CS544

LESSON 5
JPA MAPPING 2

Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
June 20 **Lesson 1** Introduction Spring framework Dependency injection	June 21 **Lesson 2** Spring Boot AOP	June 22 **Lesson 3** JDBC JPA	June 23 **Lesson 4** JPA mapping 1	June 24 **Lesson 5** JPA mapping 2	June 25 **Lesson 6** JPA queries	June 26
June 27 **Lesson 7** Transactions	June 28 **Lesson 8** MongoDB	June 29 **Midterm Review**	June 30 **Midterm exam**	July 1 **Lesson 9** REST webservices	July 2 **Lesson 10** SOAP webservices	July 3
July 4 **Lesson 11** Messaging	July 5 **Lesson 12** Scheduling Events Configuration	July 6 **Lesson 13** Monitoring	July 7 **Lesson 14** Testing your application	July 8 **Final review**	July 9 **Final exam**	July 10
July 11 **Project**	July 12 **Project**	July 13 **Project**	July 14 **Presentations**			

MAPPING IDENTITY

Primary key

- A primary key is
  - Unique
    - No duplicate values
  - Constant
    - Value never changes
  - Required
    - Value can never be null

- Primary key types:
  - Natural key
    - Has a meaning in the business domain
  - Surrogate key
    - Has no meaning in the business domain
    - Best practice

Mapping Primary Keys

- Object / Relational mismatch
  - Hibernate requires you to specify the property that will map to the primary key
- Prefer surrogate keys
  - Natural keys often lead to a brittle schema

```
@Entity
public class Person {
  @Id
  private String name;

  ...
```

Name as a natural primary key for Person can give problems

```
@Entity
public class Person {
  @Id
  private long id;
  private String name;

  ...
```

Instead use id as a surrogate key for Person

Generating Identity

- Generated identity values
  - Ensure identity uniqueness
- Private setId() methods
  - Ensure identity immutability

```
@Entity
public class Person {
  @Id
  @GeneratedValue
  private long id;
  private String name;

  public Person() {}
  public Person(String name) { this.name = name; }

  public long getId() { return id; }
  private void setId(long id) { this.id = id; }
  public String getName() { return name; }
  public void setName(String name) { this.name = name; }
}
```

Id is generated

Id can not be set by the application

Generation Strategies

JPA	Description
AUTO	Selects the best strategy for your database
IDENTITY	Use an identity column (MS SQL, MySQL, HSQL, …)
SEQUENCE	Use a sequence (Oracle, PostgreSQL, SAP DB, …)
TABLE	Uses a table to hold last generated values for PKs

Specifying Identity Generation

- @GeneratedValue

```
@Entity
public class Person {
  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private long id;
  private String name;
```

Specify the generation strategy

```
@Entity
public class Person {
  @Id
  @GeneratedValue
  private long id;
  private String name;
```

Defaults to 'AUTO' when not specified

Identity Column

- Identity columns are columns that can automatically generate the next unique id

```
@Entity
public class Person {
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private long id;
  private String name;
```

JPA identity strategy

- If your database support identity columns the native strategy will default to using them

Sequences

- By default Hibernate only uses a single sequence called ‘hibernate-sequence’
- You can specify additional custom sequences

Using Custom Sequences

Create Custom Sequence

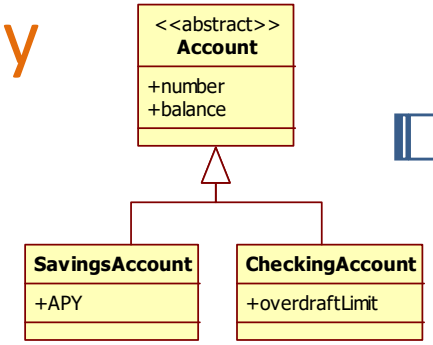
```
@Entity
@SequenceGenerator(name="personSeq", sequenceName="PERSON_SEQUENCE")
public class Person_annotated_sequence {
  @Id
  @GeneratedValue(generator="personSeq")
  private long id;
  ...
```

