Cours Java

Introduction

- Java est un langage de programmation orienté objet créé par la société « Sun Microsystems » en 1995.

- Le langage Java est un langage généraliste de programmation synthétisant les principaux langages existants.

- Il permet une programmation orientée-objet et reprend une syntaxe très proche de celle du langage C.

Introduction

- Le langage Java a l'avantage d'être:
  - ✓ **modulaire:** on peut écrire des portions de code génériques, c-à-d utilisables par plusieurs applications,
  - ✓ **Rigoureux:** la plupart des erreurs se produisent à la compilation et non à l'exécution et
  - ✓ **Portable:** un même programme compilé peut s'exécuter sur différents environnements.

- En contre-partie, les applications Java ont le défaut d'être plus lentes à l'exécution que des applications programmées en C par exemple.

1.1 Environnement Java

- **Java** est un langage interprété, ce qui signifie qu'un programme compilé n'est pas directement exécutable par le système d'exploitation mais il doit être interprété par un autre programme, qu'on appelle interpréteur.

Exemple :

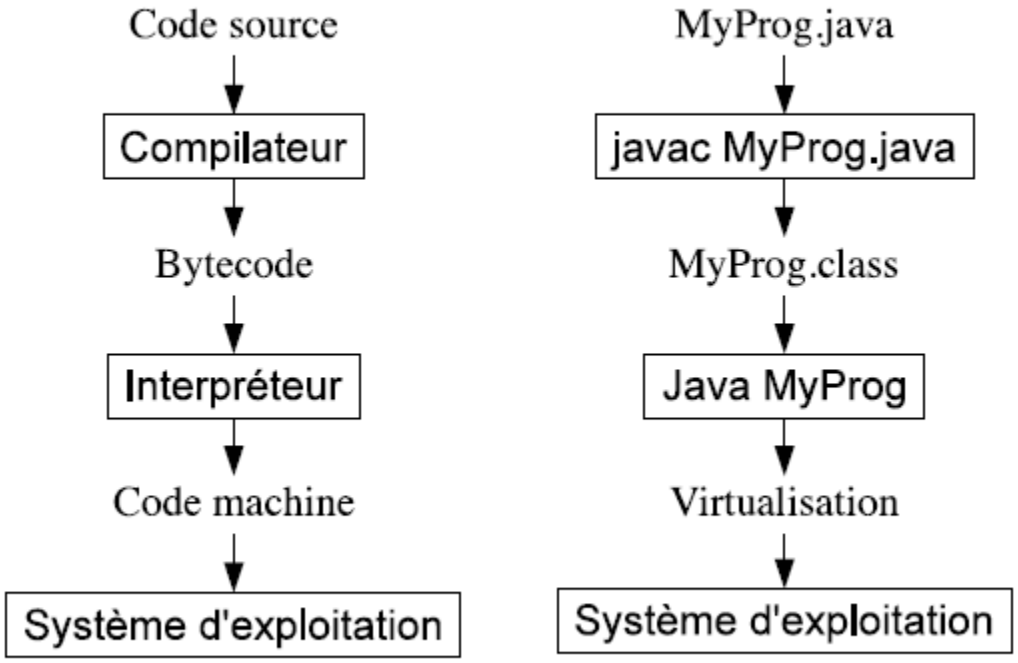

FIGURE 1.1 – Interprétation du langage

1.1 Environnement Java

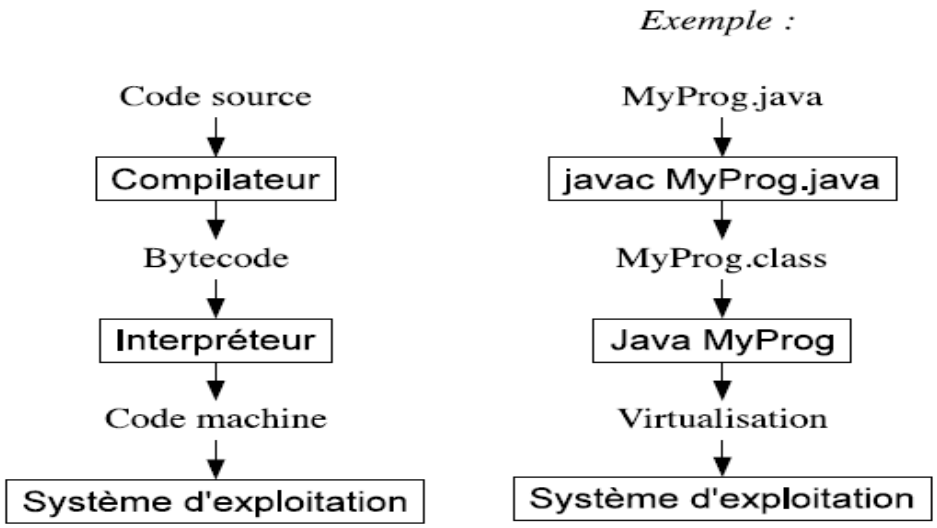

FIGURE 1.1 – Interprétation du langage

- Un développeur Java écrit son **code source**, sous la forme de **classes**, dans des fichiers dont l'extension est **.java**.
- **Le code source** est compilé par le compilateur **javac** en un langage appelé **bytecode** et enregistre le résultat dans un fichier dont l'extension est **.class**.

