The purpose of this assignment is to practice with advanced functionality of java generic types and methods.

Exercise 1 – Gas station

Open the source code S2Es1.java in your IDE. The program simulates a gas station with various types of cars. Each car has a tank that contains *Petrol*, *Diesel* or *NaturalGas* as fuel. As it has been implemented so far, different fuel types can be mixed, causing the refill method to throw an exception in case of wrong fuel type. A Runtime exception is also thrown when *HybridCars* are created *with NaturalGas*, what should not be possible.
The goal of this exercise is to remove the possibility to mix different fuel types and to erroneously instantiate *HybridCars* that provide *NaturalGas* as alternative fuel, by introducing appropriate support of generics.

1. Remove the possibility to mix fuel types
    a. Modify the *Car* and *HybridCar* classes by adding a generic type parameter to specify the *Fuel* type. Make sure that the generic type can only be a subtype of Fuel.
    b. Change the *tank* variable type, declared in the *Car* class, from *Fuel* to the generic type. Update the constructors (*Car* and *HybridCar*) and the *Car.tank* method to accept the generic type instead of the *Fuel* class.
    c. Update the object construction expressions by specifying the correct generic class.
    d. Update the refill / recharge methods to accept the correct types.
2. Remove the possibility to create *HybridCar* instances with *NaturalGas* fuel
    a. Add an empty *IHybridFuelAlternative* interface.
    b. Let *Petrol* and *Diesel* classes implement that interface.
    c. Update the *HybridCar* generic type parameter, by introducing the constraint that the generic type also implements the *IHybridFuelAlternative* interface.

Exercise 2 – Food market

Starting from the FoodMarket.java file, write a program to simulate a food market that sells *Bread*, *Cakes* and *IceCreams* as provided by the class hierarchy. Each type of food is created by a *FoodManufacturer* which generates exactly one type of product using the *produce* factory method. After instantiating a food in the *produce* method, each *FoodManufacturer* must call the specific method (*freeze* / *bake*) before returning the instance to the calling method. Finally, a *FoodManufacturer* must also provide the *addToStock* method which creates a new product (using the *produce* factory method) and adds it to the collection of foods passed as parameter.
Implement the *IceCreamMaker*, *Baker* and *PastryChef* classes as *FoodManufacturer* classes to produce respectively *IceCreams*, *Bread* and *Cakes*.
Add one specific sales counter for each product (e.g., using a List) and simulate the production of 5 products by each *FoodManufacturer* by invoking the *addToStock* method and passing the related sales counter to the method. Finally introduce a charity box (e.g., using a List) that collects one product from each manufacturer using the *addToStock* method.

Important:
Use wildcards to solve this exercise! Make sure that the *addToStock* method prevents adding wrong food to wrong counters. For instance, a *Baker* should not be allowed to add its products to the *IceCream* counter. The *Baker* should only be allowed to add its products to the charity box and to Baker's sale counters.