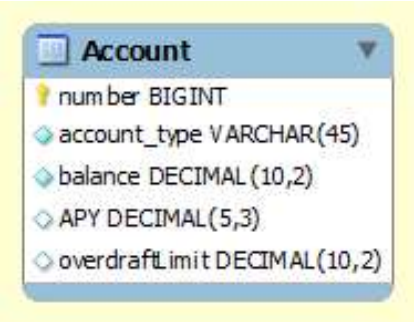
Use Custom Sequence

INHERITANCE MAPPING

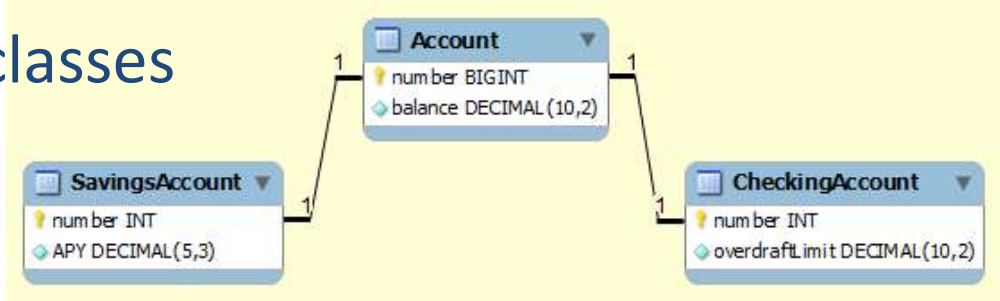
Three ways to map

- You can map inheritance in one of three ways:
  - Single Table per Hierarchy
    - De-normalized schema
    - Fast polymorphic queries





  - Joined Tables
    - Normalized & similar to classes
    - Slower queries



  - Table per Concrete Class
    - Uses UNION instead of JOIN
    - All needed columns in each table

Single Table

ACCOUNT_TYPE	NUMBER	BALANCE	OVERDRAFTLIMIT	APY
checking	1	500	200	
savings	2	100		2.3
checking	3	23.5	0	

APY is null for checking accounts, overdraft limit is null for savings

+ Simple, Easy to implement
+ Good performance on all queries, polymorphic and non polymorphic

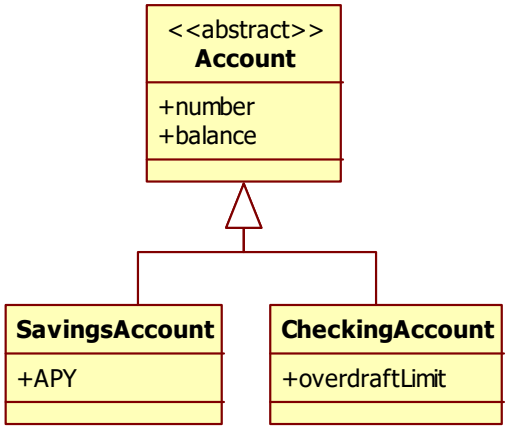
– Nullable columns / de-normalized schema
– Table may have to contain lots of columns
– A change in any class results in a change of this table

Single Table

Specify the SINGLE_TABLE strategy

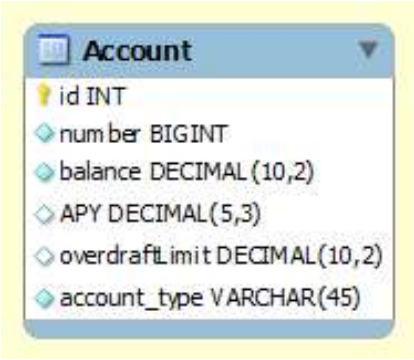
```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name="account_type",
    discriminatorType=DiscriminatorType.STRING
)
public abstract class Account
  @Id
  @GeneratedValue
  private long number;
  private double balance;

  ...
```

Optional annotation @DiscriminatorColumn




```
@Entity
@DiscriminatorValue("savings")
public class SavingsAccount extends Account {
  private double APY;

  ...
```

Specify discriminator value




```
@Entity
@DiscriminatorValue("checking")
public class CheckingAccount extends Account {
  private double overdraftLimit;

  ...
```

Specify discriminator value

Joined Tables

Account Table

NUMBER	BALANCE
1	500
2	100
3	23.5

SavingsAccount

NUMBER	APY
2	2.3

CheckingAccount

NUMBER	OVERDRAFTLIMIT
1	200
3	0

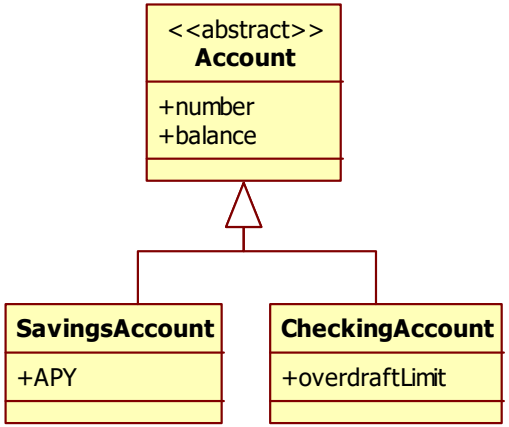
+ Normalized Schema
+ Database view is similar to domain view

