allow you to pre-define and name your queries in configuration files or using annotations. This feature enhances code organization, promotes reusability, and improves the maintainability of your Hibernate-based application. Named queries can be executed multiple times throughout your application.

Here's how to create and use named queries in HQL:

1. **Create an Entity Class:** Annotate your entity class with @NamedQuery to define a named query. For example:

```
@Entity
@Table(name = "Student")
@NamedQuery(name = "findStudentById", query = "FROM Student WHERE studentId
= :id")
public class Student {
   // ...
}
```

2. **Execute Named Query:** In your Java code, you can execute the named query as follows:

```
Query query = session.getNamedQuery("findStudentById");
query.setParameter("id", studentId);
List<Student> students = query.list();
```

Benefits of named queries:

- **Code Organization:** Named queries help in keeping your query logic separate from the application code, improving code organization and maintainability.
- **Reusability:** You can use the same named query in multiple parts of your application without duplicating the query string.
- **Type Safety:** Named queries are type-checked, and any syntax errors are caught during application startup or compilation.
- **Performance:** Hibernate can optimize named queries during session factory initialization, resulting in better performance.
- **Security:** Named queries with parameters are less susceptible to SQL injection as parameter values are bound explicitly.

Named queries are particularly useful when you need to execute the same query multiple times or when working with complex queries that are used in various parts of your application.

Let’s see the complete example:

Entity Class:

```
package com.sms;

import java.time.LocalDate;
import java.util.List;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.NamedQuery;
import javax.persistence.OneToMany;
```

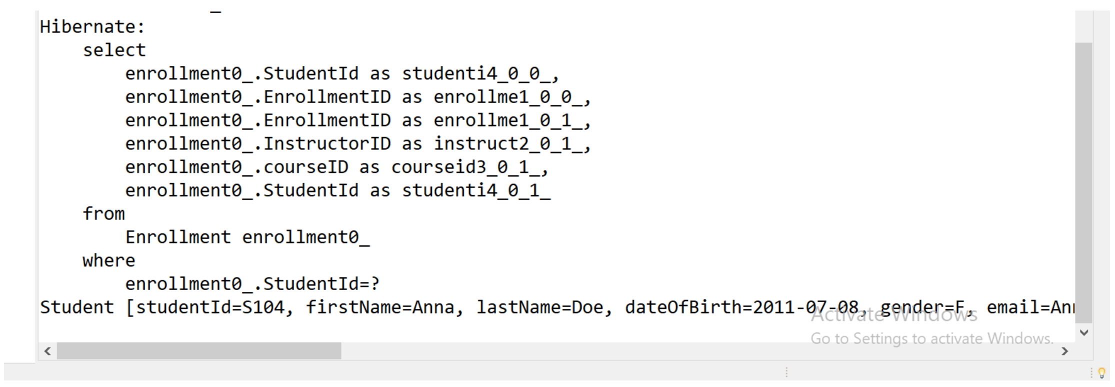
```
@Entity
@NamedQuery(name = "findStudentById", query = "FROM Student WHERE studentId
= ?1")
public class Student {
        @Id
        @Column(name = "StudentId", length = 10)
        private String studentId;

        @Column(name = "FirstName", length = 50)
        private String firstName;

        @Column(name = "LastName", length = 25)
        private String lastName;

        @Column(name = "DateOfBirth")
        private LocalDate dateOfBirth;

        @Column(name = "Gender", length = 25)
        private String gender;

        @Column(name = "Email", length = 30)
        private String email;

        @Column(name = "Phone", length = 25)
        private String phone;

        @OneToMany(mappedBy = "studentId")
        private List<Enrollment> enrollments;

        public List<Enrollment> getEnrollments() {
                return enrollments;
        }

        public void setEnrollments(List<Enrollment> enrollments) {
                this.enrollments = enrollments;
        }

        //Setter And Getter
        public String getStudentId() {
```

```
	return studentId;
}

public void setStudentId(String studentId) {
	this.studentId = studentId;
}

public String getFirstName() {
	return firstName;
}

public void setFirstName(String firstName) {
	this.firstName = firstName;
}

public String getLastName() {
	return lastName;
}

public void setLastName(String lastName) {
	this.lastName = lastName;
}

public LocalDate getDateOfBirth() {
	return dateOfBirth;
}

public void setDateOfBirth(LocalDate dateOfBirth) {
	this.dateOfBirth = dateOfBirth;
}

public String getGender() {
	return gender;
}

public void setGender(String gender) {
	this.gender = gender;
}

public String getEmail() {
		return email;
	}

	public void setEmail(String email) {
		this.email = email;
	}

	public String getPhone() {
		return phone;
	}

	public void setPhone(String phone) {
		this.phone = phone;
	}


	//Default Constructor
	public Student() {
		super();
		// TODO Auto-generated constructor stub
	}

	//All argument Constructor
	public Student(String studentId, String firstName, String lastName, LocalDate
dateOfBirth, String gender,
			String email, String phone) {
		super();
		this.studentId = studentId;
		this.firstName = firstName;
		this.lastName = lastName;
		this.dateOfBirth = dateOfBirth;
		this.gender = gender;
		this.email = email;
		this.phone = phone;
	}



	//ToString method
	@Override
	public String toString() {
		return "Student [studentId=" + studentId + ", firstName=" + firstName + ",
lastName=" + lastName
				+ ", dateOfBirth=" + dateOfBirth + ", gender=" + gender + ",
email=" + email + ", phone=" + phone
				+ ", enrollments=" + enrollments + "]";
	}

}
```

