# C# OOP Basics Exam Preparation II – Avatar

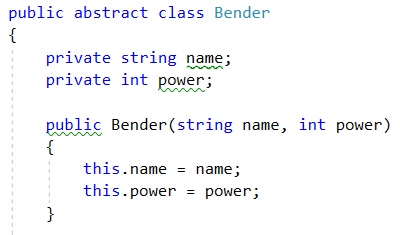
This is a step by step tutorial of the Exam Preparation II – Avatar from the C# OOP Basics course. You can submit your solution here - [Judge](https://judge.softuni.bg/Contests/698/CSharp-OOP-Basics-Exam-Sample-12-July-2017).

# Stage I: Create Class Hierarchy

## Benders

One of the main entities in the Avatar application are the Benders. Though if you have already read the whole problem description, Benders are never used standalone. Rather the specific types of benders are used in the action. Also there are some common attributes for all types of benders. This means that our **Bender** class should be **abstract** and all the other types of benders will **inherit** this main abstract class.

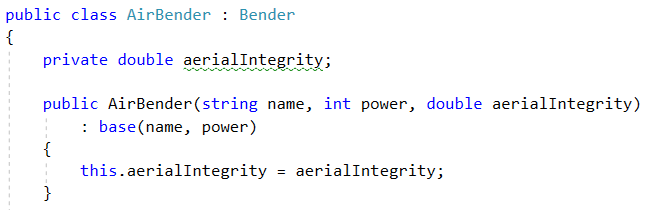
Let’s create the abstract Bender class. As stated all benders have two common characteristics – **name(string)** and **power(int)**. We can implement these characteristics through fields because we are not sure at the moment if we will need access from outside the class. Except the two fields we will also need a constructor which will set them upon creating an instance.



### Air Bender

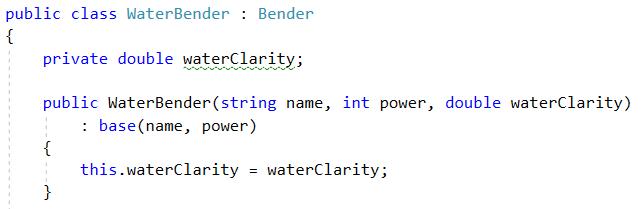
Air Bender is the first **specific** type of bender. As we mentioned it will **inherit** the abstract Bender class because it should have a name and a given power. In addition, it has one more characteristic which is typical only for this type of benders – aerialIntegrity(floating-point number). We will implement it through a field again for the same reasons.

Always keep in mind that every child class is obliged to implement the constructor of the class it inherits (the parent). The values which initialize the parent’s fields are passed to the **base** constructor. The other values which are related to the current child class’ fields are initialized in its own constructor – e.g. aerialIntegrity.



### Water Bender

Similarly, to the Air Bender, the Water Bender class will inherit from the abstract Bender class for the very same reasons. The unique field this time is called - WaterClarity(floating-point number).



### Fire Bender

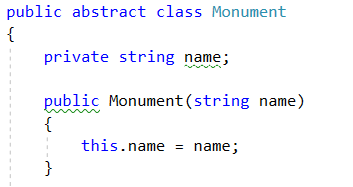
The same goes for the Fire Bender class. Its own field is called - heatAggression(floating-point number). This time the source code won’t be available but it is absolutely analogously to the previous two.

### Earth Bender

For this class the specific field is - groundSaturation(floating-point number).

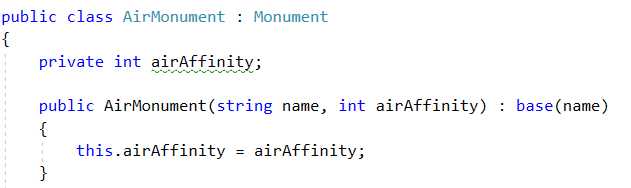
## Monuments

The second main entity are the Monuments. All the considerations for the Benders are valid here so we will create again one **abstract** class called just – **Monument.** Then all the **specific** monument classes will **inherit** from this main abstract one. The common characteristic for all the Monuments is their **name(string)** and again it will be just a field before we need any access or some kind of validation for it:



### Air Monument

Air Monuments has an additional field called – airAffinity(int):



### Water Monument

Its additional field is called – waterAffinity(int)**.**

### Fire Monument

Its additional field is called – fireAffinity(int)**.**

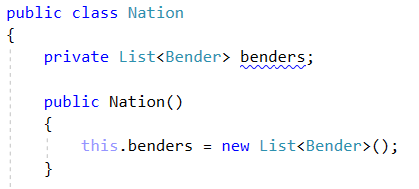
### Earth Monument

Its additional field is called – earthAffinity(int)**.**

## Nations

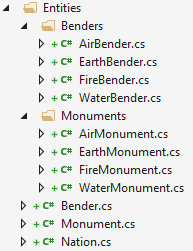
This class is **not** necessary to implement but overall it will **help** us in **designing** our code better as each nation will be supposed to **have some actions**/behaviour (methods). It is said that a nation is representing a **collection** of **Benders** from the same type.