1.1 Environnement Java

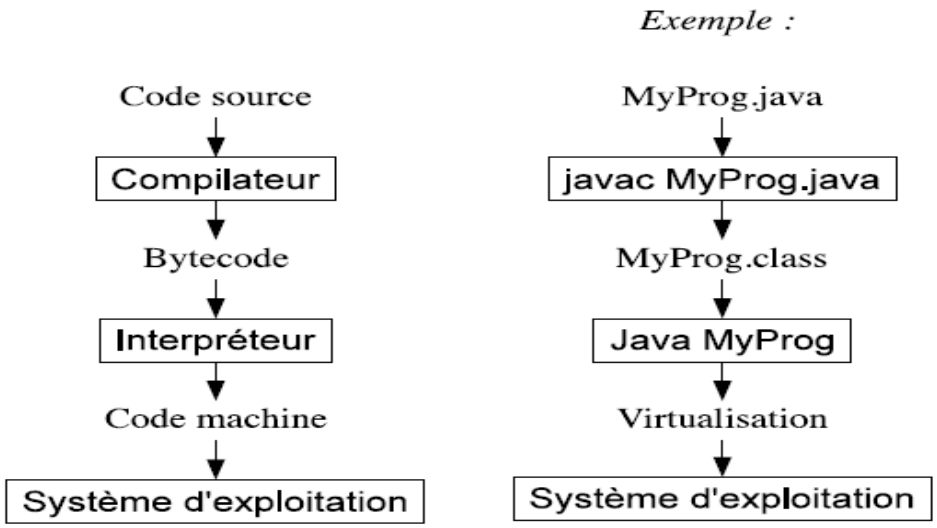

FIGURE 1.1 – Interprétation du langage

- **Le bytecode** obtenu n'est pas directement utilisable, il doit être interprété par **la machine virtuelle** de Java qui transforme alors le code compilé en code machine compréhensible par le système d'exploitation.

- Depuis 2009 la société Sun Microsystems fournit les outils de développement Java SE (Standard Edition) contenus dans le Java Development Kit (JDK).

- Dernière version stable JDK 16.0.1 (20 avril 2021).

1.1 Environnement Java

1.1.1 Compilation

- La compilation s'effectue par la commande **javac** suivie d'un ou plusieurs nom de fichiers contenant le code source de classes Java.

Exemple:

```
javac -classpath /prog/exos1:/cours MyProg.java
```

- Compilation du fichier **MyProg.java** et fait référence à d'autres classes situées dans les répertoires **/prog/exos1** et **/cours.**
- Le résultat de cette compilation est un fichier nommé **My-Prog.class** contenant le **bytecode** correspondant au source compilé.
- Il est cependant fortement souhaitable de ne pas mélanger les fichiers contenant le code source et ceux contenant le **bytecode**. Un répertoire de destination où sera créé le fichier **MyProg.class** peut être précisé par l'option **-d**, par exemple :

```
javac -d /prog/exos1 -classpath /cours MyProg.java
```

1.1 Environnement Java

1.1.2 Interprétation

- Le **bytecode** obtenu par compilation ne peut être exécuté qu'à l'aide de l'interpréteur.
- L'exécution s'effectue par la commande java suivie du nom de la classe à exécuter (sans l'extension .class).
- Comme lors de la compilation, il se peut que des classes d'autres répertoires soient nécessaires.

Exemple:

Il faut alors utiliser l'option -classpath comme dans l'exemple qui suit :

```
java -classpath /prog/exos1:/cours MyProg
```

1.2 Programmation orientée-objet

1.2.1 Classe

- Chaque langage de programmation appartient à une “famille” de langages définissant une approche ou une méthodologie générale de programmation.
- La programmation orientée-objet (introduite par le langage SmallTalk) propose une méthodologie centrée sur les données.
- Développeur Java va d’abord identifier un ensemble d’objets, tel que chaque objet représente un élément qui doit être utilisé ou manipulé par le programme, sous la forme d’ensembles de données.
- Un **objet** peut être vu comme une entité regroupant un ensemble de données et de méthodes (l’équivalent d’une fonction en C) de traitement.

1.2 Programmation orientée-objet

1.2.1 Classe

Exemple:

```
class Rectangle {

    int longueur;
    int largeur;
    int origine_x;
    int origine_y;

    void deplace(int x, int y) {
        this.origine_x = this.origine_x + x;
        this.origine_y = this.origine_y + y;
    }

    int surface() {
        return this.longueur * this.largeur;
    }
}
```

- La classe **Rectangle** utilisée pour instancier des objets représentant des rectangles, encapsule 4 entiers : la **longueur** et la **largeur** du rectangle ainsi que la position en **abscisse** et en **ordonnée** de l'origine du rectangle.

