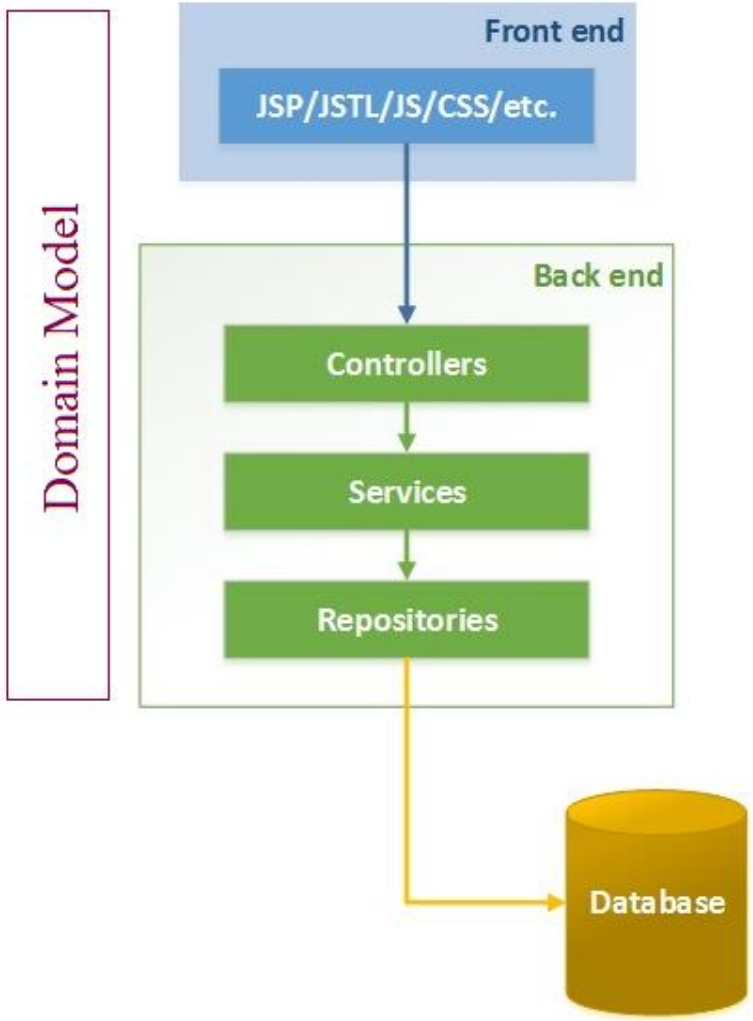
# Persistence & Transactions

Topping the Source of Pure Knowledge

# N-Tier Architecture - Spring MVC Layers

- Presentation [View & Controller] Tier
  - Communication interface to external entities
  - "View" in the model-view-controller
- Business [Services] Tier
  - Implements operations requested by clients through the presentation layer
  - Represents the "business logic"
- **Persistence** [Repository] Tier
  - Deals with different data sources of an information system
  - Responsible for storing and retrieving data



# Service Tier "manages" Persistence

- All access to Persistence through Services
- ***Services responsible for business Logic and data model composition***
  - Business logic does NOT belong in Persistence
  - Business logic does NOT belong in Presentation
- Spring/JPA/Persistence is designed with this architecture

# Object Relational Mapping

- In an application we want to focus on **business concepts, not on the** relational database structure
- Abstract from the "by-hand" communication with the DB (e.g., via JDBC)
- Allow for an automatic synchronization between Java Objects and the underlying database
- Portability
  - ORM should be mostly DB independent (with the exception of some types of features, such as identifier generation)
  - Query abstractions using e.g. JPQL or HQL - the vendor specific SQL is auto-generated
- Performance
  - Object and query caching is automatically done by the ORM

# Java Persistence API

- JPA is a specification – not an implementation.
- JPA 1.0 (2006). JPA 2.0 (2009), JPA 2.2(2017).
- Standardizes interface across industry platforms
- Object/Relational Mapping
  - **Specifically Persistence for RDBMS**

- Major Implementations [since 2006]:
  - Toplink - Oracle implementation [donated to Eclipse foundation for merge with Eclipselink 2008]
  - Hibernate - Most deployed framework. Major contributor to JPA specification.
  - OpenJPA - (openjpa.apache.org) which is an extension of Kodo implementation.

# JPA ORM Fundamentals

- Entity
  - lightweight persistence domain object
  - Annotation driven Entities - @Entity
- Persistence Context ~= Session in Hibernate
  - Like ***cache*** which contains a set of **persistent entities**
  - Within the **persistence context**, the entity instances and their **lifecycle are managed.**
- EntityManager
  - Basically a CRUD Service -- { persist, find, remove}.
  - Can Find entities by their primary key, and to query over all entities.
  - Can participate in a transaction.
- Transaction Manager
  - Java Transaction API
  - General API for managing transactions in Java
  - Start, Close, Commit, Rollback operations