As each nation’s collection of Benders will be of different type it is convenient to store the units in their common abstract class **Bender** so that each nation can store their specific type of Benders there without any changes required. We are still not sure exactly how we are going to add Benders to the collection so we’ll initialize just an empty list in the constructor without passing it from the outside of the class.



## Class Hierarchy Overview

So far you should have created all of the following classes:





# Stage II: Business Logic

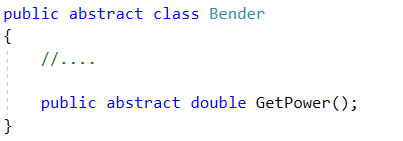
At this moment we get to know that a **nation** can **issue** a **war** and only the **strongest** nation will survive and also that a monument increases nations power. But there aren’t any further explanations on how should that happen so we will just leave it for now and get back to it when we enhance with the problem description.

## Classes Methods

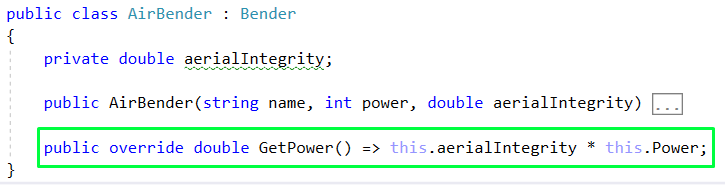
### Benders Power

The Benders are the source of power for their nation. Thus in war times we don’t need the power of each Bender, we would want to know the **total power** of the **nation** (which of course is calculated through the **sum** of **each** Bender’s **power**).

Since the benders are hold in a collection we can calculate the total power of a nation though some LINQ function over this collection. For that reason, we should **provide** our **Bender** **class** a **method** which **gives** us the **power** of each Bender. We may create a property which would grant us access to the field but this isn’t going to work because the power is calculated by **multiplying** **the** **power** of each Bender **by** their **specific field** (aerialIntegrity and etc.). At the same time each Bender should be able to provide his power and each of them will have this action but will calculate it through its own way. This will result in an **abstract method** which we will place **in the abstract class** – Bender and **then override** in each **specific class.**



Below is shown an example of the way you should override the abstract method in each specific Bender class. The rest is left to you as it is analogously.







































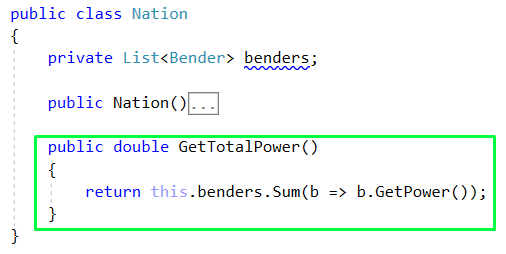






### Nations Power

Since we already have a method by which we can get each Benders power we can **provide** our **nations** with another **method** which will actually **calculate the total power** of all Benders.



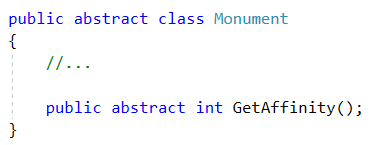
### Monuments Power Increase

The Monuments have the ability to increase a nations total power. Of course each nation could be influenced only by Monuments of the same type (e.g. air nation – air monument). A nations total power is **increased** by a **percentage** which is the **sum** of the **affinities** of all present Monuments of the needed type. Yeah, read it again, you will get it. It could be wise to store the different monuments directly in the different nations and the just modify the **GetTotalPower()** method of the **Nation** **class**.

