FICHE 4

LA STRUCTURE D'UN PROJET, PREMIERE APPROCHE

(Fiches Java)

1 Notions élémentaires sur la structure d'une classe

Chaque **classe** définit un ensemble d'**objets**. Chaque fois qu'on définit un objet de la classe en question on a une **instance** de la classe.

On trouve donc dans le code qui définit une classe à la fois les champs qui définissent les objets qu'on appelle les **variables d'instance**, et les **méthodes** qui s'appliquent à ces objets. Mais ce n'est pas tout, on y trouve aussi des **variables de classe**, **des méthodes de classe**, du **code d'initialisation**, des **variables locales** et mêmes des classes plongées.

Voyons comment tout ceci s'enchaîne.

```
/*On importe tout d'abord les packages et
les classes qui vont servir, par exemple: */

import java.io.*;

/*Comme cette classe va être utilisée par d'autres classes on
définit un package*/

package Complexes;

/*Le titre de la classe. Ici cette classe est
publique et peut donc être utilisée par une autre classe
qui lui est extérieure.*/

public class complex {
```

```
/*classes définies à l'intérieur de complex
pour les besoins futurs*/
/*Si on ne déclare pas ces classes statiques on a des
problèmes par la suite*/

  public static class zRe {
    public double valRe;
  }

  public static class zIm {
    public double valIm;
  }

  public static class zMod {
    public double valMod;
  }

  public static class zArg {
    public double valArg;
  }

  public static class complexException extends Exception {

    private static String[] tabledemessages = {
    "Données correctes",
    "Le module est négatif",
    "L'argument est exterieur à [0,2pi[",
    "Débordement"
    };

    public complexException(int numero) {
      super(tabledemessages[numero]);
    }
  }

/*Les variables d'instance. Ici on les a définies privées,
mais elles pourraient être publiques ou protected. Cependant
c'est une bonne programmation de n'acceder à ces champs que
par l'intermédiaire de méthodes qui vont contrôler
la cohérence des informations données dans ces champs.
Par exemple il n'est pas bon de modifier la partie réelle
```

```
sans modifier simultanéement le module
et l'argument*/

  private zRe Re;
  private zIm Im;
  private zMod module;
  private zArg argument;

/*variables de classe*/

  public static final complex N= new complex(0.0,0.0);
  public static final complex U= new complex(1.0,0.0);
  public static final complex I= new complex(0.0,1.0);

/************************************************/

/*methodes de classe*/

  public static zMod calcModule(zRe x, zIm y) {
    zMod rho=new zMod();
    if ((x.valRe==0) && (y.valIm==0)) rho.valMod=0;
    else if (Math.abs(x.valRe)>Math.abs(y.valIm)) rho.valMod=
          Math.abs(x.valRe)*Math.sqrt(1+(y.valIm/x.valRe)*(y.valIm/x.valRe));
         else rho.valMod=
          Math.abs(y.valIm)*Math.sqrt(1+(x.valRe/y.valIm)*(x.valRe/y.valIm));
    return rho;
  }


  public static zArg calcArgument(zRe x, zIm y) {
    zArg theta=new zArg();
    if ((x.valRe==0) && (y.valIm==0)) theta.valArg=0;
    else if ((x.valRe==0) && (y.valIm>0)) theta.valArg=Math.PI/2;
         else if ((x.valRe==0) && (y.valIm<0)) theta.valArg=3*Math.PI/2;
      else if (x.valRe>0) theta.valArg=((2*Math.PI)+
      (Math.atan(y.valIm/x.valRe))%(2*Math.PI))%(2*Math.PI);
           else theta.valArg=((2*Math.PI)+
      (Math.PI+Math.atan(y.valIm/x.valRe))%(2*Math.PI))%(2*Math.PI);
    return theta;
  }
```

```
  public static zRe calcRe(zMod rho, zArg theta) {
    zRe x=new zRe();
    x.valRe=rho.valMod*Math.cos(theta.valArg);
    return x;
  }

  public static zIm calcIm(zMod rho, zArg theta) {
    zIm y=new zIm();
    y.valIm=rho.valMod*Math.sin(theta.valArg);
    return y;
  }

/*constructeurs*/

  public complex(){
  }

  public complex(zRe x, zIm y) {
    Re=x;
    Im=y;
    module=calcModule(x,y);
    argument=calcArgument(x,y);
  }

  public complex(zMod rho, zArg theta)
    throws complexException {

    if (rho.valMod<0) throw new complexException(1);
    if ((theta.valArg<0) || (theta.valArg>=2*Math.PI))
      throw new complexException(2);
    Re=calcRe(rho,theta);
    Im=calcIm(rho,theta);
    module=rho;
    argument=theta;
  }

  public complex(double u,double v) {
    zRe x=new zRe();
    x.valRe=u;
    zIm y=new zIm();
    y.valIm=v;
```

```
    Re=x;
    Im=y;
    module=calcModule(x,y);
    argument=calcArgument(x,y);
  }

  public complex(double u,double v,boolean b)
    throws complexException {
    if (b )
      {
        zRe x=new zRe();
x.valRe=u;
zIm y=new zIm();
y.valIm=v;
Re=x;
Im=y;
module=calcModule(x,y);
        argument=calcArgument(x,y);
      }
    else
      {
        if (u<0) throw new complexException(1);
        if ((v<0) || (v>=2*Math.PI))
          throw new complexException(2);
        zMod rho=new zMod();
rho.valMod=u;
zArg theta=new zArg();
theta.valArg=v;
module=rho;
        argument=theta;
Re=calcRe(rho,theta);
        Im=calcIm(rho,theta);
      }
  }


/*methodes d'instances*/

  public zRe getRpart() {
    return Re;
  }
```

```
  public zIm getIpart() {
    return Im;
  }

  public zMod getMpart() {
    return module;
  }

  public zArg getApart() {
    return argument;
  }

/*méthodes de classe*/

  public static complex addition(complex z1,complex z2)
    throws complexException {
    complex z=
    new complex(z1.Re.valRe+z2.Re.valRe,z1.Im.valIm+z2.Im.valIm,true);
    return z;
  }

  public static complex multiplication(complex z1,complex z2)
    throws complexException {
    complex z=new complex(z1.module.valMod*z2.module.valMod,
    z1.argument.valArg+z2.argument.valArg,false);
    return z;
  }

  public static complex oppose(complex z1)
    throws complexException {
    complex z=new complex(-z1.Re.valRe,-z1.Im.valIm,true);
    return z;
  }

  public static complex inverse(complex z1)
    throws complexException {
    complex z=new complex(1/z1.module.valMod,-z1.argument.valArg,false);
    return z;
  }
```

```
  public static complex soustraction(complex z1,complex z2)
    throws complexException {
    complex z=
    new complex(z1.Re.valRe-z2.Re.valRe,z1.Im.valIm-z2.Im.valIm,true);
    return z;
  }

  public static complex division(complex z1,complex z2)
    throws complexException {
    complex z=new complex(z1.module.valMod/z2.module.valMod,
    z1.argument.valArg-z2.argument.valArg,false);
    return z;
  }
}
```

