01234567890123456789012345

Université Paul Cézanne - Licence de math-info 2e année

I4 : Programmation objet - Notes de cours

Ce module est axé sur la *programmation objet*. Chaque notion de programmation objet sera introduite en la justifiant par des motifs relatifs au *génie logiciel* et sera illustrée en *Java*. La programmation objet répond à des principes de génie logiciel, qui traite des méthodologies à adopter lorsqu'on élabore un logiciel. Le *langage Java* est étudié dans le seul but d'illustrer la programmation objet mais l'apprentissage complet de ce langage ne constitue pas l'objectif du cours.

1 Notion d'objet

Jusqu'à présent, vous avez programmé de manière *impérative* en utilisant des langages *procéduraux* (C, Pascal, etc). Cette manière de programmer est celle qui repose sur ce qui se déroule concrétement lorsqu'un programme est exécuté en langage machine par le processeur : une suite d'instructions est exécutée en séquence et modifie de proche en proche l'état global de la mémoire. La mémoire est une liste d'octets et une instruction va modifier un octet (ou un petit nombre fixe d'octets) de la mémoire. Les langages impératifs dits évolués s'abstraient quelque peu de ce schéma en définissant les notions de *structure de données* (qui regroupe des octets) et de *procédure* ou *fonction* (qui regroupe des instructions) mais gardent cette idée fondamentale de séquence d'instructions modifiant un état global. Dans les langages impératifs que vous connaissez, les déclarations des structures de données et les déclarations de fonctions restent formellement séparées dans le programme même quand des structures de données sont associées à des groupes de fonctions qui les modifient. Ex : les fonctions d'opération sur les listes chaînées qui vont servir à modifier la variable qui mémorise l'adresse du premier chaînon. Or, plutôt que de décomposer un programme en deux parties, une qui déclare les structures de données et une qui définit les fonctions associées, la programmation objet décompose le programme en *classes* regroupant chaque structure de données avec les fonctions qui la modifient. Les instances de ces classes sont appelés des *objets*. Un programme sera alors vu comme **un ensemble de composants autonomes (les objets) qui interagissent entre eux pour faire émerger un résultat global**.

Un objet se définit par :

- ses *attributs* (= variables) qui caractérisent son état qui varie au cours du temps pendant l'exécution du programme,
- ses *méthodes* (= fonctions et procédures) qui déterminent son comportement. Les méthodes d'un objet ne modifient *directement* que ses attributs.

Les objets interagissent entre eux par l'*envoi de messages*. Le message qu'envoie un premier objet à un deuxième objet consiste en l'appel d'une méthode du deuxième objet. Le deuxième objet va alors effectuer une certaine tâche et, le cas échéant, retourner un résultat au premier objet. Un message permet donc à un objet de modifier *indirectement* l'état d'un autre objet. Une méthode peut elle-même envoyer des messages à d'autres objets et les résultats qui lui sont retournés lui permettent de changer l'état de l'objet auquel elle appartient et de produire elle-même un résultat.

L'exécution d'un programme séquentiel produit donc une succession d'envois de messages qui modifient de proche en proche les états des objets tandis que des résultats partiels du problème général sont transmis en retour d'objet en objet. Mais l'autonomie des objets peut facilement permettre de définir des programmes distribués où les objets s'exécutent et s'envoient des messages en parallèle.

Pourquoi des objets ?

La modélisation d'un problème sous la forme d'objets en interaction est une façon naturelle de représenter de nombreux problèmes complexes (organismes, organisations d'individus, machines réalisées par un assemblage de composants, ...). Exemple métaphorique : le système immunitaire d'un organisme vivant (objets : cellules, messages : hormones).

Par ailleurs, l'autonomie relative des objets est une propriété intéressante du point de vue du génie logiciel. Le développement d'applications logicielles conséquentes réunit un groupe de personnes qui doivent travailler en même temps. Il faut donc décomposer les taches en modules les plus indépendants possible les uns des autres. Une fois que chaque responsable de module connait l'interface (= la liste des méthodes qu'il met à disposition des autres et leurs spécifications) des modules avec lesquels il sera en relation, ce responsable pourra écrire tout seul son propre module sans se préoccuper de la façon dont les autres modules seront écrits. Par ailleurs, la modularité permet une grande stabilité du code. On peut tester un module et changer ses aspects internes sans toucher aux autres modules. Si on étend un programme en rajoutant des modules, on ne modifiera que les anciens modules qui sont en relation avec les nouveaux modules. Cette organisation du code est déjà mise en oeuvre dans la programmation impérative par le découpage d'une application en multiples fichiers et en bibliothèques de types et de fonctions. Les langages à objets "officialisent" cette façon de concevoir une application en l'intégrant formellement dans le langage.

La conception de programmes en programmation objet diffère de celle de la programmation procédurale. En programmation procédurale, on a affaire à une imbrication de modules fonctionnelles qui accèdent collectivement à un ensemble de données. On pense en termes de décomposition de tâches en sous-tâches. En programmation objet, on identifie les acteurs (= les unités comportementales) du problème puis on détermine la façon dont ils doivent interagir pour que le problème soit résolu.

2 Aspects impératifs de Java

Avant de voir comment Java permet de faire de la programmation objet, nous allons voir ses aspects impératifs. Sur ce point, Java est très proche du langage C. Par défaut, il faut considérer que Java et C sont identiques. On ne mentionnera ici que les différences essentielles entre ces deux langages.