– Inserting or updating an entity results in multiple insert or update statements
– Necessary joins can give bad query performance

Joined

Just specify the inheritance strategy, nothing else

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Account {
  @Id
  @GeneratedValue
  private long number;
  private double balance;

  ...
```




```
@Entity
public class SavingsAccount extends Account {
  private double APY;

  ...
```

Subclasses can be mapped as normal entity classes, but without identifiers

```
@Entity
public class CheckingAccount extends Account {
  private double overdraftLimit;

  ...
```

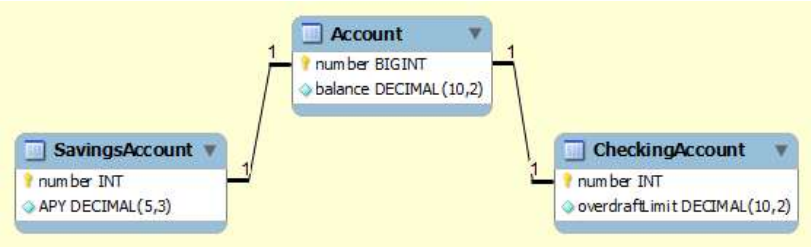


Table per Class

SavingsAccount

NUMBER	BALANCE	APY
2	100	2.3

CheckingAccount

NUMBER	BALANCE	OVERDRAFTLIMIT
1	500	200
3	23.5	0

+ Simple table structure
  + No Null values
+ Very efficient non-polymorphic queries
  + No joins needed

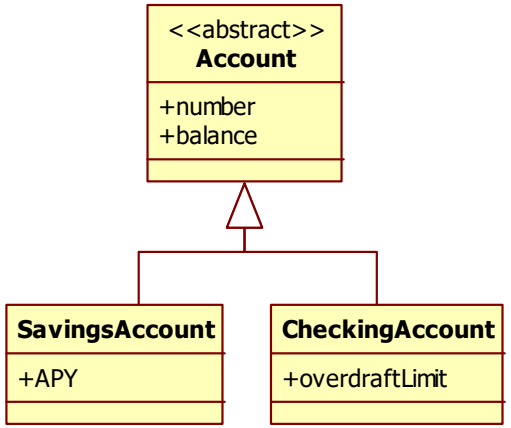
– Can not use Identity column ID generation
– JPA does not require its implementation (optional)
– Requires a UNION for polymorphic queries

Table per Class

Just specify the inheritance strategy, nothing else

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Account {
  @Id
  @GeneratedValue(strategy=GenerationType.TABLE)
  private long number;
  private double balance;

  ...
```

Id generation can not use identity column



Normal @Entity mapping

```
@Entity
public class SavingsAccount extends Account {
  private Double APY;

  ...
```

Java.util.Double instead of primitive double type


```
@Entity
public class CheckingAccount extends Account {
  private Double overdraftLimit;

  ...
```

Java.util.Double instead of primitive double type

Main point

- Class inheritance can be mapped in 3 different ways in the database.

  Science of Consciousness: The transcendental field of pure consciousness is the field of all possibilities.

COMPLEX MAPPING

Complex Mappings

- In this module we will cover:
  - Secondary tables – allow a class to be mapped to multiple tables
  - Embedded classes – allow multiple classes to be mapped to a single table
  - Composite keys – can be made using embedded classes

Secondary Tables

- Last module we used a secondary table to join a table to a single table per hierarchy strategy
- Secondary tables can be used anywhere to move properties into separate table(s)

Secondary table example from last module

```
@Entity
@DiscriminatorValue("savings")
@SecondaryTable(
  name="SavingsAccount",
  pkJoinColumns=@PrimaryKeyJoinColumn(name="number")
)
public class SavingsAccount extends Account {
  @Column(table="SavingsAccount")
  private double APY;

  ...
```

