EA Practice midterm

Question 1 [10 points] {10 minutes}

a. Suppose we have a Spring application with the following given XML configuration

```
<bean id="customerService" class="basic.CustomerService">
   <constructor-arg ref="emailService"/>
</bean>
<bean id="emailService" class="basic.EmailService">
   <constructor-arg ref="customerService"/>
</bean>
```

When we run the application, Spring gives an error. Explain clearly why Spring gives an error based on the given XML configuration.

Answer:

With constructor injection Spring first need to initialize the bean that is injected. Then it can initialize the bean by using the constructor of this bean and pass the injected bean as parameter. In the given configuration we have a circular dependency. Spring can only create the CustomerService bean after it has created the Emailservice. But it can only create the EmailService after it has created the CustomerService.

b. Explain why we need an **init()** method in Spring Boot.

Answer:

Question 2 [15 points] {20 minutes}

Suppose we need to write a **Spring Boot** application that allow us to store and find Products. A Product consists of the following attributes: productNumber, name, price and categoryName. A categoryName is something like “clothing” or “toys” or “electronics” The application should allow us to store new Products and we should be able to find products with the following functionality:

- Give all products with a price bigger than a given amount
- Give all products from a certain category

Write **ALL** necessary Java code including annotations. Do **NOT** write the Application class (that contains the main() method). Do **NOT** write imports and do **NOT** write getter and setter methods. Also do **NOT** write constructors.

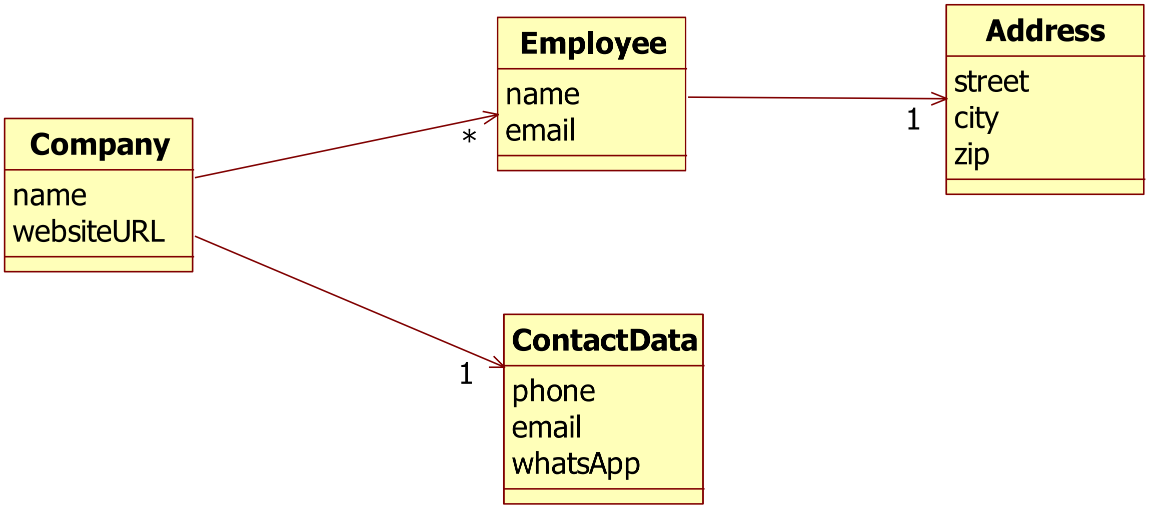
Use all the best practices we learned in this course.

```
@Entity
public class Product {
    @Id
    private String productNumber;
    private String name;
    private double price;
    private String category;


public interface ProductRepository extends JpaRepository<Product, String> {
    List<Product> findByCategory(String category);

    @Query("select p from Product p where p.price > :price")
    List<Product> findByPriceGreatherThan(@Param("price") double price);
}


@Service
public class ProductService {
    @Autowired
    ProductRepository productRepository;

    public void addProduct(ProductDTO productDto){
        Product product =
ProductAdapter.getProductFromProductDTO(productDto);
        productRepository.save(product);
    }

    public List<ProductDTO> findByCategory(String category){
        List<Product> products = productRepository.findByCategory(category);
        return ProductAdapter.getProductDTOListFromProductList(products);
    }

    public List<ProductDTO> findByPriceGreatherThan(double price){
        List<Product> products =
productRepository.findByPriceGreatherThan(price);
        return ProductAdapter.getProductDTOListFromProductList(products);
    }
}


public class ProductDTO {
    private String productNumber;
    private String name;
    private double price;
    private String category;


public class ProductAdapter {
    public static ProductDTO getProductDTOFromProduct(Product product){
        return new ProductDTO(product.getProductNumber(), product.getName(),
product.getPrice(), product.getCategory());
    }
    public static Product getProductFromProductDTO(ProductDTO productDto){
        return new Product(productDto.getProductNumber(),
productDto.getName(), productDto.getPrice(),productDto.getCategory());
```

```
    }
    public static List<ProductDTO> 
getProductDTOListFromProductList(List<Product> products){
        List<ProductDTO> productDtoList = new ArrayList<ProductDTO>();
        for (Product product : products){
            productDtoList.add(getProductDTOFromProduct(product));
        }
        return productDtoList;
    }
}
```

