ORM Usage Guide

1. Setting up

First, set up the exception logger with appropriate location. Call the logger method ExceptionLogger.getExceptionLogger(<filePath>) to set the appropriate file path for the file logger. For example:

```
ExceptionLogger.getExceptionLogger("src/main/resources/ORMExceptionLogs/");
```

Next, ensure you have a jdbc.properties file that holds information about accessing the database somewhere in your project. This will automatically be accessed by your repositories.

2. Annotate your classes

First, annotate your class with the @Entity annotation, setting the “tableName” to the exact name of the matching table in your database.

```
@Entity(tableName = "People")
```

Second, each field you want to be read from/written to the table should be annotated with @Property, setting the “fieldName" to the name of the field found in the table.

```
@Property(fieldName = "first_name")
private String firstName;
```

Note: All fields should be either wrapper classes or strings. Primitives cannot be reflected upon.

Any primary key fields should also be annotated with @PrimaryKey, setting “autoIncrement” to either true or false, depending on whether the related field in the table is set to auto-increment.

```
@PrimaryKey(autoIncrement = true)
@Property(fieldName = "id")
private Integer id;
```

The final required annotation is for the required “fake constructor” method (because I didn’t realize it was just called a “factory” at the time). All this method needs to do is:

- Take in no parameters
- Returns a new instance of the class type of the object calling it

```
@FakeConstructor
public Person fakeConstructor() {
    return new Person();
}
```

If it doesn't return a newly allocated instance of the class, functions that return variable size data structures of the class will probably not work.

The two other annotations, @Getter and @Setter, are optional, and are for marking getter and setter methods for fields. If used, set the "fieldName" to the same that was set in the related field with the @Property annotation.

```
@Getter(fieldName = "first_name")
public String getFirstName() { return firstName; }

@Setter(fieldName = "first_name")
public void setFirstName(String firstName) {
    this.firstName = firstName;
}
```

These are optional annotations if the getters and setters have conventional names:

- getField() – for getters for most field types
- isField() – for getters for Boolean fields
- setField() – for all setters

The table initializer method will look for methods following this naming convention if these annotations are not used. Regardless, fields that need to interact with the table must have:

-Public getter that takes in no parameters and returns the same type as the field they're tied to

-Public setter that takes in only one parameter of the same type as the field they're tied to

3. Making a repository

The repository constructor requires an instance of the class they are using. The most basic constructor:

```
Repository<Person> repository = new Repository<>(new Person());
```

Replacing "Person" with the class you want to be stored/read from a database table.

You can test if the constructor was successful by printing the repository object to console (automatically calling the toString() method)

```
Table "TestTableB"
    Column: id, Class field: id, Getter: getId, Setter: setId, is primary key: true
    Column: first_name, Class field: firstName, Getter: getFirstName, Setter: setFirstName, is primary key: false
    Column: last_name, Class field: lastName, Getter: getLastName, Setter: setLastName, is primary key: false
    Column: weight, Class field: weight, Getter: getWeight, Setter: setWeight, is primary key: false
    Column: age, Class field: age, Getter: getAge, Setter: setAge, is primary key: false
Fake constructor: public TestClassB TestClassB.fakeConstructor()
```

4. Using the repository methods

a. Create

The create(o) method takes the given generic object and attempts to insert it into the linked table. If the primary key is set to auto-increment, its value in the object will not be stored in the database, and instead the generated key will be placed into the object's primary key field.

```
System.out.println(repository.create(
        new Person( id: 0,  firstName: "Steve",  lastName: "Stevenson",  weight: 185.5, (byte) 31)));
```

Prints out:

```
Person{id=8, firstName='Steve', lastName='Stevenson', weight=185.5, age=31}
```

With the generated id 8.

b. Read

Two read methods exist:
- Read(O o, Object obj) – parameters are one generic and one object
- Read(O o) – only one parameter of type generic

The first method takes in a repository generic used to hold data read from the table and an object that is the primary key value that is queried. This must match the generic's primary key type. The method inserts the primary key value into the generic passed in and calls the second method, which returns a generic if successfully read into.

```
System.out.println(repository.read(new Person(),  obj: 8));
```

Prints out:

```
Person{id=8, firstName='Steve', lastName='Stevenson', weight=185.5, age=31}
```

c. Update

The update method takes a given generic and attempts to change information in the database based on the primary key field of the object.

```
System.out.println(repository.update(
        new Person( id: 8,  firstName: "James",  lastName: "Jones",  weight: 155.5, (byte) 32)));
```

Will update the row corresponding to the primary key of the field (id)

8	James	Jones	155.5	32

d. Delete

The delete function will delete the row from the table that corresponds to the given generic's primary key.

```
Person deleteMe = repository.create(new Person( id: 4,  firstName: "Delete",  lastName: "Me",  weight: 0.0, (byte) 0));
```

Will create the row:

11	Delete	Me	0	0

Since deleteMe contains the generated primary key, it can then be used to delete the row that was just created:

```
System.out.println(repository.delete(deleteMe));
```

e. ReadAll

The readAll function takes in no parameters and returns an array list of generics. It reads the table connected to the generic and attempts to create new generics (using the fake constructor), fill them with data from the query, and add them to the array list to be returned.