2) The difference between lazy and eager loading is described in the table.

S.No.	Key	Lazy	Eager
1	Fetching strategy	In Lazy loading, associated data loads only when we explicitly call getter or size method.	In Eager loading, data loading happens at the time of their parent is fetched
2	Default Strategy in ORM Layers	ManyToMany and OneToMany associations used a lazy loading strategy by default.	ManyToOne and OneToOne associations used a lazy loading strategy by default.
3	Loading Configuration	It can be enabled by using the annotation parameter : fetch = FetchType.LAZY	It can be enabled by using the annotation parameter : fetch = FetchType.EAGER
4	Performance	Initial load time much smaller than Eager loading	Loading too much unnecessary data might impact performance

Performance improvement can be explained with an example. Consider the teacher and course as a one-to-many relationship. The **Course** is a heavy object i.e. having too many attributes. For some business logic, all **Teacher** objects are loaded from the database. Business logic does not need to retrieve or use courses in it, but **Course** objects are still being loaded alongside the **Teacher** objects.
Such loading will degrade the performance of applications. JPA has considered all such problems ahead and made **One-to-Many relationships load *lazily* by default**. Lazy loading means relationships will be loaded when it is actually needed.

3) One of the main aspects of JPA is that it helps to propagate Parent to Child relationships. This behavior is possible through CascadeTypes. The CascadeTypes supported by JPA are:

Cascade Type	Description
ALL	CascadeType.ALL propagates **all operations,** including hibernate specific from a parent to a child entity.
PERSIST	CascadeType.PERSIST propagates the **save()** or **persist()** operation to the related entities.
MERGE	CascadeType.MERGE propagates only the **merge()** operation to the related entities.
REFRESH	CascadeType.REFRESH propagates only the **refresh()** operation to the related entities.
REMOVE	CascadeType.REMOVE removes all related entities associations when the owning entity is deleted.
DETACH	CascadeType.DETACH detaches all related entities if the owning entity is detached.

Cascading is useful only for Parent - Child associations where the parent entity state transitions cascade to the child entity. The cascade configuration option accepts an array of CascadeTypes. The below example shows how to add the refresh and merge operations in the cascade operations for a One-to-Many relationship:

```
@OneToMany(cascade={CascadeType.REFRESH, CascadeType.MERGE},
fetch = FetchType.LAZY)
@JoinColumn(name="EMPLOYEE_ID")
private Set<AccountEntity> accounts;
```

Check Your Progress 3

1) **One-to-Many** mapping is described as one row in a table is mapped to multiple rows in another table and Hibernate provides annotation @OnetoMany. The attribute "cascade=CascadeType.ALL" means that any change that happens to Employee Entity must cascade to all associated entities (Accounts Entity in this case) also. This means that if you save an employee into the database, then all associated accounts will also be saved into the database. If an Employee is deleted, then all accounts associated with that Employee will also be deleted. However, if we want to cascade only the save operation but not the delete operation, we need to clearly specify it using below code.

```
@OneToMany(cascade=CascadeType.PERSIST, fetch = FetchType.LAZY)
@JoinColumn(name="EMPLOYEE_ID")
private Set<AccountEntity> accounts;
```

Now only when the save() or persist() methods are called using an employee instance, the accounts will be persisted. If any other method is called on session, its effect will not affect/cascade to the accounts entity.

2) Spring Data framework has an in-built query builder mechanism that derives queries based on the method name. Method names are defined as 'find...By..', 'count...By...' etc. 'By' acts as the delimiter to identify the start of the criteria. 'And' and 'Or' are used to define multiple criteria. 'OrderBy' is used to identify the order by criteria. Few examples of query methods are as follows:
   - **findByFirstname(String firstName):** It derives the query to get the list based on the first name.
   - **findByFirstnameAndLastname(String firstName, String lastName):** It derives the query to get the list based on first name and last name
   - **findByFirstnameAndLastnameOrderByFirstnameAcs(String firstName, string last):** it derives the query to get the list based on the first name and sorted by the first name in ascending order.

3) UserRepository extends the Repository marker interface. Marker interface does not contain any method. Thus, UserRepository provides a single API to find the list of users based on the email address and last name. The query method:
   List<User>findByEmailAddressAndLastname(String emailAddress, String lastname) translates into the following query:
   select u from User u where u.emailAddress = ?1 and u.lastname = ?2.

4) BookRepository extends the CrudRepository interface, which provides the methods to read all records, specific record by id etc. All methods generate the same queries, but there are some differences.
   1. Query methods using Containing generates the following sql query.
      SELECT * FROM books WHERE title LIKE '%<pattern>%'

2. The ***findByTitleContainsIgnoreCase(String pattern)*** is similar to ***findByTitleContaining(String pattern)*** but with case insensitive.
3. The **findByTitleLike(String pattern)** generates **SELECT * FROM books WHERE title LIKE '<pattern>'**. JPA Like keyword behaves differently. It can be noticed in the generated query that it does not include wild char %. Thus, while calling this method, argument should have wild char included.
4. Sometimes we need to create queries that are too complicated for Query Methods or would result in absurdly long method names. In those cases, we can use the @Query annotation to query our database. The method findByTitile with @Query annotation is similar to the first method.

11.10 REFERENCES/FURTHER READING

- Craig Walls, "Spring Boot in action" Manning Publications, 2016. (https://doc.lagout.org/programmation/Spring%20Boot%20in%20Action.pdf)
- Christian Bauer, Gavin King, and Gary Gregory, "Java Persistence with Hibernate",Manning Publications, 2015.
- Ethan Marcotte, "Responsive Web Design", Jeffrey Zeldman Publication, 2011(http://nadin.miem.edu.ru/images_2015/responsive-web-design-2nd-edition.pdf)
- Tomcy John, "Hands-On Spring Security 5 for Reactive Applications",Packt Publishing,2018
- https://spring.io/projects/spring-data
- https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa
- https://github.com/spring-projects/spring-data-examples
- https://dzone.com/articles/magic-of-spring-data
- https://www.journaldev.com/17034/spring-data-jpa
- https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/associations.html
- https://www.baeldung.com/spring-data-rest-relationships