# CLASS METHODS IN JAVA

Class methods are methods that are called on the class itself, not on a specific object instance. The static modifier ensures implementation is the same across all class instances. Many standard built-in classes in Java (for example, Math) come with static methods (for example, Math.abs(int value)) that are used in many Java programs. The class methods are bound to the class, not to the instance. It can modify the class state means it can change class configuration globally. It can access only the class variable. The class methods are used to create the factory methods. They accept **cls** as a parameter that points to the class. The changes made by the class method reflect all instances of the class. The @classemethod decorator or classmethod() defines the class methods. Here is its Syntax:

```java
public class className {
    modifier static dataType methodName (inputParameters) { //static method
         //block of code to be executed
    }
}

//calling the method, from anywhere
className.methodName(passedParams);
```

## Object Relational Mapping

*Object Relational Mapping* is a technique used to create a mapping between a relational database and objects of software - in our case, Java objects. The idea behind this is to stop working with cursors or arrays of data obtained from the database, and rather directly obtain objects representing our business domain. To achieve that, we use techniques to map our domain objects to the database tables to automatically fill them with the data from the tables. Then, we can perform standard object manipulation on them.

## Types Of Mapping

There are a few types of relationships:

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

## One-to-Many/Many-to-One

As its name implies, One-To-One mapping represents a single-valued association where an instance of one entity is associated with an instance of another entity. In this type of association one instance of source, the entity can be mapped at most one instance of the target entity. In our example, this would be a Teacher and their Courses. A teacher can give multiple courses, but a course is given by only one teacher (that's the *Many-to-One* perspective - many courses to one teacher).
Let's create our entities:

```
@Entity
public class Teacher {
    private String firstName;
    private String lastName;
}

@Entity
public class Course {
    private String title;
}
```

Now, the fields of the Teacher class should include a list of courses. Since we'd like to map this relationship in a database, which can't include a list of entities within another entity - we'll annotate it with a @OneToMany annotation:

```
@OneToMany
private List<Course> courses;
```

We've used a List as the field type here, but we could've gone for a Set or a Map (though this one requires a bit more configuration). How does JPA reflect this relationship in the database? Generally, for this type of relationship, we must use a foreign key in a table. JPA does this for us, given our input on how it should handle the relationship. This is done via the @JoinColumn annotation:

```
@OneToMany
@JoinColumn(name = "TEACHER_ID", referencedColumnName = "ID")
private List<Course> courses;
```

Using this annotation will tell JPA that the COURSE table must have a foreign key column TEACHER_ID that references the TEACHER table's ID column. Let's add some data to those tables:

```
insert into TEACHER(ID, LASTNAME, FIRSTNAME) values(1, 'Doe', 'Jane');

insert into COURSE(ID, TEACHER_ID, TITLE) values(1, 1, 'Java 101');
insert into COURSE(ID, TEACHER_ID, TITLE) values(2, 1, 'SQL 101');
insert into COURSE(ID, TEACHER_ID, TITLE) values(3, 1, 'JPA 101');
```

And now let's check if the relationship works as expected:

```
Teacher foundTeacher = entityManager.find(Teacher.class, 1L);

assertThat(foundTeacher.id()).isEqualTo(1L);
assertThat(foundTeacher.lastName()).isEqualTo("Doe");
assertThat(foundTeacher.firstName()).isEqualTo("Jane");
assertThat(foundTeacher.courses())
        .extracting(Course::title)
        .containsExactly("Java 101", "SQL 101", "JPA 101");
```

We can see that the teacher's courses are gathered automatically when we retrieve the Teacher instance.

## One-to-One

The One-To-One mapping represents a single-valued association where an instance of one entity is associated with an instance of another entity. In this type of association one instance of source, the entity can be mapped at most one instance of the target entity.This time, instead of having a relationship between one entity on one side and a bunch of entities on the other, we'll have a maximum of one entity on each side. This is, for example, the relationship between a `Course` and its `course material`. Let's first map `CourseMaterial`, which we haven't done yet:

```
@Entity
public class CourseMaterial {
    @Id
    private Long id;
    private String url;
}
```

The annotation for mapping a single entity to another entity is, unshockingly, `@OneToOne`. Before setting it up in our model, let's remember that a relationship has an owning side - preferably the side which will hold the foreign key in the database. In our example, that would be `CourseMaterial` as it makes sense that it references a `Course` (though we could go the other way around):

```
@OneToOne(optional = false)
@JoinColumn(name = "COURSE_ID", referencedColumnName = "ID")
private Course course;
```

There is no point in having material without a course to encompass it. That's why the relationship is not `optional` in that direction. Speaking of direction, let's make the relationship bidirectional, so we can access the material of a course if it has one. In the `Course` class, let's add:

```
@OneToOne(mappedBy = "course")
private CourseMaterial material;
```

Here, we're telling Hibernate that the material within a Course is already mapped by the course field of the CourseMaterial entity.
Also, there's no optional attribute here as it's true by default, and we could imagine a course without material (from a very lazy teacher).