1.2 Programmation orientée-objet

1.2.1 Classe

- La classe Rectangle implémente une méthode permettant de déplacer le rectangle qui nécessite en entrée deux entiers indiquant la distance de déplacement en abscisse et en ordonnée.
- L'accès aux positions de l'origine du rectangle se fait directement (i.e. sans passage de paramètre) lorsque les données sont encapsulées dans la classe où est définie la méthode

Chapitre 2
Syntaxe du langage

2.1 Types de données

Le langage C a servi de base pour la syntaxe du langage Java :

- le caractère de fin d'une instruction est " ; "
- les commentaires (non traités par le compilateur) se situent entre les symboles "/*" et "*/"

ou commencent par le symbole "//" en se terminant à la fin de la ligne.

2.1.1 Types primitifs

TABLE 2.1 – Type primitifs de données en Java

Type	Classe éq.	Valeurs	Portée	Défaut
`boolean`	`Boolean`	`true` ou `false`	N/A	`false`
`byte`	`Byte`	entier signé	{-128..128}	o
`char`	`Character`	caractère	{\uoooo..\uFFFF}	\uoooo
`short`	`Short`	entier signé	{-32768..32767}	0
`int`	`Integer`	entier signé	{-2147483648..2147483647}	0
`long`	`Long`	entier signé	$\{-2^{31}..2^{31}-1\}$	0
`float`	`Float`	réel signé	$\{-3,4028234^{38}..3,4028234^{38}\}$ $\{-1,40239846^{-45}..1,40239846^{-45}\}$	0.0
`double`	`Double`	réel signé	$\{-1,797693134^{308}..1,797693134^{308}\}$ $\{-4,94065645^{-324}..4,94065645^{-324}\}$	0.0

2.1 Types de données

2.1.1 Tableaux et matrices

- Les deux syntaxes suivantes sont acceptées pour déclarer un tableau d'entiers (même si la première, non autorisée en C, est plus intuitive) :

```
int[] mon_tableau ;
int mon_tableau2[];
```

- Un tableau a toujours une taille fixe qui doit être précisée avant l'affectation de valeurs à ses indices.

  Exemple:

```
int[] mon_tableau = new int[20];
```

2.1 Types de données

2.1.3 Chaînes de caractères

- Les chaînes de caractères ne sont pas considérées en Java comme un type primitif ou comme un tableau.
- On utilise une classe particulière, nommée String, fournie dans le package `java.lang`.
- Les variables de type String ont les caractéristiques suivantes :
  - leur valeur ne peut pas être modifiée
  - on peut utiliser l'opérateur + pour concaténer deux chaînes de caractères :

```
String s1 = "hello" ;
String s2 = "world" ;
String s3 = s1 + " " + s2 ;
```

  - Après ces instructions **s3** vaut "**hello world**"

2.1 Types de données

2.1.3 Chaînes de caractères

- l'initialisation d'une chaîne de caractères s'écrit :

```
String s = new String(); //pour une chaine vide
String s2 = new String("hello world");
```

pour une chaîne de valeur "**hello world**"

- Il existe un ensemble de méthodes de la classe **java.lang.String** permettent d'effectuer des opérations ou des tests sur une chaîne de caractères.

2.2 Les opérateurs

TABLE 2.2 – Opérateurs Java

Pr.	Opérateur	Syntaxe	Résultat	Signification
1	++	++<ari>	<ari>	pré incrémentation
		<ari>++	<ari>	post incrémentation
	–	–<ari>	<ari>	pré décrémentation
		<ari>–	<ari>	post décrémentation
	+	+<ari>	<ari>	signe positif
	-	-<ari>	<ari>	signe négatif
	!	!<boo>	<boo>	complément logique
	(*type*)	(*type*)<val>	<val>	changement de type
2	*	<ari>*<ari>	<ari>	multiplication
	/	<ari>/<ari>	<ari>	division
	%	<ari>%<ari>	<ari>	reste de la division
3	+	<ari>+<ari>	<ari>	addition
	-	<ari>-<ari>	<ari>	soustraction
	+	<str>+<str>	<str>	concaténation
4	<<	<ent> << <ent>	<ent>	décalage de bits à gauche
	>>	<ent> >> <ent>	<ent>	décalage de bits à droite
5	<	<ari> < <ari>	<boo>	inférieur à
	<=	<ari> <= <ari>	<boo>	inférieur ou égal à
	>	<ari> > <ari>	<boo>	supérieur à
	>=	<ari> >= <ari>	<boo>	supérieur ou égal à
	instanceof	<val>instanceof<cla>	<boo>	test de type
6	==	<val>==<val>	<boo>	égal à
	!=	<val>!=<val>	<boo>	différent de
7	&	<ent>&<ent>	<ent>	ET bit à bit
		<boo>&<boo>	<boo>	ET booléen
8	^	<ent>^<ent>	<ent>	OU exclusif bit à bit
		<boo>^<boo>	<boo>	OU exclusif booléen
9	|	<ent>|<ent>	<ent>	OU bit à bit
		<boo>|<boo>	<boo>	OU booléen
10	&&	<boo>&&<boo>	<boo>	ET logique
11	||	<boo>||<boo>	<boo>	OU logique
12	?:	<boo>?<ins>:<ins>	<ins>	si...alors...sinon
13	=	<var>=<val>	<val>	assignation