Secondary Table

@SecondaryTables can specify multiple @SecondaryTable

pkJoinColumns can be used to specify a multi column join

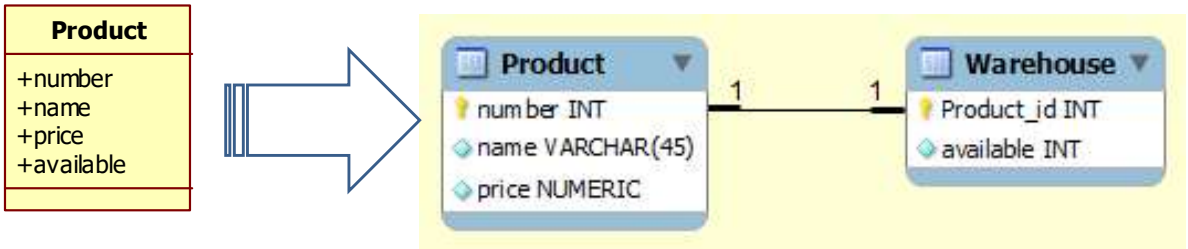
```
@Entity
@SecondaryTables(
  @SecondaryTable(name="warehouse", pkJoinColumns = {
    @PrimaryKeyJoinColumn(name="product_id", referencedColumnName="number")
  }
))
public class Product {
  @Id
  @GeneratedValue
  private int number;
  private String name;
  private BigDecimal price;
  @Column(table="warehouse")
  private boolean available;

  ...
```

JoinColumn name can differ from the referenced column

Properties need to specify the secondary table to be on it

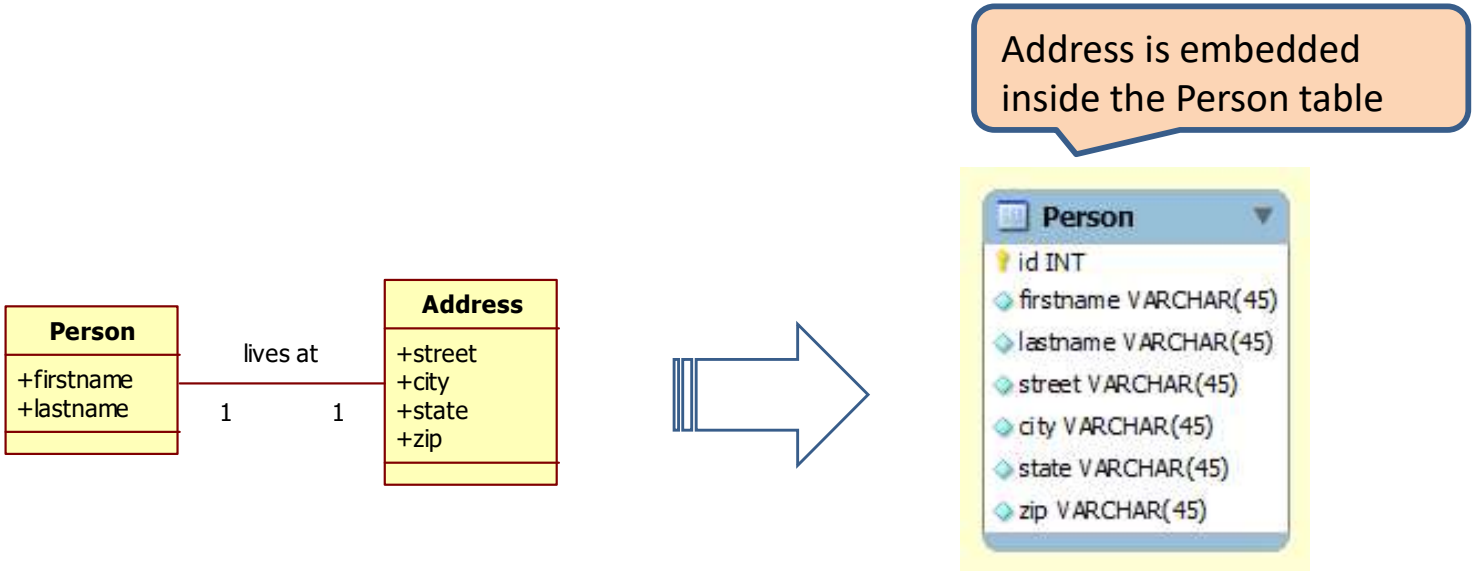
All you really need is @SecondaryTable and a name, the rest is optional

```
@Entity
@SecondaryTable(name="warehouse")
public class Product {
  @Id
  @GeneratedValue
  private int number;
  private String name;
  private BigDecimal price;
  @Column(table = "warehouse")
  private int available;

  ...
```



Embedded Classes

- Combine multiple classes in a single table
- Especially useful for tight associations
- These classes are considered value classes rather than entity classes



Embeddable

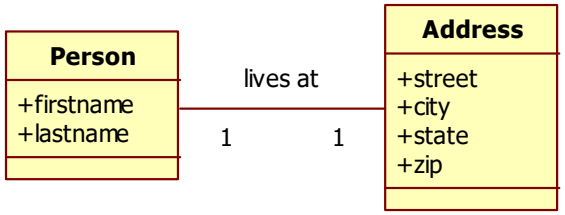
```
@Entity
public class Person {
  @Id
  @GeneratedValue
  private int id;
  private String firstname;
  private String lastname;

  @Embedded
  private Address address;

  ...
```

