1. **Pesquisar e descrever resumidamente como implementar os demais tipos de relacionamento (One to Many, Many to One e Many to Many).**

Esses relacionamentos são indicativos que é feito um mapeamento ORM para representar os dados de um banco de dados. Na explicação vou usar uma locadora de veículos como exemplo.

One to Many
Relacionamento representado por uma anotação @OneToMany, indicando que o relacionamento de uma classe com outra é 1 …* (um para muitos) onde a classe que recebe essa anotação geralmente recebe uma coleção da outra classe relacionada.

Ex:
class LocadoraDeVeiculos{ ...
@OneToMany
private List<Carro> carros;
…}

Many to One

Relacionamento representado por uma anotação @ManyToOne, indicando que o relacionamento de uma classe com outra é * …1 (muitos para um), a classe que recebe essa anotação geralmente recebe se relaciona com uma entidade relacionada.

Ex:
class Carro{ ...
@ManyToOne
private LocadoraDeVeiculos locadoraDeVeiculos;
…}

Many to Many

Relacionamento representado por uma anotação @ManyToMany, indicando que há um relacionamento entre classes, onde existe atributos que são *…* (muitos para muitos), geralmente a classe recebe uma coleção da classe relacionada e da mesma forma a outra classe recebe uma coleção dessa classe atual.

Ex:
class Carro{ ...
@ManyToMany
private List<LocadoraDeVeiculos> locadorasDeVeiculos;
…}

class LocadoraDeVeiculos{ ...
@ManyToMany
private List<Carro> carros;
…}

Existe várias propriedades que podem ser adicionadas em todos esses relacionamentos, falando do manyToMany uma anotação importante é o mappedBy que indica quem é o dono do relacionamento muito para muitos.

2. **Fazer a implementação de um relacionamento One To Many considerando que um *Departamento* pode conter vários *Funcionários* (criar os atributos que julgar necessário).**

DEPARTAMENTO

```
@Entity
public class Departamento implements Serializable {

   private static final long serialVersionUID = 1L;
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   private String nome;

   @OneToMany
   private Set<Funcionario> funcionarios;

    ...geters e seters
```

FUNCIONARIO

```
@Entity
public class Funcionario implements Serializable {

   private static final long serialVersionUID = 1L;
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;

   private String nome;
   private String sobrenome;
   @Temporal(TemporalType.DATE)
   private Date nascimento;
   @Enumerated(EnumType.STRING)
   private Sexo sexo;

      ...geters e seters
```

ENUM SEXO

```
enum Sexo { M, F }
```

3. **Fazer a implementação de um relacionamento Many to One considerando que muitos *Pedidos* podem pertencer a um *Cliente* (novamente, criar os atributos que julgar necessário).**

CLIENTE

```
@Entity
public class Cliente implements Serializable {

   private static final long serialVersionUID = 1L;
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;

   private String nome;
   private String sobrenome;
   @Temporal(TemporalType.DATE)
   private Date nascimento;
   @Enumerated(EnumType.STRING)
   private Sexo sexo;
   @OneToMany(mappedBy = "cliente")
   private Set<Pedido> pedidos;
}

   ...geters e seters
```

PEDIDO

```
@Entity
public class Pedido implements Serializable {

   private static final long serialVersionUID = 1L;
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;

   @ManyToOne
   private Cliente cliente;
}
    ...geters e seters
```

4. **Fazer a implementação de um relacionamento Many to Many considerando que muitos *Autores* podem escrever muitos *Livros* (novamente, criar os atributos que julgar necessário).**

AUTOR

```
@Entity
public class Autor implements Serializable {

   private static final long serialVersionUID = 1L;
   @Id
```

```
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String nome;
    private String sobrenome;

    @Temporal(TemporalType.DATE)
    private Date nascimento;

    @Enumerated(EnumType.STRING)
    private Sexo sexo;

   @ManyToMany
       @JoinTable(name="autorLivro", joinColumns=
      {@JoinColumn(name="autor_id")}, inverseJoinColumns=
        {@JoinColumn(name="livro_id")})
   private List<Livro> livros;

}

    ...geters e seters
```

LIVRO

```
@Entity
public class Livro implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String isbn;

    private Integer quantidadePaginas;

    @ManyToMany
    private List<Autor> autores;

}

    ...geters e seters
```

5. Pesquisar e descrever resumidamente como utilizar relacionamentos bidirecionais.

Um relacionamento bidirecional se diferencia do unidirecional por manter referências em ambas as entidades, ou seja ambos os lados podem enxergar a classe relacionada. Na prática precisamos ligar as tabelas com @JoinColum e e mappedBy para que a referência se estabeleça.