Voici une coquille de programme Java :

```
public class Programme
{
   public static void main(String args[])
   {
      ...
   }
}
```

A la place de ... on place une suite d'instructions qui correspond à ce qu'on place dans la fonction `main()` de C. Le nom du fichier source doit être identique à celui qui suit `public class` dans le source mais avec l'extension `.java`. Ici, le nom de notre fichier source est donc `Programme.java`. Pour compiler le source, on utilise la commande `javac` qui produit un code objet dont le nom du fichier est le même que le source mais avec l'extension `.class`. Ici, la commande `javac Programme.java` produit le fichier `Programme.class`. Pour exécuter le code objet, on utilise la commande `java` en mentionnant le nom du fichier objet sans son extension. Ici, la commande `java Programme` exécute le programme.

Les instructions `if`, `switch`, `while`, `for`, etc existent, s'écrivent et fonctionnent comme en C. Idem pour les opérateurs arithmétiques, logiques, etc.

Certains types primitifs diffèrent. Les types entiers (signés) sont `byte` (1 octet), `short` (2 octets), `int` (4 octets) et `long` (8 octets). Les types flottants sont `float` (4 octets) et `double` (8 octets). Les flottants permettent aussi de représenter l'infini positif ou négatif et la valeur non représentable (par exemple, le résultat de 0 divisé par 0). Le type caractère `char` est codé sur 2 octets. Par ailleurs, il existe le type

`boolean` qui représente les 2 valeurs `true` ou `false`. *Le résultat d'un test est de type* `boolean` *et non 0 ou 1 comme en C.*

Si on accède à la valeur d'une variable locale non initialisée, il se produit une erreur à la compilation. Le mot-clé `final` permet de spécifier qu'une variable est une constante (ex : `final int n;`). Une seule affectation est autorisée et toute tentative de réaffectation produira une erreur à la compilation.

Un tableau `tab` regroupant des éléments de type `X` se déclare ainsi : `X tab[];`. Après cette déclaration, `tab` n'est encore qu'un pointeur. On lui alloue dynamiquement une taille ainsi : `tab = new X[taille];`, où `taille` représente une valeur entière. `tab.length` a pour valeur la taille de `tab`.

Il existe un type (une classe, en fait) `String` pour représenter les chaînes de caractères. Ex : `String s;` déclare la chaîne `s`. On peut affecter une constante à une chaîne : `s = "bonjour";`. L'opérateur + permet la concaténation : `s = s + " Au revoir Monsieur " + nom;`. `s.length()` a pour valeur la longueur de `s`. Pour afficher une chaîne `s` : `System.out.println(s);`

Pour lire une chaîne, en Java et en Java 5.0 :

```
String chaine;                                                | Scanner s = new Scanner(System.in);
InputStreamReader isr = new InputStreamReader(System.in); | String param = s.next();
BufferedReader entree = new BufferedReader(isr);          | int value = s.nextInt();
chaine = entree.readLine();                                   | s.close();
```

3 Les classes et leurs instances

Une *classe* est un type de structure de données qui unit la liste des attributs d'un objet et les méthodes qui opèrent sur ces attributs. Voici l'exemple classique de la classe Compte qui représente un compte en banque qui possède 3 attributs (`solde`, `numero` et `proprietaire`) et définit 3 méthodes (`creation`, `depot`, `retrait`) :

EN JAVA :

```
class Compte
{
   int solde;
   int numero;
   String proprietaire;

   void creation(int num, String prop)
   {
      solde = 0;
      numero = num;
      proprietaire = prop;
   }

   void depot(int montant)
   {
      solde = solde + montant;
   }

   void retrait(int montant)
   {
      solde = solde - montant;
   }
}
```

EQUIVALENT EN C :

```
typedef struct
{
   int solde;
   int numero;
   char* proprietaire;
}Compte;

void creation(Compte *c, int num, char* prop)
{
   c->solde = 0;
   c->numero = num;
   c->proprietaire = prop;
}

void depot(Compte* c, int montant)
{
   c->solde = c->solde + montant;
}

void retrait(Compte* c, int montant)
{
   c->solde = c->solde - montant;
}
```

On peut alors déclarer des objets de type `Compte`. On dit que ces objets sont des *instances* de la classe `Compte`. En Java, la déclaration d'un objet ne crée qu'une **référence** (= un pointeur) sur cet objet. Il faut

ensuite allouer cet objet en mémoire grâce à l'opérateur `new`.

EN JAVA :	EQUIVALENT EN C :
Compte unCompte; unCompte = new Compte();	Compte* unCompte; unCompte = (Compte*) malloc(sizeof(Compte));

Une fois l'objet créé, on peut accéder à ses attributs ou lui demander d'exécuter ses méthodes. Pour ceci, on mentionne le nom de l'objet suivi d'un point suivi d'un nom d'attribut ou de méthode et de ses paramètres :

EN JAVA :	EQUIVALENT EN C :

unCompte.creation(1234, "Jean Dupont");
unCompte.depot(1000);
unCompte.retrait(450);
System.out.println("Il reste " + unCompte.solde + " a "
+ unCompte.proprietaire);
	creation(unCompte, 1234, "Jean Dupont");
	depot(unCompte, 1000);
	retrait(unCompte, 450);
	printf("Il reste %f a %s", unCompte->solde,
	unCompte->proprietaire);

Lorsqu'à l'intérieur de la définition d'une méthode on veut désigner un attribut ou une méthode appartenant à l'objet sur lequel on appellera la méthode en train d'être définie, on ne peut connaître le nom de l'objet. On se contente alors d'écrire le nom de l'attribut ou de la méthode (sans le/la préfixer par un nom d'objet). Ex : on ajoute la méthode suivante à la classe `Compte` :

```
void virement(int montant, Compte destinataire)
{
   retrait(montant);
   destinataire.depot(montant);
}
```

Surcharge des méthodes

Puisque chaque méthode est liée à sa classe, il est possible que des méthodes ayant la même signature (= nom + nombre et type des paramètres) soient définies dans des classes différentes.

