EPITECH. TECHNOLOGY

JAVA SEMINAR

DAY 05 - ABSTRACT CLASSES AND INTERFACES

JAVA SEMINAR

Contents

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

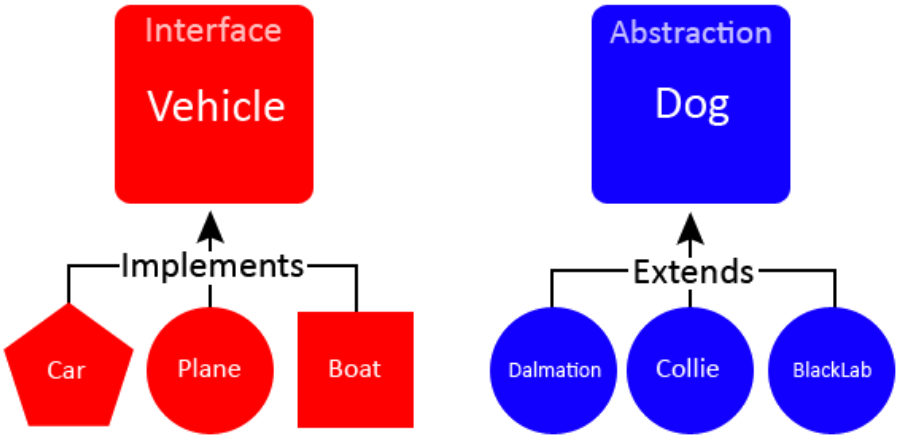
These are real cornerstones for Java language.

An **abstract class** is a class that can not be instantiated.
That 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.

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 interfaces, thanks to the `implements` keyword.

By convention, interfaces start with an `I`, and abstract classes start with an `A`.

Unless specified otherwise, all messages must be followed by a newline and the names of the getter and setter for `Attribute` will always be like `getAttribute` and `setAttribute`.
For instance, attribute `Bobby` will have `getBobby` and `setBobby`.
FYI, this name convention is known as *CamelCase*.

Exercise 01

Delivery: `./Weapon.java`

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

- ✓ `name` (string): the name of the weapon ;
- ✓ `apcost` (integer): the action point cost to use the weapon ;
- ✓ `damage` (integer): the amount of damage dealt by the weapon ;
- ✓ `melee` (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` in this 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

Delivery: `./Weapon.java`, `./PlasmaRifle.java`, `./PowerFist.java`

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

attribute	`PlasmaRifle`	`PowerFist`
`name`	Plasma Rifle	Power Fist
`damage`	21	50
`apcost`	5	8
`output`	PIOU	SBAM
`melee`	false	true

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

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

Exercise 03

Delivery: `./Fighter.java`

Let's create your first interface. It will be called `Fighter`. It'll determine the base methods that we want your 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

Delivery: `./Weapon.java`, `./PlasmaRifle.java`, `./PowerFist.java`, `./Fighter.java`, `./Unit.java`, `./Monster.java`

Create a abstract class, named `Unit`, that implements the generic methods of the `Fighter` interface.

A `Unit` has three main attributes:

- ✓ its `name` ;
- ✓ its health points `hp` ;
- ✓ its action points `ap` which is the resource used to make an action.

It had only one protected constructor that takes these values as parameters in this order.

Unit

The `Unit` class must implement the following methods:

- ✓ `getName`, `getHp` and `getAp`
- ✓ `receiveDamage`, `moveCloseTo` and `recoverAP`.

receiveDamage
This method always receives 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 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 moves the Unit closer to its target (a later call to `attack` could 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 `[Unit's 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 closer to yourself?

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

Monster

You will implement the base for your `Monster` and `SpaceMarine` (next exercise), both abstract classes extending `Unit`. Let's first concentrate on the `Monster` class:

- ✓ add a `damage` attribute and its getter, `getDamage` ;
- ✓ set it to 0 for the time being (and later will be set differently for each monster) ;
- ✓ add an `apcost` attribute (and it's getter), also set to 0 for now.

The `apcost` and `damage` attributes need to be protected.

equip
Implement the `equip` method that displays `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.

If `Monster` is not in the `melee` range of the `Fighter` target given as parameter, display `[Monster's name]: I'm too far away from [target's name].`

If your monster is in `melee` range, it must check if it has enough `ap`. 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.
Before calling `receiveDamage`, you should display `[Monster's name] attacks [target's name].`

Exercise 05

Delivery: `./Weapon.java`, `./PlasmaRifle.java`, `./PowerFist.java`, `./Unit.java`, `/SpaceMarine.java`

Create the `SpaceMarine` abstract class extending `Unit`.
Add a `weapon` attribute and its getter `getWeapon`.

equip

Your SpaceMarine needs to take this new Weapon and equip it.
Display `[SpaceMarine's name] has been equipped with a [weapon's name].` if it's done successfully.

