Java Programming 2 – Lab Sheet 3

This Lab Sheet is based on material up to and including the lectures on objects and inheritance.

The deadline for Moodle submission of this lab exercise is 4:30pm on Thursday 15 October 2020.

Aims and objectives

- Writing Java classes including fields and methods
- Choosing appropriate data types for fields

Submission material

In this lab, you will create Java classes that will represent a simplified monster in a monster battling game. A **Monster** has the following properties:

- A string representing its **name** (e.g., "Charizard")
- One or two strings representing its **types** (e.g., "Fire", "Flying")
- Up to four **moves** which can be used in battle – the moves are represented by the **Move** class described below.

Each **Move** has the following properties:

- A string representing its **name** (e.g., "Fire Blast", "Air Slash")
- A string representing its **type** (e.g., "Fire", "Flying")
- A positive integer representing its **power** (e.g., 110, 75)

Moves can be accessed individually by their index (0-based), and can be swapped out for one another. For example, a **Monster** might have the following set of **Moves** to start with:

0. Fire Blast (Fire): 110
1. Air Slash (Flying): 75

The Monster's trainer might later add a third **Move** to the set, in position 2; the moves would then be:

0. Fire Blast (Fire): 110
1. Air Slash (Flying): 75
2. ***Inferno (Fire): 100***

At a later time, the trainer might decide to change the second **Move** for a different one, after which the moves would be

0. Fire Blast (Fire): 110
1. ***Mega Punch (Normal): 80***
2. Inferno (Fire): 100

Your task is to create two Java classes, one that should represent a **Move** and one that should represent a **Monster**.

The specification below describes the **public methods** that must be supported by each class. It is up to you what fields you want to define, and what the types of those fields should be.

If you complete the first two sections (**Move** class and basic **Monster** class), you will receive up to 4* on the lab depending on correctness. You must complete the "managing moves" section to receive 5*.

Move

Your **Move** class should have a **public** access modifier and should be saved in a file called **Move.java**. It should have exactly the following structure:

- One public constructor with the following signature:
  - `public Move (String name, String type, int power)`
- **Get** methods for all of the properties, with the following signatures:
  - `public String getName()`
  - `public String getType()`
  - `public int getPower()`
- An overridden version of the **toString()** method that produces a human readable representation of the Move, with the following signature:
  - `public String toString()`

Your **Move** class should have no **public** methods other than those listed above, and all of the fields must be declared as **private**.

Monster – basic behaviour

Just as with the **Move** class, the **Monster** class should be declared **public**, and should be saved in a file called **Monster.java**. It should have exactly the following set of public methods:

- Two public constructors – one for monsters with exactly one type, and one for monsters with two types – with the following signatures:
  - `public Monster (String name, String type)`
  - `public Monster (String name, String type1, String type2)`
- A **get** method for the monster name, with the following signature:
  - `public String getName()`
- A method to test whether the monster has a given type, with the following signature:
  - `public Boolean hasType (String type)`
  - *This method should return **true** if **type** is one of this Monster's types, and **false** if not*
- An overridden version of the **toString()** method that produces a human readable representation of the Monster, with the following signature:
  - `public String toString()`

Monster – managing moves

Once the classes above have been created, your final task is to put them together to implement the full behaviour described on the front page.

Each **Monster** should be able to have up to four **Move**s in its list. The program implementation should allow each of the moves to be accessed by its position in the list, and it should be possible to change the move at each position.

Concretely, to implement this behaviour, you will need to add the following two methods to your **Monster** class:

- `public Move getMove (int index)`
  - This method should return the **Move** as position **index** in the Monster's list. You can assume that **index** is a number between 0 and 3. If the monster has no move at position **index**, this method should return **null**.
- `public void setMove (int index, Move move)`
  - This method should set the **Move** at position **index** in the Monster's list to the given move, where again **index** can be assumed to be a number between 0 and 3.

You can add whatever field(s) you require to your **Monster** class to support the above behaviour. Note that any newly added fields must have a **private** access modifier (just like the originally specified fields above). If your implementation requires it, you can also add additional methods, but any methods other than those above must also be **private**.