Main Class:

```
package com.sms;
import java.util.List;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.query.Query;
public class HqlOperation {
	public static void main(String[] args) {
		// Obtain a Hibernate SessionFactory
		SessionFactory factory = HibernateUtil.getSessionFactory();
		// Open a new session
		Session session = factory.openSession();
		// Begin a transaction
		Transaction transaction = session.beginTransaction();
		Query query = session.getNamedQuery("findStudentById");
		query.setParameter(1, "S104");
		Student student = (Student) query.getSingleResult();
System.out.println(student);
	}
}
```

Output:



```
Hibernate:
    select
        enrollment0_.StudentId as studenti4_0_0_,
        enrollment0_.EnrollmentID as enrollme1_0_0_,
        enrollment0_.EnrollmentID as enrollme1_0_1_,
        enrollment0_.InstructorID as instruct2_0_1_,
        enrollment0_.courseID as courseid3_0_1_,
        enrollment0_.StudentId as studenti4_0_1_
    from
        Enrollment enrollment0_
    where
        enrollment0_.StudentId=?
Student [studentId=S104, firstName=Anna, lastName=Doe, dateOfBirth=2011-07-08, gender=F, email=An
```

IV. Inheritance in hibernate

In Hibernate, inheritance is a way to represent the object-oriented inheritance hierarchy in a relational database. Hibernate provides several strategies for mapping inheritance in your object-relational model. These strategies include:

- **Table per Hierarchy (Single Table Inheritance):** In this strategy, all classes in the inheritance hierarchy are mapped to a single database table. A discriminator column is used to differentiate between different types of objects in the table. This approach is simple but can lead to a wide table with many nullable columns.

- **Table per Class (Concrete Table Inheritance):** In this strategy, each class in the inheritance hierarchy is mapped to a separate database table. There is no discriminator column, and each table contains only the fields for that specific class. This approach can lead to more normalized tables but might require joins when querying across the hierarchy.

- **Table per Subclass (Joined Table Inheritance):** In this strategy, each class in the inheritance hierarchy is mapped to a separate database table. A shared primary key is used to establish a relationship between the tables. This approach results in a more normalized database schema and can be efficient for querying specific classes.

- **Table per Concrete Class:** This is an extension of the Table per Class strategy. It allows concrete classes in the hierarchy to be used independently as root classes, each with its table, rather than requiring them to be subclassed.

Here's an example of how to map inheritance using Hibernate annotations:

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "vehicle_type", discriminatorType =
DiscriminatorType.STRING)
public abstract class Vehicle {
   @Id
   @GeneratedValue
   private Long id;
   private String manufacturer;
   // Other common fields
}

@Entity
@DiscriminatorValue("Car")
public class Car extends Vehicle {
   private int numberOfDoors;
   // Car-specific fields
}

