PLAN

A/ Quelques concepts de POO .
- a/ - L'objet caractérisé par composantes : un état (valeur), un comportement, une identité (
- b/ - Premier exemple : l'objet "rectangle"
- c/ - Le cycle de vie d'un objet
- d/ - Type abstrait et interface
- e/ - Implémentation d'un type abstrait en Java

B/ Heritage et polymorphisme
- / - Heritage simple et héritage multiple
- / - Multi_typage et interfaces
- / - Redefinition et surcharge des méthodes
- / - Le polymorphisme
- / - Variables de classe et héritage

REMARQUE. J'ai concocté ce qui suit à partir de ce livre qu'on peut trouver en ligne :

http : // ica developpez com/ fichiers/ java/ manuelpoo pdf

Dans ce qui suit, on "survole" quelques notions dont nos étudiants pourraient avoir besoin . On n'a pas vraiment le temps physique d'approfondir ces notions en quelques heures .

A/ Quelques concepts de POO

Un objet peut etre considéré comme "une structure de données obéissant à un ensemble de regles et régie par un ensemble d'opérations" .

b/ Premier exemple : l'objet "rectangle"

On peut par exemple considérer un rectangle comme étant la donnée de son point (x,y) le plus haut à gauche, sa hauteur et sa largeur.

Un objet rectangle r_1 a donc un certain état :
<x=2, y= , largeur = 20, hauteur = 0>.

r_1 a un comportement et un ensemble d'opérations qui permettent d'agir sur cet objet. Notamment, sa création

r_1 = new Rectangle(2, , 20, 0);
r_1.déplacer(5,6) par exemple pour déplacer x de 5 et y de 6. r_1 vaut alors (7,9,20, 0);
r_1.agrandir(,) permet par exemple d'agrandir la hauteur de et la largeur de .
largeur = r_1.getLargeur() permet de récupérer la largeur de l'objet.

r_1 a une identité indépendamment de ses caractéristiques qui peuvent être modifiées (ex : déplacer, agrandir).

r_1 = new Rectangle(2, , 20, 0); $r_2 = r_1$; r_1 = new Rectangle(, , ,);

c/ Cycle de vie d'un objet

La vie d'un objet peut être caractérisée par un diagramme d'états et de transitions, qui présentera la création de l'objet, sa destruction, ainsi que les différents états caractérisés chacun par un comportement spécifique de l'objet.

Nous avons vu les piles (LIFO). Un tel objet comporte les méthodes suivantes :
- empiler(unElement)
- depiler() (renvoie l'élément au sommet de la pile)
- estVide() (test si la pile est vide)
- estPleine() (si la pile est limitée en taille, on doit pouvoir tester)

Nous avons alors le diagramme suivant :

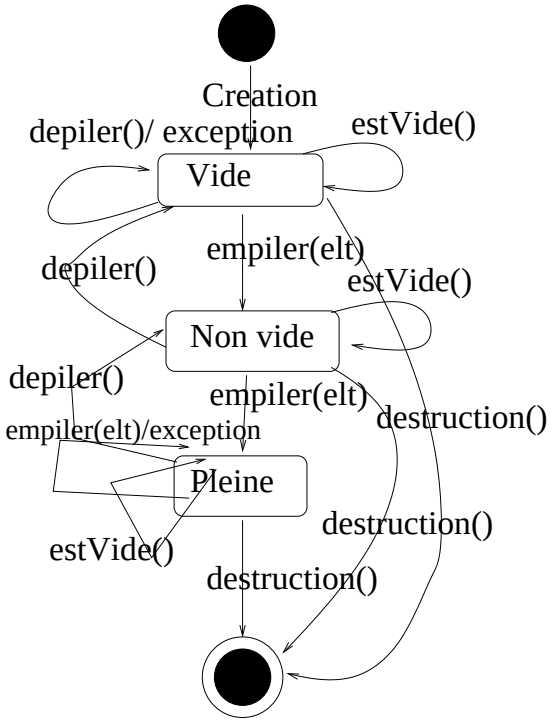
d/ Type abstrait

Un type abstrait définit une structure de données par un ensemble d'opérations applicables aux objets de ce type et par une description de ces opérations.

