# Exercise: Prelude to OOP in Java

This document defines in-class exercise problems from the [“Java Fundamentals“ Course @ Software University](https://softuni.bg/trainings/1232/java-fundamentals-october-2015). You are presented with some problems and certain steps you need to take in order to accomplish the tasks.

## The Task:

Design a simple program that simulates a battle between a mage and a warrior. Both characters have similar features as well as some unique abilities.

Both classes have:

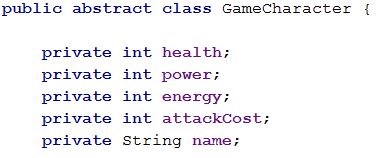
* Health
* Power
* Energy
* AttackCost
* Name

Both classes can:

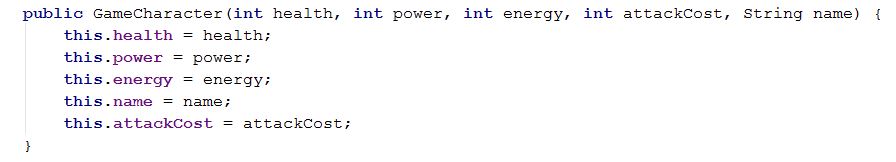
* Attack
* Respond to attack

### Abstract class:

All of these features are common, so we can define them in a base class called GameCharacter

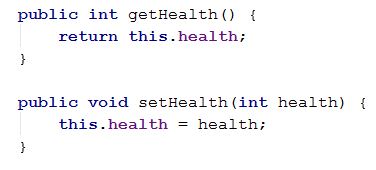


This is only a template. To create an actual object out of that template, we need a special method called a **constructor.** Constructors are defined such as:



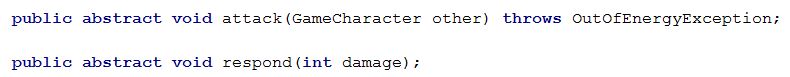
Hint: In **IntelliJ**, you can use **Alt+Insert** to auto-generate code for **constructors, getters, setters, etc** In **Eclipse -> right-click -> Source -> Generate Getters and Setters**

Those variables are called fields and are usually made private. Only the current object can access its own fields. To give access to other objects in our application, we already need the already mentioned getters and setters. For instance, for the ***health*** field, we need:



**Do the same for the other fields.**

We just defined our classes’ *state*. It is time to



The exception is a *custom* exception which will be defined later. Leave it **without** exception for now. We are done with the base class. Let’s go to the actual fighters. : )

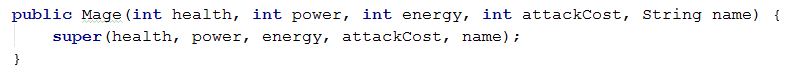
### Mage:

The keyword to inherit from a class in java is ***extends.*** When extending a class, we can use all of its public and protected features.

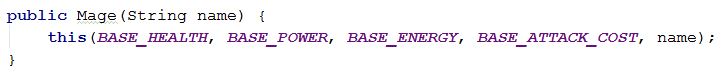
C:\Users\Edu\Pictures\mage.JPG

You will notice that our Mage class has health, energy, power, etc. All of these features come from the base class. You will also have to implement the two missing methods - ***attack*** and ***respond****.* Before that, we should define the constructors for the mage class. We can use overloaded constructors for convenience. Sometimes we only need a mage with basic statistics and a unique name, sometimes we may want to create a more powerful mage with more power and/or health.

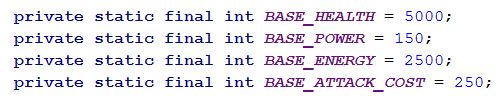
So, one of the constructors will require all the parameters needed to create a mage:



*Super* is a keyword that passes the arguments to the constructor in the base class. This way, we can store them in the private fields.

****

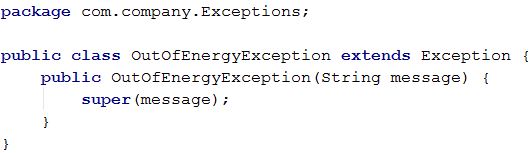
*This* will pass the parameters to the other constructor, which in turn will send them to the base class. The actual parameters are defined as constants, because it’s **never** a good idea to have **magic** variables in our code.