Par ailleurs, une classe peut définir des méthodes ayant le même nom mais des listes de paramètres qui différent. Ex : on peut ajouter la méthode

```
boolean retrait(int montant, String prop)
{
   if(prop == proprietaire && montant <= solde)
   {
      retrait(montant);
      return true;
   }
   else
      return false;
}
```

à la classe `Compte`. On dit que la méthode est *surchargée*.

Constructeurs

A chaque classe est associé au moins un *constructeur*. C'est une méthode particulière qui porte le *même nom* que la classe et qui est automatiquement exécutée lors de la création d'un objet (avec l'opérateur `new`). La syntaxe d'un constructeur est la même que celle d'une méthode ordinaire à ceci près qu'on n'indique aucun type de paramètre de retour dans sa signature. Il peut y avoir plusieurs constructeurs. Lors de la création d'un objet, il faut passer une liste des valeurs en paramètre. Cette liste permet de sélectionner le constructeur à exécuter et de lui fournir les valeurs dont il a besoin.

Un constructeur est utile pour initialiser les attributs de l'objet qui vient d'être créé. Ex : la méthode `creation` de la classe `Compte` a déjà un rôle de constructeur. On peut donc la remplacer par :

```
Compte(int num, String prop)
{
    solde = 0;
    numero = num;
    proprietaire = prop;
}
```

Pour créer une instance de `Compte`, on écrira : `unCompte = new Compte(1234, "Jean Dupont");`

Un constructeur ne peut pas être appelé explicitement comme on le ferait avec une autre méthode. Si aucun constructeur n'est défini dans une classe alors un constructeur par défaut est automatiquement ajouté. Il n'a aucun paramètre et n'effectue aucune opération.

Initialisation d'un objet

Les attributs d'un objet sont toujours initialisés par défaut. Si l'attribut est un nombre, il sera initialisé à 0. Si c'est une référence à un objet, il sera initialisé à `null`. On peut aussi indiquer explicitement la valeur initiale d'un attribut lorsqu'il est déclaré (dans la définition de sa classe) : `int solde = 100;`. Par ailleurs, un attribut peut être initialisé dans un constructeur. Cela a un sens si sa valeur initiale dépend des paramètres passés au constructeur.

Destruction d'un objet

Lorsqu'un objet n'est plus utile, il faut libérer l'emplacement mémoire qu'il occupe. En C (resp. en C++), on le fait explicitement grâce à la fonction `free` (resp. l'opérateur `delete`). En Java, il n'existe pas l'opérateur inverse de `new` permettant de libérer la mémoire occupée par un objet devenu inutile. En Java, le programmeur n'a pas besoin de s'occuper de libérer la mémoire car un ramasse-miettes [garbage collector] s'en occupe tout seul : dès que Java s'aperçoit qu'un objet n'est plus référencé (c-a-d plus aucune variable ne contient son adresse), il devient candidat à l'élimination et sa mémoire est libérée automatiquement à terme.

Juste avant de détruire un objet en libérant sa mémoire, Java fait appel à sa méthode `finalize()` si elle a été définie (dans sa classe). Ne définissez `finalize()` qu'avec précaution : on ne peut déterminer avec précision quand ses instructions seront exécutées.

Attributs et méthodes de classe

Jusqu'à présent, nous avons considéré que tous les attributs et les méthodes se rapportaient à une instance particulière. En fait, on peut aussi définir des *attributs de classe* qui existent en un seul exemplaire au niveau de la classe et donc en dehors de toute instance. Pour déclarer un attribut de classe, il suffit de préfixer sa déclaration du mot-clé `static`. Pour désigner cet attribut, il faut le préfixer du nom de sa classe (et non pas d'un nom d'instance comme pour les attributs d'instance). Une *méthode de classe* est une méthode qui s'exécute en réference à sa classe et non plus à une instance particulière. Cette méthode ne peut désigner et modifier que des attributs de classe. La méthode `main` en est un exemple particulier. Elle s'exécute en dehors de toute création d'instance de la classe où elle est définie. Quand on veut exécuter un programme en Java, lancer la commande `java UneClasse` provoque l'exécution de la méthode `main` de la classe `UneClasse`.

Exemple :

```
public class A
{
   static int nbi = 0;

   static void nbInstances()
   {
      System.out.println("La classe A a " + nbi + " instances");
   }

   A()
   {
      nbi = nbi + 1;
   }
   public void finalize()
   {
      nbi = nbi - 1;
   }

   public static void main(String args[])
   {
      A a1 = new A();
      A a2 = new A();

      if(A.nbi > 0)
         A.nbInstances();
   }
}
```

Par ailleurs un bloc `static {...}`, servant de *constructeur de classe* peut être intégré dans la définition d'une classe pour initialiser des attributs de classe. L'utilité en est la même que pour les constructeurs et les attributs d'instance : on s'en sert lorsqu'il faut calculer des valeurs d'attributs de classe.

Composition de classes

Les attributs d'un objet peuvent eux-même être des objets. En fait, comme pour une variable dont le type est une classe, un attribut ne mémorise que la référence de son objet. La composition peut être récursive : un attribut d'une instance de la classe A peut être une instance de A.

Du fait qu'on ne mémorise que la référence d'un objet, il faut faire attention lorsqu'on veut affecter un objet à un autre du même type, comparer deux objets du même type ou effectuer la duplication d'un objet.