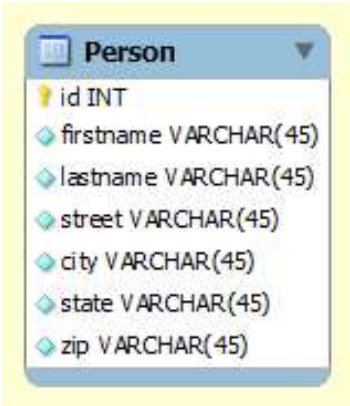
@Embedded annotation is used for embeddable objects

```
@Embeddable
public class Address {
  private String street;
  private String city;
  private String state;
  private String zip;

  ...
```

@Embeddable instead of @Entity

No @Id in embeddable





ID	FIRSTNAME	LASTNAME	STREET	CITY	STATE	ZIP
1	Frank	Brown	45 N Main St	Chicago	Illinois	51885

Multiple Embedded Addresses

```
@Entity
public class Customer {
  @Id
  @GeneratedValue
  private int id;
  private String firstname;
  private String lastname;

  @Embedded
  @AttributeOverrides( {
    @AttributeOverride(name="street", column=@Column(name="ship_street")),
    @AttributeOverride(name="city", column=@Column(name="ship_city")),
    @AttributeOverride(name="state", column=@Column(name="ship_state")),
    @AttributeOverride(name="zip", column=@Column(name="ship_zip"))
  })
  private Address shipping;

  @Embedded
  @AttributeOverrides( {
    @AttributeOverride(name="street", column=@Column(name="bill_street")),
    @AttributeOverride(name="city", column=@Column(name="bill_city")),
    @AttributeOverride(name="state", column=@Column(name="bill_state")),
    @AttributeOverride(name="zip", column=@Column(name="bill_zip"))
  })
  private Address billing;

  ...
```

Rename the column names for the embedded object using @AttributeOverrides

ID	FIRSTNAME	LASTNAME	SHIP_STREET	SHIP_CITY	SHIP_STATE	SHIP_ZIP	BILL_STREET	BILL_CITY	BILL_STATE	BILL_ZIP
1	Frank	Brown	45 N Main St	Chicago	Illinois	51885	100 W Adams St	Chicago	Illinois	60603

Composite Keys

- Composite Keys are multi-column Primary Keys
  - By definition these are natural keys
  - Have to be set by the application (not generated)
  - Generally found in legacy systems
  - Also create multi-column Foreign Keys

Composite Ids

@Embeddable

```
@Embeddable
public class Name implements Serializable {
  private String firstname;
  private String lastname;

  ...
```

Also requires hashCode and equals methods (see next slide)

```
@Entity
public class Employee {
  @Id
  private Name name;
  @Temporal(TemporalType.DATE)
  private Date startDate;

  ...
```

Embeddable object as identifier creates composite key

PK is made of Both firstname and lastname

equals() & hashCode()

```
@Embeddable
public class Name {
  private String firstname;
  private String lastname;

  ...

  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if ((obj == null) || obj.getClass() != this.getClass())
      return false;
    Name n = (Name) obj;
    if (firstname == n.firstname || (firstname != null && firstname.equals(n.firstname))
      && lastname == n.lastname || (lastname != null && lastname.equals(n.lastname))) {
      return true;
    } else {
      return false;
    }
  }

  public int hashCode() {
    int hash = 1234;
    if (firstname != null)
      hash = hash + firstname.hashCode();
    if (lastname != null)
      hash = hash + lastname.hashCode();
    return hash;
  }
```

Compares object contents for equality

Generates a unique int based on the class contents

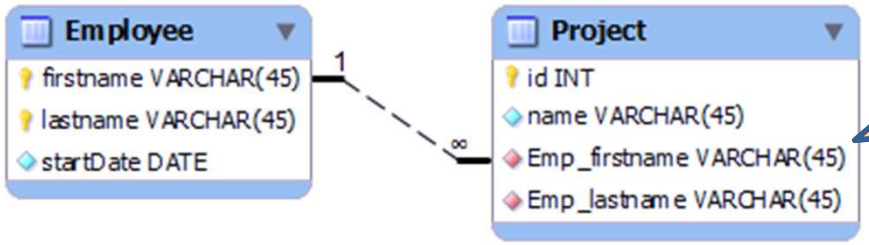
Foreign Keys to Composite Ids

```
@Entity
public class Employee {
  @Id
  private Name name;
  @Temporal(TemporalType.DATE)
  private Date startDate;
  @OneToMany(mappedBy = "owner")
  private List<Project> projects = new ArrayList<Project>();

  ...
```