Prenons l'exemple du type abstrait "Rectangle". Un objet du type "Rectangle" sera caractérisé par informations : les coordonnées x et y du point le plus haut à gauche, sa largeur et sa hauteur. Ces informations seront variables et leur valeur pourra être modifiée. Prises ensemble, ces informations constituent une structure de données.

Un type abstrait est défini par un "ensemble d'opérations"

- Les constructeurs. Ces opérations sont exécutées au moment de la création de l'objet. Elles servent à donner un état initial à l'objet : une valeur initiale aux différentes variables de la structure de données. Typiquement, on rencontrera espèces de construc- teurs : le



constructeur sans parametre, qui donnera une valeur par defaut à l'objet, un ou plusieurs constructeurs avec parametres et enfin un constructeur de copie qui cree un objet en copiant la valeur d'un objet passé en parametre .

- Les modificateurs et interrograteurs .Ecrits souvent sous la forme setXxx(.), ces operations changent l'etat de l'objet .

- Les accesseurs .Ecrits souvent sous la forme getXxx(.), ces operations permettent "d'interroger l'objet et retournent des informations concernant l'etat de l'objet .

Pour le type abstrait "Rectangle", nous avons l'ensemble d'operations suivantes .

Opération	Catégorie	Notes
new Rectangle()	( ) Constructeur	Création d'un nouveau rectangle avec une valeur par défaut <0,0,0,0>
new Rectangle(x, y, [illegible])	( ) Constructeur	Création d'un nouveau rectangle avec les valeurs spécifiées
agrandir(δ,)	(x) Modificateur	
deplacer(,)	(x) Modificateur	
setLargeur(x)	(x) Modificateur	
setHauteur(x0)	(x) Modificateur	
setX(0)	(x) Modificateur	
setY(x)	(x) Modificateur	
getLargeur()	( ) Accesseur	
getHauteur()	( ) Accesseur	
getX()		
getY()		
contientLePoint(δ,6)	( ) Interrogateur	

Une operation est donc definie par son nom, la liste de ses parametres, le type de valeur retournée par l'operation .

e/ Implémentation d'un type abstrait en Java

Un objet concrétise un "type abstrait" .On peut définir à partir du type abstrait "Rectangle" précédent l'interface suivant .

interface Rectangle$_I$

Modificateurs public void agrandir(int deltaL, int deltaH) ; public void deplacer (int deltaX, int deltaY) ;

Observateurs public int getX() ; public int getY() ; public int getLargeur() ; public int getHauteur() ; public boolean contientLePoint(int px, int py) ;

Une interface décrit la liste des méthodes mises à disposition par le type abstrait .Elle spécifie les noms et la maniere d'utiliser ces méthodes .

Une fois definie, une interface pourra etre utilisée pour déclarer des variables ou plus généralement des parametres de méthodes qui désigneront des objets susceptibles de répondre aux opé- rations spécifiées dans l'interface .

Par exemple .

```
Rectangle_I r; // ‘‘r’’ est une variable.
  La variable ‘‘r’’ sera utilisée pour désigner un objet de type
  ‘‘Rectangle_I’’.
  A ce stade, il n’y a pas eu de création d’objet: cette variable vaut
  ‘‘null’’, elle ne désigne pour l’instant aucun objet.
:
:
public void uneMethode (Rectangle_I r) {
```

[illegible]

```
  De manière générale, les interfaces sont utilisées pour déclarer des par-
  amètres de méthodes. Dans le cadre de la procédure ‘‘uneMethode’’,
  ‘‘r’’ désignera un objet répondant exclusivement aux messages spécifiés
  dans l’interface ‘‘Rectangle_I’’
      /* code de la méthode ‘‘uneMethode’’ */
}
```

On remarquera qu'une interface ne donne aucune information au sujet des constructeurs potentiels .

Une interface ne permet pas de creer un objet .En e et, la creation de l'objet dependra directement de la maniere dont sera "implementee l'interface et notamment de la struc- ture de donnees mise en oeuvre, il n'est donc pas prevu de constructeur à ce stade .

