Exercises module 23 – Type conversion

23.1 – Casting

1. Given this interface and classes:

```
interface Car {
    public void start();
    public void stop();
}

class Volvo implements Car {
    @Override
    public void start() {    }

    @Override
    public void stop() {    }
}

class Tesla implements Car {

    @Override
    public void start() {    }

    @Override
    public void stop() {    }

    public void ChargeBattery() {
        System.out.println("Charging battery");
    }
}
```

2. Given this code in the main class:

```
public static void main(String[] args) {
// write your code here

    Volvo volvo = new Volvo();
    Tesla tesla = new Tesla();

    Method1(volvo);
    Method2(tesla);
    Method3(tesla);

    Car car1 = new Volvo();
    Car car2 = new Tesla();
    Car car3 = volvo;
    Car car4 = tesla;
}
```

3. And finally given these methods in the Main class:

```
private static void method1(Volvo car) {
    car.start();
    car.stop();
}

private static void method2(Tesla car) {
    car.start();
    car.stop();
}

private static void method3(Car car) {
    car.start();
    car.stop();
}
```

4. Review the code and it **<u>is very crucial you understand</u>** how it works and what happens here.
5. As you notice the Tesla class contain one extra method **chargeBattery()**.

   In what methods is this method available? Understand why.
6. Can you add this line at the end of the main method?

```
Tesla car5 = car4;
```

Why not? How can you make it compile?

Make the code compile.

7. Add this code at the end:

```
Tesla tesla1 = new Tesla();
Car tesla2 = tesla1;
Tesla tesla3 = tesla2;
```

Understand these questions:

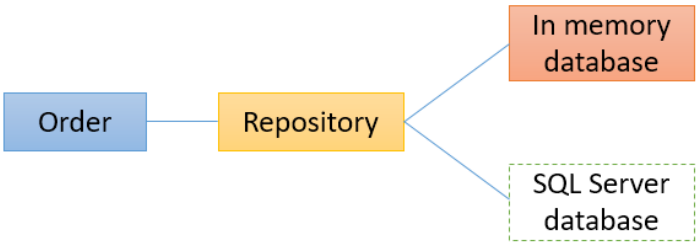
- Is tesla3 the same instance/object as tesla1?
- Do we lose any information in the steps above?
- On what tesla (1-3) can we call the chargeBattery method?

23.2 – Repository

1. The **repository pattern** is a common pattern when **loading/saving** objects **to/from** the database. Start by reading about the repository pattern:
   a. http://blog.sapiensworks.com/post/2014/06/02/The-Repository-Pattern-For-Dummies.aspx
   b. https://thinkinginobjects.com/2012/08/26/dont-use-dao-use-repository/
   c. http://stackoverflow.com/questions/11985736

2. Create a new project and copy the **Order** and **OrderLine** classes from exercises 19.2, also do add the **YodaMoney** library.

3. We want to create an **interface** that gives us an abstraction for persisting and retrieving an Order objects to/from **storage**, like:



The **Repository** interface also allows us to replace the underlying storage implementation without breaking the system. It's quite convenient and common to have an in-memory database to start with when developing a system because we don't need to depend on the availability of an actual database server.

4. Create a new Repository interface:

```
public interface Repository {

    Order getOrder(int orderID, int customerID);

    List<Order> getAllOrderForCustomer(int customerID);

    void deleteOrder(int orderID, int customerID);

    int createNewOrder(Order order);
}
```