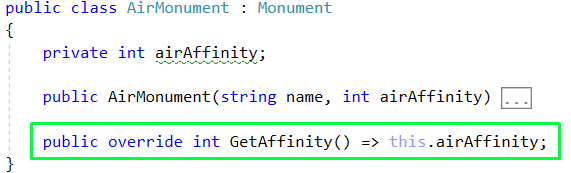
Let’s first add one more filed in the Nation class which will hold all the monument. Don’t forget to initialize it in the constructor with an empty list:



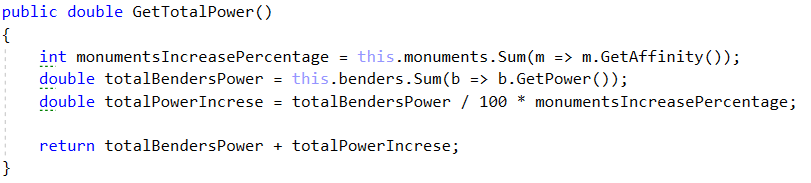
The reason we store just **Monument** inside is the same for which we store just Bender in the other collection. But since we are going to need every monuments affinity (which has a different name in every specific class) we will assign again an **abstract method** in the **abstract Monument** **class** which will be **overridden** **in** the **inheriting classes** and **give back** each monuments specific **affinity**.



Then in every specific Monument class override the above method like this:

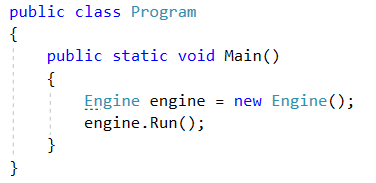


Now is time to modify the **GetTotalPower()** method so that includes the increase in power from the Monuments:



## Engine

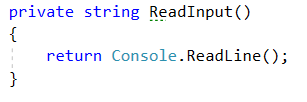
The Main method is not the best place to implement our Business Logic. Let’s create a new class which will process the main action – **Engine** and just call its method **Run** in the Main method after calling an instance of it.



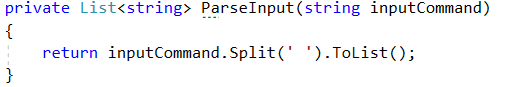
Inside the Engine class we will implement the continuous reading of input commands. This is done in the **Run method** and we will just add one helping variable of type **bool** which will **indicate** when the reading of commands should be **terminated** (turn off the Engine). Initially we set it true in the constructor before the Run method is called.



As you will notice there are two helper methods **ReadInput()** and **ParseInput()** which just make our code more easily readable:



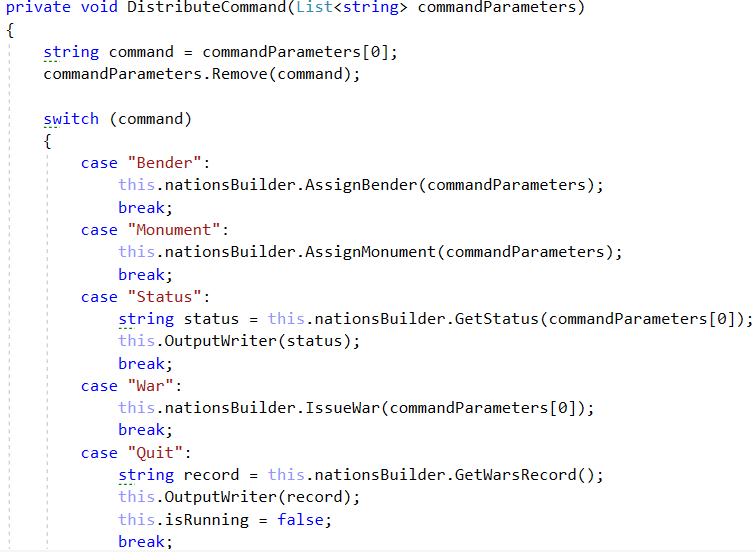
There is a reason why we give back the elements of the input command in list instead of just an array but you will see this in a while.



Another thing you might be wondering about is what is the **nationsBuilder** field and why is it there. We get an instance of it in the constructor and then we will place all our commands inside the **NationsBuilder** **class**. We do this separation because it **doesn’t** really sound like Engine’s job to create Benders, Monuments and etc. Its task will be to just distribute the commands that are received from the input.

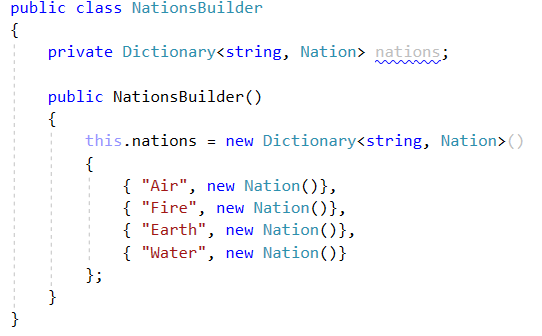
## Commands

The **distribution** of the **commands** will happen in the **DistributeCommand()** method which receives the already **parsed elements of the input** **command**. The implementation of each command will result in a new method **inside** the NationsBuilder class. After reading the command type we will call the according method from the NationsBuilder. In this way we will leave our switch **clean** and **tidy**.