Notamment, lors de la creation d'un objet, le systeme devra lui reserver une zone memoire, qui contiendra sa valeur .La taille de cette zone memoire et sa valeur depend bien entendu de la maniere dont sera implementee la structure de donnees .

Pour creer l'objet, il faudra realiser l'implementation de l'interface au moyen d'une classe .Ce que nous verrons dans le point suivant .Et quand l'objet aura ete cree, il sera possible de lui envoyer l'un ou l'autre des messa- ges specifies dans l'interface .Comme par exemple :

```
r.déplacer (4, 5);
r.agrandir (4);
int unEntier; // Déclaration d'une variable de type entier
unEntier= r.getLargeur();                // ‘‘unEntier’’ vaut la largeur
                                         // actuelle du rectangle
```

Implementation du type abstrait Rectangle en Java .L'interface decrit precedemment per- met de declarer un objet mais pas de le creer .Pour creer un objet du type "Rectangle$_I$", il nous faut maintenant realiser le type abstrait .Le concretiser .En informatique, on parle d'implementation .

```
class Rectangle implements Rectangle_I{
// Variables d'instance
     private   int x;      // Point supérieur gauche
     private   int y;
     private   int lg;     // Largeur et Hauteur
     private   int ht;
// Constructeur (s)
     public Rectangle () {
           x = y = 0;
           lg = ht = 0;
     }
     public Rectangle (Rectangle r) {
           x = r.x; y = r.y; lg = r.lg; ht = r.ht;
                     this (r.x, r.y, r.lg, r.ht);
```

```
            // ou:
            //         utilisation du constructeur avec paramètres
      }
      public Rectangle(int pX, int pY, int largeur,int hauteur) {
            x = pX; y = pY;
            lg = largeur; ht = hauteur;
      }
// Modificateurs
      public void agrandir(int deltaL, int deltaH) {
            lg = lg + deltaL;
            ht = ht + deltaH;
      }
      public void déplacer (int deltaX, int deltaY) {
            x = x + deltaX;
            y = y + deltaY;
      }
      public  void  setX(int x) {this.x = x;}
      public  void  setY(int y) {this.y = y;}
      public  void  setLargeur(int lg) {this.lg = lg;}
      public  void  setHauteur(int ht) {this.ht = ht;}
// Observateurs
      public  int  getX () {return x;}
      public  int  getY () {return y;}
      public  int  getLargeur() {return lg;}
      public  int  getHauteur () {return ht;}
      public boolean contientPoint (int pX, int pY) {
            return    (pX > x) && (pX < x+lg) && (pY > y) &&
                      (pY < y+ht);
      }
}
```

Nous pouvons maintenant créer véritablement un objet de type "Rectangle$_I$, la classe "Rectangle a prévu à cet e et trois constructeurs :

```
Programme: Création d'objets
       Rectangle$_I$ r1, r2, r3;
             Déclaration de 3 variables
       r1 = new Rectangle ();
             Création de l'objet en utilisant le constructeur sans paramètre: la lar-
             geur, la hauteur ainsi que les coordonnées de son point supérieur gau-
             che valent respectivement <0, 0, 0, 0>
       r2 = new Rectangle (3, 4, 20, 10);
             Création de l'objet en utilisant le constructeur avec paramètres
       r3 = new Rectangle (r2);
             Création de l'objet en utilisant le constructeur de copie: la largeur, la
             hauteur ainsi que les coordonnées de son point supérieur gauche valent
```

```
respectivement <3, 4, 20, 10>
```

Structure d'une classe Java La déclaration d'une classe obéit à la syntaxe suivante :

```
[Modificateurs de classe] class NomDeLaClasse
                        [extends nomDeLaSuperclasse]
                         [implements interface1, interface2,..]
{
      [Variables]
      [Constructeurs]
      [Méthodes]
}
```

La notion d'interface est un concept spécifique à Java, et vient palier au fait que Java, contrairement à C++, ne connaît pas l'héritage multiple. En revanche, la notion d'interface offre à Java la possibilité d'opérer le typage multiple : le type d'une variable peut être multiple !