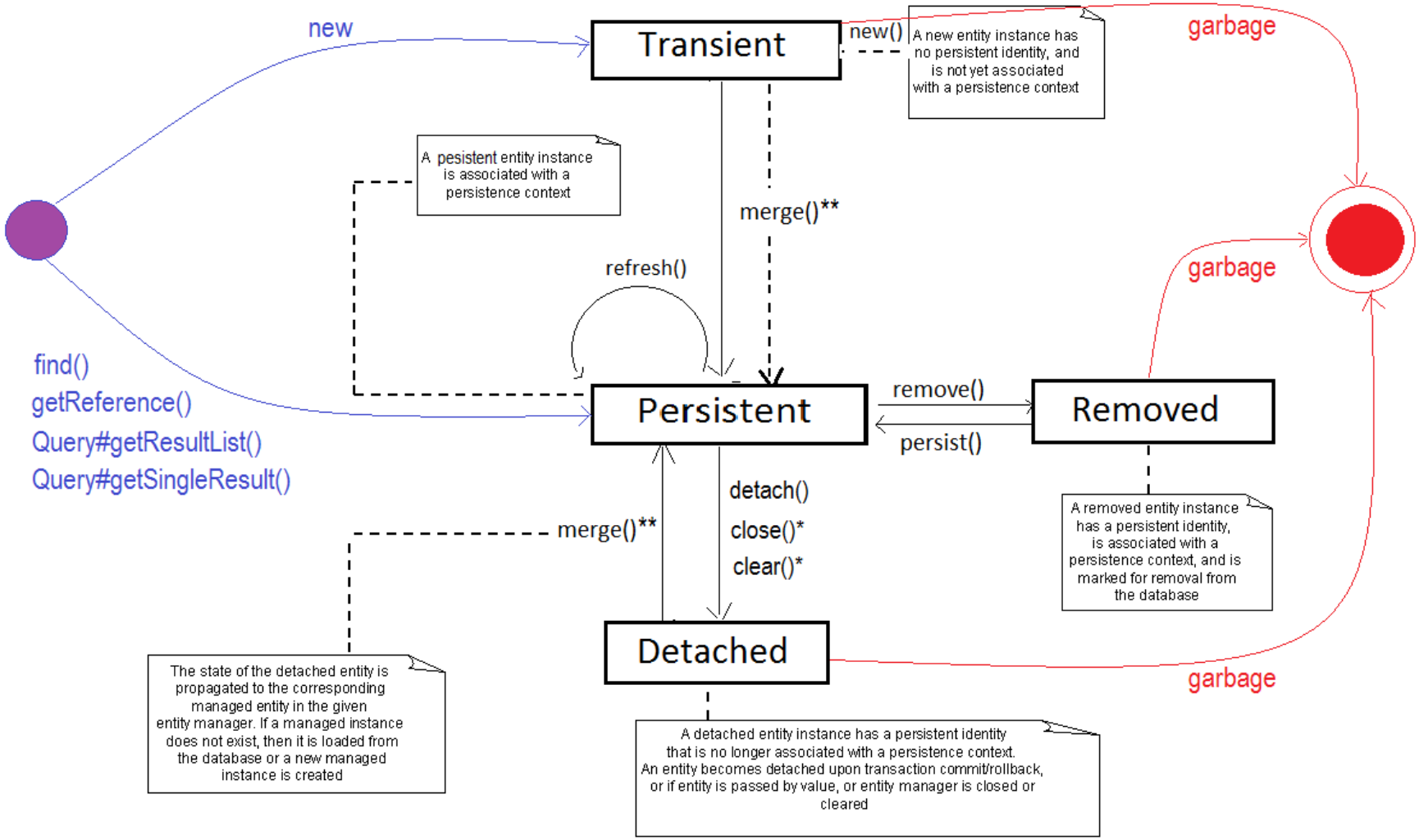
# Entity

- Are POJOs (Plain Old Java Objects)
- Lightweight persistent domain object
- Typically represent a table in a relational database
- Each entity instance corresponds to one row in that table
- Have a persistent identity
- May have both, persistent and transient (non-persistent) state
  - Simple types (primitive data types, wrappers, enums)
  - Composite types (e.g., Address)
  - Non-persistent state (using identifier `transient` or `@Transient` annotation)

# JPA ORM Fundamentals

- Entity
  - lightweight persistence domain object
  - Annotation driven Entities - @Entity
- Persistence Context ~= Session in Hibernate
  - Like ***cache*** which contains a set of **persistent entities**
  - Within the **persistence context**, the entity instances and their **lifecycle are managed.**
- EntityManager
  - Basically a CRUD Service -- { persist, find, remove}.
  - Can Find entities by their primary key, and to query over all entities.
  - Can participate in a transaction.
- Transaction Manager
  - Java Transaction API
  - General API for managing transactions in Java
  - Start, Close, Commit, Rollback operations

# Entity Lifecycle




*Affects all instances in the persistance context

**Merging returns a persistence instance, Original doesn't change state


JavaByDeveloper

# An Entity's Object States Relationship with the ORM Persistence Context

- **New, or Transient -** the entity has just been instantiated and is not associated with a persistence context. It has no persistent representation in the database and no identifier value has been assigned.
- **Managed, or Persistent -** the entity has an associated identifier and is associated with a persistence context.
- **Detached -** the entity has an associated identifier, but is no longer associated with a persistence context (usually because the persistence context was closed or the instance was evicted from the context).
- **Removed -** the entity has an associated identifier and is associated with a persistence context, however it is scheduled for removal from the database.

# JPA ORM Fundamentals

- Entity
  - lightweight persistence domain object
  - Annotation driven Entities - @Entity
- Persistence Context ~= Session in Hibernate
  - Like ***cache*** which contains a set of **persistent entities**
  - Within the **persistence context**, the entity instances and their **lifecycle are managed.**
- EntityManager
  - Basically a CRUD Service -- { persist, find, remove}.
  - Can Find entities by their primary key, and to query over all entities.
  - Can participate in a transaction.
- Transaction Manager
  - Java Transaction API
  - General API for managing transactions in Java
  - Start, Close, Commit, Rollback operations

Persistence Context ~= Session

Spring "manages" through @Transactional

# Transaction Management

- Unit of work

- **START –**

  Open a Session

  Open a single database connection

  Start a Transaction

- **Do the Work –**

  Associate & Manage entities W/R the session

  Exercise DB CRUD operations

- **END –**

  End Transaction

  Close a Session

# Cascade and Fetch
# Configurable Parent-Child operations

- Cascade Types
  - ALL, PERSIST, MERGE, REMOVE, REFRESH, DETACH
  - Default is none
  - **Persist**
    - If the parent is persisted so are the children
  - **Remove**
    - If the parent is "removed" so are the children
  - **Merge [ a detatched object]**
    - If the parent is merged so are the children
      - Merge modifications made to the detached object are merged into a corresponding **DIFFERENT** managed object

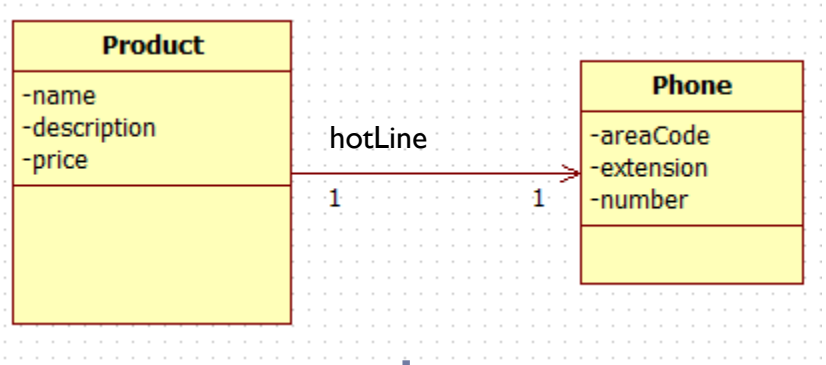
- Fetching Strategies
  - Define how object hierarchies are loaded
  - EAGER = load all related objects immediately
  - LAZY = load the related objects only if they are accessed for the first time
  - Be careful with EAGER, as large object graphs may be loaded unintentionally