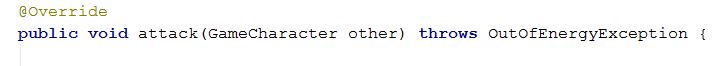
Now, it’s time for the two methods that we have left not implemented. However, to attack, we need some energy because our attacks have energy cost. Once we have no more energy, we can no longer attack. It’s a good idea to throw an exception at that moment, to notify the rest of the program. In order to be more descriptive, and to separate that unique case from other possible scenarios, we can define a custom exception.

We should always maintain a project structure. In other words, create a sub-package called Exceptions. This way, any other developer that works on our code, knows what to expect there.

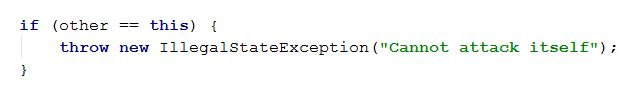
In the Exceptions package, add a new class called OutOfEnergyException. It should inherit from the base Exception class.



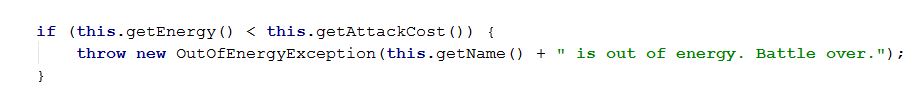
Once done with that, we can return to the Mage class. We were just about to implement the ***attack*** method.



In the actual method, it is a good idea to start with a few checks. First of all, the mage is also a GameCharacter. It is entirely possible to make a mistake and try to attack ourselves in our code. So, we need to be protected against that. It’s done with a simple if-clause:



IllegalStateException describes an event that is not normal for our problem. It is usually a ***coding mistake***, not a ***runtime*** event such as ***running out of energy when attacking.*** That’s why we didn’t put it in the method signature, such as the **OutOfEnergyException**. It, on the other hand, is something that we expect to happen and it is something that we have to catch. We will throw it when we – *you guessed it!* – are out of energy.



Now, let’s continue with the normal events in the method. First of all, as we already described, a mage ***may*** critically hit. Let’s say that this will happen 50% of the time. How can we decide if it’s a critical hit or not? Look at the Math.random () method. It produces a floating point number between 0 and 1. So, 50% of the time, that number will be smaller than 0.5. We can store this info in a Boolean variable.

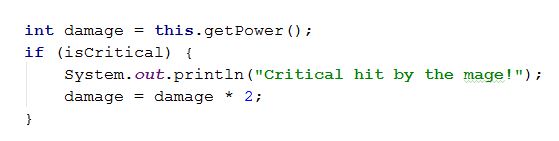
First, create a new constant:

C:\Users\Edu\Pictures\const.JPG

Then, store the random outcome in a Boolean variable:

C:\Users\Edu\Pictures\crit.JPG

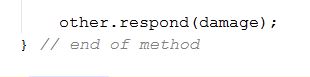
If the attack is critical, we will double our damage; however, it will not be a good idea to change the normal damage of the mage. This is introducing a side effect and may cause problems later. So, we can store the damage in a temporary variable and double it, if needed.



Finally, take some energy.

C:\Users\Edu\Pictures\eng.JPG

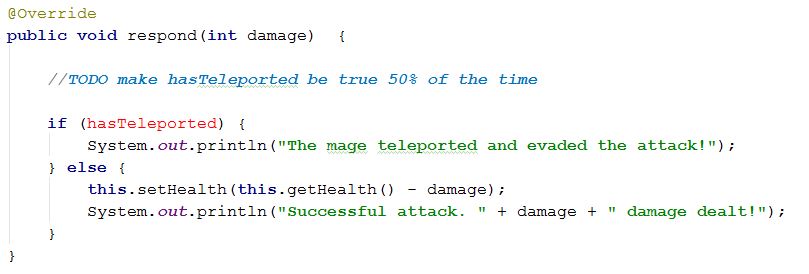
Now, we are done with our attack. It’s time for the other character to respond to it:



The whole method should look something like this:



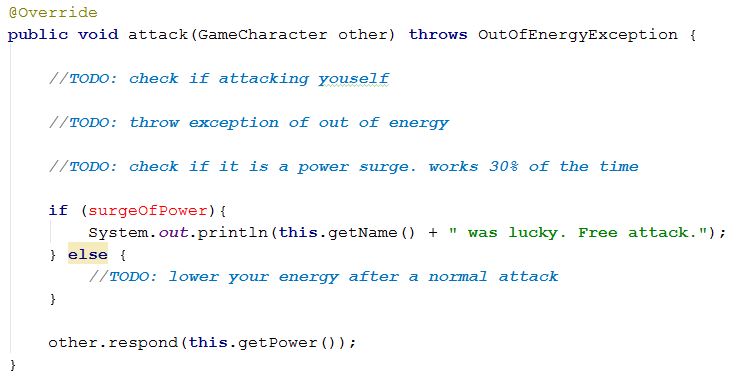
Now, we will implement the respond method of the mage. It will be slightly different compared to the warrior’s one. The mage has a chance to teleport and completely avoid the damage. Let’s say that this chance is 50%.



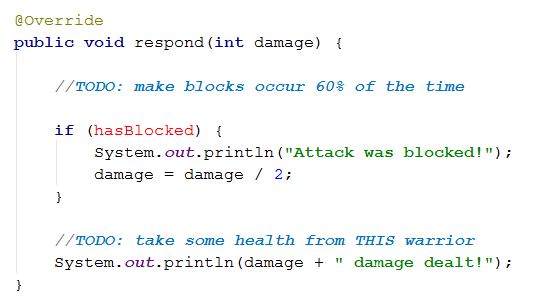
### Warrior:

The warrior is similar to the mage; it has health, power, etc, etc. He should also inherit from the base class. You can follow the instructions provided for the mage, and design the warrior class yourself. The steps that you need to take are:

1. Define the base statistics: **Health**: 5000; **Power**: 200; **Energy**: 1000; **AttackCost**: 150;
2. Define two overloaded constructors
3. Implement ***attack*** and ***respond*** methods from the base class.



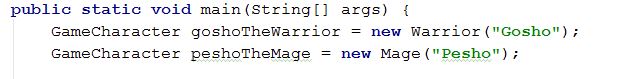
The respond method of the warrior is not very different from that of the mage. The warrior cannot teleport but can block. A successful block lowers the incoming attack damage by 50%.



We are ready with the mage and the warrior. As you saw, after moving all the similar features in a base class, it was much easier to design the two ***derived*** classes. The powerful feature of object oriented programming is that we can now create many additional classes with a very small amount of code, and all of those classes can interact with each other.

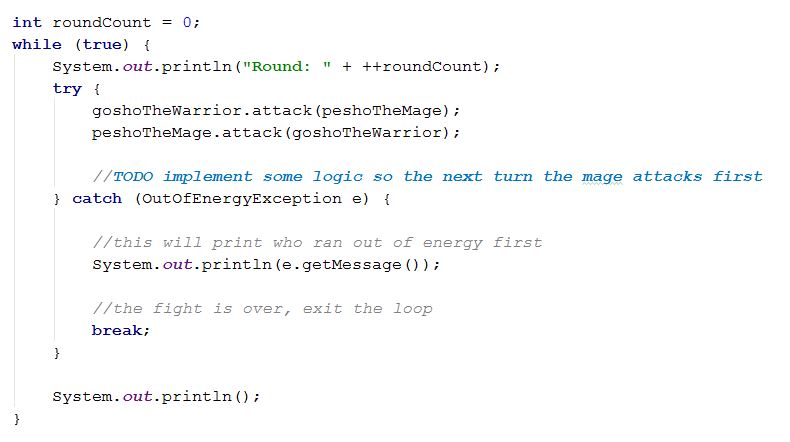
### The main method

We can now create a mage and warrior and let them fight. Since that are a lot of random critical hits, blocks, teleports and so on, we have no idea who will win. Also, both classes **have a lot of health,** do relatively small damage, and have / do not have a lot of energy. We are pacifists, so we not really going to kill them, it will be just a little spar. After **one** of them **runs out of energy**, the fight is **over.**

****

Notice that both the ***Warrior*** and the ***Mage*** are created stored in a variable of type ***GameCharacter.*** This is another feature of object-oriented programming called polymorphism. It is possible because each of the classes is derived from the base class. It also allows us to have only one attack method, where the class we attack is defined as a **GameCharacter.** Otherwise, we would have to have two identical methods, one for Warrior, and one for Mage. And what happens if we added a third class? Fourth? You get the point.

Now, let’s simulate the fight. An iteration of the loop is defined as *round*



After the battle is over, we have to define the winner. This is the class that has more health at the end. Print it.