

Inheritance, Overriding and Dynamic Binding

OOP in Scala

2016

1 OOP Design

This exercise introduce a systematic method that can be used to design object systems: CRC cards ($\{\text{Class, Responsibilities, Collaborators}\}$)¹. Each card describe a class of objects, abstracting away its implementation. Each card has three components:

- **Name of class:** This name creates a vocabulary between the conceptors of an application. It must thus be wisely chosen, to correctly describe the class purpose. It must be as indicative as possible, since it will be used in a much wider scope.
- **Responsibilities:** They identify the problems solved by this class. To define them, ask yourself the following questions: *what should my object **know**?* and *what should my object **do** in my application?*

The first category of responsibilities (know things) encompasses mainly the values it should save while the second category of responsibility (do things) encompass the following:

- do a given computation ;
- modify its internal state in some way;
- create and initialize other objects ;
- control and coordinate the activity of other objects.

- **Collaborators:** The names of classes with which this class must cooperate to achieve its responsibilities. So they are the classes to which this one will send messages (call methods) or from which it will receive messages.

Here is an example CRC card, form a simple dice.

Dice	
Responsibilities	Collaborators
<ul style="list-style-type: none">– Save the value (a char) of each of the 6 sides– Save the currently visible side– Allow to retrieve the visible side– Allow to cast another dice (the visible side is updated)	<ul style="list-style-type: none">– java.util.Random

★ **Exercise 1:** L'objectif de cet exercice est de déterminer les classes nécessaires à la réalisation d'un interpréteur rudimentaire LOGO.

Le langage LOGO a été crée au dans les années 1960 au Massachusetts Institute of Technology (MIT) par Wally Feurzeig et Seymour Papert. C'est un bon langage d'initiation à la programmation en particulier pour les enfants grâce au côté ludique de la tortue graphique. La tortue graphique peut effectuer les actions suivantes : avancer de N pixels, tourner à droite de N degrés, tourner à gauche de N degrés, reculer de N pixels, se cacher, se montrer, lever le crayon, poser le crayon, changer la couleur du crayon.

La classe `tortue.Screen` représente l'écran graphique de notre application. L'interface publique offerte par cette classe est la suivante :

¹K. Beck and W. Cunningham. A Laboratory for Teaching Object-Oriented Thinking, in Proceedings of OOPSLA'89. pp.~1–6, 1989. ACM Press. <http://doi.acm.org/10.1145/74877.74879>

```

1 class Screen {
2   def Screen(width:Int, height:Int)
3   def setForegroundColor(c:Color) :Unit
4   def drawLine(xA:Int, yA:Int, xB:Int, yB:Int) ;
5   def fillRectangle(xA:Int, yA:Int, xB:Int, yB:Int) ;
6   def setBackgroundColor(c:Color) ;
7   def clear() :Unit
8 }

```

- ▷ **Question 1:** Réaliser la carte CRC correspondant à la classe `tortue.Screen`.
- ▷ **Question 2:** Réaliser la carte CRC de la classe `tortue.Tortoise`, modélisant la tortue.
- ▷ **Question 3:** À partir de la carte CRC déterminer une interface publique possible pour la classe `tortue.Tortoise`.
- ▷ **Question 4:** Déterminer la carte CRC pour la classe principale dénommée `tortue.Main`. Sachant que l'application fonctionne de la manière suivante. Une fois exécutée, l'écran graphique apparaît et il est demandé à l'utilisateur de saisir une commande. Une fois la commande saisie, si celle-ci est valide, elle est exécutée et l'application demande à nouveau à l'utilisateur de saisir une commande.

Les commande possibles sont les suivantes :

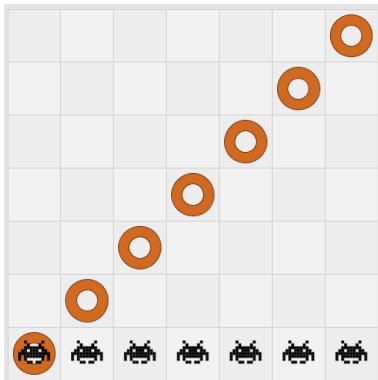
```

1 FD x      # pour faire avancer la tortue de x pixels
2 BD x      # pour faire reculer la tortue de x pixels
3 LT d      # pour faire tourner à gauche la tortue de d degrés
4 RT d      # pour faire tourner à droite la tortue de d degrés
5 PENUP     # pour lever le crayon
6 PENDOWN   # pour poser le crayon
7 CLEAR     # pour effacer l'écran
8 BC c      # pour choisir la couleur numéro c du crayon (0: blanc, 1: noir, etc)
9 EXIT      # pour quitter l'application

```

- ▷ **Question 5:** Déterminer une interface publique de la classe principale de l'application.
- ▷ **Question 6:** (*facultative*) Écrivez une petite application implémentant ces classes `Tortue.Main` et `Tortue.Tortoise` en supposant que la classe `Tortue.Screen` vous est fournie.