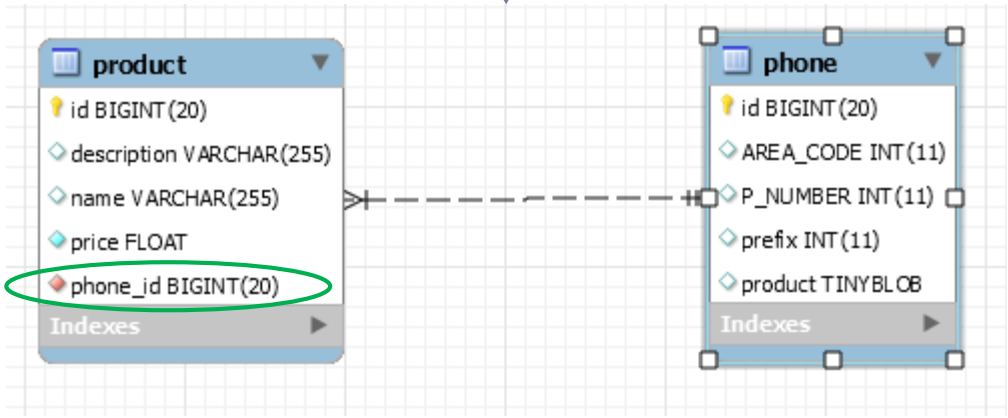
# ORM Relationships

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

- Unidirectional - Bidirectional

# OneToOne Unidirectional



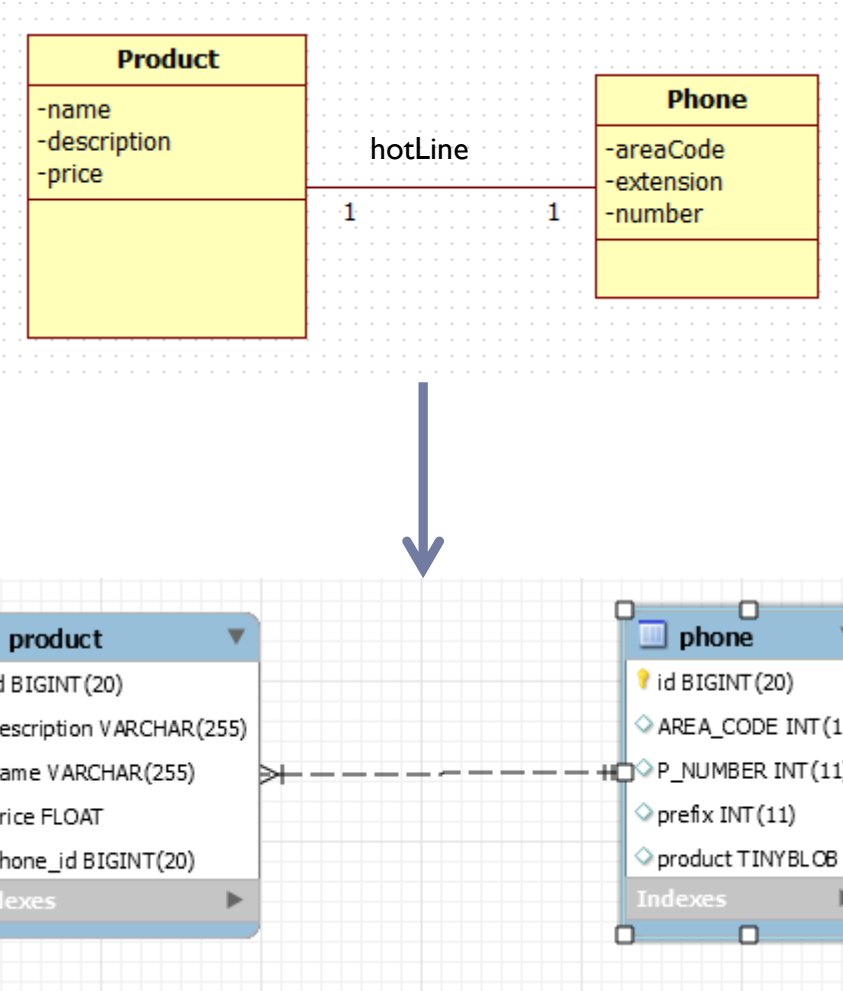



# OneToOne Unidirectional

```
@Entity
public class Product implements
   Serializable {

   @Id
   @GeneratedValue(strategy=Generat
      ionType.AUTO)
   private long id;

   private String name;
   private String description;
   private float price;

   @OneToOne(cascade =
      CascadeType.ALL)
   @JoinColumn(name = "phone_id",
      nullable = false)
   private Phone hotLine;
}
```

```
@Entity
public class Phone implements
   Serializable {

   @Id
   @GeneratedValue(strategy=Generat
      ionType.AUTO)
   private long id;

   private Integer areacode;
   private Integer number;
   private Integer prefix;
}
```

# OneToOne Bi-directional



# OneToOne Bi-directional

- Annotation the OTHER side of the relationship ALSO…

```
@Entity
public class Product implements
   Serializable {

   @Id
   @GeneratedValue(strategy=GenerationT
      ype.AUTO)
   private long id;

   private String name;
   private String description;
   private float price;

   @OneToOne(cascade = CascadeType.ALL)
   @JoinColumn(name = "phone_id",
      nullable = false)
   private Phone hotLine;
}
```

```
@Entity
public class Phone implements
   Serializable {

   @Id
   @GeneratedValue(strategy =
      GenerationType.AUTO)
   private long id;

   private Integer areacode;
   private Integer number;
   private Integer prefix;

   @OneToOne(mappedBy = "hotLine",
      cascade = CascadeType.ALL)
   private Product product;
}
```