Après les instructions suivantes :

```
Compte unCompte, unAutreCompte;
unCompte = new Compte(1,"Dupont");
unAutreCompte = new Compte(2,"Martin");
unCompte = unAutreCompte;
```

Les variables `unCompte` et `unAutreCompte` font référence au même objet (celui qui a Martin pour propriétaire). L'autre objet n'est plus référencé. `unCompte.depot(100);` et `unAutreCompte.depot(200);` ajoutent en tout 300 au solde du compte dont le propriétaire est Martin. Si l'opération qu'on veut effectuer est la création d'une copie d'un objet alors il faut créer un nouvel objet et y copier (éventuellement récursivement) un par un tous les attributs de l'objet original. Une classe peut définir une méthode retournant

(une référence sur) un objet qui est la copie de l'instance sur laquelle est invoquée cette méthode :

```
class A
{
   int n;
   B b;

   A copie()
   {
      A copieA = new A();
      copieA.n = n;
      copieA.b = b; // OU mieux : copieA.b = b.copie();
      return copieA;
   }
   ...
}
```

De même, si on veut comparer deux objets `a1` et `a2` pour savoir s'ils sont identiques, `a1 == a2` ne permet que de savoir si `a1` et `a2` référencent le même objet. Pour comparer leur contenu, il faut comparer récursivement leurs attributs.

Autoréférence

Le mot-clé `this`, utilisé à l'intérieur de la définition d'une méthode, fait référence à l'instance sur laquelle est appelée la méthode. Quand on désigne un attribut ou une méthode sans les préfixer d'une référence à une instance, tout ce passe comme si on les préfixait par `this`. On peut le faire mais ça n'est utile que lorsqu'il y a une ambiguïté entre un nom d'attribut et un nom de paramètre ou de variable locale car le nom tout seul désigne le paramètre.

Par ailleurs, utilisé comme nom de méthode à l'intérieur de la définition d'un constructeur, `this` fait référence à un constructeur de la même classe.

Exemple illustrant ces divers aspects dans la classe `Compte`:

```
class Compte
{
   ...
   Compte(int numero, String prop)
   {
      solde = 0; // inutile
      this.numero = numero; // indispensable
      this.proprietaire = prop; // 'this' est inutile
   }

   Compte(int numero)
   {
      this(numero, "anonyme");
   }

   boolean transfertPerso(int montant, Compte source)
   {
      if(proprietaire == source.proprietaire)
      {
         source.virement(montant, this);
         return true;
      }
      else return false;
   }
}
```

Paquetage

Un paquetage [package] est le regroupement sous un nom d'un ensemble de classes. Les paquetages de Java sont hiérarchisés sous la forme d'une arborescence. En chaque noeud de l'arborescence peut se trouver un paquetage. Un paquetage se désigne par un ensemble d'identifiants séparés par des points. Ex : `java.awt.geom` est un paquetage de Java contenant des classes définissant des objets géométriques. A l'arborescence logique des paquetages correspond l'arborescence physique des répertoires et fichiers mémorisants les fichiers *.class des paquetages. Ex : les classes du paquetage `java.awt.geom` sont stockées dans le répertoire `java/awt/geom` à partir du répertoire racine des paquetages.

Pour indiquer depuis un fichier source que ses fichiers *.class créés seront stockés dans le répertoire `MonPaquetage`, il faut placer la commande
`package MonPaquetage;` en tête du fichier.

Pour utiliser une ou plusieurs classes définies dans un paquetage, il y a plusieurs solutions :

- Expliciter le paquetage de la classe en préfixant le nom de la classe par son nom de paquetage : `MonPaquetage.A a = new MonPaquetage.A();`.
- Utiliser en tête du fichier l'instruction `import` suivi du nom du paquetage et de la classe : `import MonPaquetage.A;` On n'a alors plus besoin de préfixer le nom de la classe dans le reste du fichier : `A a = new A();`
- Importer toutes classes d'un paquetage : `import MonPaquetage.*;` On n'a alors plus besoin de préfixer aucun nom de classe de ce paquetage dans le reste du fichier. Ceci ne concerne pas les sous-paquetages du paquetage importé.

4 Encapsulation

L'*encapsulation* est une notion importante de la programmation objet et du génie logiciel. Elle consiste à masquer le plus possible les détails d'implémentation et le fonctionnement interne des objets. Cette dissimulation permet de découpler les classes constituants un programme afin que la modification de la structure interne de l'une n'oblige pas à modifier l'autre. L'encapsulation permet aussi la réutilisation d'une classe dans un autre contexte (cas des bibliothèques).

Niveaux d'accès

En Java, on peut restreindre l'accès à des classes, des méthodes et des attributs. Cette restriction les rend invisibles et inréférençables en dehors de leur niveaux d'accès. Concernant les membres (= méthodes et attributs) des classes, il existe 4 niveaux d'accès :

- public (`public`) : le membre est accessible de n'importe où.
- protégé (`protected`) (cette restriction est liée à la notion d'héritage que nous verrons plus tard.)
- privé paquetage (pas de mot-clé, niveau d'accès par défaut) : le membre n'est accessible que depuis les autres classes du paquetage où il est déclaré.
- privé (`private`) : le membre n'est accessible que depuis l'intérieur de la classe où il est déclaré.

Concernant les classes, il y a deux niveaux d'accès : public ou privé paquetage. Dans un même fichier, une seule classe au maximum peut être déclarée publique. Si une classe contient la méthode statique `main` alors c'est cette classe qui doit être publique.

Pourquoi restreindre les accès ?

Le principe de base de l'encapsulation est de rendre tout le moins accessible possible : par défaut, tous les attributs et les méthodes sont privés et les classes sont privées paquetage. L'allégement de toute restriction doit être motivée.

Si l'accès d'une classe est privé paquetage alors on sait que si on la modifie ou on la supprime, aucune classe extérieure au paquetage n'en sera affectée. Il n'y aura qu'à modifier éventuellement certaines classes du paquetage.