Légende

<ari> valeur arithmétique
<boo> valeur booléenne
<cla> classe
<ent> valeur entière
<ins> instruction
<str> chaîne de caractères (String)
<val> valeur quelconque
<var> variable

2.2 Les structures de contrôle

Les structures de contrôle permettent d'exécuter un **bloc d'instructions soit plusieurs fois** (instructions itératives) soit selon la valeur d'une expression (instructions conditionnelles ou de choix multiple).

2.3.1 Instructions conditionnelles

Syntaxe :

```
if (<condition>) <bloc1> [else <bloc2>]
```

ou

```
<condition>?<instruction1>:<instruction2>
```

`<condition>` doit renvoyer une valeur booléenne. Si celle-ci est vraie c'est `<bloc1>` (respectivement `<instruction1>`) qui est exécuté sinon **`<bloc2>`** (respectivement **`<instruction2>`**) est exécuté. La partie **`else <bloc2>`** est facultative.

2.2 Les structures de contrôle

2.3.2 Instructions itératives

Les instruction itératives permettent d'exécuter plusieurs fois un bloc d'instructions, et ce, jusqu'à ce qu'une condition donnée soit fausse. Les trois types d'instruction itératives sont les suivantes :

- **TantQue...Faire...**

```
while (<condition>)
        <bloc>
```

Exemple :

```
while (a != b)
        a++;
```

- **Faire...Tant Que...**

```
do <bloc> while (<condition>);
```

Exemple :

```
do a++
while (a != b);
```

- **Pour...Faire**

```
for (<init>;<condition>;<instr_post_itération>)
  <bloc>
```

Exemple :

```
for (int i = 0, j = 49 ; (i < 25) && (j >= 25); i++, j--)
{
if (tab[i] > tab[j]) {
int tampon = tab[j];
```

2.2 Les structures de contrôle

2.3.3 Instructions *break* et *continue*

L'instruction break est utilisée pour sortir immédiatement d'un bloc d'instructions (sans traiter les instructions restantes dans ce bloc). Dans le cas d'une boucle on peut également utiliser l'instruction continue avec la différence suivante :

- **break :** l'exécution se poursuit après la boucle (comme si la condition d'arrêt devenait vraie) ;
- **continue :** l'exécution du bloc est arrêtée mais pas celle de la boucle. Une nouvelle itération du bloc commence si la condition d'arrêt est toujours vraie.

Exemple :

```
for (int i = 0, j = 0 ; i < 100 ; i++) {
    if (i > tab.length) {
            break ;
    }
    if (tab[i] == null) {
            continue ;
    }
        tab2[j] = tab[i];
        j++;
}
```

Chapitre 3
Éléments de programmation Java

3.1 Premiers pas

- Un programme écrit en Java consiste en un ensemble de classes représentant les éléments manipulés dans le programme et les traitements associés.
- L'exécution du programme commence par l'exécution d'une classe qui doit implémenter une méthode particulière :

```
public static void main(String[] args)
```

- Les classes **implémentant** cette méthode sont appellées **classes exécutables**.

Exemple:

```
public class HelloWorld {
        public static void main(String[] args) {
                System.out.println("Hello world");
        }
}
```

3.1 Premiers pas

Packages

- Un grand nombre de classes, fournies par Java SE, implémentent des données et traitements génériques utilisables par un grand nombre d'applications. Ces classes forment l'**API** (**A**pplication **P**rogrammer **I**nterface) du langage Java.

- Toutes ces classes sont organisées en packages (ou bibliothèques) dédiés à un thème précis. Parmi les packages les plus utilisés, on peut citer les suivants :

Package	Description
java.awt	Classes graphiques et de gestion d'interfaces
java.io	Gestion des entrées/sorties
java.lang	Classes de base (importé par défaut)
java.util	Classes utilitaires
javax.swing	Autres classes graphiques

3.1 Premiers pas

Packages

- Pour accéder à une classe d'un package donné, il faut préalablement **importer** cette classe ou son package en utilisant le mot clé « **import** ».

- Pour importer toutes les classes du package (même les classes non utilisées) on utilise le caratétre: « ***** »:

  Exemple:

```
import java.util.* ;
```

- Il est possible de créer vos propres packages en précisant, avant la déclaration d'une classe, le package auquel elle appartient.