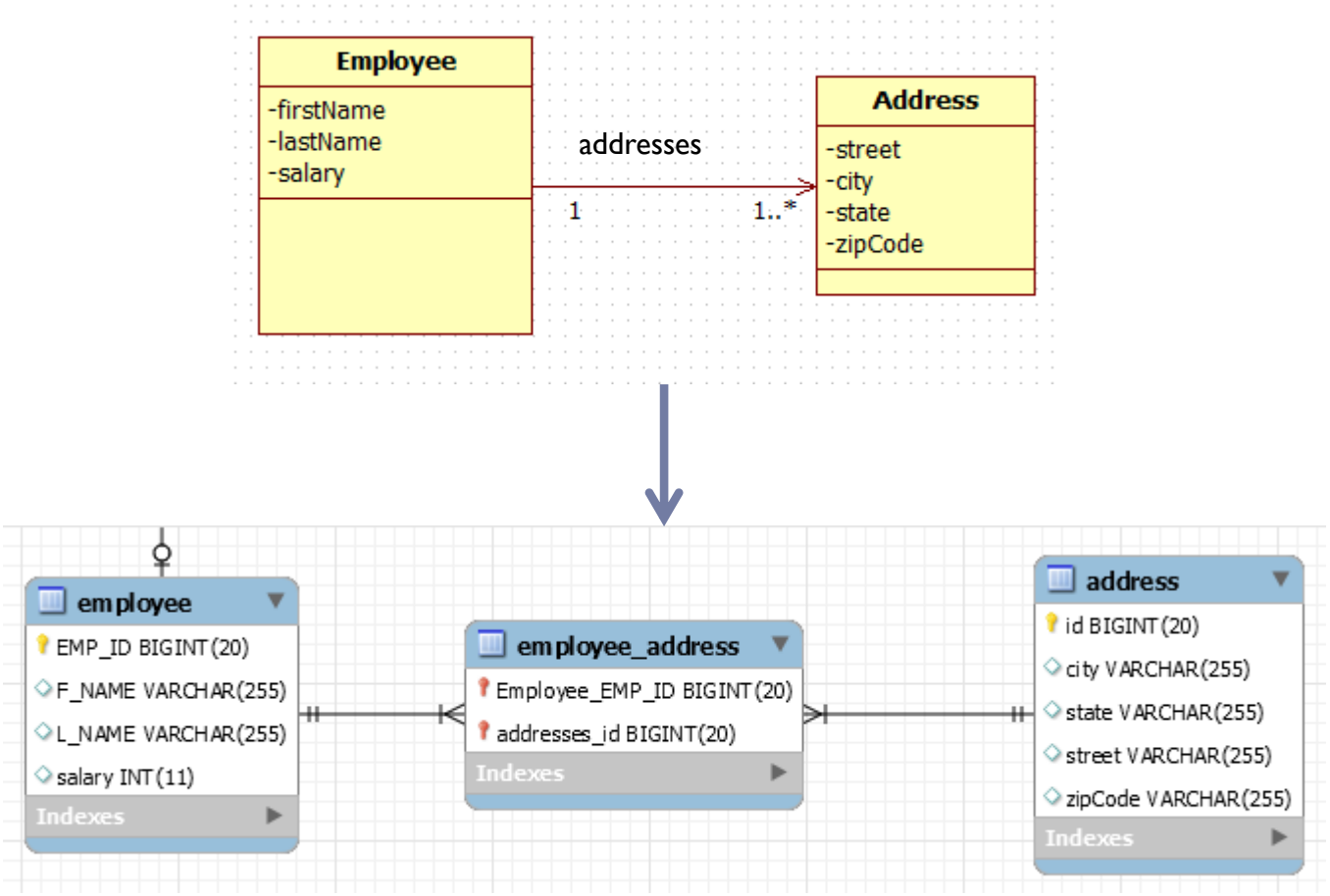
mappedBy – use the foreign key and mapping in the *source* to define the *target* mapping

# Bi-directional Relationships

**WARNING NOTICE**

- If you add or remove to one side of the collection, you **must** also add or remove from the other side
- Database will be updated correctly **ONLY** if you add/remove from the owning side of the relationship
- Your object model can get out of sync if you do not pay attention…

# One-to-Many Join Table



# OneToMany Unidirectional JoinTable

```java
@Entity
@Table(name = "employee")
public class Employee implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "EMP_ID")
    private long id;

    @Column(name = "F_NAME")
    private String firstName;

    @Column(name = "L_NAME")
    private String lastName;

    private Integer salary;

    @OneToMany(cascade = CascadeType.ALL)
    // FetchMode.JOIN will do eager load also
    @Fetch(FetchMode.JOIN)
    private List<Address> addresses;
}
```

```java
@Entity
public class Address implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    private String street;
    private String city;

    private String state;

    private String zipCode;
}
```