@Entity
@DiscriminatorValue("Bike")
public class Bike extends Vehicle {
   private int numberOfWheels;
   // Bike-specific fields
}
```
In the above example, we have used the Single Table Inheritance strategy. The @Inheritance annotation specifies the inheritance strategy, and the @DiscriminatorColumn and @DiscriminatorValue annotations define the discriminator column and its values.

Hibernate also supports the use of XML configuration files for mapping inheritance. The choice of inheritance strategy depends on your specific use case and the database design, and you should choose the one that best fits your requirements for querying and database performance.

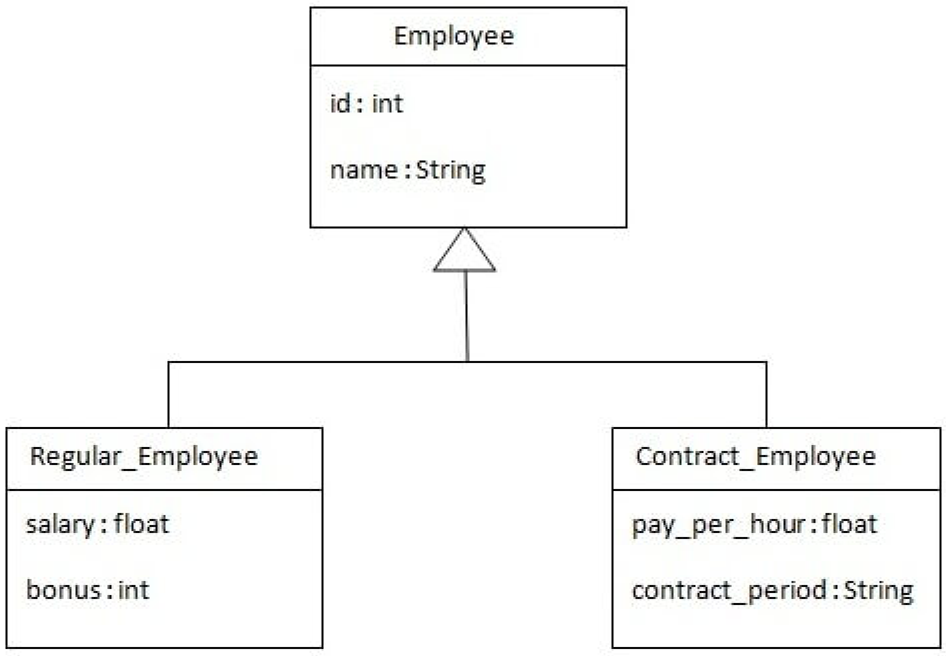
1. **Table Per Hierarchy**

In table per hierarchy mapping, a single table is required to map the whole hierarchy, an extra column (known as discriminator column) is added to identify the class. But nullable values are stored in the table .

@Inheritance(strategy=InheritanceType.SINGLE_TABLE), @DiscriminatorColumn and @DiscriminatorValue annotations for mapping table per hierarchy strategy.

In the case of table per hierarchy, only one table is required to map the inheritance hierarchy. Here, an extra column (also known as discriminator column) is created in the table to identify the class.

Let's see the inheritance hierarchy:



There are three classes in this hierarchy. Employee is the super class for Regular_Employee and Contract_Employee classes.

Example of Hibernate Table Per Hierarchy using Annotation

You need to follow the following steps to create simple example:

- Create the persistent classes

- Edit pom.xml file
- Create the configuration file
- Create the class to store the fetch the data

Create the Persistent classes

You need to create persistent classes representing the inheritance. Let's create the three classes for the above hierarchy:

Employee.java

```
package inheritance;
import javax.persistence.Column;
import javax.persistence.DiscriminatorColumn;
import javax.persistence.DiscriminatorType;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="type",discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue(value="employee")
public class Employee {
	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)

	@Column(name = "id")
	private int id;

	@Column(name = "name")
	private String name;
	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 Employee(int id, String name) {
		super();
		this.id = id;
		this.name = name;
	}
	public Employee() {
		super();
		// TODO Auto-generated constructor stub
	}
	@Override
	public String toString() {
		return "Employee [id=" + id + ", name=" + name + "]";
	}


}
```

Regular_Employee.java

```
package inheritance;
import javax.persistence.Column;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
@Entity
@DiscriminatorValue("regularemployee")
public class Regular_Employee extends Employee{
	@Column(name="salary")
	private float salary;

	@Column(name="bonus")
	private int bonus;
	public float getSalary() {
		return salary;
	}
	public void setSalary(float salary) {
		this.salary = salary;
	}
	public int getBonus() {
		return bonus;
	}
	public void setBonus(int bonus) {
		this.bonus = bonus;
	}
	@Override
	public String toString() {
		return "Regular_Employee [salary=" + salary + ", bonus=" + bonus + "]";
	}
}
```

Contract_Employee.java