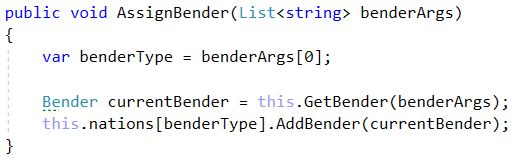
### Nations Builder

In the begging our **NationBuilder** **class** will have just one field – a dictionary which will keep our four nations and initialize them in the constructor. The rest is to come in the next subtasks:

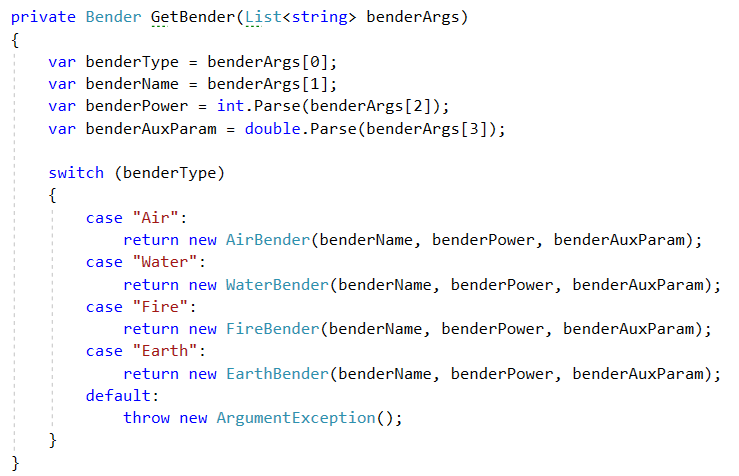


### Bender Command

In this command our task is to **create** one new instance of some of the four types of Benders and **add** it to the according nation of the Benders type. This will be done in the NationsBuilder class’s method **AssignBender().** This method should create and add the **new** Bender to its nation.



Create and assign have **different** **meanings** though so we will extract the creation itself in another **private method** called **GetBender()**. It’s function is very similar to a factory in a Factory pattern but we won’t complicate the code that further.



Also a new method in the **Nation class** is needed – **AddBender()**. Its function is just to add the newly created bender to the nations collection of benders.



### Monument Command

The Monument command has exactly the same structure as the previous one and the same methods should be implemented. You can repeat the steps but instead of Benders we are working with Monuments. The methods you should have created in the end are:

* **AssignMonument(List<string> monumentArgs) -** inside the NationsBuilder class
* **GetMonument(List<string> monumentArgs) -** inside the NationsBuilder class
* **AddMonument(Monument monument)-** inside the Nation class

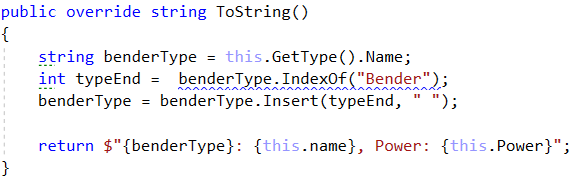
### Status Command

This command asks us to **print detailed information** about the nation whose **type** is passed as a parameter. These details include information about the Benders and Monuments for the current nation.

Let’s create the Benders string representation first. It is supposed to have the following pattern:



The information for the first two parameters is hold in the abstract Bender class which means we can override its **ToString()** method like this:



The name of the current class is taken through the **GetType()** method and we just insert a single space between the two words of the name which fully satisfies the needed output. The additional parameter that’s needed in the end of the message is unique for every type of Bender. That’s why we will finish this message by overriding the **ToString()** once again in every child class of the Bender just like this:

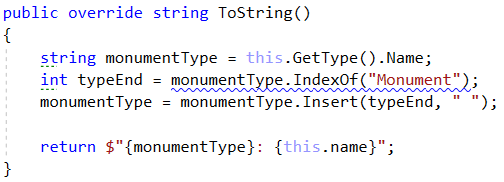


If we check the examples, we will notice that the final parameter should be formatted to the second digit after the decimal point. You should be able to replicate this structure for the other three types of Benders by yourself.