Si l'accès d'un membre d'une classe est privé (resp. privé paquetage) alors on pourra le modifier ou le supprimer sans qu'aucune autre classe (resp. aucune classe en dehors du paquetage) ne soit affectée.

En pratique, afin de rendre opaque la structure d'un objet pour permettre de modifier cette structure ultérieurement, **tous les attributs seront privés** dans toutes les classes. Dans notre exemple de la classe `Compte`, on va rendre privé l'accès aux attributs et à la méthode `retrait` à un seul paramètre, afin qu'un objet extérieur ne puisse pas modifier une instance de Compte n'importe comment :

```
class Compte
{
    private int solde;
    private int numero;
    private String proprietaire;

    public Compte(int numero, String prop){...}
    public Compte(int numero){...}
    public void depot(int montant){...}
    private void retrait(int montant){...}
    public boolean retrait(int montant, String prop){...}
    public boolean transfertPerso(int montant){...}
    public void virement(int montant, Compte dest)
}
```

Néanmoins, on peut avoir besoin de connaître le solde d'un compte. Comme on doit interdire de modifier l'attribut `solde` de l'extérieur de la classe, on va simplement ajouter la méthode `public int solde(){return solde;}`.

De façon plus générale, si on a besoin de connaître la valeur d'un attribut ou de le modifier, on effectuera cette opération via une méthode (non privée) dite *d'accès*. Les méthodes fournissant la valeur d'un attribut sont appelées des *accesseurs*. Les méthodes transmettant une nouvelle valeur à affecter à un attribut sont appelées des *modifieurs*. En les déclarant privés, on a donc interdit de référencer les attributs en dehors de leur classe mais les méthodes d'accès permettent de choisir pour chacun leur degré d'accessibilité en lecture et/ou en écriture.

On peut encore aller plus loin que ce qu'intègre officiellement Java pour l'encapsulation. On peut aussi faire en sorte qu'à l'intérieur d'une classe, chaque méthode qui n'est pas une méthode d'accès ne référence ses attributs qu'aux moyens des méthodes d'accès. Ainsi, si les attributs de cette classe sont réécrits, **seuls les méthodes d'accès doivent être réécrites**.

Exemple : on veut créer une classe `Complexe` pour représenter des nombres complexes. Un nombre complexe z a deux formes : $z = x + i.y$ ou $z = \rho.e^{i.\theta}$. Comme ρ et θ peuvent se calculer en fonction de x et y, on peut décider de déclarer les attributs flottants `partieReelle` et `partieImaginaire` ainsi que les 8 méthodes suivantes : `float getRe()`, `void setRe(float)`, `float getIm()`, `void setIm(float)`, `float getRho()`, `void setRho(float)`, `float getTheta()`, `void setTheta(float)`. Les 4 premières lisent et affectent directement les attributs `partieReelle` et `partieImaginaire` tandis que les 4 dernières le font indirectement via des calculs (mais l'utilisateur de la classe n'est pas censé le savoir). Plus tard on peut décider de remplacer les attributs `partieReelle` et `partieImaginaire` par `rho` et `theta` ou d'ajouter ces deux derniers. Il faudra alors modifier (une partie) de ces 8 méthodes d'accès mais il n'y aura pas besoin de modifier le code des classes utilisant la classe `Complexe` ni même les autres méthodes de cette classe (typiquement: celles effectuant des opérations sur les complexes).

5 Héritage

L'héritage est une notion-clé de la programmation objet. Le mécanisme d'héritage consiste à créer une nouvelle classe à partir d'une **classe qui existe déjà en la complétant** pour qu'elle permette de créer des objets **plus spécifiques**. Si la classe B hérite (ou dérive) de A alors on dira que A est la *superclasse* ou la *classe-mère* de B, tandis que B est la *sous-classe* ou la *classe-fille* de A.

Quand B dérive de A, B possède implicitement tous les attributs et toutes les méthodes de A. Dans la déclaration de B, on ajoute des attributs et des méthodes supplémentaires ou de remplacement. Faire dériver une classe B à partir d'une classe A n'a de sens que si *toutes les instances possibles de B forment un sous-ensemble de l'ensemble de toutes les instances possibles de A*. Par exemple, faire hériter `Chien` d'`Animal` a un sens. Faire hériter `Complexe` de `Réel` est une *erreur de conception*, même si on se dit qu'on aimerait reprendre les opérations sur les réels, rajouter un attribut pour la partie imaginaire du nombre complexe et compléter les opérations, etc. C'est le contraire qui aurait un sens (faire dériver `Réel` de `Complexe`) car tous les nombres réels sont des nombres complexes (particuliers). En résumé, ne pas confondre héritage (tout B est A) et composition (tout A possède un B, un B appartient à un A).

En Java, pour déclarer une classe B qui hérite de A, on ajoute `extends A` derrière l'en-tête de la déclaration (`class B`). Dès lors, la classe B possède implicitement tous les attributs et toutes les méthodes de la classe A. Dans le corps de la déclaration de la sous-classe, on déclare les attributs et les méthodes supplémentaires.