Le typage multiple

```
interface Rectangle-I {

}
interface ObjetGraphique {
     public void dessiner();
}
class RectangleA implements Rectangle_I {
     // implémentation de l'interface Rectrangle_I
          :
           ...
          :
          :
     // méthodes propres à la classe RectangleA
          public void dessiner() {
                /* instructions..*/
          }
          public void effacer() {
                /* instructions.. */
          }
}
class RectangleB {
// Pour des rectangle de taille et de position non modifiables
     private int x, y, lg, ht;
     public RectangleB (int x, int y, int lg, int ht) {
           this.x = x; this.y = y; this.lg = lg; this.ht = ht;
     }
```

```
     public  int  getX() {return x;}
     public  int  getY() {return y;}
     public  int  getLargeur() {return lg;}
     public  int  getHauteur() {return ht;}
}
class Fenetre implements Rectangle_I, ObjetGraphique {
// Implémentation de l'interface Rectangle_I
     :
           Implémentation de toutes les méthodes de l'interface
     :
     :
// Implémentation de l'interface ObjetGraphique
     public void dessiner() { /* instructions */ }
```

```
class Jeu implements Configuration {
      :
      public void run {
            while (NoEssai <= NOMBRE_MAX_ESSAIS) {
                  :
                  :
            }
      }
}
```

B/ Héritage et polymorphisme

1/ Héritage simple et héritage multiple

Une classe utilisera l'héritage pour reprendre à son compte toutes les proprietes qui ont ete definies dans une autre classe .C'est ce que l'on appelle l'héritage simple, mécanis- me propose en Java .D'autres langages comme C++ proposent l'héritage multiple, qui permet de reprendre les proprietes de plusieurs classes à la fois .

Considérons les cas suivant d'héritages

En JAVA (héritage simple)

classe Singe
+ manger()
+ grimperAuxArbres()

classe Homme // hérite de Singe
+ manger()

classe Etudiant // hérite d'Homme

En C++ (heritage multiple)

```
Homme                          Singe
+ manger()                   + manger()
                             + grimperAuxArbres()
```

Etudiant // hérite à la fois de Singe et d'Homme

Dans la version "héritage simple , la classe Homme redefinit l'implementation de la me- thode mangerBanane() (l'homme, en e et, a des manieres que le singe ne conna t pas) . L'héritage multiple, si puissant qu'il soit, a le grand défaut de présenter certaines ambigu -

tés : un étudiant mange à la maniere d'un singe ou à la maniere d'un homme ? le program_ meur devra lever l'ambigu té en specifiant l'origine de la méthode mangerBanane() . Dans la plupart des cas, l'ambigu té est simple à lever . Toutefois il peut arriver que le graphe d'héritage devienne fort complexe à force d'héritage multiple et di cile à apprehender . Dans le cas de figure precedent, il pourrait arriver que la classe Homme hérite d'autres classes dont certaines héritent elles_meme de la classe Singe .

Pourtant, grâce à l'héritage multiple, il est possible de déclarer des variables qui seront caracterisées par un typage multiple . Ainsi, un étudiant pourra dans certaines parties du programme etre considéré comme un objet de type Singe, lorsqu'il s'agira par exem_ ple de le faire grimper aux arbres . Dans d'autres parties du programme, on le manipu_ lera comme un objet de type Homme, quand on lui demandera par exemple de payer sa cotisation . Java résout ce probleme par le biais du concept d'interface : une classe Y peut hériter d'une seule classe, la classe X, mais peut par contre implementer plusieurs interfaces, comme par exemple les interfaces A et B . Un objet de la classe X possede ainsi plu_ sieurs types : X, Y, A et B .

2/ Multi-typage et interfaces

Dans l'exemple suivant, la classe Etudiant hérite de la classe Homme et implemente les deux interfaces Singe et MangeurDeMacDo . Un étudiant obéit ainsi aux définitions d'au moins [redacted]es di erents : Etudiant, Homme, Singe et MangeurDeMacDo . C'est ce que l'on appelle le typage multiple, à défaut d'héritage multiple .