5. Modify the Order class and add these two **public fields**:

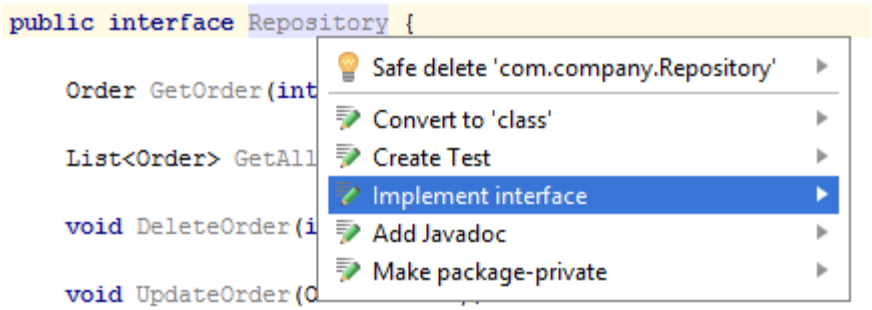
```
public int orderID;
public int customerID;
```

6. Update the **constructor** so that the customerName and customerID can be set via the constructor. The constructor signature should be:

```
public Order(String customerName, int customerID)
```

(We don't set the orderID via the constructor)

7. Place the cursor on the Interface name and press alt+**enter** to implement the interface. Name the class **InMemoryRepository**:



Remove any **@Override** in the generated class.

8. To get a better sense of what we are trying to achieve, this is the code in the **main** method that we want to run:

```
Repository repo = new InMemoryRepository();
int customerID = 42;

Order order1 = new Order("Edument AB", customerID);
order1.addOrderLine(new OrderLine("Widget A", 10,
                                  Money.of(CurrencyUnit.EUR, 3.14)));
order1.addOrderLine(new OrderLine("Widget B", 15,
                                  Money.of(CurrencyUnit.EUR, 9.95)));

Order order2 = new Order("Edument AB", customerID);
order2.addOrderLine(new OrderLine("Widget C", 2,
                                  Money.of(CurrencyUnit.EUR, 5.95)));
order2.addOrderLine(new OrderLine("Widget D", 1,
                                 Money.of(CurrencyUnit.EUR, 250)));

int order1ID = repo.createNewOrder(order1); //should be #1
int order2ID = repo.createNewOrder(order2); //should be #2
System.out.println("Order1 ID=" + order1ID + ", Order2 ID=" + order2ID);

//Get all orders, should return #2
System.out.println("Number of items: " +
repo.getAllOrderForCustomer(customerID).size());

repo.deleteOrder(order2ID, customerID);

//Get all orders, should return #1
System.out.println("Number of items: " +
repo.getAllOrderForCustomer(customerID).size());
```

9. The **InMemoryRepository** class will act like a simple in memory database. Let's implement it:
    a. Add a private **List<Order> orders** field and a private integer **orderID** field (set it to Zero)

    b. Implement the **createNewOrder** method. It should
        i. When adding a new order object to a database, the database will usually set and return a new OrderID (primary key) for us. To simulate this, we use the **orderID** field counter to track the ID when creating new items to the repository.

        Read more about Primary keys on the web, like:
        http://stackoverflow.com/questions/9551195
        https://en.wikipedia.org/wiki/Unique_key

        So we set the **orderId** in the order object like this:

```
order.orderID = ++orderId;
```

        ii. Then add the order object to the list
        iii. Return the new orderID

    c. Implement the **getAllOrderForCustomer** method, it should:
        i. Return the orders list to the caller, but only the orders that matches the provided customerID

    d. Implement the **deleteOrder** method, it should:
        i. Locate the first order in the private orders list that **matches both** the orderID and CustomerID.
        ii. Remove the found item and if not found do nothing
        iii. Why do we need to pass in both the **orderID** and **customerID**, why is not the **orderID** enough to pass in? In theory just passing in orderID would be enough, but what can the reasons be for providing the customerID as well?

    e. Implement the **getOrder**, it should:
        i. Locate the first order in the private orders list that **matches both** the orderID and customerID and return it. If no match is found, return null.

10. **Creating this kind of code, repositories and similar is a very common and important thing for us as developer to do and master**.

    Important keywords to investigate further:

    - Repository
    - Interface
    - CRUD (Create Read Update Delete)

- Primary key
- Entity objects
- Unit of work
- Cloning objects
  http://stackoverflow.com/questions/869033/how-do-i-copy-an-object-in-java