The same goes for the Monuments. They also need some custom sting representation in the following pattern:



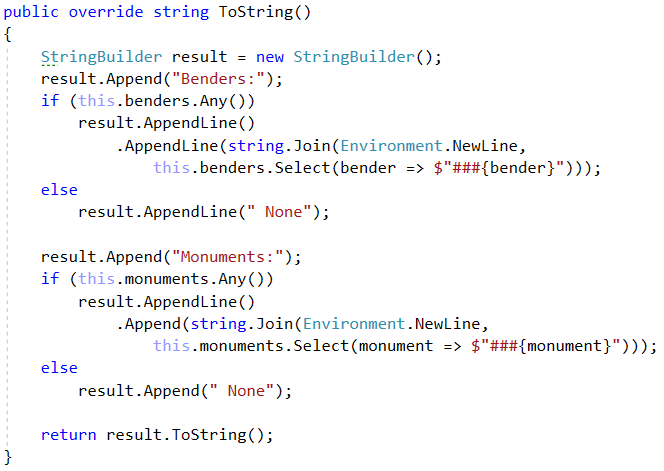
Overriding the **ToString()** methodin the abstracts class Monument looks like this:



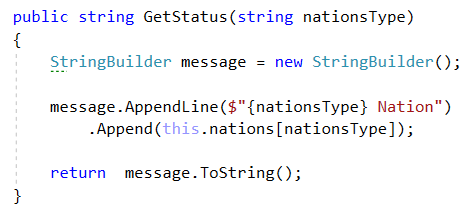
Again in every **specific** Monument class you should have something like this:



Next we will put everything together to form the whole nations detailed description. Let’s override the **ToString()** method in the Nation class:



Finally, in the **NationsBuilder** class create a method called – **GetStatus()**. It should be something like this:



The last thing we need to do is to provide our Engine with a method to show the output on the Console:

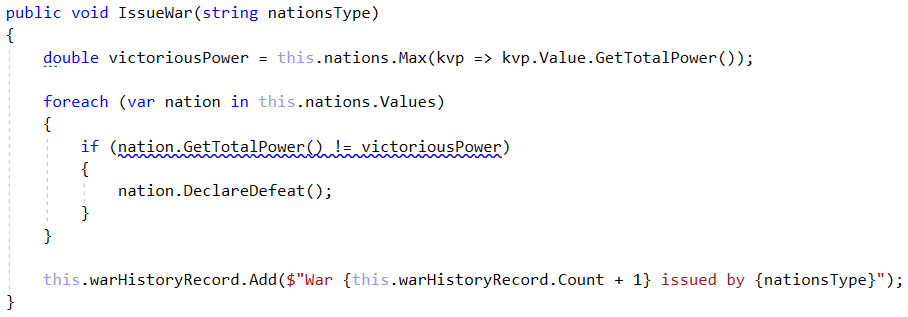


Now is a good idea to test the code you have written so far.

### War Command

This command is pretty simple. All we need to do is to involve all nations in war which means **getting** their **total** **power** and choosing the victor. Victorious is the nation with the **biggest** total power score – you are told there **won’t** be **two** nations with the **same** score.

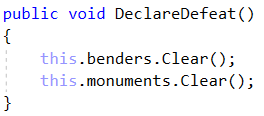
So let’s create another method in the NationsBuilder class called – **IssueWar()** and pass the name of the nation who issued the war. Inside we can get the winning score of the total powers of all the nations. Then iterate through the nations and **DeclareDefeat()** for all nations whose score is not the victorious one.



In the end of every issued war we should **keep** a record of it because in the end we will be **asked** to **print** all issued wars. For this reason, we are creating **new list** of strings and we will add a **new** **element** for **every** war. **Don’t forget** to **initialize** this new list in the **constructor**.



Declaring a defeat represents **erasing** **all** the **Benders** and **Monuments** a nation has collected. In order to simplify the process of defeating we will **add** a new **method** in the **Nation** **class** – **DeclareDefeat()**.

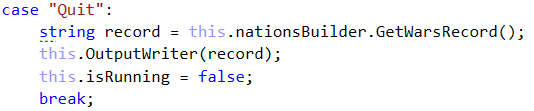


### Quit Command

This command has to tell us that our program should get terminated. But just before doing it we should print the information about all the wars that have been issued. Let’s create one last method in the NationsBuilder class - **GetWarsRecord()**. It will give back a single string holding all the history.



Of course in the Engine we should receive this message and print it on the Console:



Also you might have already noticed that we do one more thing under the Quit command and this is switching the value of the field **isRunning** to false. This has to tell our Engine to break the while cycle and terminate the program (turn off the Engine).