Question 3 [15 points] {15 minutes}

Suppose we have the following JPA entities:



We need to write the following queries:

These queries should be defined by the method name in the repository:

- **Give all Companies with a given name. Name is a parameter.**
- **Give all streets given a certain city and a certain zip**

These queries should be defined by **@Query** in the repository:

- **Give the name of all companies from a given city**
- **Give the name of the company given a certain phone number**
- **Give all Companies where an employee works with a certain given name.**

Write the queries in the corresponding repositories. Write the **complete Java code** of all necessary repositories including the methods and the annotations. **Do not write Java imports**

```
@Repository
public interface CompanyRepository extends JpaRepository<Company, Long>{

List<Company> findByName(String name);

@Query("select distinct c.name from Company c join c.employees e where e.address.city=
:city)
List<Company> findByCity(@Param("city") String city);


@Query("select c.name from Company c where c.contactData.phone= :phone)
Company findByPhone(@Param("phone") String phone);

@Query("select distinct c from Company c join c.employees e where e.name= :name)
List<Company> findByEmployeeName(@Param("name") String name);
}


@Repository
public interface AddressRepository extends JpaRepository<Address, Long>{

  List<String> findStreetByCityAndZip(String city, String zip);
}
```

Question 4 [20 points] {20 minutes}

Given are the following entities:

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
      name="vehicle_type",
      discriminatorType=DiscriminatorType.STRING
)
public abstract class Vehicle {
   @Id
   @GeneratedValue
   private long id;
   private String brand;
   private String color;

   public Vehicle() { }

   public Vehicle(String brand, String color) {
      this.brand = brand;
      this.color = color;
   }
}


@Entity
public abstract class Car extends Vehicle{
   private String licencePlate;
   public Car() { }
   public Car(String brand, String color, String licencePlate) {
      super(brand, color);
      this.licencePlate = licencePlate;
   }
}

@Entity
@DiscriminatorValue("RentalBycicle")
public class RentalBycicle extends Vehicle{
   private double pricePerHour;
   public RentalBycicle() {   }
   public RentalBycicle(String brand, String color, double pricePerHour) {
      super(brand, color);
      this.pricePerHour = pricePerHour;
   }
}


@Entity
@DiscriminatorValue("SellableCar")
public class SellableCar extends Car {
   private double sellPrice;
   public SellableCar() { }
   public SellableCar(String brand, String color, String licencePlate, double
sellPrice) {
      super(brand, color, licencePlate);
      this.sellPrice = sellPrice;
   }
}


@Entity
@DiscriminatorValue("RentalCar")
public class RentalCar extends Car {
   private double pricePerDay;
   public RentalCar() {   }
   public RentalCar(String brand, String color, String licencePlate, double
pricePerDay) {
      super(brand, color, licencePlate);
      this.pricePerDay = pricePerDay;
   }
}

public interface RentalBycicleRepository extends JpaRepository<RentalBycicle,
Long> {
}
public interface RentalCarRepository extends JpaRepository<RentalCar, Long> {
}
public interface SellableCarRepository extends JpaRepository<SellableCar,
Long> {
}