Etudiant // hérite d'Homme et implement deux interfaces
+ grimperAuxArbres() ;
+ mangerUnMacDo() ;

«Interface Singe
_ mangerBanane() ;
_ grimperAuxArbres() ;

«Interface MangeurDeMacDo
_ mangerUnMacDo() ;

Héritage simple en Java

Pour présenter brievement ce mécanisme, supposons que le programmeur ait défini un lien d'héritage entre deux classes "A et "B en écrivant que la classe "B hérite de la classe "A .

```
class A {
      /* code de la classe A */
};
```

```
class B extends A {
      // La classe B ‘‘ hérite ’’ des propriétés de la
      // classe A et rajoute son propre code
      /* code rajouté par la classe B */
};
```

Si “B hérite de “A : tous les membres qui ont été définis dans la classe “A devien_ nent des membres à part entière de la classe “B , et toute modification ultérieure de “A sera automatiquement reportée sur “B : il s'agit en fait d'un “ super COPIER_ COLLER , un copier_coller dont la mise à jour est automatique .On dira que la classe “B est une classe dérivée de “A ou encore une sousclasse de “A , et que “A est la superclasse de “B

De manière plus précise, on peut dire que la classe “B , par héritage, partage toutes les propriétés de la classe “A , à savoir : sa structure et son comportement .Héritage de la structure La sousclasse hérite des variables d'instance et des variables de classe de sa superclasse .Elle a la possibilité de rajouter des variables qui seront propres à son usage personnel .Par contre, elle ne pourra pas en supprimer : l'héritage doit être accepté dans son ensemble ! Héritage du comportement La sousclasse hérite des méthodes d'instance et des méthodes de classe qui ont été définies au niveau de sa superclasse .Ici encore, la sousclasse a la possibilité de rajouter de nouvelles méthodes .Elle pourra par ailleurs redéfinir les méthodes dont elle hérite dans le but d'adapter leur comportement aux spécificités de la nouvelle classe .

Héritage : quel intérêt ? On peut citer au moins avantages à retirer de l'exploitation du concept d'héritage : une meilleure structuration du programme, une réutilisation potentielle du code, et enfin une économie de la mémoire .Une meilleure structuration du programme Obtenue par une mise en évidence des parties générales (superclasses) et des parties spé_ cialisées (sousclasses) .Cette amélioration est bénéfique du point de vue de la mise au point des programme et de leur modification future (maintenance), et donc la qualité du logiciel en général .La réutilisation de code Placé sous l'angle de la réutilisation de code, l'héritage permet d'économiser beaucoup d'écriture : les sousclasses réutilisent le code qui a été écrit _ une fois seulement _ au ni_ veau des méthodes de la classe de base .Par la suite, toute modification apportée aux méthodes de la classe de base sera automatiquement propagée au niveau des sousclas_ ses ! L'héritage et l'utilisation de la mémoire L'héritage permet de gagner de la place en mémoire au niveau des méthodes : les métho_ des héritées ne seront pas dupliquées, leur code sera placé au niveau de la classe de base .Par contre, on ne gagne rien aux niveau des variables d'instances : ces dernières sont du_ pliquées .Un objet occupe en mémoire toute la place qui est nécessaire pour stocker ses propres variables d'instance, mais aussi toutes les variables d'instance dont il hérite .

Considérons l'exemple suivant :

```
class Animal      {
      private    String nom;            // Variable: nom de l'animal
      :
}
class Oiseau      extends Animal {
      private    boolean peutVoler;
```

```
        public int m(int x) {
        // Surcharge: signature différente (paramètres différents)
               System.out.println("surcharge");
               return 0;
        }
        public int m() {
        // Redéfinition: en-t\^ete quasiment identique
               System.out.println("redéfinition");
               return 0;
        }
        /*
        public float m() {
               /* Interdit: ambigu\"{\i}té avec la méthode précédente
               Voici un exemple d'ambigu\"{\i}té:
               B unObjet = new B();
               float f = unObjet.m();
                      Ambigu\"{\i}té: cette méthode comme la précédente
                      peuvent \^etre invoquées !
               */
               System.out.println("interdit");
               return 0;
        }
        */
        public String p(int x) {
        // Surcharge: signature différente (paramètres différents)
               System.out.println("surcharge");
               return "";
        }
        /*
        public Point p() {
               // Interdit: ambigu\"{\i}té avec la méthode définie dans A
               System.out.println("interdit");
               return new Point(0, 0);
        }
        */
}
```