Si dans la sous-classe on déclare un attribut qui a le même nom qu'un attribut de la superclasse alors ce dernier est masqué, c'est-à-dire : il existe toujours mais il n'est plus accessible directement par son nom. Si dans la sous-classe on déclare une méthode qui a la même signature qu'une méthode de la superclasse, cette dernière est masquée. Si on veut quand même désigner un attribut masqué ou une méthode masquée de la superclasse alors il faut les préfixer avec le mot-clé `super`, qui référence la superclasse.

```
class A
{
   int val = 1;
   int val2 = 2;

   void salutations()
   {
      System.out.println("Bonjour");
   }

   void affiche()
   {
      System.out.println(val + " " + val2);
   }
}

public class B extends A
{
   int val = 10;
   int val3 = 3;

    void aurevoir()
   {
      System.out.println("Au revoir");
   }

   void affiche()
   {
      super.affiche(); // affiche : 1 2
      System.out.println(val + " " + val3); // affiche : 10 3
      System.out.println(super.val); // affiche : 1
   }

   public static void main(String[] args)
   {
      B b = new B();

      b.salutations(); // appel de la méthode de A
      b.aurevoir(); // appel de la méthode de B
      b.affiche(); // appel de la méthode de B
   }
}
```

Quand un attribut ou une méthode sont désignés, on les cherche d'abord dans la classe de l'instance sur laquelle on les invoque, puis, s'ils n'y sont pas définis, dans sa superclasse, puis dans la superclasse de sa superclasse, etc.

Accessibilité

Tout attribut et toute méthode sont directement accessibles depuis une sous-classe sauf s'ils sont privés. Un attribut privé peut être accessible indirectement via un accesseur (non privé).

Le mot-clé `protected` sert à ne permettre l'accès d'un attribut ou d'une méthode qu'aux méthodes d'un même paquetage ou d'un paquetage des sous-classes de cette classe où ils sont déclarés. En pratique, son utilisation est déconseillée.

Quand une méthode est redéfinie, on ne peut changer sa visibilité que si on l'augmente (privé < privé-paquetage < protégé < public).

En préfixant la déclaration d'une méthode avec le mot-clé `final` on interdit qu'une de ses sous-classes puisse redéfinir cette méthode.

En préfixant la déclaration d'une classe avec le mot-clé `final` on interdit sa dérivation.

Les constructeurs

En Java, on ne récupère pas directement les constructeurs de la classe dont on hérite. Il faut explicitement écrire tous les constructeurs dont la classe a besoin. On peut cependant faire appel au constructeur de la superclasse grâce au mot-clé `super` suivi d'une liste de paramètres correspondant au constructeur. L'appel au constructeur de la superclasse doit se faire en début de la définition du constructeur de la sous-classe. Si aucun appel explicite n'est fait à un constructeur de superclasse alors l'appel implicite `super();` est effectué dans le constructeur. Attention: si on ne définit aucun constructeur, c'est le constructeur par défaut (sans paramètres) qui est défini avec appel automatique au constructeur sans paramètre de la superclasse. En conséquence, si la superclasse n'a pas de constructeur sans paramètre (car il a été désactivé par la définition explicite d'un constructeur avec paramètres), il y a une erreur à la compilation (constructeur sans paramètre non défini).

L'utilisation standard d'un constructeur étant l'initialisation des attributs de la classe, il est habituel et sain de faire appel à un constructeur de la superclasse pour qu'il initialise lui-même les attributs déclarés dans la superclasse puis d'initialiser explicitement les nouveaux attributs introduits dans la sous-classe.

La classe Object

Par défaut, toute classe écrite en Java hérite directement de la classe `Object`. Si une classe hérite explicitement d'une autre classe alors elle hérite quand même indirectement d'`Object` par l'intermédiaire de cette autre classe.

6 Le polymorphisme en Java

En Java, on peut affecter à une variable a de type A (une instance de la classe A) la valeur d'une variable b de type B sous-type de A (une instance de la classe B qui hérite de A). a référence alors le même objet que référence b. On dit que le type de a est devenu B de façon dynamique. Le polymorphisme est la capacité d'une variable de changer dynamiquement son type. On peut alors appeler des méthodes définies dans B sur a :

- si la méthode invoquée n'a été définie que dans la superclasse, elle est invoquée.
- si la méthode invoquée a été redéfinie dans la sous-classe, c'est la méthode redéfinie qui est invoquée.
- si la méthode invoquée n'est définie que dans la sous-classe, l'invocation directe provoque une erreur de type à la compilation. Pour pouvoir invoquer la méthode, il faut d'abord transtyper a en B pour invoquer la méthode de la sous-classe. En effet, le compilateur Java contrôle le **type déclaré** de la variable et non son type réel, celui de l'objet pointé par la variable, qui ne peut être connu qu'à l'exécution du programme.

Exemple :

```
B b = new B();
A a = new A();

a.salutations(); —> la méthode de A est appelée
a.affiche(); ———> la méthode de A est appelée
// b = a ; ————> INTERDIT : provoque une erreur de type à la compilation
a = b; ————> la variable 'a' pointe maintenant sur l'objet référencé par 'b'
a.salutations(); —> la méthode de A (non redéfinie dans B) est appelée
// a.aurevoir(); ——> INTERDIT : provoque une erreur de type à la compilation
((B)a).aurevoir(); –> la méthode de B est appelée
a.affiche(); ———> la méthode de B est appelée !
```

Grâce au polymorphisme, on peut stocker dans une structure de données destinée à regrouper plusieurs objets de type A (un tableau de A, une liste de A, etc) des objets hétéroclites dont les types sont des sous-types de A puis invoquer la même méthode sur ces objets. Ceux-ci vont alors chacun exécuter l'opération qui correspond à leur type réel. Afin de ne pas invoquer une méthode sur une instance dont la classe ne reconnaît pas la méthode, on peut tester le type de l'objet pour savoir si on peut invoquer la méthode : *variable* `instanceof` *classe* teste si la variable référence une instance de la classe indiquée. Exemple :

```
A tab[] = new A[100];
int i;
Random alea = new Random(); // création d'une instance d'entier aléatoire

for(i = 0; i < tab.length; i++)
   if(alea.nextInt() % 2 == 0) // une chance sur deux
      tab[i] = new B();
   else
      tab[i] = new A();

for(i = 0; i < tab.length; i++)
{
   tab[i].salutations();
   tab[i].affiche(); // la méthode définie dans A ou dans B (selon le type réel)
   if(tab[i] instanceof B)
      ((B)tab[i]).aurevoir(); // transtypage à cause du test de type à la compilation
}
```

Remarque: toute variable de type `Object` peut être instanciée avec un objet de n'importe quel type.