@SpringBootApplication
public class Application implements CommandLineRunner {
   @Autowired
   RentalCarRepository rentalCarRepository;
   @Autowired
   SellableCarRepository sellableCarRepository;
   @Autowired
   RentalBycicleRepository rentalBycicleRepository;

   public static void main(String[] args) {
      SpringApplication.run(Application.class, args);
   }

   @Override
   public void run(String... args) throws Exception {
      RentalCar rentalCar = new RentalCar("BMW", "Black", "KL-980-1", 67.00);
      rentalCarRepository.save(rentalCar);
      SellableCar sellableCar = new SellableCar("Audi", "White", "KM-956-2",
45980.00);
      sellableCarRepository.save(sellableCar);
      RentalBycicle rentalBycicle = new RentalBycicle("Moof", "Grey", 10.50);
      rentalBycicleRepository.save(rentalBycicle);

   }

}
```

a. In the given code above, add the necessary mapping annotations so that the whole inheritance hierarchy is mapped according the **single table per hierarchy** strategy. Do **NOT** rewrite any code. Only write the correct annotations in the given code.

b. Explain **ALL** advantages and disadvantages we learned about the **single table per hierarchy** strategy.

Advantages

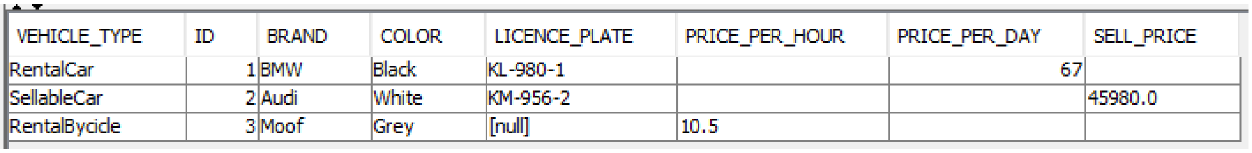
+ Simple, Easy to implement

+ Good performance on all queries, polymorphic and non-polymorphic

Disadvantages

- Nullable columns / de-normalized schema
- Table may have to contain lots of columns
- A change in any class results in a change of this table

c. Draw the corresponding database table with all the columns and corresponding data if we run Application.java.



VEHICLE_TYPE	ID	BRAND	COLOR	LICENCE_PLATE	PRICE_PER_HOUR	PRICE_PER_DAY	SELL_PRICE
RentalCar	1	BMW	Black	KL-980-1		67	
SellableCar	2	Audi	White	KM-956-2			45980.0
RentalBycicle	3	Moof	Grey	[null]	10.5		

d. Suppose we map the given inheritance hierarchy with the strategy **Joined Tables**. Draw the corresponding database tables with all the columns and corresponding data if we use the strategy **Joined Tables**

e. Suppose we map the given inheritance hierarchy with the strategy **Table per concrete class**. Draw the corresponding database tables with all the columns and corresponding data if we use the strategy **Table per concrete class**

Question 5 [10 points] {15 minutes}

Circle all statements that are correct:

a. When we add a version attribute to an entity and we annotate this with @Version then you will never have the dirty read problem on this entity.

b. If we do not allow the phantom read problem in our application, we cannot run 2 transactions at the same time.

c. In a Spring boot application that uses JPA, you cannot use dependency injection on JPA entities.

d. When you make one method of a Spring bean transactional the 2 phase commit protocol will never be used. If you make 2 or more methods of a Spring bean transactional the 2 phase commit protocol will be used.

e. Cascading is only applicable for inserts, updates and deletes.

f. In JPA, a @OneToOne relation is stored in the database as a @ManyToOne relation.

g. A named query cannot contain a join.

h. An entity class in a Spring Boot JPA application is always a singleton.

i. With the TransactionReadCommitted isolation level, you can never have the lost update problem

j. In databases that use sequences, every table contains a sequence column.