I4 - Programmation objet

N. Prcovic

Contenu général du cours

La *programmation objet*

- Pourquoi ? Répond à des principes de *génie logiciel* (= méthologie à adopter pour réaliser un logiciel).
- Comment ? En utilisant le langage *Java*.

Evolution de la programmation impérativ

- **Langage machine** : suite d'instructions + suite de données (mémorisés sous forme d'octets).
  Une instruction modifie un (petit nombre d') octet(s).
- **Langages procéduraux** : instructions regroupées en *procédures* et données regroupées en *structures* (types). On modifie toujours un état global.
- **Langages objets** : regroupement des structures de données avec les fonctions qui les modifient (= classes).

Notion d'objet

- Une *classe* (= type) définit la structure de donnée et les fonctions qui s'appliquent à ses *instances*, des *objets* (= variables).
- Un objet se définit par :
  - ses *attributs* (= champs) 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.

Exécution d'un programme objets

- En séquentiel : une succession d'envois de messages qui modifient de proche en proche les états des objets.
- Programmes distribués : les objets s'exécutent et s'envoient des messages en parallèle.

Pourquoi des objets ?

- Façon naturelle de représenter de nombreux problèmes complexes (organismes, organisations d'individus, machines réalisées par un assemblage de composants, ...).
- La modularité répond à des principes de génie logiciel :
  - Classes écrites en parallèle par différents programmeurs.
  - Le changement d'une classe a peu d'impact sur les autres -> réutilisabilité.

Remarque : "officialisation" de la façon de concevoir une application dans les autres langages impératifs.

Conception d'un programme objets

Diffèrence fondamentale entre programmation procédurale et programmation objet:

- En programmation procédurale: 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 intergir pour que le problème soit résolu.

Exemple

- Classe `Verre` contenant les attributs entiers `quantite` et `contenance` et les méthodes `emplir(int)` et `boire(int)`.
- Classe `Bouteille` (d'un litre) contenant l'attribut entier `quantite` et l'attribut booléen `est_ouverte` et les méthodes `ouvrir()`, `fermer()`, `verserDans(Verre, int)`.
- Programme : on crée une bouteille d'orange, une bouteille de vodka, un verre de 20cl, on verse 8cl de vodka puis 12cl d'orange dans le verre et on boit tout le verre.

Aspects impératifs de Java

Coquille du programme **Programme**`.java` :

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

La commande `javac` **Programme**`.java` produit le fichier **Programme**`.class`.

Exécution : commande `java` **Programme**.

Aspects impératifs de Java

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

Aspects impératifs de Java

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.
- 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*.

Aspects impératifs de Java

- 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.

Aspects impératifs de Java

- 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`.

Aspects impératifs de Java

- Type (classe) `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`.

Aspects impératifs de Java

- Pour afficher une chaîne `s` :

```
System.out.println(s);
```

- Pour lire une chaîne, en Java 5.0 :

```
Scanner s = new Scanner(System.in);
String param = s.next();
int value = s.nextInt();
s.close();
```

Les classes et leurs instances

- Rappel : une *classe* est un type qui unit la liste des *attributs* d'un objet avec les *méthodes* qui opèrent sur ces attributs.
- Exemple classique de la classe `Compte` qui représente un compte en banque qui définit
  - 3 attributs : `solde`, `numero` et `proprietaire`.
  - 3 méthodes : `creation`, `depot`, `retrait`.

