**Exercise: Design Patterns**

Problems for exercise and homework for the ["C# OOP" course @ SoftUni"](https://softuni.bg/trainings/4377/csharp-oop-february-2024).

Problem solutions are provided if you struggle doing it on your own as it can be too abstract. The solutions are just examples, you don’t need to follow them completely. There is no automated testing for this exercise, you have to validate what you’ve done by yourself.

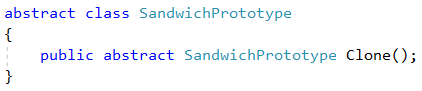
## I. Prototype

Your task is to create a console application for building sandwiches implementing the Prototype Design Pattern.

### Abstract Class

First, you have to create an abstract class to represent a sandwich and define a method by which the abstract Sandwich class can clone itself.

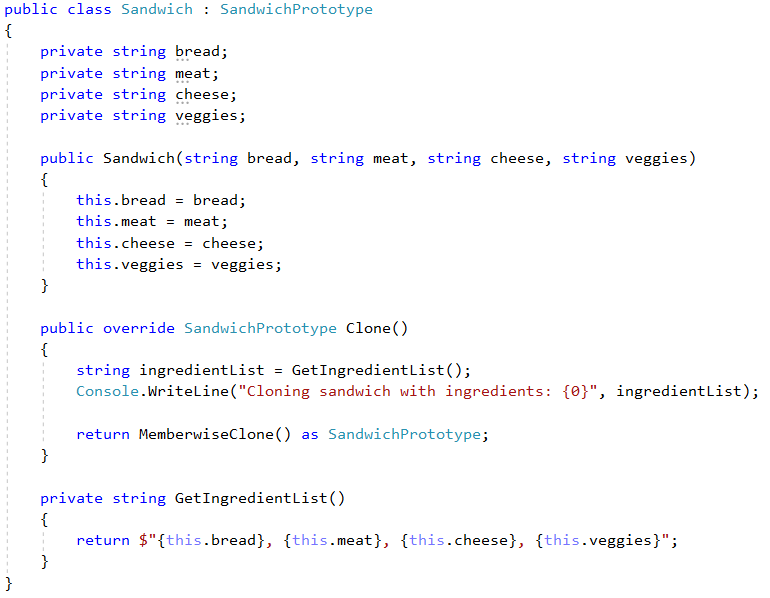
#### Solution



**ConcretePrototype Participants**

Now you need the **ConcretePrototype** participant class that can clone itself to create more Sandwich instances. Let’s say that a Sandwich consists of four parts: the meat, cheese, bread, and veggies.

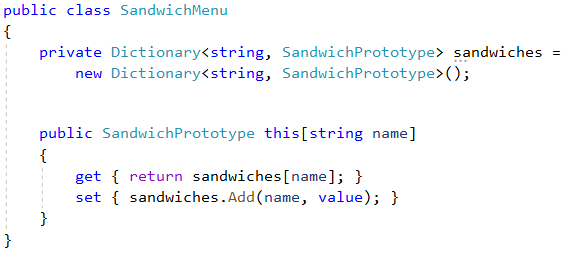
#### Solution



### Sandwich Menu

Let’s create a class that will have the purpose to store the sandwiches we’ve made. It will be like a repository.

#### Solution

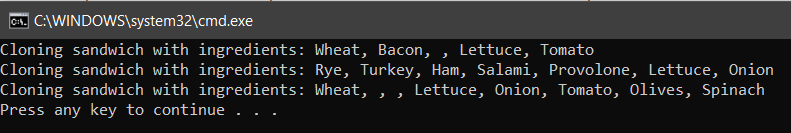


**Use What You’ve Done**

Now is the time to test what you have done by trying to use it. In your **Main()** method you can do just that by instantiating the prototype and then cloning it, thereby populating your **SandwichMenu**.

#### Solution





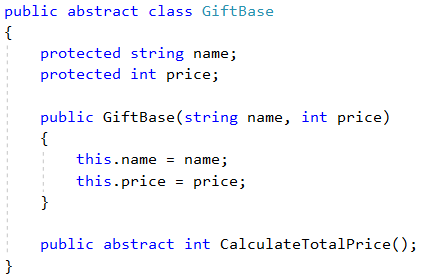
## II. Composite

Your task is to create a console application that calculates the total price of gifts that are being sold in a shop. The gift could be a single element (toy) or it can be a complex gift that consists of a box with two toys and another box with maybe one toy and the box with a single toy inside. We have a tree structure representing our complex gift so, implementing the Composite design pattern will be the right solution for us.