3.2 Variables et méthodes

3.2.1 Visibilité des champs

- Dans des exemples, le mot-clé public apparaît parfois au début d'une déclaration de classe ou de méthode. Ce mot-clé autorise n'importe quel objet à utiliser la classe ou la méthode déclarée comme publique.
- La portée de cette autorisation dépend de l'élément à laquelle elle s'applique.

TABLE 3.1 – Portée des autorisations

Élément	Autorisations
Variable	Lecture et écriture
Méthode	Appel de la méthode
Classe	Instanciation d'objets de cette classe et accès aux variables et méthodes de classe

3.2 Variables et méthodes

3.2.1 Visibilité des champs

- Le mode public n'est, bien sûr, pas le seul type d'accès disponible en Java. Deux autres mot-clés peuvent être utilisés en plus du type d'accès par défaut : **protected** et **private**.
- Si aucun mot-clé ne précise le type d'accès, celui par défaut est appliqué.
- Le tableau suivant récapitule ces différents types d'accès:

TABLE 3.2 – Autorisations d'accès

	`public`	`protected`	défaut	`private`
Dans la même classe	Oui	Oui	Oui	Oui
Dans une classe du même package	Oui	Oui	Oui	Non
Dans une sous-classe d'un autre package	Oui	Oui	Non	Non
Dans une classe quelconque d'un autre package	Oui	Non	Non	Non

3.2 Variables et méthodes

3.2.2 Variables et méthodes de classe

- Dans certains cas, il est plus judicieux d'attacher une variable ou une méthode à une classe plutôt qu'aux objets instanciant cette classe. Par exemple, la classe java.lang.Integer possède une variable MAX_VALUE qui représente la plus grande valeur qui peut être affectée à un entier.
- Cette variable étant commune à tous les entiers, elle n'est pas dupliquée dans tous les objets instanciant la classe Integer mais elle est associée directement à la classe Integer. Une telle variable est appelée **variable de classe.**
- De la même manière, il existe des **méthodes de classe** qui sont associées directement à une classe.
- Pour déclarer une variable ou méthode de classe, on utilise le mot-clé static qui doit être précisé avant le type de la variable ou le type de retour de la méthode.

3.2 Variables et méthodes

3.2.2 Variables et méthodes de classe

- La classe ***java.lang.Math*** nous fournit un bon exemple de variable et de méthodes de classes.

```
public final class Math {
    ...
    public static final double PI = 3.14159265358979323846 ;
    ...
    public static double toRadians(double angdeg) {
            return angdeg / 180.0 * PI ;
}
...
}
```

- La classe Math fournit un ensemble d'outils (variables et méthodes) très utiles pour des programmes devant effectuer des opération mathématiques complexes.
- Dans la portion de classe reproduite ci-dessus, on peut notamment y trouver une approximation de la valeur de π et une méthode convertissant la mesure d'un angle d'une valeur en degrés en une valeur en radians.

Chapitre 3 : Héritage

4.1 Principe d'héritage

- Dans certaines applications, les classes utilisées ont en **commun** certaines **variables**, **méthodes** de traitement ou même des signatures de méthode.

- Avec un langage de programmation orienté objet, on peut définir une classe à différent niveaux d'abstraction permettant ainsi de factoriser certains attributs communs à plusieurs classes.

- Une classe générale définit alors un ensemble d'attributs qui sont partagés par d'autres classes, dont on dira qu'elles ***héritent*** de cette classe générale.

4.1 Principe d’héritage

Exemple:

Les classes **Carre** et **Rectangle** peuvent **partager** une méthode **surface**() renvoyant le résultat du calcul de la surface de la figure. Plutôt que d’écrire **deux fois** cette méthode, on peut définir une relation d’héritage entre les classes Carre et Rectangle.

Dans ce cas, seule la classe Rectangle contient le code de la méthode surface() mais celle-ci est également utilisable sur les objets de la classe Carre si elle hérite de Rectangle.

4.1 Principe d'héritage

- L'idée principale de l'héritage est d'organiser les classes de manière hiérarchique.

- La relation d'héritage est unidirectionnelle et, si une classe B hérite d'une classe A, on dira que B est une sous-classe de A.

- Cette notion de sous-classe signifie que la classe B est un cas particulier de la classe A et donc que les objets instanciant la classe B instancient également la classe A.

4.1 Principe d'héritage

Exemple:

La figure ci-dessous propose une organisation hiérarchique de ces classes **Carre**, **Rectangle** et **Cercle,** telle que Carre hérite de **Rectangle** qui hérite, ainsi que **Cercle**, d'une classe Forme.