La classe `Compte` en Java

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;
}
```

Création d'objet

En deux temps :

- déclaration d'une variable mémorisant un objet.
  En Java, la déclaration d'un objet ne crée qu'une **référence** (= un pointeur) sur cet objet.

EN JAVA :	EQUIVALENT EN C :
Compte unCompte;	Compte* unCompte;

- Il faut ensuite allouer cet objet en mémoire grâce à l'opérateur new.

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

Modification d'un objet

Une fois l'objet créé, on peut accéder à ses attributs ou lui demander d'exécuter ses méthodes.

Syntaxe:

- désignation d'un attribut: *nom de l'objet*.*nom d'attribut*
- appel de méthode
  nom de l'objet.*nom 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);

Modification d'un objet

- On peut ne pas préfixer un nom d'attribut ou de méthode si on l'écrit à l'intérieur de la définition d'une méthode.
  L'attribut ou la méthode désignés sont alors ceux de l'objet auquel appartient la méthode où ils sont écrits.
- Ex : on ajoute la méthode suivante à la classe `Compte` :

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

Surcharge des méthodes

- Il est possible de définir des méthodes ayant la **même signature** dans des classes différentes.
- Une classe peut définir des méthodes ayant le **même nom** mais des **listes de paramètres qui différent.**
  Ex : dans la classe `Compte`, ajout de la méthode

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

On dit que la méthode `retrait` 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 qui se différencient par leur liste de paramètres.

Constructeurs

- 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");
```

Constructeurs

- 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;`.
- 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

- Principe : 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`. Le programmeur n'a pas besoin de s'occuper de libérer la mémoire car un ramasse-miettes [garbage collector] s'en occupe tout seul.
- Juste avant de détruire un objet en libèrant sa mémoire, on fait appel à sa méthode `void finalize()` (si elle a été définie dans sa classe). Attention : on ne peut déterminer avec précision quand elle sera appelée.

Attributs et méthodes de classe

- *Attribut de classe* :
  - existe en un seul exemplaire au niveau de la classe (et donc en dehors de toute instance).
  - Déclaration : la préfixer du mot-clé `static`
  - Désignation : le préfixer du nom de sa classe (et non pas d'un nom d'instance).
- *Méthode de classe* :
  - s'exécute en réference à sa classe (et non plus à une instance particulière).
  - ne peut désigner et modifier que des attributs de classe.
  - Déclaration et désignation : idem attribut de classe.

Exemple : la méthode `main`

- La méthode `main` est un exemple particulier de méthode de classe.
- 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 fi nalize() { nbi = nbi - 1; }

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

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

"Constructeur" de classe

- Un bloc `static {...}` peut être intégré dans la définition d'une classe pour initialiser des attributs statiques.
- 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.

Conséquences des variables-références

Du fait qu'on ne mémorise que la référence d'un objet, il faut faire attention lorsqu'on veut :

- comparer deux objets du même type :
  a == b ne compare que les adresses.
- effectuer la duplication d'un objet :
  a = b ne fait que recopier dans a l'adresse contenue dans b.

A ne pas faire

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** (l'autre objet n'est plus référencé).

Les instructions `unCompte.depot(100);` et `unAutreCompte.depot(200);` ajoutent en tout 300 au solde du compte dont le propriétaire est Martin.

Comparaison d'objets

- Pour comparer deux objets, il faut comparer leur attribut un par un.
- Bonne idée : inclure une *méthode de comparaison* dans la classe :

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

    boolean egale(A a) {
        return n == a.n && b == a.b;
    }
    ...
}
```

Copie d'un objet

Pour copier un objet, il faut :

- créer une instance du même type que cet objet.
- recopier ses attributs un par un.
  Bonne idée : inclure une *méthode de copie* dans la classe :

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

    A copie() {
        A copieA = new A();
        copieA.n = n;
        copieA.b = b;
        return copieA;
    }
    ...
}
```

Constructeur de copie

Meilleure idée : inclure un *constructeur de copie* dans la classe :

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

    A(A a)
    {
        n = a.n;
        b = a.b;
    }
    ...
}
```

Et la copie se fait par l'instruction :

```
A copie = new A(x);
```

Composition de classes

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

```
class A
{
    B b;
    A a;
    ...
}
```

Composition : comparaison et copie

- Comparaison : il faut comparer **récursivement** les attributs.
- Copie : il faut copier **récursivement** les attributs.