### Component

First, you have to create an abstract class to represent the base gift. It should have two fields (name and price) and a method that calculates the total price. These fields and methods are going to be used as an interface between the Leaf and the Composite part of our pattern.

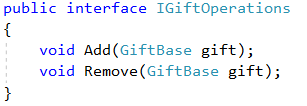
#### Solution



**Basic Operations**

Create an interface **IGiftOperations** that will contain two operations - Add and Remove (a gift). You should create the interface because the Leaf class doesn’t need the operation methods.

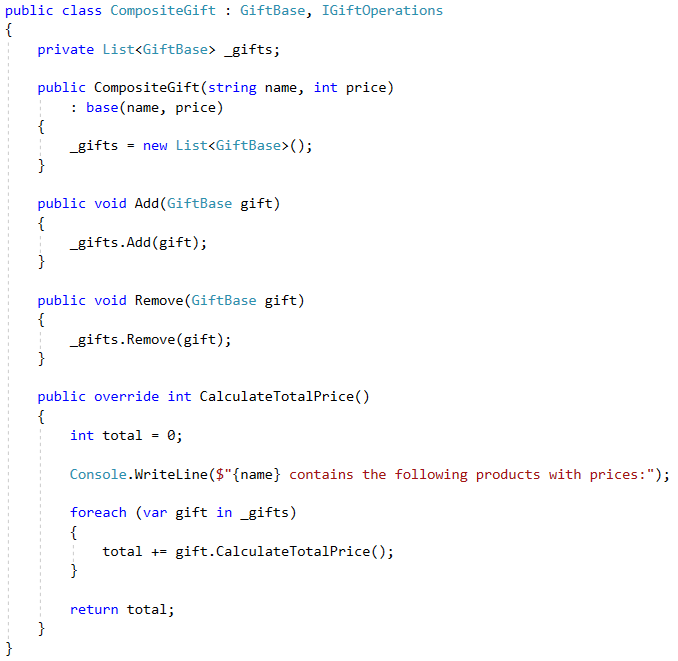
#### Solution



### Composite Class

Now you have to create the composite class (**CompositeGift**). It should inherit the **GiftBase** class and implement the **IGiftOperations** interface. Therefore, the implementation is pretty forward. It will consist of many objects from the **GiftBase** class. The **Add** method will add a gift and the **Remove** - will remove one. The **CalculateTotalPrice** method will return the price of the **CompositeGift**.

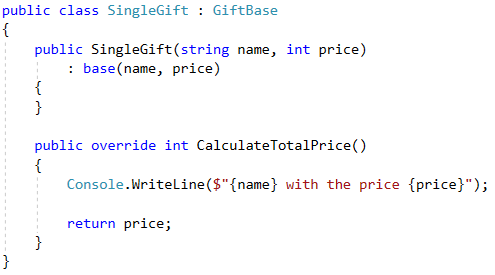
#### Solution



### Leaf Class

You should also create a Leaf class (**SingleGift**). It will not have sub-levels so it doesn’t require to add and delete operations. Therefore, it should only inherit the **GiftBase** class. It will be like a single gift, without component gifts.

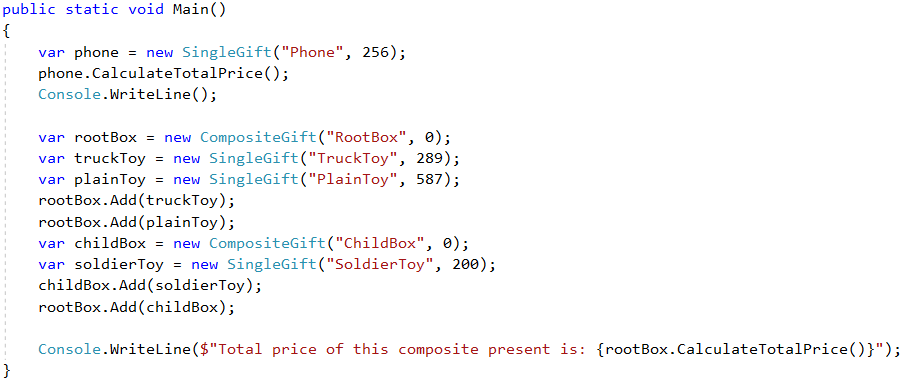
#### Solution

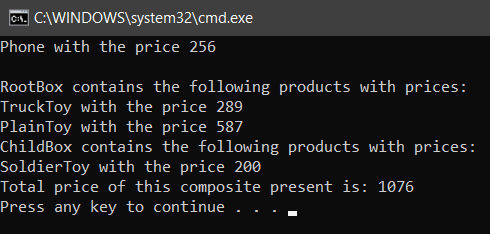


**Use What You’ve Done**

Now is the time to test what you have done by trying to use it. In your **Main()** method you can do just that by instantiating the Leaf class (**SingleGift**) and the Composite class (**CompositeGift**) and using their methods.