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

attack

If a `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 the `SpaceMarine` is not in range, he must say `[SpaceMarine's name]: I'm too far away from [target's name].`.

Like `Monster`, the `SpaceMarine` needs enough `ap` to attack.
If `ap` are at least equal to his weapon's `apcost`, 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 `apcost` from the `SpaceMarine`'s `ap` and display `[SpaceMarine's name] attacks [target's name] with a [weapon's name].` before calling the weapon's `attack` method.

receiveDamage

This function works exactly the same for `SpaceMarine` that for `Monster`.

moveCloseTo

If your `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

Delivery:
`./Weapon.java`, `./PlasmaRifle.java`, `./PowerFist.java`, `./Unit.java`, `./Monster.java`, `/SpaceMarine.java`, `./TacticalMarine.java`, `./AssaultTerminator.java`

It is time to create the space marines.
As you've guessed, each of them inherits from the `SpaceMarine` class.

Let's begin with `TacticalMarine`. At creation:

- ✓ its `name` is necessarly given ;
- ✓ display `[TacticalMarine's name] on duty.` ;
- ✓ `TacticalMarine` is equiped with a `PlasmaRifle` ;
- ✓ `TacticalMarine` has 100 `hp` and 20 `ap` by default ;
- ✓ `TacticalMarine` takes back 12 `ap` instead of 9 when `recoverAP` is called.

Let's talk about `AssaultTerminator` now. At creation:

- ✓ its `name` is necessarly given ;
- ✓ display `[AssaultTerminator's name] has teleported from space.` ;
- ✓ `AssaultTerminator` is equiped with a `PowerFist` ;
- ✓ `AssaultTerminator` has 150 `hp` and 30 `ap` by default ;
- ✓ `AssaultTerminator` reduces the damage by 3 when `receiveDamage` is called.

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

Exercise 07

Delivery:
`./Weapon.java`, `./PlasmaRifle.java`, `./PowerFist.java`, `./Unit.java`, `./Monster.java`, `/SpaceMarine.java`, `./TacticalMarine.java`, `./AssaultTerminator.java`, `./RadScorpion.java`, `./SuperMutant.java`

You will now create the monsters!
As you've guessed, each of them inherits from the `Monster` class.

Monsters have generic names: they are called `RadScorpion` or `SuperMutant`, followed by an id.
For instance, the first RadScorpion is named `RadScorpion #1`, the second one `RadScorpion #2`, ...

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

Let's create the `RadScorpion` first. When created:

- ✓ it displays `[RadScorpion's name]: Crrr!`
- ✓ it has 80 `hp` and starts with the maximum `ap` (50) ;
- ✓ each basic attack deals out 25 `damage` and costs 8 `ap` ;
- ✓ each attack on a marine who is not an `AssaultTerminator` deals out double `damage`.

Now, create the `SuperMutant`. When created:

- ✓ it displays `[SuperMutant's name]: Roaarrr!` ;
- ✓ it starts with 170 `hp` and 20 `ap` (170 `hp` being their full health that they can't exceed);
- ✓ each attack deals out 60 `damage` and costs 20 `ap` ;
- ✓ when SuperMutant recover `ap`, they also recover 10 `hp` by call.

Exercise 08

Delivery:
`./Weapon.java`, `./PlasmaRifle.java`, `./PowerFist.java`, `./Unit.java`, `./Monster.java`, `/SpaceMarine.java`, `./TacticalMarine.java`, `./AssaultTerminator.java`, `./RadScorpion.java`, `./SuperMutant.java`, `./SpaceArena.java`

Create a `SpaceArena` class.
It will simulate fights between teams of monsters and teams of space marines.

This class has 3 methods:

- ✓ `enlistMonsters` takes a list of `Monster` as parameter and add them to the the list of registered ;
- ✓ `enlistSpaceMarines` takes a list of `SpaceMarine` as parameter and add them to the list of registered.

It should not be possible to add a `Fighter` that is already registered.

- ✓ `fight`:
  - takes no parameters ;
  - returns a boolean indicating whether there was at least one round or not ;
  - makes the enlisted teams of monsters and space marines fight themselves ;
    - if no monsters are registered, it outputs `No monsters available to fight.` ;
    - if no space marines are registered, it outputs `Those cowards ran away.` ;
    - if at least one of the two teams is missing, it stops and returns `false`.

When a new round begins, the space marine always goes first.

The first `Fighter` 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 a space marine wins, a second monster comes in the arena and the whole process starts again until one of the two teams has been defeated (if a monster wins, then a second space marine enters the fray).

Every time a `Fighter` (monster or a space marine) join the arena to fight, display `[Fighter's name] has entered the arena.`

If 2 fighters enter at the same time, the space marine 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.

v 3.4