Same Name embeddable @Id as before

Normal mappedBy on this side

```
@Entity
public class Project {
  @Id
  @GeneratedValue
  private int id;
  private String name;
  @ManyToOne
  @JoinColumns( {
    @JoinColumn(name = "Emp_firstname", referencedColumnName = "firstname"),
    @JoinColumn(name = "Emp_lastname", referencedColumnName = "lastname")
  })
  private Employee owner;

  ...
```

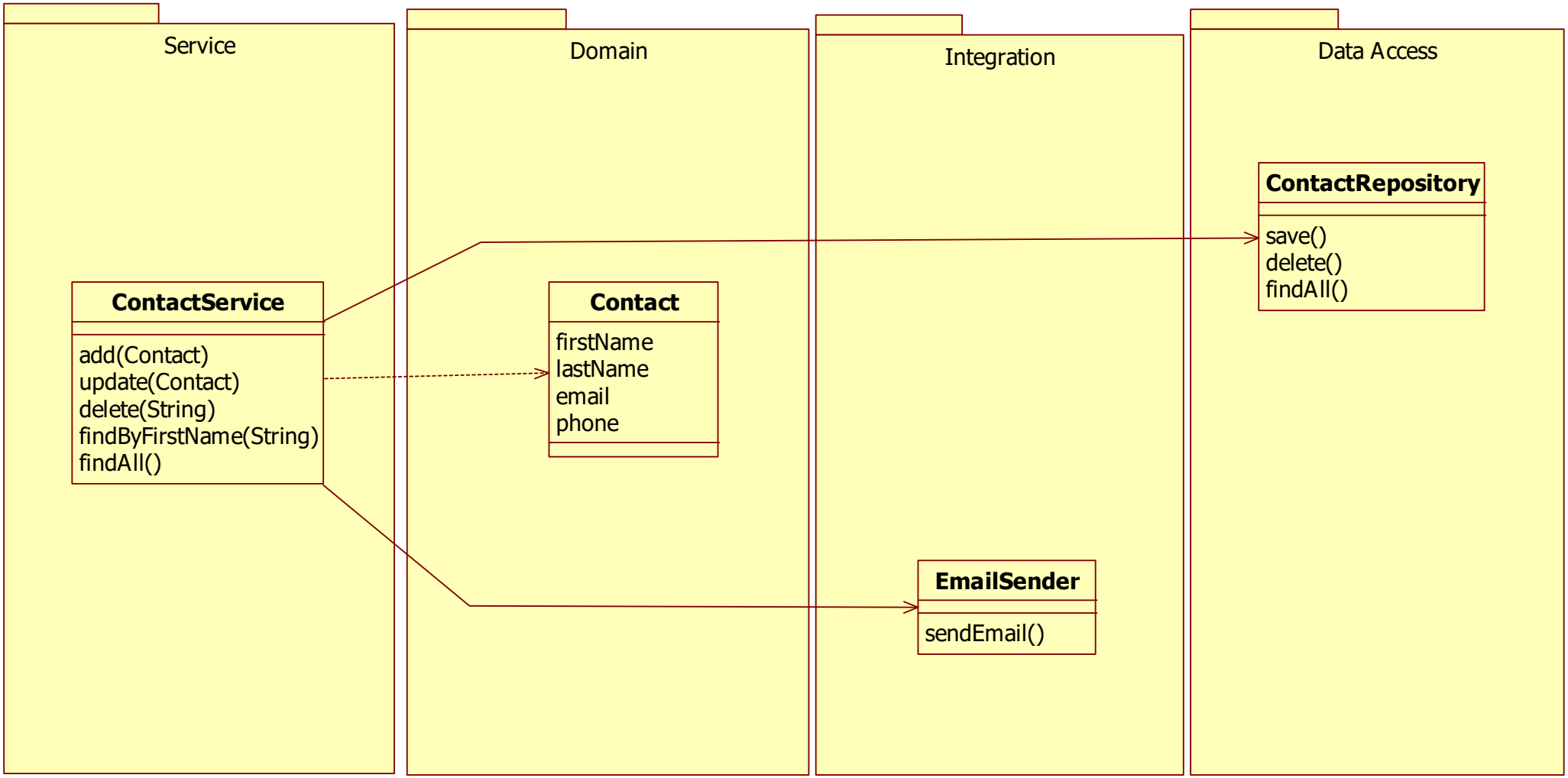


Two column Foreign Key

Two column FK specification

DATA TRANSFER OBJECTS (DTO)

What does findByFirstName return?