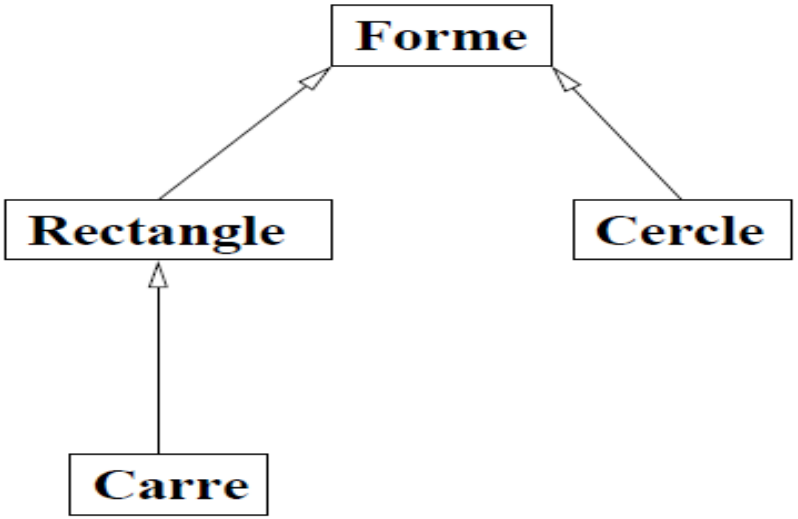

FIGURE 4.1 – Exemple de relations d'héritage

4.1 Principe d'héritage

Exemple:

```
public class Rectangle extends Forme {

   private int largeur;
   private int longueur;

   public Rectangle(int x, int y) {
      this.largeur = x;
      this.longueur = y;
   }

   public int getLargeur() {
      return this.largeur;
   }

   public int getLongueur() {
      return this.longueur;
   }

   public int surface() {
      return this.longueur * this.largeur;
   }

   public void affiche() {
      System.out.println("rectangle " + longueur + "x" + largeur);
   }
}
```

4.1 Principe d'héritage

Exemple:

- la classe **Carre** peut bénéficier de la classe **Rectangle** et ne nécessite pas la réécriture de ces méthodes si celles-ci conviennent à la sous-classe.
- Toutes les **méthodes** et **variables** de la classe Rectangle ne sont néanmoins pas accessibles dans la classe Carre.
- Pour qu'un attribut puisse être utilisé dans une sous-classe, il faut que son type d'accès soit **public** ou **protected**, ou, si les deux classes sont situées dans le **même package**, qu'il utilise le type d'accès par défaut.
- Dans cet exemple, les variables longueur et largeur ne sont pas accessibles dans la class Carre qui doit passer par les méthodes getLargeur() et getLongueur(), déclarées comme publiques.

4.1 Principe d'héritage

4.1.1 Redéfinition

- L'héritage intégral des attributs de la classe Rectangle pose deux problèmes :
  1. il faut que chaque carré ait une longueur et une largeur égales ;
  2. la méthode affiche écrit le mot "rectangle" en début de chaîne. Il serait souhaitable que ce soit "carré" qui s'affiche.
- Les constructeurs ne sont pas hérités par une sous-classe. Il faut donc écrire un constructeur spécifique pour Carre. Ceci nous permettra de résoudre le premier problème en écrivant un constructeur qui ne prend qu'un paramètre qui sera affecté à la longueur et à la largeur. Pour attribuer une valeur à ces variables (qui sont privées), le constructeur de Carre doit faire appel au constructeur de Rectangle en utilisant le mot-clé super qui fait appel au constructeur de la classe supérieure comme suit :

```
public Carre(int cote) {
        super(cote,cote);
}
```

4.1 Principe d'héritage

4.1.1 Redéfinition

- Le second problème peut être résolu par une redéfinition de méthode.
- On dit qu'une méthode d'une sous-classe redéfinit une méthode de sa classe supérieure, si elles ont la même signature mais que le traitement effectué est ré-écrit dans la sous-classe.
- Le code de la classe Carre où sont résolus les deux problèmes soulevés :

```
public class Carre extends Rectangle {

   public Carre(int cote) {
      super(cote, cote);
   }

   public void affiche() {
      System.out.println("carré " + this.getLongueur());
   }
}
```

4.1 Principe d'héritage

4.1.1 Redéfinition

- Lors de la redéfinition d'une méthode, il est encore possible d'accéder à la méthode redéfinie dans la classe supérieure. Cet accès utilise également le mot-clé **super** comme préfixe à la méthode.
- Dans notre cas, il faudrait écrire **super.affiche()** pour effectuer le traitement de la méthode affiche() de Rectangle.

```
public class Carre extends Rectangle {

   public Carre(int cote) {
      super(cote, cote);
   }

   public void affiche() {
      System.out.println("carré " + this.getLongueur());
   }
}
```

