Auto Generation of ID's - Sequences
(for databases that supports this, like Oracle and Derby)

cphbusiness

We can control how a Sequence is generated or map it to an existing sequence.

```
Class Book{
...
  @ID
  @GeneratedValue(strategy = GenerationType.SEQUENCE,generator="s1")
  @SequenceGenerator(name="s1",sequenceName = "My_SEQ",
                          initialValue = 200000,allocationSize = 1)
```

These annotations will:

- **Create a sequence as sketched below, if we are creating tables from Entities**
- **Map to the existing sequence if we are creating Entities from tables**

Table Create Script

```
DROP SEQUENCE My_SEQ RESTRICT;
CREATE SEQUENCE My_SEQ START WITH 200000 INCREMENT BY 1;
```

Auto Generation of ID's - Tables

All databases can use a separate Table as their strategy to provide a "next id" value

This is usually the default when you select: **`GenerationType.AUTO`**

```
Class Book{
...
  @Id
  @GeneratedValue(strategy = GenerationType.TABLE,generator="s1")
  @TableGenerator(name="s1",table = "My_SEQ",
  initialValue = 200000,allocationSize = 50)
```

These annotations will:

- **Create a table for auto id's if we are creating tables from Entities**
- **Map to the existing table if we are creating Entities from tables**

Auto Generation of ID's - IDENTITY

MySQL does not provide **Sequences** to generate a unique value for new Rows.
MySQL provides a construct AUTO_INCREMENT as sketched below:

```
CREATE TABLE Persons
(
  ID int NOT NULL AUTO_INCREMENT,
  Name varchar(80),
  PRIMARY KEY (ID)
)
```

This is how you Signal JPA to use this strategy for automatic key generation:

```
@GeneratedValue(strategy = GenerationType.IDENTITY)
 private Integer id;
```

There is no way, as for the other two strategies, to provide a start value and allocation size via JPA.
Se exercises for an example script you can use to insert data without conflicting with JPA.

Composite Primary Keys

cphbusiness

Composite primary keys can be defined in two ways:

Using an **Id Class**

```
@Entity @IdClass(ProjectId.class)
public class Project {
    @Id int departmentId;
    @Id long projectId;
     :
}

Class ProjectId {
    int departmentId;
    long projectId;
}
```

Using an **Embeddable Class**

```
@Entity
public class Project {
  @EmbeddedId ProjectId id;
     :
}
@Embeddable
Class ProjectId {
    int departmentId;
    long projectId;
}
```

The main purpose of both the `IdClass` and the `Embeddable` Class is to be used as the structure passed to the `EntityManager find()` and `getReference()` AP

Date Time and Transient Properties

```
@Temporal(TemporalType.DATE)
private Date dateOfBirth;

@Temporal(TemporalType.TIMESTAMP)
private Date creationDate;

@Transient
private int age;
```

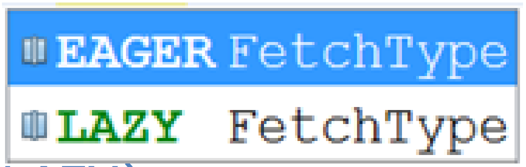
Enums

```
public enum CustomerType {
  GOLD,
  SILVER,
  IRON,
  RUSTY
}
```

```
      public class Customer {
        ...
        @Enumerated(EnumType.STRING)
        private CustomerType customerType;

      }
```

Collections and Maps of Basic Types


EAGER FetchType
LAZY FetchType

```
@ElementCollection(fetch = FetchType.LAZY)
private List<String> hobbies= new ArrayList();


@ElementCollection(fetch = FetchType.LAZY)
@MapKeyColumn(name = "PHONE")
@Column(name = "Description")  //Name of the Value column
private Map<String, String> phones = new HashMap();
```