7 Les classes abstraites

Les *classes abstraites* sont des classes qui ne sont pas destinées à avoir d'instance (et ne peuvent pas en avoir). En général, en plus d'un certain nombre de méthodes "concrètes", elles contiennent des *méthodes abstraites* qui sont simplement des déclarations de méthodes contenant leur signature mais pas leur code. En Java, pour déclarer une méthode abstraite, il faut préfixer sa signature avec le mot-clé `abstract`. Pour déclarer une classe abstraite, il faut préfixer sa déclaration avec le même mot-clé `abstract`. Toute classe contenant une méthode abstraite doit être déclarée abstraite mais il n'est pas nécessaire qu'une classe contienne une méthode abstraite pour qu'on puisse la déclarer abstraite.

```
abstract class A
{
   void methode1() { ... }
   abstract int methode2(int x); // méthode abstraite
   ...
}
```

Toute classe qui hérite d'une classe abstraite est abstraite et doit être explicitement déclarée abstraite sauf si elle redéfinit toutes les méthodes abstraites de sa superclasse.

Les classes abstraites servent à modéliser des types d'objets qui n'ont pas de réalité concrète. Par exemple, un "moyen de transport" est un concept abstrait tandis qu'une "voiture" peut correspondre à un objet concret. Si dans une application on a besoin de manipuler des objets dont les comportements sont voisins, il peut être utile de créer une superclasse abstraite qui caractérise et factorise ce qu'ils ont en commun.

Exemple : on a affaire à des vélos, des voitures, des bus et des tramways. Tous sont caractérisables par leur position, leur conducteur et le fait qu'ils puissent avancer (chacun à leur manière). En outre, les voitures, bus et tramways peuvent prendre des passagers. Les voitures ne peuvent prendre que 3 passagers en plus du conducteur tandis que les tramways et bus, en tant que transports en commun, peuvent en prendre un nombre plus grand. Voici une modélisation possible :

```
abstract class MoyenDeTransport
{
    private Passager conducteur;
    private Position position;
    MoyenDeTransport(Passager c, Position p)
    {
        conducteur = c;
        position = p;
    }
    abstract public void avance();
}

class Velo extends MoyenDeTransport
{
    void avance() { ... }; // Comment avance un vélo
    ...
}

abstract class VehiculeCollectif extends MoyenDeTransport
{
    abstract public void accueillePassager(Passager p);
}

class Voiture extends VehiculeCollectif
{
    private Passager passagerAvant;
    private Passager passagerArriereDroit;
    private Passager passagerArriereGauche;

    public void accueillePassager(Passager p) { ... }
    void avance() { ... }; // Comment avance une voiture
    void recule() { ... }; // Comment recule une voiture
    ....
}
```

```
abstract class TransportEnCommun extends VehiculeCollectif
{
    private Passager[] LesPassagers;
    TransportEnCommun(Passager c, Position p, int capacite)
    {
        super(c, pos);
        LesPassagers = new Passager[capacite];
    }
    void accueillePassager(Passager p) { ... }
    ...
}

class Bus extends TransportEnCommun
{
    void avance() { ... }; // Comment avance un bus
    void recule() { ... }; // Comment recule un bus
    ...
}

class Tramway extends TransportEnCommun
{
    void avance() { ... }; // Comment avance un tramway
    ...
}
```

Si maintenant on veut modéliser un flux de véhicules arrivant à un croisement avec des feux tricolores, on pourra mémoriser les véhicules de toutes sortes (vélo, voiture, bus, tramway) dans un tableau d'instances de `MoyenDeTransport` et invoquer sur chacun la méthode `avance` lorsque le feu est vert.

8 Les interfaces

Les interfaces sont une spécificité du langage Java. Il s'agit de classes abstraites particulières : tous leurs attributs doivent explicitement être `static` et `final` et toutes leurs méthodes sont implicitement abstraites. On les déclare comme des classes mais en remplaçant le mot-clé `class` par `interface`. Elles sont

implicitement abstraites. Exemple :

```
interface MonInterface
{
    final static int uneConstante = 10;
    void uneMethode(); // méthode implicitement abstraite
}
```

Contrairement à une classe qui ne peut hériter que d'une seule classe, une interface peut hériter de plusieurs interfaces. Dans ce cas, on fait comme habituellement, on utilise le mot-clé `extends` mais en indiquant la suite d'interfaces desquelles on hérite en les séparant par des virgules. Une classe peut aussi hériter de plusieurs interfaces mais il faut alors utiliser le mot-clé `implements` à la place de `extends`. D'ailleurs, une classe peut hériter d'une autre classe et de plusieurs interfaces. Exemple :

```
interface Interface1 extends InterfaceMere
{
    void nouvelleMethode(int x);
}

class UneClasse extends ClasseMere implements Interface1, Interface2
{
    [On implémente toutes les méthodes déclarées dans Interface1
    et Interface2 ainsi que les méthodes abstraites de ClasseMere]
}
```

