T5 - Java Seminar

T-JAV-500

Day 05

Abstract classes and Interfaces

3.3

Day 05

language: Java

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.

Today you will delve even deeper into OOP.
You will keep using all the previous days' concepts, and also discover a few new things:

- Abstract class
- Abstract methods
- Interfaces

These are real cornerstones for Java language.

An abstract class is a class that can not be instantiated (which means that no object of this class can be created directly).
To do so, one need to create a child class that inherits from this abstract class.
Abstract java classes require the "abstract" keyword.

As you will see, it is extremely useful! It allows polymorphism (it is not a swear word, you can google it).

An abstract method is a method without implementation.
It is not mandatory for an abstract class to contain abstract methods, but if a class contains abstract methods, it MUST be defines as abstract too.

An interface is a similar to a class with only abstract methods.
But it is not technically a class.
A class can implement one *or several* interface(s), thanks to the **implements** keyword.

By convention, interfaces usually start with an uppercase 'i', and abstract classes usually start with an uppercase 'a'. It is not mandatory and it could start with the letter in lower case, or simply not start with those letters at all.

Unless specified otherwise, all messages must be followed by a newline.

Unless specified otherwise, the getter and setter name will always add “get” or “set” in front of the name of the attribute, in CamelCase.

EXERCISE 01

File to hand in: ./Weapon.java

Create an `Weapon` abstract class with the following protected attributes:

- **name** (a string): the name of the weapon,
- **apcost** (an integer): the action point cost to use the weapon,
- **damage** (an integer): the amount of damage dealt by the weapon,
- **melee** (a boolean): true if the weapon is used for close combat false otherwise.

These attributes will have getters (**getName**, **getApcost**, **getDamage**, **isMelee**) but no setter.
This class' constructor will take the name, the apcost, the damage and the melee status in this very order.

Also add an abstract public method named **attack** that takes no parameter and returns nothing.

Be careful! You cannot instantiate an abstract class.

The constructor should **NOT** be public.

`boolean` is not exactly the same as `Boolean`.

EXERCISE 02

Files to hand in: ./Weapon.java
./PlasmaRifle.java
./PowerFist.java

Create these two classes, inheriting from `Weapon`, with the given features:

```
PlasmaRifle

Name: "Plasma Rifle"
Damage: 21
AP cost: 5
Output of attack method: "PIOU"
Melee: false
```

```
PowerFist

Name: "Power Fist"
Damage: 50
AP cost: 8
Output of attack method: "SBAM"
Melee: true
```

A call to attack() must display the specific output of the weapon followed by a newline.

Don't rewrite the constructors fully, use the one from the parent class!

EXERCISE 03

File to hand in: ./Fighter.java

Let's create our first interface. It will be called Fighter.
Its goal is to determine the base methods that we want our fighting units to implement when they are created.

Add a few public methods to this interface in order to do that:

- **boolean equip(Weapon)**
- **boolean attack(Fighter)**
- **void receiveDamage(int)**
- **boolean moveCloseTo(Fighter)**
- **void recoverAP()**
- **String getName()**
- **int getAp()**
- **int getHp()**

`boolean` is not exactly the same as `Boolean`.

EXERCISE 04

Files to hand in: ./Fighter.java
./Weapon.java
./PlasmaRifle.java
./PowerFist.java
./Unit.java
./Monster.java

First we will create an intermediate abstract class `Unit` that implements the generic methods of the `Fighter` interface.
A `Unit` has three main attributes: “name”, “hp”, and “ap”.
The first one is the name of the unit.
The second is its health point.
The third one is “action points”, the resource that our unit uses to make an action.
This classe must have only one constructor: a protected constructor taking these values as parameters in the declared order.

UNIT

The `Unit` class must implement the following methods : **getName**, **getHp** and **getAp**, **receiveDamage**, **moveCloseTo** and **recoverAP** .

RECEIVEDAMAGE

This **receiveDamage** function must always receive an integer representing the damage suffered by the unit. The unit’s HP must be reduced by this amount.

If the HP gets to 0 (or below), the unit must be considered dead and each of its methods (except the getter) should return false from that point on.

Is there any reason to store a negative HP value? Dead is dead.

moveCloseTo

This function will make the unit move closer to its target, which means that a later call to "attack" will be successful.

We will consider that the unit can only be close to one target at a time.