4/ Le polymorphisme

Le polymorphisme est un concept qui découle de la possibilité donnée au programmeur de redéfinir des méthodes et qui s'appuie sur le mécanisme d'édition de liens dynamique. Le polymorphisme est un concept fondamental de la programmation objet ; son application permet d'écrire des blocs d'instructions indépendants du type des objets manipulés.

Le polymorphisme s'applique principalement aux parametres des méthodes. C'est en effet au niveau des parametres de méthode que nous exploiterons au mieux le polymor-

phisme, comme par exemple :

```
public void m (X unParamètre) {
      /* instructions */
}
```

Cette méthode admet un parametre de type “X .Au moment de l'execution de cette me_ thode, le parametre e ectif pourra prendre n'importe qu'elle forme, pour autant qu'il soit de type “X , à savoir :

.Si “X est une classe : le parametre e ectif peut etre soit une instance de la classe “X , soit une instance de n'importe qu'elle sousclasse de “X .
2.Si “X est une interface : le parametre e ectif peut etre l'instance de toute classe qui implémente l'interface “X .

Prenons comme exemple la classe Vector de la librairie Java .Cette classe met en œuvre les listes dynamiques de Java .En consultant cette classe, vous constaterez que les méthodes de cette classe admettent des arguments de type Object qui peuvent prendre n'importe quelle forme à l'execution : public void addElement(Object obj) ; Ainsi, la classe Vector est une classe susceptible de contenir n'importe quel type d'élé_ ment : en e et, l'argument de type Object admet des parametres de n'importe quelle forme (rappelons que la classe Object est la superclasse de toutes les classes de Java) .Cette généricité est une exploitation typique du concept de polymorphisme .

Le polymorphisme est tres interessant dans la mesure où il permet de realiser des méthodes générales : méthodes dont le type courant des parametres n'est pas imposé, mais connu uniquement à l'execution .La possibilite de mettre en œuvre des méthodes générales permet par ricochet de créer des classes plus générales, en favorisant ainsi la réutilisation du code .Java et Smalltalk permettent par exemple de realiser une classe Pile, dans laquelle il sera possible d'empiler n'importe quel type d'élement .Il su rait de definir une méthode Empiler avec un argument de type Object, qui admettra des parametres de n'importe quelle nature .

5/ Variables de classe et héritage

En Java, une variable de classe peut etre manipulee par les sous_classes comme si cette variable était déclarée dans les sous_classes elles_memes .Mais il n'y a pas de duplication de la variable .Il ne s'agit donc pas d'un héritage à proprement parler .

Par exemple, considérons la hierarchie Animal -> Oiseau -> Perroquet .

On peut imaginer declarer une variable de classe population au niveau meme de la classe Animal .Cet variable, de type int, permettrait de compter le nombres d'ani_ maux crées par instanciation :

```
class Animal {
       public static int population;// Variable de classe
```

```
        Animal () {
        // constructeur
                population++;            // Variable incrémentée
                                         // à chaque création d'objet
                :
        }
        :
}
class Oiseau extends Animal {
        :
        :
        if (population > 10)...
                // Manipulation de la variable
                // de classe définie dans Animal
}
class Perroquet extends Oiseau {
        :
}
```

La variable population, déclarée au niveau de la classe Animal n'existe qu'en un seul exemplaire ! Elle n'est pas dupliquée dans Oiseau, ni dans la classe Perroquet. Déclarée public dans la classe Animal, elle est par contre visible tant au niveau de la classe Oiseau qu'au niveau de la classe Perroquet.

A l'instar des variables de classe, les méthodes de classe peuvent etre manipulée par les sousclasses comme si ces méthodes etait déclarées dans les sousclasses elles_memes. On peut parler ici d'héritage.