4.1 Principe d'héritage

4.1.1 Redéfinition

- il est possible d'interdire la redéfinition d'une méthode ou d'une variable en introduisant le mot-clé **final** au début d'une signature de méthode ou d'une déclaration de variable.
- Il est aussi possible d'interdire l'héritage d'une classe en utilisant **final** au début de la déclaration d'une classe (avant le mot-clé class).

```
public class Carre extends Rectangle {

   public Carre(int cote) {
      super(cote, cote);
   }

   public void affiche() {
      System.out.println("carré " + this.getLongueur());
   }
}
```

4.1 Principe d'héritage

4.1.2 Polymorphisme

- Le polymorphisme est la faculté attribuée à un objet d'être une instance de plusieurs classes.
- Il a une seule classe "réelle" qui est celle dont le constructeur a été appelé en premier (c'est-à-dire la classe figurant après le new) mais il peut aussi être déclaré avec une classe supérieure à sa classe réelle.
- Cette propriété est très utile pour la création d'ensembles regroupant des objets de classes différentes comme dans l'exemple suivant :

```
Forme[] tableau = new Forme[4];
tableau[0] = new Rectangle(10,20);
tableau[1] = new Cercle(15);
tableau[2] = new Rectangle(5,30);
tableau[3] = new Carre(10);
```

4.1 Principe d'héritage

4.1.2 Polymorphisme

- L'opérateur **instanceof** peut être utilisé pour tester l'appartenance à une classe comme suit :

```
for (int i = 0 ; i < tableau.length ; i++) {
   if (tableau[i] instanceof Forme)
      System.out.println("element " + i + " est une forme");
   if (tableau[i] instanceof Cercle)
      System.out.println("element " + i + " est un cercle");
   if (tableau[i] instanceof Rectangle)
      System.out.println("element " + i + " est un rectangle");
   if (tableau[i] instanceof Carre)
      System.out.println("element " + i + " est un carré");
}
```

- L'exécution de ce code sur le tableau précédent affiche le texte suivant :

```
element[0] est une forme
element[0] est un rectangle
element[1] est une forme
element[1] est un cercle
element[2] est une forme
element[2] est un rectangle
element[3] est une forme
element[3] est un rectangle
element[3] est un carré
```

4.1 Principe d'héritage

4.1.2 Polymorphisme

- L'ensemble des classes Java, y compris celles écrites en dehors de l'API, forme une hiérarchie avec une racine unique. Cette racine est la classe Object dont hérite toute autre classe.
- En effet, si vous ne précisez pas explicitement une relation d'héritage lors de l'écriture d'une classe, celle-ci hérite par défaut de la classe Object.
- Grâce à cette propriété, des classes génériques de création et de gestion d'un ensemble, plus élaborées que les tableaux, regroupent des objets appartenant à la classe Object (donc de n'importe quelle classe).
- Une des propriétés induites par le polymorphisme est que l'interpréteur Java est capable de trouver le traitement à effectuer lors de l'appel d'une méthode sur un objet.

4.1 Principe d'héritage

4.1.2 Polymorphisme

- Ainsi, pour plusieurs objets déclarés sous la même classe (mais n'ayant pas la même classe réelle), le traitement associé à une méthode donné peut être différent.
- Si cette méthode est redéfinie par la classe réelle d'un objet (ou par une classe située entre la classe réelle et la classe de déclaration), le traitement effectué est celui défini dans la classe la plus spécifique de l'objet et qui redéfinie la méthode.
- Dans notre exemple, la méthode **affiche()** est redéfinie dans toutes les sous-classes de Forme et les traitements effectués sont

```
for (int i = 0 ; i < tableau.length ; i++) {
   tableau[i].affiche();
}
```

Résultat :

```
rectangle 10x20
cercle 15
rectangle 5x30
carré 10
```

4.1 Principe d'héritage

4.1.2 Polymorphisme

- Dans l'état actuel de nos classes, ce code ne pourra cependant pas être compilé. En effet, la fonction **affiche()** est appelée sur des objets dont la classe déclarée est Forme mais celle-ci ne contient aucune fonction appelée affiche() (elle est seulement définie dans ses sous-classes).

- Pour compiler ce programme, il faut transformer la classe Forme en une interface ou une classe abstraite tel que cela est fait dans les sections suivantes.

```
for (int i = 0 ; i < tableau.length ; i++) {
   tableau[i].affiche();
}
```

Résultat :

```
rectangle 10x20
cercle 15
rectangle 5x30
carré 10
```

4.2 Interfaces

- Une interface est un type, au même titre qu'une classe, mais abstrait et qui donc ne peut être instancié (par appel à new plus constructeur).
- Une interface décrit un ensemble de signatures de méthodes, sans implémentation, qui doivent être implémentées dans toutes les classes qui *implémentent* l'interface.
- L'utilité du concept d'interface réside dans le regroupement de plusieurs classes, tel que chacune implémente un ensemble commun de méthodes, sous un même type.