```
class A {
    B b;
    A a;

    A copie(){
        A copieA = new A();
        copieA.b = b.copie();
        if(a == null)
            copieA.a = null;
        else
            copieA.a = a.copie();
        return copieA;
    }
    ...
}
```

Autoréférence

- Dans la définition d'une méthode, il faut parfois pouvoir référencer l'objet sur lequel sera appelé la méthode.
- Le mot-clé `this` fait référence à l'instance sur laquelle est appelée la méthode.
- Exemple : dans la classe `Compte`, on rajoute la méthode :

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

Autoréférence

- 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.
- Exemple : dans la classe `Compte`, on modifie le constructeur :

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

Autoréférence

- Utilisé comme nom de méthode, `this` fait référence à un constructeur.
- Exemple : dans la classe `Compte`, on rajoute un autre constructeur :

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

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.
- 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.

Paquetage

- 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.

Paquetage

- Pour inclure les classes d'un fichier source dans un paquetage, il faut placer la commande `package MonPaquetage;` en tête du fichier.
- Les fichiers `*.class` créés seront stockés dans le répertoire `MonPaquetage`.

Paquetage

Pour utiliser les classes d'un paquetage, 3 solutions :

- Expliciter le paquetage de la classe en préfixant le nom de la classe par son nom de paquetage :
  `MonPaquetage.A a = new 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.*;` Plus besoin de préfixer aucun nom de classe du paquetage.

Encapsulation

- **Principe** : masquer le plus possible les détails d'implémentation et le fonctionnement interne des objets.
- **Effet** : découpler les classes constituants un programme afin que la modification de la structure interne de l'une n'oblige pas à modifier l'autre.

Niveaux d'accès

- En Java, on peut restreindre l'accès à des **classes**, des **méthodes** et des **attributs**.
- Cette restriction les rend inréférençables en dehors de leur niveaux d'accès.
- Concernant les membres (méthodes et attributs) des classes, il existe 4 niveaux d'accès :
  - public (`public`) : accessible de n'importe où.
  - protégé (`protected`) : cf héritage.
  - privé paquetage (pas de mot-clé, niveau d'accès par défaut) : accessible uniquement depuis les autres classes du paquetage où il est déclaré.
  - privé (`private`) : accessible uniquement depuis l'intérieur de la classe où il est déclaré.

Niveaux d’accès

- 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 de classe `main` alors c’est cette classe qui doit être publique.

Restriction des accès

- **Principe de base** : 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 extérieur 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.

Restriction des accès

- En pratique, dans toutes les classes, **tous les attributs seront privés.**
- Exemple (classe `Compte`) : on rend 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 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)
}
```

Méthodes d'accès

- Remarque : 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 ecrire 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*.
  - *accesseur* : méthode fournissant la valeur d'un attribut.
  - *modifieur* : méthode transmettant une nouvelle valeur à affecter à un attribut.

Exemple

Exemple : liste chaînée dont on veut connaître la longueur (`int getLongueur()`).

- soit la méthode parcourt la liste chaînée à chaque fois.
- soit la méthode lit l'attribut `longueur`. Il est modifié par les autres méthodes (ajout ou suppression de chaînons).

Encapsulation "interne"

- On peut même limiter l'accès aux attributs dans la classe où ils sont déclarés : on y accède que grâce à leurs méthodes d'accès.
- **Intérêt** : si les attributs changent, seuls les méthodes d'accès doivent être modifiées.

Exemple

On veut créer une classe `Complexe`.
Un nb 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 déclare les attributs flottants `x` et `y` et les 8 méthodes : `float getX()`, `void setX(float)`, `float getY()`, `void setY(float)`, `float getRho()`, etc. Les 4 dernières simulent l'accès à ρ et θ via des calculs.
- Plus tard, on peut remplacer les attributs `x` et `y` par `rho` et `theta` ou ajouter ces deux derniers. Il faudra alors modifier les méthodes d'accès mais pas les autres méthodes de `Complexe`.

A ne plus faire

```
class A
{
    X x;
    Y y;

    void f(...){ x = ...}
    int g(...){ int a = y; ...}
    ...
}
```

On change les attributs => on change toutes les classes qui les désignent.