```
package inheritance;
import javax.persistence.Column;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
@Entity
@DiscriminatorValue("contractemployee")
public class Contract_Employee extends Employee{

	@Column(name="pay_per_hour")
private float pay_per_hour;
@Column(name="contract_duration")
private String contract_duration;
	public float getPay_per_hour() {
		return pay_per_hour;
	}
	public void setPay_per_hour(float pay_per_hour) {
		this.pay_per_hour = pay_per_hour;
	}
	public String getContract_duration() {
		return contract_duration;
	}
	public void setContract_duration(String contract_duration) {
		this.contract_duration = contract_duration;
	}
	@Override
	public String toString() {
		return "Contract_Employee [pay_per_hour=" + pay_per_hour + ",
contract_duration=" + contract_duration + "]";
	}
}
```

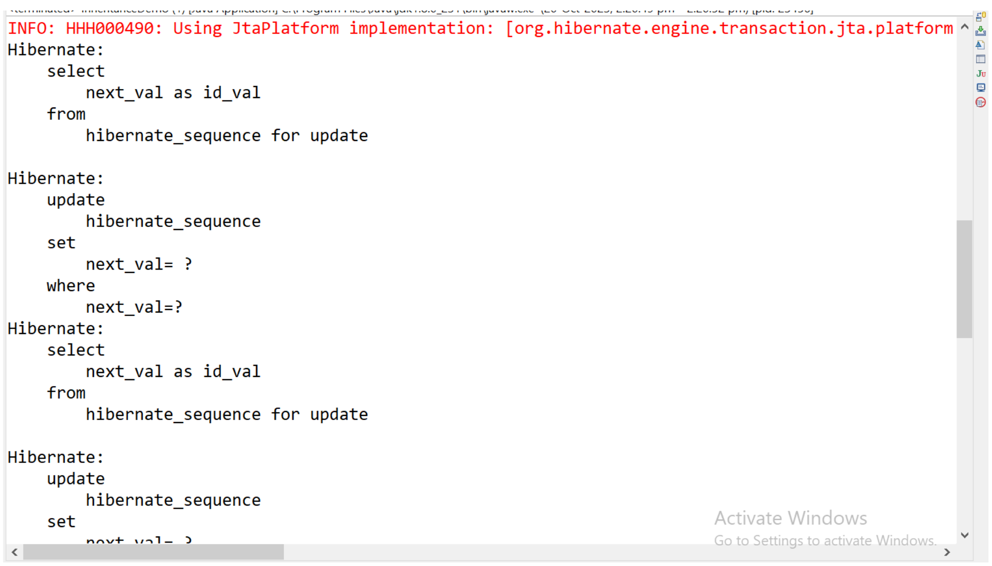
HibernateUtil.java

```
package inheritance;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
public class HibernateUtil {
	private static final SessionFactory sessionFactory = buildSessionFactory();
	private static SessionFactory buildSessionFactory() {
		try {
			return new
Configuration().configure("hibernate.cfg.xml").addAnnotatedClass(Employee.class)