★ Exercice 2: Spécifier les buggles de la PLM






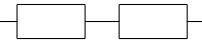
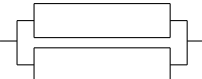
Les buggles habitent sur un monde en forme de grille. Tout comme les tortues, elles peuvent se déplacer sur ce monde (avancer, reculer, tourner d'un quart de tour à droite, tourner d'un quart de tour à gauche, lever une brosse, baisser une brosse, définir la couleur d'une brosse). Le monde des buggles peut quand à lui contenir des Baggles (les fameux biscuits tant appréciés par les buggles), mais également des murs qui empêchent les buggles d'avancer. Les buggles doivent donc être capables de savoir si elles sont faces à un mur ou non. De même elles peuvent savoir si elles sont au dessus d'un baggle, le prendre et le déposer.

- ▷ **Question 1:** Réaliser les différentes cartes CRC nécessaires à la conception de cette application. On réutilisera la carte CRC de la classe `tortue.Screen` vue dans l'exercice précédent.
- ▷ **Question 2:** Déterminer les interfaces publiques des classes correspondantes aux cartes CRC que vous avez réalisées.

2 Electric Dipoles

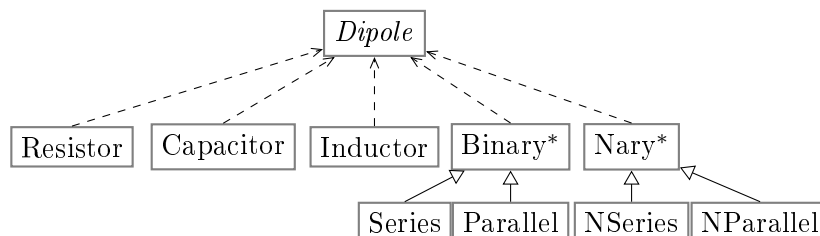
Every electric circuit is composed of differing components such as resistors, capacitors, diodes and electromagnetic coils. They can be assembled in either series or parallel circuits. Depending on their component, each circuit present a specific resistance to the current when a voltage is applied. The **impedance** extends this notion of resistance to alternating currents.

Given ω the angular frequency of the current, the impedance z of the circuit can be computed as follows (with the constant $i = 1\angle\frac{\pi}{2} = e^{j\frac{\pi}{2}}$).

Symbol	Description	Impedance
r in Ω 	A resistor of value r expressed in ohms (noted Ω)	$z = r$
l in H 	An inductor of value l expressed in henries (noted H)	$z = i(\omega * l)$
c in F 	A capacitor of value c expressed in farad (noted F)	$z = i(\frac{-1}{\omega * c})$
	A series circuit with 2 dipoles of impedance z_1 and z_2	$z = z_1 + z_2$
	A parallel circuit with 2 dipoles of impedance z_1 and z_2	$z = \frac{1}{\frac{1}{z_1} + \frac{1}{z_2}}$

2.1 Modeling Dipole

We will use the following class hierarchy to model the electric dipoles.



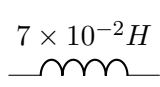
The *Dipole* trait provides an abstract method `impedance(omega:Double):Complex`, implemented in each concrete class of the hierarchy. The parameter `omega` represents the angular frequency of the current. The impedance is a complex number (only resistors have a real impedance).

2.2 Implementing Dipole

Download, unpack the [provided code](#) to your local disk and open it with your favorite editor (or **geany** if you have no favorite editor yet). The provided code contains a `build.sbt` file for easy compilation, as well as a set of unit tests checking the features that you should implement. You should run the tests often during your work² to track your progress. Of course, the tests for a given feature will fail until you implement that feature.