A faire au moins

```
class A
{
    private X x;
    private Y y;

    X getX() { return x; }
    void setX(X x) { this.x = x; }
    Y getY() { return y; }
    void setY(Y y) { this.y = y; }

    void f(...){ x = ...}
    int g(...){ int a = y; ...}
    ...
}
```

On change les attributs => on change toutes les méthodes de la classe.

A faire au mieux

```
class A
{
    private X x;
    private Y y;

    X getX() { return x; }
    void setX(X x) { this.x = x; }
    Y getY() { return y; }
    void setY(Y y) { this.y = y; }

    void f(...){ setX(...); ...}
    int g(...){ int a = getY(); ...}
    ...
}
```

On change les attributs => on ne change que les méthodes d'accès de la classe.

Héritage

- Héritage : Définition d'une nouvelle classe à partir d'une classe qui **existe déjà** en la **complétant** de nouveaux attributs et/ou de nouvelles méthodes afin qu'elle permette de créer des objets **plus spécifiques**.
- 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 remplacements.
- Vocabulaire :

  La classe B *hérite* de la classe A $\Leftrightarrow$ B *dérive* de A

  $\Leftrightarrow$ A est la *superclasse* de B $\Leftrightarrow$ A est la *classe-mère* de B $\Leftrightarrow$ B est la *sous-classe* de A $\Leftrightarrow$ B est la *classe-fille* de A.

Utilisation de l'héritage

- 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**.
  - faire hériter `Chien` d'`Animal` a un sens.
  - faire hériter `Complexe` de `Réel` est une **erreur de conception** (mauvaise idée : ajouter un attribut pour la partie imaginaire du nombre complexe et compléter les opérations).
  - faire hériter `Réel` de `Complexe` est Ok car tous les nombres réels sont des nombres complexes.
- **Ne pas confondre héritage (tout B est A) et composition (tout A possède un B, un B appartient à un A).**

Héritage en Java

- Pour déclarer une classe B qui hérite de A, on ajoute `extends A` derrière l'en-tête de la déclaration (`class B`).
- 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.

```
class A {
    int x;
    void f() { ... }
}

public class B extends A {
    int y;
    void g() { ... }
}
```

```
B b = new B();
b.x = 3;
b.y = 4;
b.f();
b.g();
```

Redéfinitions

- 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** : 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.

Exemple

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

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

    void affi che() {
        System.out.print(val + " ");
        System.out.println(val2);
    }
}

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

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

    void affi che() {
        super.affi che(); // affi che : 1 2
        System.out.println(val); // affi che : 10
        System.out.println(val3); // affi che : 3
        System.out.println(super.val); // affi che : 1
    }

    public static void main(String[] args) {
        B b = new B();
        b.salutations(); // appel méthode de A
        b.aurevoir(); // appel méthode de B
        b.affi che(); // appel méthode de B
    }
}
```

Mécanisme de l’héritage

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.**
- Le mot-clé `protected` sert à ne permettre l'accès d'un membre que depuis un **même paquetage** ou **une des sous-classes.**
  `private` < *privé-paquetage* < `protected` < `public`
- Quand une méthode est redéfinie, on ne peut changer sa visibilité que si on l'**augmente.**
- 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.**

L'héritage et les constructeurs

- La définition d'un constructeur ayant la même signature que celui de sa superclasse remplace ce dernier.
- Appel au constructeur de la superclasse grâce au mot-clé `super` suivi d'une liste de paramètres correspondant à un constructeur.
- L'appel au constructeur de la superclasse doit se faire en début de la définition du constructeur.
- Si aucun appel explicite n'est fait à un constructeur de superclasse alors appel implicite à `super();`.
- Utilisation standard d'un constructeur :
  - appel à un constructeur de la superclasse : initialisation des attributs de la superclasse.
  - initialisation des attributs déclarés dans la classe.

Le polymorphisme en Java

- Le polymorphisme est la capacité d'une variable de changer dynamiquement son type.
- En Java, on peut affecter à une variable a de type A la valeur d'une variable b de type B dérivant 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 en Java

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, cette dernière 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.

Exemple

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

a.salutations(); —–> la méthode de A est appelée
a.affi che(); ———> 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 redéfi nie 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.affi che(); ———> la méthode de B est appelée !

Intérêt du polymorphisme

- On peut stocker dans une structure de données destinée à regrouper des instances de A des objets hétéroclites dont les types dérivent de A,
- puis invoquer la même méthode sur ces objets. Ceux-ci vont alors chacun exécuter **l'opération qui correspond à leur type.**
- 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].affi che(); // la méthode défi nie 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
}
```