This is the Default

# OneToMany Bidirectional JoinTable

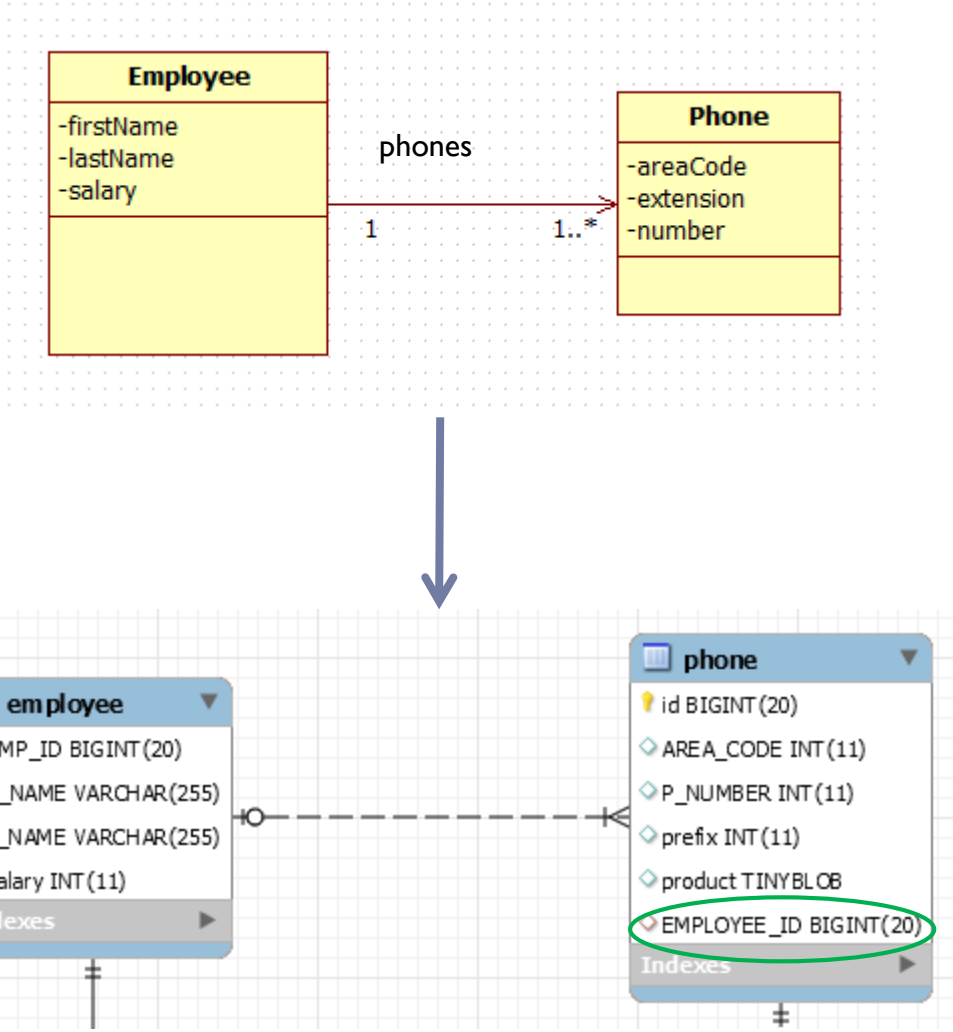
## OneToMany side same as unidirectional example

```
@Entity
@Table(name = "emp")
public class Employee implements Serializable {


    @Id
    @GeneratedValue(strategy =
        GenerationType.AUTO)
    @Column(name = "EMP_ID")
    private long id;


    @Column(name = "F_NAME")
    private String firstName;


    @OneToMany(cascade = CascadeType.ALL,
        mappedBy="employee")
    // FetchMode.JOIN will do eager load also
    @Fetch(FetchMode.JOIN)
    private List<Address> addresses;
}
```

## Simply Add ManyToOne on child object

```
@Entity
public class Address implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    private String street;
    private String city;

    private String state;

    private String zipCode;


    @ManyToOne
@JoinTable(name="emp_addr", joinColumns =
@JoinColumn(name="emp_id"),
 inverseJoinColumns =
@JoinColumn(name="add_id"))
    private Employee employee;
}
```

# OneToMany Unidirectional JoinColumn



# OneToMany Unidirectional JoinColumn

```
@Entity
@Table(name = "emp")
public class Employee implements
   Serializable {

   @Id
   @GeneratedValue(strategy =
      GenerationType.AUTO)
   @Column(name = "EMP_ID")
   private long id;

   @Column(name = "F_NAME")
   private String firstName;

   @OneToMany(cascade =
      CascadeType.ALL)
   @Fetch(FetchMode.JOIN)
   @JoinColumn(name = "EMPLOYEE_ID")
   private List<Phone> phones;

}
```

```
@Entity
public class Phone implements
   Serializable {

   @Id
   @GeneratedValue(strategy=GenerationT
      ype.AUTO)
   private long id;

   private Integer areacode;
   private Integer number;
   private Integer prefix;
}
```

HIBERNATE REFERENCE DOC:
A *unidirectional one-to-many association on a foreign key* is an unusual case, and is not recommended. You should instead use a join table for this kind of association.

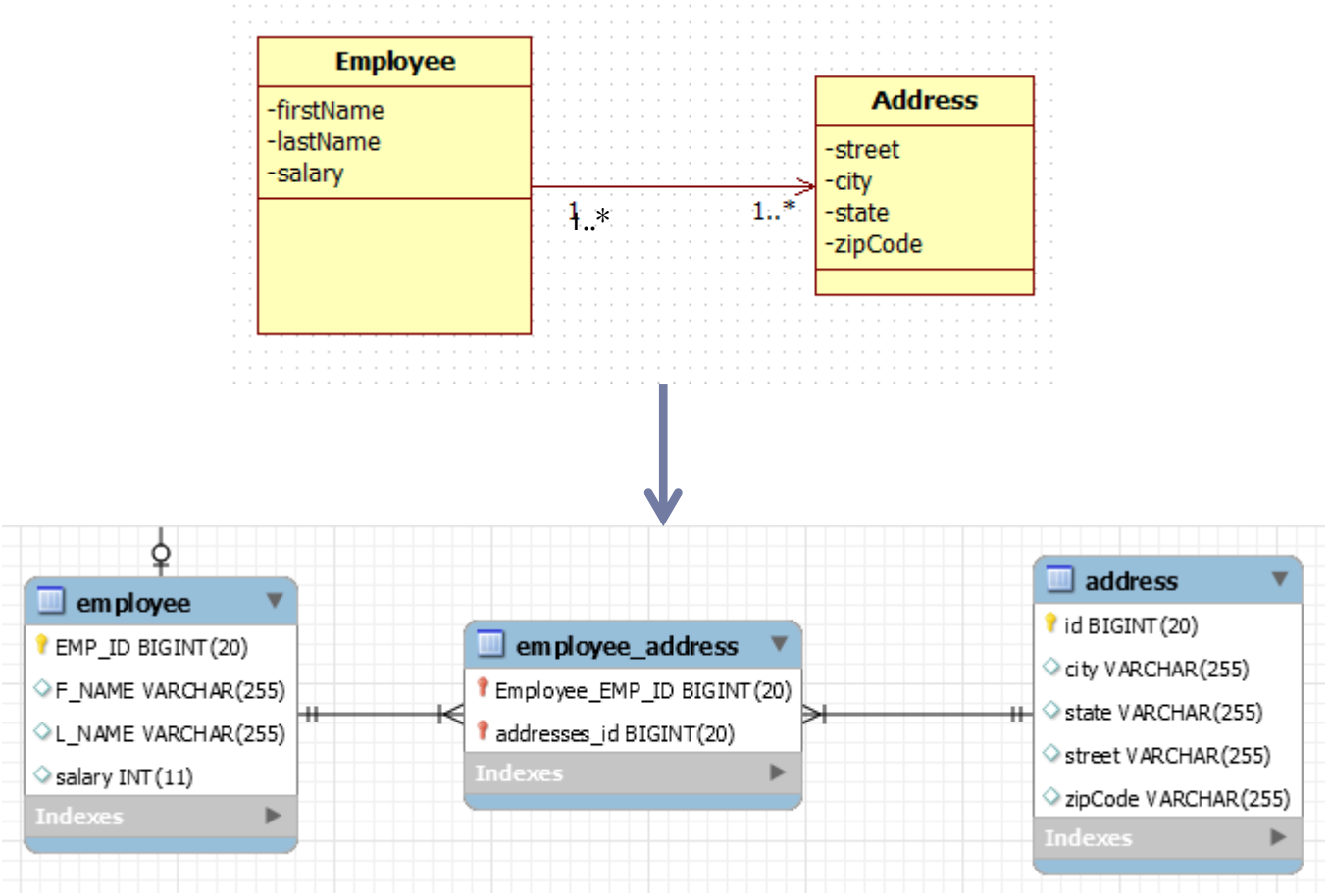
# OneToMany Bi-directional JoinColumn

```
@Entity
@Table(name = "emp")
public class Employee implements
   Serializable {


   @Id
   @GeneratedValue(strategy =
      GenerationType.AUTO)
   @Column(name = "EMP_ID")
   private long id;


   @Column(name = "F_NAME")
   private String firstName;


   @OneToMany(cascade = CascadeType.ALL,
      mappedBy = "employee")
   private List<Phone> phones;

}
```

```
@Entity
public class Phone implements
   Serializable {


   @Id
   @GeneratedValue(strategy =
      GenerationType.AUTO)
   private long id;


   private Integer areacode;
   private Integer number;
   private Integer prefix;


   @ManyToOne
   @JoinColumn(name = "EMP_ID")
   private Employee employee;


}
```