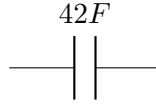
²Run the tests with the following command: `sbt test`

▷ **Question 3:** We will first implement the simple dipoles. Fill the classes `Resistor`, `Capacitor` and `Inductor` in file `src/main/scala/dipole/SimpleDipoles.scala` to pass the first set of tests. You have to implement both `impedance()` and `toString()`.



$$(z \approx 22j \, \Omega)$$

Tested Inductor.



$$(z \approx -7.6 \times 10^{-5}j \, \Omega)$$

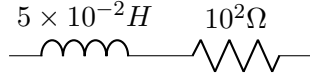
Tested Capacitor.



$$(z = 100 \, \Omega)$$

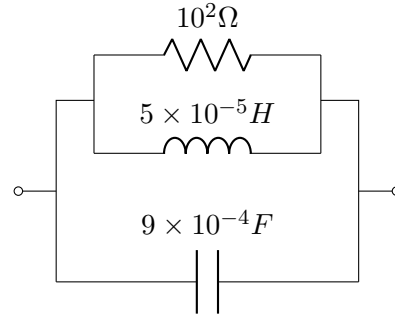
Tested Resistor.

▷ **Question 4:** We will now implement the binary circuits, either built in parallel or in series. Check your implementation with the provided tests, that use the following circuits.



$$(z \approx 100.0 + 15.70j \, \Omega)$$

Tested Series Circuit.



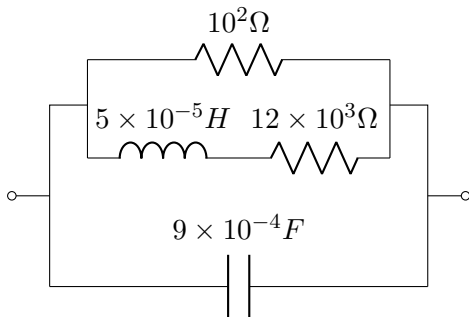
$$(z \approx 0.2079 + -4.55j \, \Omega)$$

Tested Parallel Circuit.

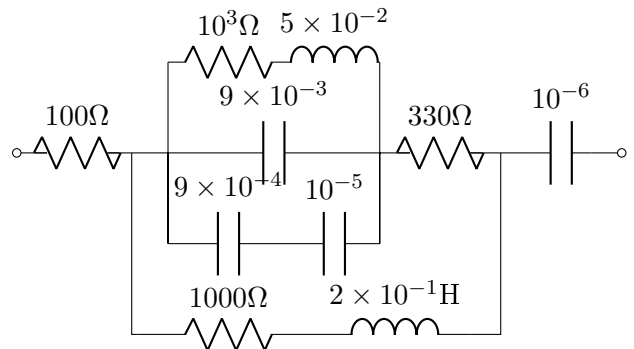
▷ **Question 5:** We will now implement N-ary circuits. The impedance of a series circuit is simply given by $\sum_{i=1}^n \omega_i$ while the impedance of a parallel circuit is given by $\frac{1}{\sum_{i=1}^n \frac{1}{\omega_i}}$.

Note that you also have to implement a `Nary.::(head:Dipole)` method, enabling to build a new circuit by appending a dipole to the currently defined one.

▷ **Question 6:** You will now fill the file `src/main/scala/dipole/Instances.scala` to define the instances of dipoles depicted below.



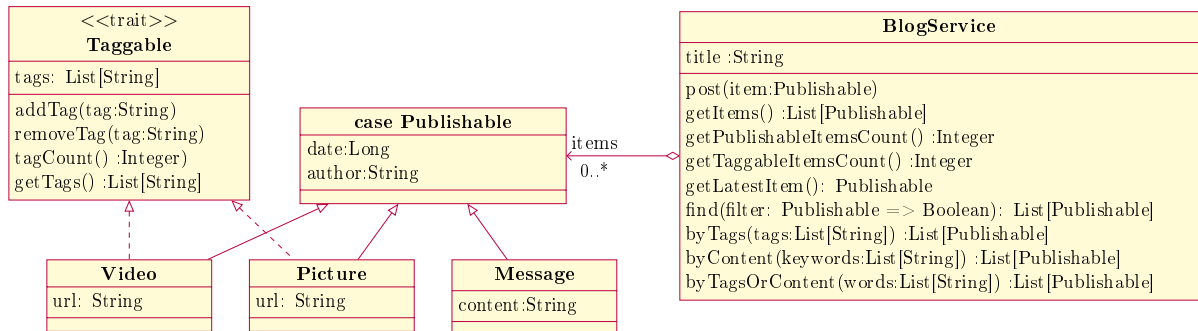
The dip1 dipole.



The dip2 dipole.

3 Blogging

We will now implement a blogging micro-system: a web site constituted of posts aggregated over time. These posts can be text messages, images or videos. *Tags* can be attached to images or videos, to select the ones that match a given set of keywords. The textual messages cannot be tagged, but the textual search should operate on their content directly.



▷ **Question 1:** Implement this hierarchy of classes, and test your work with an appropriate specification.