The entity and the repository

```
@Entity
public class Contact {
   @Id
   private long id;

   private String firstName;
   private String lastName;
   private String email;
   private String phone;
```

```
public interface ContactRepository extends JpaRepository<Contact, Long> {
   public Contact findByFirstName(String firstName);
}
```

The service

```
@Service
public class ContactService {
   @Autowired
   ContactRepository contactRepository;
   @Autowired
   EmailSender emailSender;

   public void add(Contact contact){
      contactRepository.save(contact);
      emailSender.sendEmail(contact.getEmail(), "Welcome");
   }
   public void update(Contact contact){
      contactRepository.save(contact);
   }
   public Contact findByFirstName(String firstName){
      return contactRepository.findByFirstName(firstName);
   }
   public void delete(String firstName){
      Contact contact = contactRepository.findByFirstName(firstName);
      emailSender.sendEmail(contact.getEmail(), "Good By");
      contactRepository.delete(contact);
   }
   public Collection<Contact> findAll(){
      return contactRepository.findAll();
   }
}
```

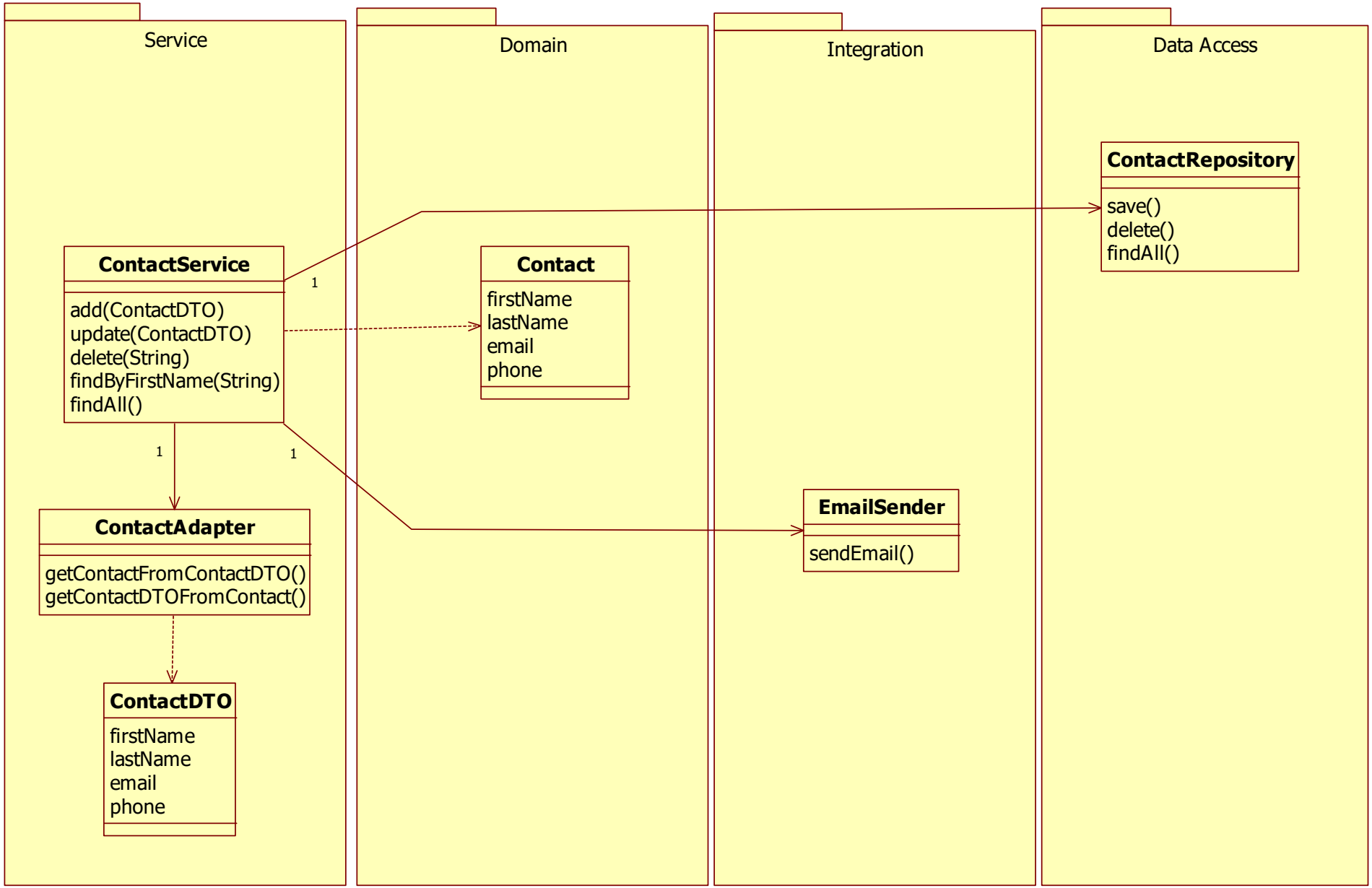
The Contact class is exposed to the client