Owns relationship

# Many-to-Many


Employee
-firstName
-lastName
-salary
1..*
1..*
Address
-street
-city
-state
-zipCode
employee
EMP_ID BIGINT(20)
F_NAME VARCHAR(255)
L_NAME VARCHAR(255)
salary INT(11)
Indexes
employee_address
Employee_EMP_ID BIGINT(20)
addresses_id BIGINT(20)
Indexes
address
id BIGINT(20)
city VARCHAR(255)
state VARCHAR(255)
street VARCHAR(255)
zipCode VARCHAR(255)
Indexes

# Many-To-Many

```
@Entity
@Table(name = "emp")
public class Employee implements
   Serializable {


   @Id
   @GeneratedValue(strategy =
      GenerationType.AUTO)
   @Column(name = "EMP_ID")
   private long id;


   @ManyToMany(cascade = {
      CascadeType.ALL })
   @JoinTable(name="emp_project")
   Set<Project> projects = new
      HashSet<>();
}
```

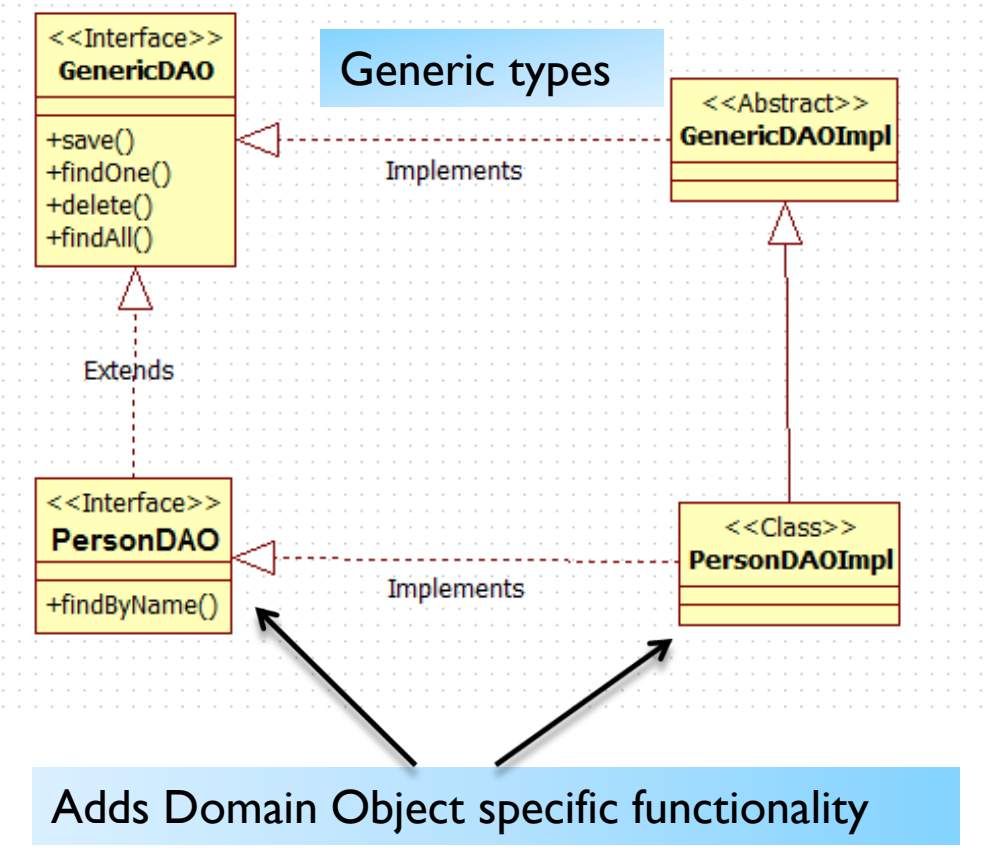
```
@Entity
public class Project implements
   Serializable{


   @Id
   @GeneratedValue(strategy =
      GenerationType.AUTO)
   private long id;


   private String name;


   @ManyToMany(mappedBy = "projects")
   private Set<Employee> employees =
      new HashSet<>();
}
```

If Converting from OneToMany [Join table] – The ManyToMany is achieved by simply dropping the unique constraint on the JoinTable created by OneToMany

# Main Point

- JPA is a specification not an implementation. It provides a consistent, reliable mechanism for data storage and retrieval that alleviates the application developer from the details involved in the persistence layer.
- *The mechanism of transcending allows the individual to tap into Transcendental Consciousness and enlivens its qualities in activity.*