.addAnnotatedClass(Contract_Employee.class).addAnnotatedClass(Regular_Employee
.class)
					.buildSessionFactory();
		} catch (Throwable ex) {
			throw new ExceptionInInitializerError(ex);
		}
	}
	public static SessionFactory getSessionFactory() {
		return sessionFactory;
	}
}
```

InheritanceDemo.java

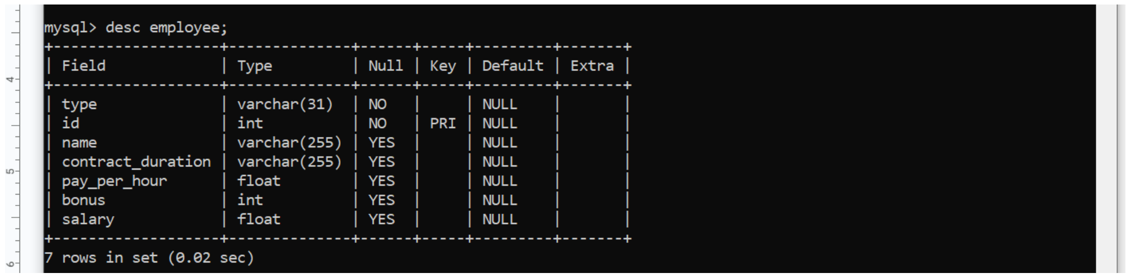
```
package inheritance;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
public class InheritanceDemo {
        public static void main(String[] args) {
                // Obtain a Hibernate SessionFactory
                SessionFactory factory = HibernateUtil.getSessionFactory();
                // Open a new session
                Session session = factory.openSession();
//              // Begin a transaction
                Transaction transaction = session.beginTransaction();
                Employee e1 = new Employee();
                e1.setName("Gaurav Chawla");
                Regular_Employee e2 = new Regular_Employee();
                e2.setName("Vivek Kumar");
                e2.setSalary(50000);
                e2.setBonus(5);
                Contract_Employee e3 = new Contract_Employee();
                e3.setName("Arjun Kumar");
                e3.setPay_per_hour(1000);
                e3.setContract_duration("15 hours");
                session.save(e1);
                session.save(e2);
                session.save(e3);
                transaction.commit();
                session.close();
                factory.close();
        }
}
```

Output:

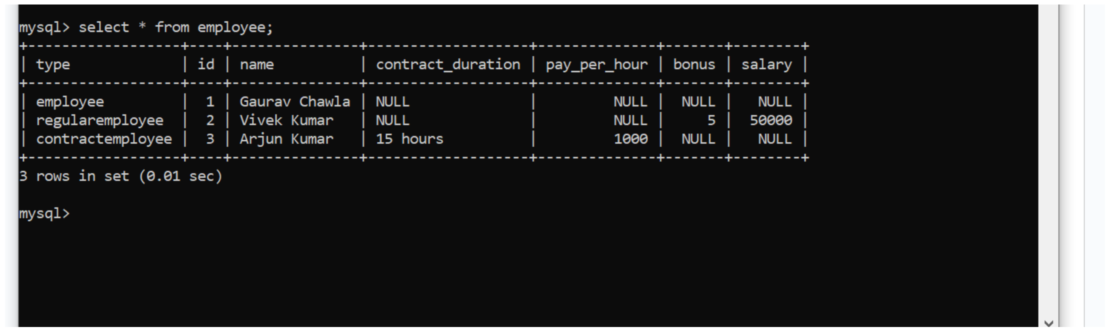


```
INFO: HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.jta.platform
Hibernate: 
    select
        next_val as id_val 
    from
        hibernate_sequence for update
            
Hibernate: 
    update
        hibernate_sequence 
    set
        next_val= ? 
    where
        next_val=?
Hibernate: 
    select
        next_val as id_val 
    from
        hibernate_sequence for update
            
Hibernate: 
    update
        hibernate_sequence 
    set
```

Database:



```
mysql> desc employee;
+-------------------+--------------+------+-----+---------+-------+
| Field             | Type         | Null | Key | Default | Extra |
+-------------------+--------------+------+-----+---------+-------+
| type              | varchar(31)  | NO   |     | NULL    |       |
| id                | int          | NO   | PRI | NULL    |       |
| name              | varchar(255) | YES  |     | NULL    |       |
| contract_duration | varchar(255) | YES  |     | NULL    |       |
| pay_per_hour      | float        | YES  |     | NULL    |       |
| bonus             | int          | YES  |     | NULL    |       |
| salary            | float        | YES  |     | NULL    |       |
+-------------------+--------------+------+-----+---------+-------+
7 rows in set (0.02 sec)
```



```
mysql> select * from employee;
+------------------+----+--------------+-------------------+--------------+-------+--------+
| type             | id | name         | contract_duration | pay_per_hour | bonus | salary |
+------------------+----+--------------+-------------------+--------------+-------+--------+
| employee         |  1 | Gaurav Chawla | NULL             |         NULL |  NULL |   NULL |
| regularemployee  |  2 | Vivek Kumar  | NULL              |         NULL |     5 |  50000 |
| contractemployee |  3 | Arjun Kumar  | 15 hours          |         1000 |  NULL |   NULL |
+------------------+----+--------------+-------------------+--------------+-------+--------+
3 rows in set (0.01 sec)