Une interface, comme son nom l'indique, sert de relais entre une classe qui l'implémente et une classe qui l'utilise. Elle indique les méthodes qu'une classe met à la disposition des autres classes mais sans indiquer son implémentation ni sa structure interne. Lorsqu'une application nécessite de définir de nombreuses classes qui vont interagir et qui seront écrites par des programmeurs différents, chaque programmeur n'a besoin de connaître que les interfaces des classes qu'il n'a pas à écrire mais dont les classes qu'il écrit ont besoin pour fonctionner. Exemple :

```
// fichier InterfaceDeA.java
public interface InterfaceDeA
{
    void methodeDeA();
}
```

```
// fichier A.java
public class A implements InterfaceA
{
    ...
    void methodeDeA() { ... }
    ...
}
```

```
// fichier B.java (n'a pas besoin de connaître la classe A)
public class B
{
    private InterfaceDeA a;
    ...

    public B(InterfaceDeA a)
    {
        this.a = a; // fonctionnera grâce au polymorphisme
    }
    public void methodeDeB()
    {
        a.methodeDeA(); // fonctionnera grâce au polymorphisme
        ...
    }
    ...
}
```

```
// fichier Principale.java
// (classe servant à l'exécution de l'application, connaît la classe A)
public class Principale
{
   public static void main(String[] args)
   {
      A a = new A();
      B b = new B(a); // fonctionne grâce au polymorphisme

      b.methodeDeB();
   }
}
```

Celui qui implémente la classe `B` ne connaît que l'interface de `A`. Ce n'est que lorsqu'on utilisera les classes pour écrire la classe `Principale`, qui ne sert qu'à exécuter le programme, qu'on aura besoin de connaître la classe `A` pour créer un objet de ce type.

Un autre type d'utilisation des interfaces est la simulation de l'héritage multiple.

9 Héritage multiple

Contrairement à C++, Java ne permet pas à une classe d'hériter de plusieurs classes. L'héritage multiple pose des difficultés que Java a décidé d'éluder :

- Si un attribut (resp. une méthode) est défini sous le même nom (resp. même signature) dans les deux classes dont une classe-fille hérite, à quel attribut (resp. méthode) se réfère la classe-fille quand elle le nomme ? (en C++, il faut préfixer l'attribut (resp. la méthode) par le nom de sa classe pour les différencier)
- Si un attribut est déclaré dans une classe A, dont hérite deux classes B1 et B2 qui l'initialisent chacune avec une valeur différente, une classe C qui hérite de B1 et de B2 aura quelle valeur initiale pour cet attribut ? Cet attribut doit-il exister en un ou plusieurs exemplaires ? Si une méthode a été définie dans A, redéfinie dans B1 mais pas dans B2, à quelle définition de méthode doit-on se référer dans C ? (en C++, utilisation de l'héritage virtuel)

Simulation de l'héritage simple grâce à la composition

Pour récupérer les attributs et les méthodes d'une classe A, on peut déclarer une classe B qui contient un attribut de type A. Il reste alors à déclarer dans B les méthodes qu'il y a dans A. Ces méthodes ne consistent qu'en l'appel de la méthode correspondante sur l'attribut de type A. Exemple :

```
class A
{
   private int attribut;
   ...
   A(...) { ... }
   void methode1() {...}
   ...
}
```

```
class B
{
   A a;
   ... [attributs supplémentaires]
   B(...)
   {
      a = new A(...);
      ...
   }
   void methode1()
   {
      a.methode1();
   }
   ...
}
```

Simulation de l'héritage multiple grâce à la composition

On peut simuler un héritage multiple par une combinaison entre un héritage simple et une composition. Si on veut qu'une classe B hérite de deux classes A1 et A2, on crée une interface à A2 et on déclare que B hérite de A1, implémente l'interface de A2 et contient un attribut de type de l'interface de A2. On résout ainsi les problèmes posés par l'héritage multiple :

- On différencie explicitement les attributs et méthodes communes à A1 et A2 : si le nom est le même, c'est l'attribut/la méthode de A1, si on préfixe le nom par le nom de l'attribut de type A2, il s'agit évidemment de l'attribut/la méthode défini(e) déclaré par A2.
- Les attributs aux noms communs existent automatiquement en un exemplaire chacun.
- Si une classe Z hérite deux de classes Y1 (par héritage) et Y2 (par implémentation d'une interface) qui héritent toutes deux d'une classe X, que X définit une méthode qui a été redéfinie uniquement par Y1, on peut décider facilement à quelle définition de méthode on va se référer lors de son invocation : dans Z, l'invocation directe de la méthode (ou de celle de la superclasse si on redéfinit cette méthode dans Z) fait référence à la méthode redéfinie dans Y1 et son invocation sur l'attribut de type Y2 fait référence indirectement à sa définition dans X.

Exemple :

```
class A1
{
    ...
    A1(...) { ... }
    void methodeA1() {...}
    void affiche() {...}
    ...
}

interface IA2 {
    void methodeA2();
    void affiche();
    ...
}

class A2 implements IA2
{
    ...
    A2(...) { ... }
    void methodeA2() {...}
    void affiche() {...}
    ...
}
```

```
class B extends A1 implements IA2
{
    IA2 a2;
    ... [attributs supplémentaires]
    B(...)
    {
        super(...); // appel constructeur de A1
        a2 = new A2(...); // appel constructeur de A2
        ...
    }

    void methodeA2() // Redéfinition des méthodes de A2
    {
        a2.methodeA2();
    }

    void affiche() // redéfinition possible d'une méthode commune
    {
        super.affiche(); // appel de la méthode de A1
        a2.affiche(); // appel de la méthode de A2
    }
    ...
}
```

Bibliographie

- H. Bersini, "L'orienté objet", Eyrolles.
- Bruce Eckel, "Thinking in Java", téléchargeable à *http://www.MindView.net/Books/TIJ*.
- Niemeyer et Knudsen , "Introduction à Java", Eyrolles.
- F. Delannoy, "Programmer en Java", Eyrolles.
- J. Bloch, "Java efficace", Vuibert.