# “Classic” ORM GenericDAO



# Spring Version of DAO

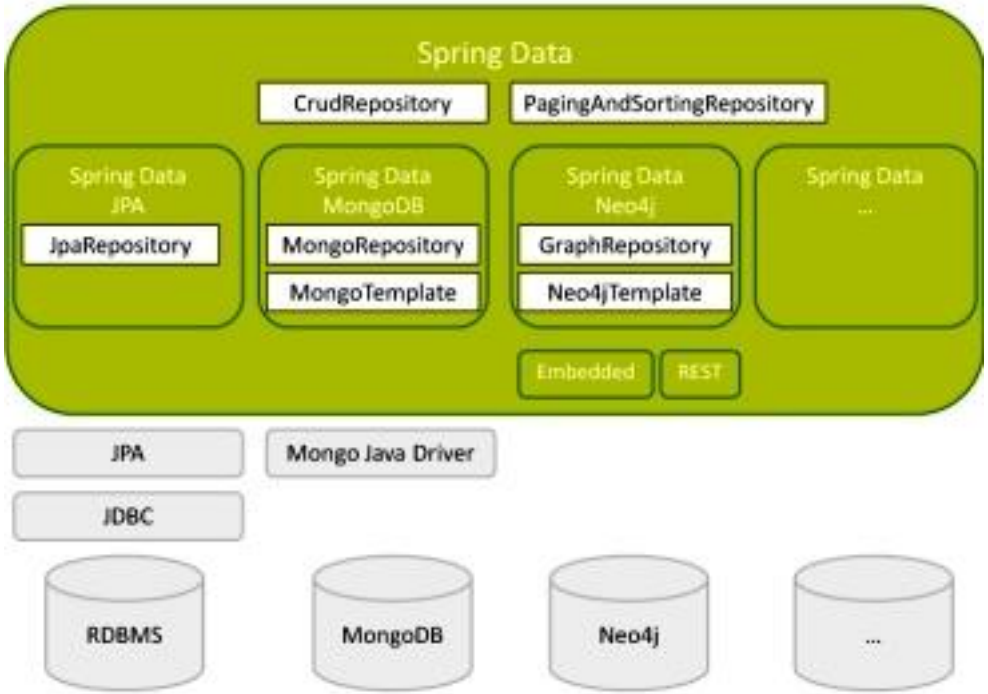
```
@Repository
public interface PhoneRepository extends CrudRepository<Phone, Long> {
}
```

# Spring Data

- Spring Data
  - High level SpringSource project whose purpose is to unify and ease the access to different kinds of persistence stores, both relational database systems and NoSQL data stores.

- Hibernate ORM
  - (Hibernate for short) is an object-relational mapping Java library;
  - a framework for mapping an object-oriented domain model to a traditional relational database. Distributed under the GNU Lesser (General Public License).

# Spring Data Project



# Spring Data Repositories

- Spring Data repository abstraction
- Significantly reduce the amount of boilerplate code required to implement data access layers
- Domain Object specific wrapper that provides capabilities on top of EntityManager
- Performs function of a Base Class DAO

- Three Types:
  - **`CrudRepository`**: provides CRUD functions.
  - **`PagingAndSortingRepository`**: provide methods to do pagination and sorting records.
  - **`JpaRepository`**: provides methods such as flushing the persistence context and delete record in a batch.

# Wiring the Components (XML)

```
<bean id="entityManager"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
   <property name="dataSource" ref="dataSource" />
   <property name="packagesToScan" value="edu.mum.domain" />
   <property name="jpaVendorAdapter">
   <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
   </property>
   <property name="jpaProperties">
      <props>
        <prop key="hibernate.hbm2ddl.auto">create</prop>
        <prop key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>
      </props>
   </property>
</bean>

<bean id="transactionManager"
   class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="entityManager" />
</bean>
```

# Wiring Continued

- Scan for interfaces extending Repository

```
<jpa:repositories base-
  package="com.packt.webstore.domain.repository"/>
```

- Scan for transaction-based resources

```
<context:component-scan base-package=
"com.packt.webstore.service" />

<context:component-scan base-package=
"com.packt.webstore.repository"/>

<tx:annotation-driven transaction-
  manager="transactionManager"/>
```