mysql>
```

2. Table Per Concrete class

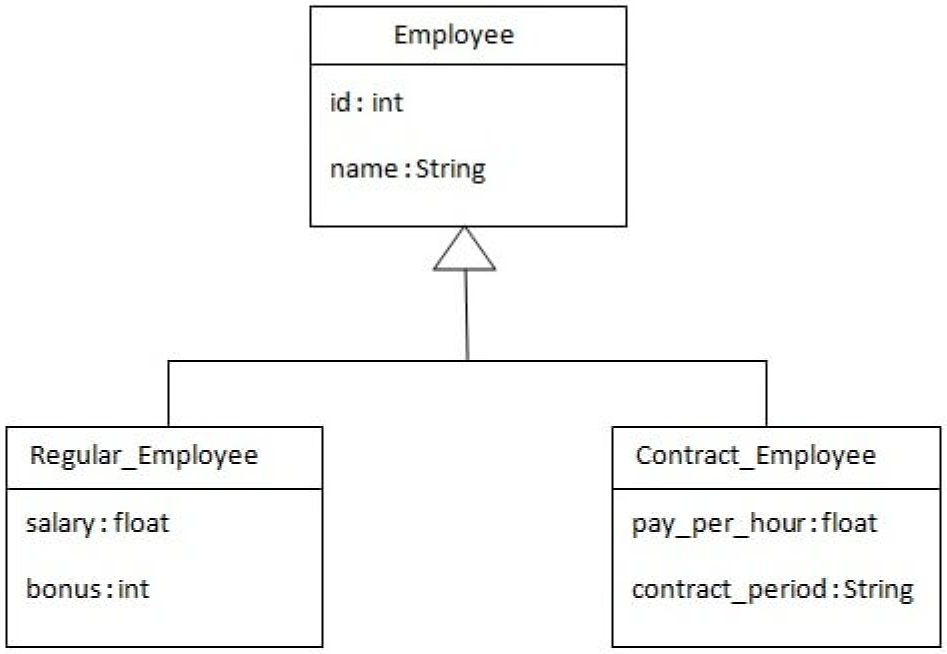
In the case of Table Per Concrete class, tables are created per class. So there are no nullable values in the table. Disadvantage of this approach is that duplicate columns are created in the subclass tables.

Here, we need to use @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS) annotation in the parent class and @AttributeOverrides annotation in the subclasses.

@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS) specifies that we are using a table per concrete class strategy. It should be specified in the parent class only.

@AttributeOverrides defines that parent class attributes will be overridden in this class. In table structure, parent class table columns will be added in the subclass table.

The class hierarchy is given below:



Example of Table per concrete class

In this example we are creating the three classes and providing mapping of these classes in the employee.hbm.xml file.

1) Create the Persistent classes

You need to create persistent classes representing the inheritance. Let's create the three classes for the above hierarchy:

Employee.class

```
package inheritance;
import javax.persistence.Column;
import javax.persistence.DiscriminatorColumn;
import javax.persistence.DiscriminatorType;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;
@Entity
@Table(name = "employee1")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Employee {
	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)

	@Column(name = "id")
	private int id;

	@Column(name = "name")
	private String name;
	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 Employee(int id, String name) {
		super();
		this.id = id;
		this.name = name;
	}
	public Employee() {
		super();
		// TODO Auto-generated constructor stub
	}
	@Override
	public String toString() {
		return "Employee [id=" + id + ", name=" + name + "]";
	}


}
```

Regular_Employee.class

```
package inheritance;
import javax.persistence.AttributeOverride;
import javax.persistence.AttributeOverrides;
import javax.persistence.Column;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
import javax.persistence.Table;
@Entity
@Table(name="regularemployee1")
@AttributeOverrides({
@AttributeOverride(name="id", column=@Column(name="id")),
@AttributeOverride(name="name", column=@Column(name="name"))
})
public class Regular_Employee extends Employee{
	@Column(name="salary")
	private float salary;

	@Column(name="bonus")
	private int bonus;
	public float getSalary() {
		return salary;
	}
	public void setSalary(float salary) {
		this.salary = salary;
	}
	public int getBonus() {
		return bonus;
	}
```

```
        public void setBonus(int bonus) {
                this.bonus = bonus;
        }
        @Override
        public String toString() {
                return "Regular_Employee [salary=" + salary + ", bonus=" + bonus + "]";
        }
}
```

Contract_Employee.class

```
package inheritance;
import javax.persistence.AttributeOverride;
import javax.persistence.AttributeOverrides;
import javax.persistence.Column;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
import javax.persistence.Table;
@Entity
@Table(name="contractemployee1")
@AttributeOverrides({
@AttributeOverride(name="id", column=@Column(name="id")),
@AttributeOverride(name="name", column=@Column(name="name"))
})
public class Contract_Employee extends Employee{