If the unit is not already close to its target, a call to this function will display:

```
[name] is moving closer to [target's name].
```

It returns `true` is the unit moved closer to the target and `false` otherwise.

Can you move close to yourself?

recoverAP

A call to the **recoverAP** method increases the monster's AP by 7 at most.
It should never go over 50.

Monster

Now we will actually implement the base for our Monsters and our SpaceMarines (in the next exercise), both abstract classes extending `Unit`.

Let's first concentrate on the `Monster` class.
Add a "damage" attribute and its getter, **getDamage**.
It must be set to 0 for the time being (and later will be set differently for each monster).
Also add an "apcost" attribute (and it's getter), also set to 0 for now.

equip

Implement the **equip** method.
It must display:

```
Monsters are proud and fight with their own bodies.
```

ATTACK

All monsters have a “melee” type, which means that they first need to get within melee range (see **moveCloseTo** method) before being able to attack their target.
So if the monster is not in the melee range of the `Fighter` passed as parameter, display:

```
[name]: I'm too far away from [target's name].
```

where [name] is the name of the monster and [target’s name] its target’s name.

If our monster is in melee range, it must check if it has enough AP to attack; that’s when its “ap” and “apcost” attributes matter.
In order to attack, it should have at least the same “ap” available than an attack’s “apcost”.

If the attack is successful you must deduct “apcost” from “ap” and call the target’s **receiveDamage** method while passing the monster’s “damage” as parameter.
You should display the following before calling **receiveDamage**:

```
[name] attacks [target's name].
```

EXERCISE 05

Files to hand in: ./Unit.java
./SpaceMarine.java
./Weapon.java
./PlasmaRifle.java
./PowerFist.java

Let’s now create our `SpaceMarine` abstract class extending Unit and add a “weapon” attribute and its getter, **getWeapon**.

EQUIP

Our SpaceMarine needs to take this new Weapon and equip it.
If it’s done successfully, display:

```
[name] has been equipped with a [weapon's name].
```

where [name] is our SpaceMarine’s name and [weapon’s name] his weapon’s name.

If the weapon has already been taken by another SpaceMarine, the function will do nothing.
It’s up to you to decide how to handle this.

ATTACK

If our SpaceMarine doesn't have any equipped weapons, the function do nothing but output:

```
[name]: Hey, this is crazy. I'm not going to fight this empty-handed.
```

If the equipped weapon is a melee one and our SpaceMarine is not in range, he must say:

```
[name]: I'm too far away from [target's name].
```

Like in Monster, our SpaceMarine needs enough AP to attack.
If its available AP are at least equal to his weapon's cost, and if he is in range (or is using an in-range weapon), call the equipped weapon's **attack** method in addition to the target's **receiveDamage** method while passing the weapon's damage as parameter.

Also, if the attack has been successful, you should deduct the weapon's cost from the SpaceMarine's AP and display the following before calling the weapon's **attack** method:

```
[name] attacks [target's name] with a [weapon's name].
```

RECEIVEDAMAGE

This function works exactly like Monster.

MOVECLOSETO

If our SpaceMarine has a melee weapon, this function works exactly like Monster.
Otherwise, this function does nothing and returns false.

RECOVERAP

This function works exactly like Monster, except that it will recover 9 AP instead of 7.

EXERCISE 06

Files to hand in: ./Unit.java
./Monster.java
./SpaceMarine.java
./Weapon.java
./PlasmaRifle.java
./PowerFist.java
./TacticalMarine.java
./AssaultTerminator.java

It is finally time to create our actual Space Marines.
As you should have guessed, all SpaceMarines need to inherit from the SpaceMarine class.

Let's begin with TacticalMarine.
At the beginning of it creation, he should say

```
[name] on duty.
```

where [name] is his name, necessarly given during his creation.

During his creation, the TacticalMarine is equiped with a PlasmaRifle.
By default, a TacticalMarine has 100HP and 20AP.
Since a Tactical Marine is... tactical, it regenerates his AP faster than other Marines: he will actually take back 12 AP instead of 9, when **recoverAP** is called.

Let's talk about AssaultTerminator now.
When being created, he also receives his name and displays (before all):

```
[name] has teleported from space.
```

By default, he possess 150HP and 30AP.
During his creation, he is equiped with a PowerFist.
Also, since AssaultTerminator is a bit more resistant than other Marines, when his **receiveDamage** method is called, he reduces the damage by 3.