# Wiring the Components (Java Config)

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(basePackages = { "edu.mum.repository" })
public class RootApplicationContextConfig {
    @Bean
    public EntityManagerFactory entityManagerFactory() {
        HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
        vendorAdapter.setGenerateDdl(true);
        LocalContainerEntityManagerFactoryBean factory = new LocalContainerEntityManagerFactoryBean();
        factory.setJpaVendorAdapter(vendorAdapter);
        factory.setPackagesToScan("edu.mum.domain");
        factory.setDataSource(dataSource());
        factory.afterPropertiesSet();
        factory.setJpaProperties(additionalProperties());
        return factory.getObject();
    }
    @Bean
    public PlatformTransactionManager transactionManager() {
        JpaTransactionManager txManager = new JpaTransactionManager();
        txManager.setEntityManagerFactory(entityManagerFactory());
        return txManager;
    }
}
```

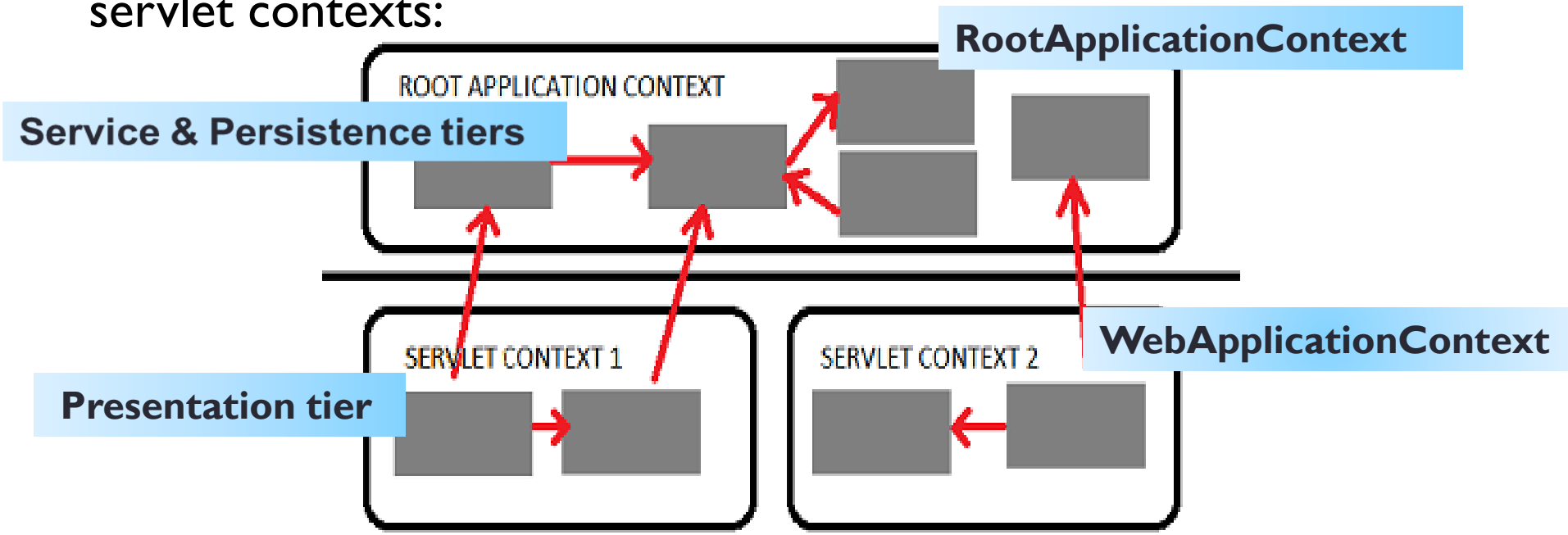
# Spring Boot Configuration

- No – if follow standard, in-memory database

- Not in-memory databse configuration
  - spring.jpa.hibernate.ddl-auto=create
  - spring.datasource.url=jdbc:mysql://localhost:3306/boot_onetoone
  - spring.datasource.username=root
  - spring.datasource.password=root
  - spring.datasource.driver-class-name=com.mysql.jdbc.Driver

# Web Application Context

- Spring has multilevel application context hierarchies.
- Web apps by default have two hierarchy levels, root and servlet contexts:




- Presentation tier has a WebApplicationContext
- [Servlet Context] which inherits all the resources already defined in the root ApplicationContext [ Services, Persistence]

# Spring DAO

```
@Service
@Transactional
public class PhoneServiceImpl implements PhoneService {

  @Autowired
  PhoneRepository phoneRepository;

  public List<Phone> getAll() {
    return Util.iterableToCollection(phoneRepository.findAll());
  }

  public Phone save(Phone phone) {
    return phoneRepository.save(phone);
  }

}
```

```
@Repository
public interface PhoneRepository extends CrudRepository<Phone, Long> {
}
```

# CrudRepository

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {

    <S extends T> S save(S entity);

    <S extends T> Iterable<S> saveAll(Iterable<S> entities);

    Optional<T> findById(ID id);

    boolean existsById(ID id);

    Iterable<T> findAll();

    Iterable<T> findAllById(Iterable<ID> ids);

    long count();

    void deleteById(ID id);

    void delete(T entity);

    void deleteAll(Iterable<? extends T> entities);

    void deleteAll();
}
```

LOOKS just Like [what is Known as] a "genericDAO interface"
HOWEVER, Spring provides [default] implementations – effectively Java 8-like default methods in an interface

# Spring Data Repository Query Resolution Query examples - CREATE example

- CREATE example
  - attempts to construct a store-specific query from the query method name.
  - The name of our query method must start with one of the following prefixes: *find…By*, *read…By*, *query…By*, *count…By*, and *get…By*.
  - limit the number of returned query results: *findTopBy*, *findTop8By*, *findFirstBy*, and *findFirst8By*
  - select unique results: *findTitleDistinctBy* or *findDistinctTitleBy*

```
@Repository
public interface PhoneRepository extends CrudRepository<Phone, Long> {
    public List<Phone> findByAreacodeOrPrefix(String areacode, String prefix);
    public long countByAreacode(String areacode);
}
```

Query Key Words  Query lookup strategies

# JPQL - Data Object Queries

- JPQL is similar to SQL, but operates on objects, attributes and relationships instead of tables and columns.

```
@Entity
public class Product implements Serializable {
    private String name;
    @OneToOne(cascade = CascadeType.ALL)
    private Phone hotLine;
}
```

- **JPQL:**
  - **SELECT p FROM Product p**
- Will Yield:
  - **Product with Phone;**
- Where:
  - **product.getHotLine().getNumber();** is populated

- **NOTE: JPA `OneToOne` relationship defaults to eager**

# Spring Data Repository Query Resolution Query examples - USE_DECLARED_QUERY

- USE_DECLARED_QUERY
  - tries to find a declared query and will throw an exception in case it can't find one.

- **JPQL Queries**

```
@Query(value = "SELECT e FROM Employee e WHERE e.lastName = :lastname")
public List<Employee> findByLastName(String lastname);
```

- **SQL Queries**

```
@Query(value = "SELECT * FROM emp e WHERE e.F_NAME = ?1", nativeQuery = true)
public List<Employee> findByFirstName(String firstName);
```

- **CLASS LEVEL DECLARED**

```
public final static  String FIND_BY_SALARY_QUERY = "SELECT e FROM
   Employee e WHERE e.salary = :salary";

@Query(FIND_BY_SALARY_QUERY)
public List<Employee> findByAddress(@Param("salary") Integer salary);
```

# Spring Data Repository Query Resolution Query examples - JPA Named Query

- Declaration:

```
@Entity
@Table(name = "emp")
@NamedQuery(name = "Employee.findEmployeesByLastName", query = "SELECT e FROM Employee e
   WHERE LOWER(e.lastName) = LOWER(:lastName)")
public class Employee implements Serializable {


}
```

*Query name convention:* @{EntityName}.{queryName}

- Usage:

```
@Repository
public interface EmployeeRepository extends CrudRepository<Employee, Long> {
   public List<Employee> findEmployeesByLastName(@Param("lastName") String lastName);
}
```

# Main Point

- Spring provides a Transactional capability for ORM applications.
- The *mechanism of transcending allows the individual to tap into Transcendental Consciousness and enlivens its qualities in activity.*