        @Column(name="pay_per_hour")
private float pay_per_hour;
@Column(name="contract_duration")
private String contract_duration;
        public float getPay_per_hour() {
                return pay_per_hour;
        }
        public void setPay_per_hour(float pay_per_hour) {
                this.pay_per_hour = pay_per_hour;
        }
        public String getContract_duration() {
                return contract_duration;
        }
        public void setContract_duration(String contract_duration) {
```

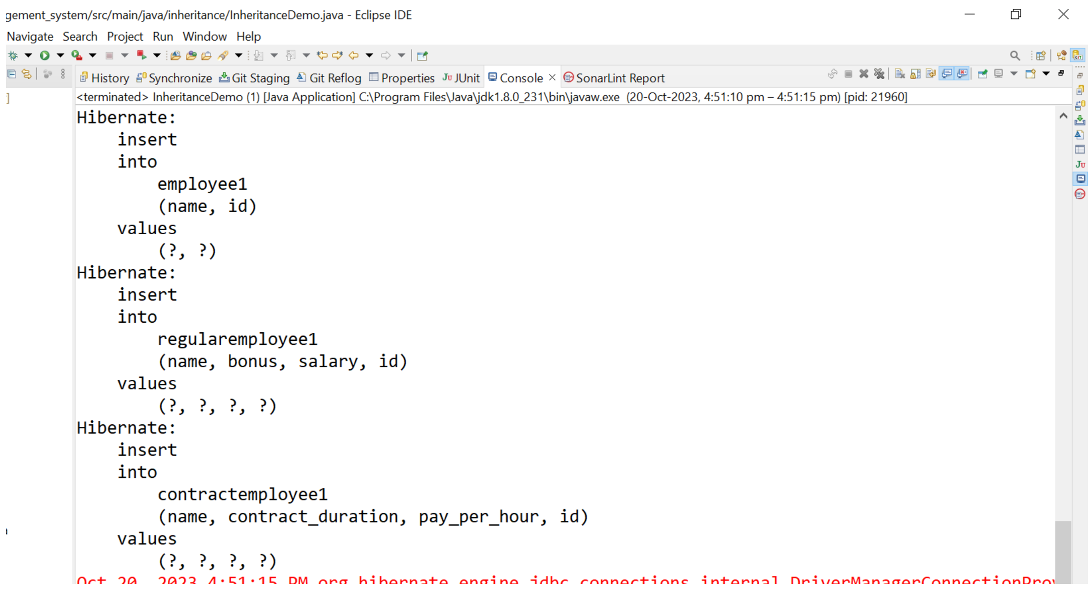
```
		this.contract_duration = contract_duration;
	}
	@Override
	public String toString() {
		return "Contract_Employee [pay_per_hour=" + pay_per_hour + ",
contract_duration=" + contract_duration + "]";
	}
}
```

InheritanceDemo.java

```
package inheritance;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
public class InheritanceDemo {
	public static void main(String[] args) {
		// Obtain a Hibernate SessionFactory
		SessionFactory factory = HibernateUtil.getSessionFactory();
		// Open a new session
		Session session = factory.openSession();
//		// Begin a transaction
		Transaction transaction = session.beginTransaction();
		 Employee e1=new Employee();
	 e1.setName("Gaurav Chawla");

	 Regular_Employee e2=new Regular_Employee();
	 e2.setName("Vivek Kumar");
	 e2.setSalary(50000);
	 e2.setBonus(5);

	 Contract_Employee e3=new Contract_Employee();
	 e3.setName("Arjun Kumar");
	 e3.setPay_per_hour(1000);
	 e3.setContract_duration("15 hours");
		session.save(e1);
		session.save(e2);
		session.save(e3);
		transaction.commit();
		session.close();
```