Remarquons que nous avons défini par exemple la fonction addition comme étant statique. Elle prend en entrée 2 nombres complexes et donne en sortie leur somme. On aurait pu choisir de travailler avec une méthode d'instance ajouterA qui aurait pris en paramètre d'entrée un nombre complexe et l'aurait ajouté à l'instance courante.

Voici aussi un exemple de code d'initialisation d'une classe. Cet exemple est un extrait d'un analyseur syntaxique. On remarquera en passant la déclaration d'une variable locale à la boucle for.

```
static{
    tableconst[0]="E";
    tableconst[1]="PI";
    freels=null;
    for (short i=TAILLE_TABLE_FONCT-1; i>=0; i--){
      Allocation P1=new Allocation();
      P1.place=i;
      P1.suivant=freels;
      freels=P1;
    }
    insere_nouvelle_fonction_1("ABS");
    insere_nouvelle_fonction_1("ACH");
    insere_nouvelle_fonction_1("ACOS");
    insere_nouvelle_fonction_1("ASH");
    insere_nouvelle_fonction_1("ASIN");
    insere_nouvelle_fonction_1("ATAN");
```

```
    insere_nouvelle_fonction_1("ATH");
    insere_nouvelle_fonction_1("CH");
    insere_nouvelle_fonction_1("COS");
    insere_nouvelle_fonction_1("EXP");
    insere_nouvelle_fonction_1("FRAC");
    insere_nouvelle_fonction_1("INT");
    insere_nouvelle_fonction_1("LOG");
    insere_nouvelle_fonction_1("LN");
    insere_nouvelle_fonction_1("SH");
    insere_nouvelle_fonction_1("SIN");
    insere_nouvelle_fonction_1("SQR");
    insere_nouvelle_fonction_1("SQRT");
    insere_nouvelle_fonction_1("TG");
    insere_nouvelle_fonction_1("TH");
    insere_nouvelle_fonction_1("CONJ");
    insere_nouvelle_fonction_1("RE");
    insere_nouvelle_fonction_1("IM");
    insere_nouvelle_fonction_1("ARG");
    insere_nouvelle_fonction_2("MAX");
    insere_nouvelle_fonction_2("MIN");
  }
```

2 Cas particulier d'une classe exécutable

Une classe exécutable doit avoir une méthode **main** qui est le point d'entrée du programme. A titre d'exemple voici une classe qui se sert du package "Complexes" et qui affiche le module d'un nombre complexe entré par son module et son argument.

```
import Complexes.*;

public class calComp

{
  public static void main(String[] args)
    throws complex.complexException
    {
      complex z=new complex(1.0,1.05,false);
      System.out.println(z.getMpart().valMod);
```

```
    }
}
```

3 Cas particulier d'une applet

Une applet doit être écrite suivant le modèle suivant :

```
import java.awt.Graphics;
import java.awt.Color;
import java.applet.Applet;

public class UneApplet extends Applet{

  String text = "Une Applet";

  public void init()
    {
      setBackground(Color.cyan);
    }

  public void start()
    {
      System.out.println("starting ...");
    }

  public void stop()
    {
      System.out.println("stopping ...");
    }

  public void destroy()
    {
      System.out.println("preparing to unload ...");
    }

  public void paint(Graphics g)
    {
      System.out.println("Paint");
      g.setColor(Color.blue);
      g.drawRect(0,0,getSize().width-1,getSize().height-1);
```

```
      g.setColor(Color.red);
      g.drawString(text,40,50);
    }
}
```

Elle est appelé par le code HTML suivant :

```
<HTML>
<BODY>
<APPLET CODE=SimpleApplet.class WIDTH=200 HEIGHT=100>
</APPLET>
</BODY>
</HTML>
```

4 L'organisation d'un projet

Lors du traitement d'un projet on va être amené à élaborer un ou plusieurs **packages**. Il est important de bien réfléchir au **découpage en packages** du projet au **découpage en classes** de chaque package. Il faut ensuite déterminer pour chaque classe, chaque objet, ce qui est **public**, **protected**, **private** ou par défaut. Il faut penser à ce qu'on veut pouvoir laisser comme latitude à l'utilisateur et aussi au développeur qui serait amener à écrire des extensions de classes.