The application

```
@SpringBootApplication
public class SpringBootDemoApplication implements CommandLineRunner {
  @Autowired
  private ContactService contactService;

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

  @Override
  public void run(String... args) throws Exception {
    contactService.add(new Contact("Frank","Brown","fbrown@gmail.com","4723459800"));
    System.out.println(contactService.findByFirstName("Frank"));
  }
}
```

The client knows about the Contact class

Data Transfer Objects (DTO)



The entity and the repository

```
@Entity
public class Contact {
   @Id
   private long id;

   private String firstName;
   private String lastName;
   private String email;
   private String phone;
```

```
public interface ContactRepository extends JpaRepository<Contact, Long> {
   public Contact findByFirstName(String firstName);
}
```

The DTO and the Adapter

```
public class ContactAdapter {
   public static Contact getContactFromContactDTO(ContactDTO contactDTO){
      return new Contact(contactDTO.getFirstName(),
            contactDTO.getLastName(),
            contactDTO.getEmail(),
            contactDTO.getPhone());
   }
   public static ContactDTO getContactDTOFromContact(Contact contact){
     return new ContactDTO(contact.getFirstName(),
          contact.getLastName(),
          contact.getEmail(),
          contact.getPhone());
   }

   public static List<ContactDTO> getContactDTOsFromContacts(List<Contact> contacts){
      List<ContactDTO> contactDTOs = new ArrayList<ContactDTO>();
      for (Contact contact: contacts){
         contactDTOs.add(getContactDTOFromContact(contact));
      }
      return contactDTOs;
   }
}
```

```
public class ContactDTO {

   private String firstName;
   private String lastName;
   private String email;
   private String phone;
```

The service(1/2)

Only the ContactDTO class is exposed to the client

```
@Service
public class ContactService {
   @Autowired
   ContactRepository contactRepository;
   @Autowired
   EmailSender emailSender;

   public void add(ContactDTO contactDTO){
      Contact contact = ContactAdapter.getContactFromContactDTO(contactDTO);
      contactRepository.save(contact);
      emailSender.sendEmail(contact.getEmail(), "Welcome");
   }
   public void update(ContactDTO contactDTO){
      Contact contact = ContactAdapter.getContactFromContactDTO(contactDTO);
      contactRepository.save(contact);
   }
   public ContactDTO findByFirstName(String firstName){
      Contact contact = contactRepository.findByFirstName(firstName);
      return ContactAdapter.getContactDTOFromContact(contact);
   }
```

The service(2/2)

```
    public void delete(String firstName){
        Contact contact = contactRepository.findByFirstName(firstName);
        emailSender.sendEmail(contact.getEmail(), "Good By");
        contactRepository.delete(contact);
    }

    public Collection<ContactDTO> findAll(){
        return ContactAdapter.getContactDTOsFromContacts(contactRepository.findAll());
    }
}
```

Only the ContactDTO class is exposed to the client

The application

```
@SpringBootApplication
public class SpringBootDemoApplication implements CommandLineRunner {
  @Autowired
  private ContactService contactService;

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

  @Override
  public void run(String... args) throws Exception {
    contactService.add(new ContactDTO("Frank","Brown","fbrown@gmail.com","4723459800"));
    System.out.println(contactService.findByFirstName("Frank"));
  }
}
```

The client only knows about the ContactDTO class

Main point

- Using DTO's gives loose coupling through information hiding.

  Science of Consciousness: Through the daily practice of TM one gets more and more access to the intelligence of creation.

Connecting the parts of knowledge with the wholeness of knowledge

1. Using JPA requires that the OO domain model looks very similar as the Relational database model.
2. Collections can be mapped as a Set, a Map, an unordered List and an ordered List

3. **Transcendental consciousness** is the most abstract field at the basis of all creation, with the greatest flexibility and power.
4. **Wholeness moving within itself:** In Unity Consciousness, we see that all layers of creation, from completely abstract to completely relative are nothing but the Self.