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.
- La conséquence principale est que **toute** variable de type `Object` peut être instanciée avec un objet de **n'importe quel type.**

Les classes abstraites

- Les *classes abstraites* sont des classes qui ne peuvent pas avoir d'instance.
- En général, elles déclarent des *méthodes abstraites* = méthodes sans corps.
- **Utilité** : **oblige** les sous-classes (concrètes) à définir (concrétement) les méthodes abstraites.

Les classes abstraites en Java

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

- On préfixe la signature de la méthode abstraite avec le mot-clé `abstract`.
- On préfixe la déclaration d'une classe abstraite avec le même mot-clé `abstract`.
- Remarques :
  - Toute classe **contenant une méthode abstraite** doit être **déclarée abstraite**.
  - Il n'est **pas nécessaire** qu'une classe contienne une méthode abstraite pour qu'on puisse la déclarer abstraite.

Classes abstraites, héritage et polymorphisn

- 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.
- On peut **déclarer une variable** dont le type est une classe abstraite.
- On peut alors lui **affecter** une instance d'une **sous-classe concrète**.

Intérêt des classes abstraites

- Les classes **abstraites** servent à modéliser des types d'objets qui n'ont **pas de réalité concrète.**
- Ex : 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 modéliser 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 :
  - méthodes **concrètes** : ce qu'ils font **tous de la même manière.**
  - méthodes **abstraites** : ce qu'ils font **chacun à leur manière.**

Exemple

- On veut modéliser un trafic routier mettant en jeu 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**.
- Les voitures et les bus peuvent **reculer**.

Exemple

```
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
    ...
}
```

Exemple

```
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
    ....
}
```

Exemple

```
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
    ...
}
```

Exemple

- Si maintenant on veut modéliser un flux de véhicules arrivant à un feu tricolore, 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.

```
MoyenDeTransport fi le[] = new MoyenDeTransport[100];
[remplissage du tableau avec des instances de Velo, Voiture, etc]
int i = 0;
while(fi le[i] != null)
    fi le[i].avance();
...
```

Les interfaces (uniquement en Java)

- Ce sont des classes abstraites particulières :
  - tous leurs attributs doivent **explicitement** être `static` et `final`.
  - 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.

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

Interface vs classe abstraite

```
interface A {
    void f();
    int g(fbat x);
}
```

```
abstract class A {
    abstract void f();
    abstract int g(fbat x);
}
```

Ecrire une interface **équivaut presque** à écrire une classe abstraite ne déclarant **que des méthodes abstraites**

sauf qu'une classe peut hériter de **plusieurs interfaces.**

Interface et héritage

- Contrairement à une classe standard, une interface peut hériter de **plusieurs** interfaces.
- On utilise le mot-clé `extends` mais en indiquant la suite des interfaces 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`.
- Une classe peut hériter d'une autre classe et de plusieurs interfaces.

```
interface Interface1 extends InterfaceMere {
    void nouvelleMethode(int x);
}
class UneClasse extends ClasseMere implements Interface1, Interface2 {
    void nouvelleMethode(int x) { ... }
    ...
}
```

Intérêt des interfaces

- 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 autres classes.

Exemple

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

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

Exemple

```
// 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
        ...
    }
    ...
}
```

Celui qui implémente la classe B ne connaît que l'interface de A.

Exemple

```
// 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();
    }
}
```

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.