4.2 Interfaces

- Une interface possède les caractéristiques suivantes :
  - elle contient des signatures de méthodes ;
  - elle ne peut pas contenir de variables ;
  - une interface peut hériter d'une autre interface (avec le mot-clé **extends**) ;
  - une classe (abstraite ou non) peut implémenter plusieurs interfaces. La liste des interfaces implémentées doit alors figurer après le mot-clé implements placé dans la déclaration de classe, en séparant chaque interface par une virgule.

4.2 Interfaces

- Dans notre exemple, **Forme** peut être une **interface** décrivant les méthodes qui doivent être implémentées par les classes **Rectangle** et **Cercle**, ainsi que par la classe Carre (même si celle-ci peut profiter de son héritage de Rectangle).
- L'interface Forme s'écrit alors de la manière suivante :

```
public interface Forme {
    public int surface() ;
    public void affiche() ;
}
```

- Pour obliger les classes **Rectangle**, **Cercle** et **Carre** à implémenter les méthodes **surface**() et **affiche**(), il faut modifier l'héritage de ce qui était la classe **Forme** en une implémentation de l'interface définie ci-dessus :

```
public class Rectangle implements Forme {
    ...
}
```

et

```
public class Cercle implements Forme {
    ...
}
```

4.3 Classes abstraites

- Le concept de classe abstraite se situe entre celui de classe et celui d'interface. C'est une classe qu'on ne peut pas directement instancier car certaines de ses méthodes ne sont pas implémentées.
- Une classe abstraite peut donc contenir des variables, des méthodes implémentées et des signatures de méthode à implémenter.
- Une classe abstraite peut implémenter (partiellement ou totalement) des interfaces et peut hériter d'une classe ou d'une classe abstraite.
- Le mot-clé **abstract** est utilisé devant le mot-clé class pour déclarer une classe abstraite, ainsi que pour la déclaration de signatures de méthodes à implémenter.

4.3 Classes abstraites

- Imaginons que l'on souhaite attribuer deux variables, **origine_x** et **origine_y**, à tout objet représentant une forme. Comme une interface ne peut contenir de variables, il faut transformer Forme en classe abstraite comme suit :

```
public abstract class Forme {
    private int origine_x;
    private int origine_y;

    public Forme() {
        this.origine_x = 0;
        this.origine_y = 0;
    }

    public int getOrigineX() {
        return this.origine_x;
    }

    public int getOrigineY() {
        return this.origine_y;
    }

    public void setOrigineX(int x) {
        this.origine_x = x;
    }

    public void setOrigineY(int y) {
        this.origine_y = y;
    }

    public abstract int surface();

    public abstract void affiche();
}
```

```
public class Rectangle extends Forme {
    ...
}
```

et

```
public class Cercle extends Forme {
    ...
}
```

4.3 Classes abstraites

Lorsqu'une classe hérite d'une classe abstraite, elle doit :

- soit implémenter les méthodes abstraites de sa **super-classe** en les dotant d'un corps ;
- soit être elle-même abstraite si au moins une des méthodes abstraites de sa **super-classe** reste abstraite.

4.4 Classes et méthodes génériques

- Il est parfois utile de définir des classes paramétrées par un type de données (ou une classe).
- Par exemple, dans le package **java.util**, de nombreuses classes sont génériques et notamment les classes représentant des ensembles (Vector, ArrayList, etc.). Ces classes sont génériques dans le sens où elles prennent en paramètre un type (classe ou interface) quelconque E. E est en quelque sorte une variable qui peut prendre comme valeur un type de donné. Ceci se note comme suit, en prenant l'exemple de **java.util.ArrayList:**

```
package java.util ;

public class ArrayList<E>
      extends AbstractList<E>
      implements List<E>, ...
{
   ...
   public E set(int index, E element) {
      ...
   }

   public boolean add(E e) {
      ...
   }
   ...
}
```

4.4 Classes et méthodes génériques

```
package java.util ;

public class ArrayList<E>
      extends AbstractList<E>
      implements List<E>, ...
{
   ...
   public E set(int index, E element) {
      ...
   }

   public boolean add(E e) {
      ...
   }
   ...
}
```

- Nous pouvons remarquer que le type passé en paramètre est noté entre chevrons (ex : <E>), et qu'il peut ensuite être réutilisé dans le corps de la classe, par des méthodes (ex : la méthode set renvoie un élément de classe E).

4.4 Classes et méthodes génériques

- Il est possible de définir des contraintes sur le type passé en paramètre, comme par exemple une contrainte de type **extends** :

```
public class SortedList<T extends Comparable<T>> {
    ...
}
```

- Ceci signifie que la classe **SortedList** (liste ordonnée que nous voulons définir) est paramétrée par le type T qui doit être un type dérivé (par héritage ou interfaçage) de **Comparable<T>**.
- En bref, nous définissons une liste ordonnée d'éléments comparables entre eux (pour pouvoir les trier), grâce à la méthode int **compareTo(T** o) de l'interface **Comparable** qui permet de comparer un **Comparable** à un élément de type T.