```
                factory.close();
        }
}
```

Output:

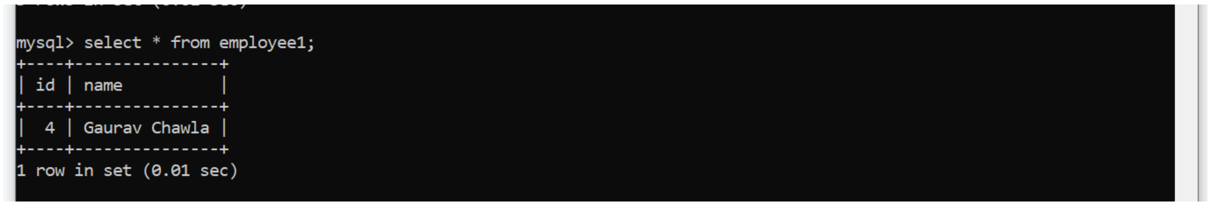


gement_system/src/main/java/inheritance/InheritanceDemo.java - Eclipse IDE
Navigate Search Project Run Window Help
History Synchronize Git Staging Git Reflog Properties JUnit Console SonarLint Report

```
<terminated> InheritanceDemo (1) [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (20-Oct-2023, 4:51:10 pm – 4:51:15 pm) [pid: 21960]
Hibernate: 
    insert 
    into
        employee1
        (name, id) 
    values
        (?, ?)
Hibernate: 
    insert 
    into
        regularemployee1
        (name, bonus, salary, id) 
    values
        (?, ?, ?, ?)
Hibernate: 
    insert 
    into
        contractemployee1
        (name, contract_duration, pay_per_hour, id) 
    values
        (?, ?, ?, ?)
Oct 20, 2023 4:51:15 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionPro
```

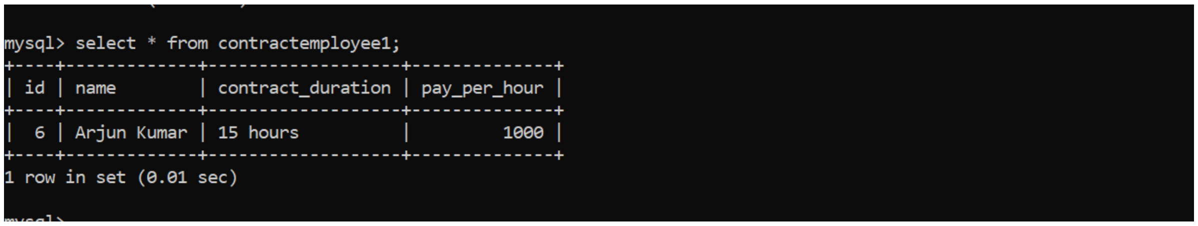
Database:

Employee table



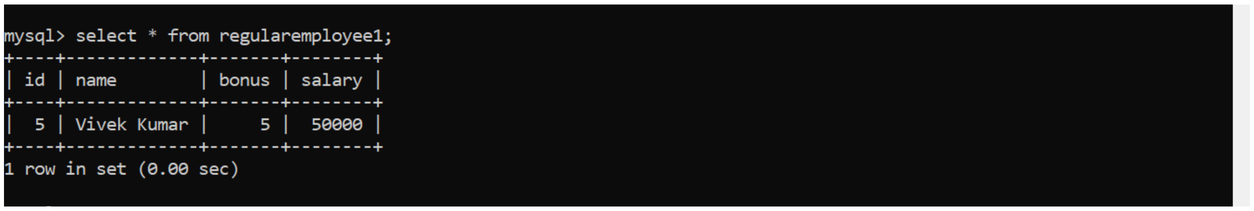
```
mysql> select * from employee1;
+----+---------------+
| id | name          |
+----+---------------+
|  4 | Gaurav Chawla |
+----+---------------+
1 row in set (0.01 sec)
```

contractemployee1



```
mysql> select * from contractemployee1;
+----+-------------+-------------------+--------------+
| id | name        | contract_duration | pay_per_hour |
+----+-------------+-------------------+--------------+
|  6 | Arjun Kumar | 15 hours          |         1000 |
+----+-------------+-------------------+--------------+
1 row in set (0.01 sec)
```

regularemployee1



```
mysql> select * from regularemployee1;
+----+------------+-------+--------+
| id | name       | bonus | salary |
+----+------------+-------+--------+
|  5 | Vivek Kumar |     5 |  50000 |
+----+------------+-------+--------+
1 row in set (0.00 sec)
```

Exercise:

Use ChatGPT to explore inheritance in hibernate:

Put the below problem statement in the message box and see what ChatGPT says.

I've been studying Hibernate inheritance, and I'm interested in delving into the topic of Table per Subclass (Joined Table Inheritance).Kindly give me some tutorial and code snippets.