#### Solution





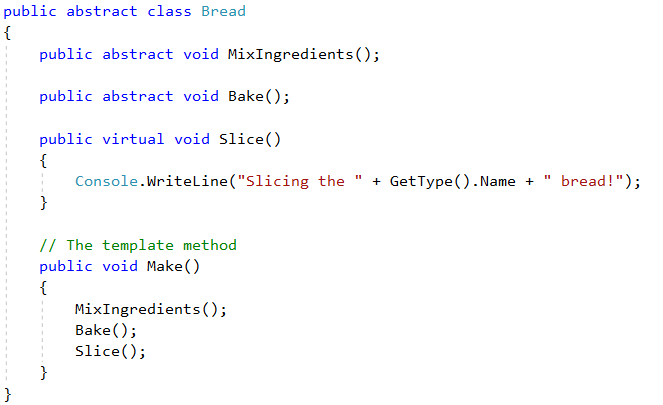
## III. Template Pattern

There are easily [hundreds of types of bread](https://en.wikipedia.org/wiki/List_of_breads) currently being made in the world, but each kind involves specific steps to make them. Your task is to model a few different kinds of bread that all use this same pattern, which is a good fit for the Template Design Pattern.

### Abstract Class

First, you have to create an abstract class (**Bread**) to represent all loaves of bread we can bake. It should have two abstract void methods **MixIngredients()**, **Bake()**, one virtual void method **Slice(),** and the template method - **Make()**.

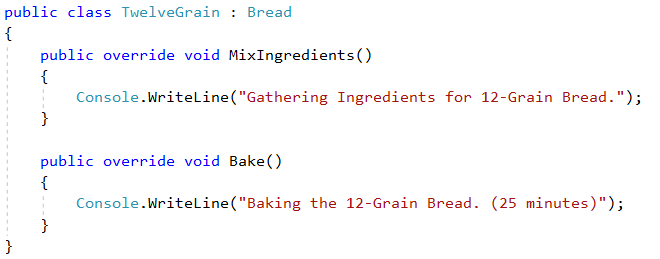
#### Solution

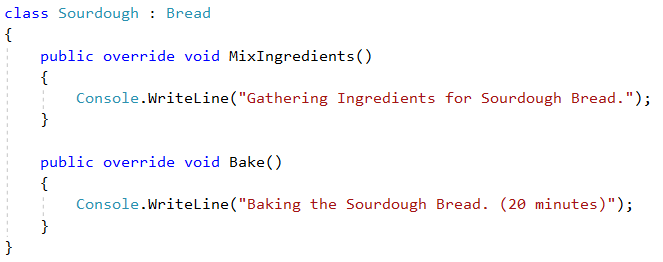


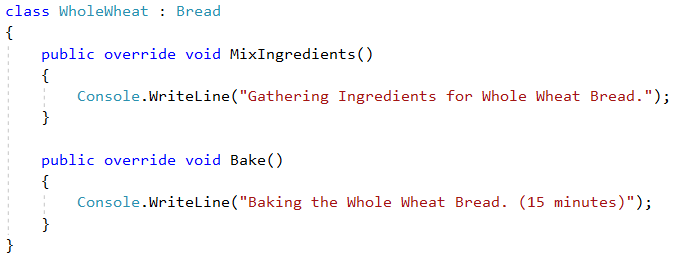
### Concrete Classes

Extend the application by adding several Concrete Classes for different types of **Bread**. Examples: **TwelveGrain**, **Sourdough**, **WholeWheat**.

#### Solution







**Use What You’ve Done**

Now is the time to test what you have done by trying to use it. In your **Main()** method you can do just that by instantiating objects of the classes you’ve just made. It was that simple. This, this might be something you’ve been already using but you didn’t know it was a Design Pattern.

#### Solution