## Many-to-Many

The Many-To-Many mapping represents a collection-valued association where any number of entities can be associated with a collection of other entities. In a relational database, any number of rows of one entity can be referred to as any number of rows of another entity. Effectively, in a database, a *Many-to-Many* relationship involves a middle table referencing both other tables. So, for our example, the *Many-to-Many* relationship will be the one between Student and Course instances as a student can attend multiple courses, and a course can be followed by multiple students. In order to map a *Many-to-Many* relationship we'll use the @ManyToMany annotation. However, this time around, we'll also be using a @JoinTable annotation to set up the table that represents the relationship:

```
@ManyToMany
@JoinTable(
  name = "STUDENTS_COURSES",
  joinColumns = @JoinColumn(name = "COURSE_ID", referencedColumnName = "ID"),
  inverseJoinColumns = @JoinColumn(name = "STUDENT_ID", referencedColumnName = "ID")
)
private List<Student> students;
```

Now, go over what's going on here. The annotation takes a few parameters. First of all, we must give the table a name. We've chosen it to be STUDENTS_COURSES. After that, we'll need to tell Hibernate which columns to join in order to populate STUDENTS_COURSES. The first parameter, joinColumns defines how to configure the join column (foreign key) of the owning side of the relationship in the table. In this case, the owning side is a Course. Let's set up a data set with students and courses:

```java
Student johnDoe = new Student();
johnDoe.setFirstName("John");
johnDoe.setLastName("Doe");
johnDoe.setBirthDateAsLocalDate(LocalDate.of(2000, FEBRUARY, 18));
johnDoe.setGender(MALE);
johnDoe.setWantsNewsletter(true);
johnDoe.setAddress(new Address("Baker Street", "221B", "London"));
entityManager.persist(johnDoe);

Student willDoe = new Student();
willDoe.setFirstName("Will");
willDoe.setLastName("Doe");
willDoe.setBirthDateAsLocalDate(LocalDate.of(2001, APRIL, 4));
willDoe.setGender(MALE);
willDoe.setWantsNewsletter(false);
willDoe.setAddress(new Address("Washington Avenue", "23", "Oxford"));
entityManager.persist(willDoe);

Teacher teacher = new Teacher();
teacher.setFirstName("Jane");
teacher.setLastName("Doe");
entityManager.persist(teacher);

Course javaCourse = new Course("Java 101");
javaCourse.setTeacher(teacher);
entityManager.persist(javaCourse);

Course sqlCourse = new Course("SQL 101");
sqlCourse.setTeacher(teacher);
entityManager.persist(sqlCourse);
```

Of course, this won't work out of the box. We'll have to add a method that allows us to add students to a course. Let's modify the `Course` class a bit:

```java
public class Course {

    private List<Student> students = new ArrayList<>();

    public void addStudent(Student student) {
        this.students.add(student);
    }
}
```

Now, we can complete our dataset:

```java
Course javaCourse = new Course("Java 101");
javaCourse.setTeacher(teacher);
javaCourse.addStudent(johnDoe);
javaCourse.addStudent(willDoe);
entityManager.persist(javaCourse);

Course sqlCourse = new Course("SQL 101");
sqlCourse.setTeacher(teacher);
sqlCourse.addStudent(johnDoe);
entityManager.persist(sqlCourse);
```

Once this code has run, it'll persist in our `Course`, `Teacher,` and `Student` instances as well as their relationships. For example, let's retrieve a student from a persistent course and check if everything's fine:

```java
Course courseWithMultipleStudents = entityManager.find(Course.class, 1L);

assertThat(courseWithMultipleStudents).isNotNull();
assertThat(courseWithMultipleStudents.students())
  .hasSize(2)
  .extracting(Student::firstName)
  .containsExactly("John", "Will");
```