Examples

The following are some example behaviours that should be supported. Note that the exact format of the **toString()** output is up to you, so your output may not look exactly the same as this, but the overall behaviour should be the same.

```
jshell> Monster charizard = new Monster("Charizard", "Fire", "Flying");
charizard ==> Charizard [Fire, Flying]

jshell> charizard.getName();
$4 ==> "Charizard"

jshell> charizard.hasType("Fire");
$5 ==> true

jshell> charizard.hasType("Flying");
$6 ==> true

jshell> charizard.hasType("Normal");
$7 ==> false

jshell> Monster eevee = new Monster("Eevee", "Normal");
eevee ==> Eevee [Normal]

jshell> eevee.hasType("Normal");
$9 ==> true
```

```
jshell> eevee.hasType("Fire");
$10 ==> false

jshell> Move fireBlast = new Move("Fire Blast", "Fire", 110);
fireBlast ==> Fire Blast (Fire): 110

jshell> fireBlast.getName();
$12 ==> "Fire Blast"

jshell> fireBlast.getType();
$13 ==> "Fire"

jshell> fireBlast.getPower();
$14 ==> 110

jshell> charizard.getMove(0);
$15 ==> null

jshell> charizard.setMove(0, fireBlast);

jshell> charizard.getMove(0);
$17 ==> Fire Blast (Fire): 110

jshell> charizard.getMove(1);
$18 ==> null

jshell> Move bodySlam = new Move("Body Slam", "Normal", 85);
bodySlam ==> Body Slam (Normal): 85

jshell> eevee.setMove(2, bodySlam);

jshell> eevee.getMove(2);
$21 ==> Body Slam (Normal): 85
```

Hints and tips

1. Make sure that none of your methods produce any output through **System.out.println** or similar mechanisms. You are implementing a class that will (in theory) be used as part of a larger system, so output is not appropriate unless specifically requested.
2. You must follow the specifications given exactly. **This will be checked automatically on your submission**. In particular:
    - Both classes must have the correct names (**Move** and **Monster**), must be declared **public**, and must be in files called **Move.java** and **Monster.java**, respectively.
    - All fields of both classes must be declared **private**
    - Your classes must have exactly the **public** methods and constructors listed above, with exactly the given signatures.
        - If your implementation requires it, you can add additional methods and/or constructors. Be sure that any extra methods or constructors are declared **private**.
3. It is entirely up to you what data types you use for the class fields. You can complete this lab using only the types we have learned so far (primitives, Strings, arrays) – if you want to use some other data representation, you can do so, but the tutors may not be able to help you fully.

How to submit

You should submit your work before the deadline no matter whether the programs are fully working or not.

When you are ready to submit, go to the JP2 moodle site. Click on **Laboratory 3 Submission**. Click 'Add Submission'. Open Windows Explorer and browse to the folder that contains your Java source code and drag only the *two* Java files **Monster.java** and **Move.java** into the drag-and-drop area on the moodle submission page. Then click the blue save changes button. Check the .java files are uploaded to the system. Then click **submit assignment** and fill in the non-plagiarism declaration. Your tutor will inspect your files and return feedback to you via moodle.

Outline Mark Scheme

Your tutor will mark your work and return you a score in the range "Excellent" (*****) to "Very poor" (*). We will automatically execute your submitted code and check its output, and will also automatically test all fields and methods of both classes to make sure that they meet the specification. We will also look at the code before choosing a final mark.

Example scores might be:

5*: you complete the code for all parts, including the additional methods on **Monster**, with no errors

4*: you complete all of the code except for the additional **Monster** methods with no errors, OR you complete all code for all parts but with minor errors

3*: you complete all code but with more major errors, OR you complete only the basic classes with minor errors

2*: some attempt made

1*: minimal effort

For this assignment, we will again consider correctness and will not deduct marks for inappropriate coding style. However, your tutors may comment on your style, and you are advised to follow the examples of the code given during lectures regarding indentation, variable names, formatting, etc. Starting from Lab 4 onwards, style will be taken explicitly into account in your lab grades.

Possible extensions

If you have completed the tasks for the lab, here are some possible extensions you could try. **Do not submit code for any of these tasks** – the tutors only want to see your solutions to the required problems. Note that some of these extensions are things that we will be doing during the labs in the coming weeks, but you might want to try your own version of them in advance.

- Add error checking in various places – for example, you could ensure that all monsters and moves only have valid types,[1] and that all numeric values are in range.
- Try to write code to make monsters "battle" each other with their moves. You might need to add additional features such as hit points to the monsters, to allow them to damage each other with their moves.
- If you implement battling, you might also want to incorporate features such as type effectiveness[2] and Same Type Attack Bonus (STAB)[3] into the battle process.

[1] See https://bulbapedia.bulbagarden.net/wiki/Type for a list
[2] https://bulbapedia.bulbagarden.net/wiki/Type#Type_chart
[3] https://bulbapedia.bulbagarden.net/wiki/Same-type_attack_bonus