However, the received damage can't be reduced under 1.

Exercise 07

Files to hand in: ./Unit.java
./Monster.java
./SpaceMarine.java
./Weapon.java
./PlasmaRifle.java
./PowerFist.java
./TacticalMarine.java
./AssaultTerminator.java
./RadScorpion.java
./SuperMutant.java

We will now finally create our Monsters!
As you should have guessed, all of the monsters classes must inherit from the `Monster` class.

Our monsters will have generic names: they will all be called "RadScorpion" or "SuperMutant", followed by an id.
For example, the first RadScorpion will be called "RadScorpion #1", the second one will be "RadScorpion #2",...

Thus, no need to give monsters any parameters when they are created.

Let's create the RadScorpion first. When created, it displays:

```
[name]: Crrr!
```

where [name] is its name.

RadScorpion is a fairly common monster, it only has 80HP.
It starts with the maximum AP, 50, and each one of its attacks deals out 25 damages, and costs 8 AP.
However, they can be pretty scary. That's why they will deal out double damage if they are attacking a marine, who is not an "AssaultTerminator" (because the other marines are too scared to run away).

Now let's create our SuperMutant. When created, it displays:

```
[name]: Roaarrr!
```

It starts with 170HP and 20AP.
Each one of its attacks deals out 60 damages, but costs 20Ap.
When SuperMutant recover AP, they also recover HP, with a maximum of 10HP recovered by call (170HP being their full health).

EXERCISE 08

Files to hand in: ./Unit.java
./Monster.java
./SpaceMarine.java
./Weapon.java
./PlasmaRifle.java
./PowerFist.java
./TacticalMarine.java
./AssaultTerminator.java
./RadScorpion.java
./SuperMutant.java
./SpaceArena.java

Create a SpaceArena class, to simulate fights between teams of Monsters and teams of SpaceMarines.

This class must have 3 methods:

- **enlistMonsters**
  It takes a List of `Monster` as parameter and add theses monsters to the one that is already registered to fight.
- **enlistSpaceMarines**
  It takes a `List` of `SpaceMarine` as parameter and add them to the one that is already registered to fight.

It should not be possible to add a warrior that is already enlisted for a fight.

- **fight**.
  It takes no parameters and returns a boolean indicating whether there was at least one round or not.
  It makes the enlisted teams of monsters and spaceMarines fight amongst themselves.
  If no monsters are registered, it outputs:
  `No monsters available to fight.`
  Else, if no spaceMarines are registered, it outputs:
  `Those cowards ran away.`
  If at least one of the two teams is missing, the function stops there by returning false.

Here is how a round should proceed:

- When a new round between a monster and spaceMarine begins, the spaceMarine always goes first.
- The one playing will try to attack, if it's successful, its turn is over.
  If it failed because he wasn't in range, he will go closer.
  If it failed because he didn't have enough AP, he will call his **recoverAP** method once.
- It is then the opponent's turn.
  This process repeats until one of the opponents has fallen.
- The winner calls his **recoverAP** function once before starting the next fight.
- If the spaceMarine wins, the second monster comes in the arena and the whole process starts again until one of the two teams has been defeated (if the monster has won, then the second spaceMarine enters the fray).

Every time a monster or a spaceMarine enters the arena to fight, display:

```
[name] has entered the arena.
```

If ever both fighters enter at the same time, the SpaceMarine is always introduced first.

At the end of the fight (when one of team doesn't have any warrior left), display:

```
The [team's name] are victorious.
```

where [team's name] is whether "monsters" or "spaceMarines".

Remember, each winner stays in the arena waiting for the next round.

Here is a little example…

```
import java.util.*;

public class Example {
    public static void main(String[] args) {
        SpaceArena arena = new SpaceArena();

        arena.enlistMonsters(Arrays.asList(new RadScorpion(), new SuperMutant(), new
            RadScorpion()));
        arena.enlistSpaceMarines(Arrays.asList(new TacticalMarine("Joe"), new
            AssaultTerminator("Abaddon"), new TacticalMarine("Rose")));
        arena.fight();

        arena.enlistMonsters(Arrays.asList(new SuperMutant(), new SuperMutant()));
        arena.fight();
    }
}
```

The result of this example can be seen in a text file given alongside this subject.