6 soirs - le 7ème est pour la présentation

Modalité de l'examen : Juger sur projet

1. Introduction Java
2. Introduction IHM: Swing(composants, conteneurs, layouts),[réflexion projet]
3. Les évènements, [Cahier des charges]
4. Boites de dialogues, [Projets]
5. MVC, [Projets]
6. Composants complexes, [Projets]
7. Présentations

INTRODUCTION AU LANGAGE JAVA

Internet

- https://www.oracle.com/java/technologies/downloads/?er=221886 jdk : Java dévloppment Kit
- https://docs.oracle.com/javase/tutorial/
- https://docs.oracle.com/javase/tutorial/getStarted/cupojava/index.html

Introduction : avantages

Le langage Java est un langage capable de s'exécuter sur n'importe quelle plate-forme. Il est semi-compilé, semi-interprété.

Le code source Java est transformé en de simples instructions binaires.
(Byte Code= Instructions générées par le compilateur qu'un ordinateur abstrait peut exécuter).

Introduction : avantages

Robuste et sûr :

- Pas de pointeurs.
- Compilateur très strict car toutes les valeurs doivent être initialisées.
- Traitement des exceptions obligatoire.
- Les erreurs à l'exécution sont vérifiées tout comme les limites des tableaux.

Introduction : avantages

Sécurisé :

Désallocation de la mémoire grâce au ***Garbage Collector*** (ramasse miettes) qui permet un accroissement de la sécurité en résolvant les problèmes liés aux allocations et à la libération explicite de mémoire.

Quand on lance java, il y a plein de chose qui se lancent, y compris le ramasse-miette - on n'a pas besoin de s'en occuper, on peut le forcer, mais il fait son boulot tout seul (il nettoie la mémoire)

Thread : on lance le programme qui fait comme un fil, mais en parallèle il y a plein de chose qui s'exécutent (garbage collector) - on doit retenir qu'il y a plein de choses qui tournent en même temps.

Introduction : avantages

Portable :

Le compilateur Java génère du *byte-code.* Celui-ci constitue les instructions pour la machine virtuelle JVM). Java Virtual Machine (certaine gratuite, ou Open JDK)

La **J**ava **V**irtual **M**achine (JVM) est disponible gratuitement sur le site de Oracle, et ceci, pour la plupart des plateformes (Linux, Windows, Mac, Solaris…).

Le code ne dépend pas de l'OS, mais de son type à lui, l'Unicode (64 bits)

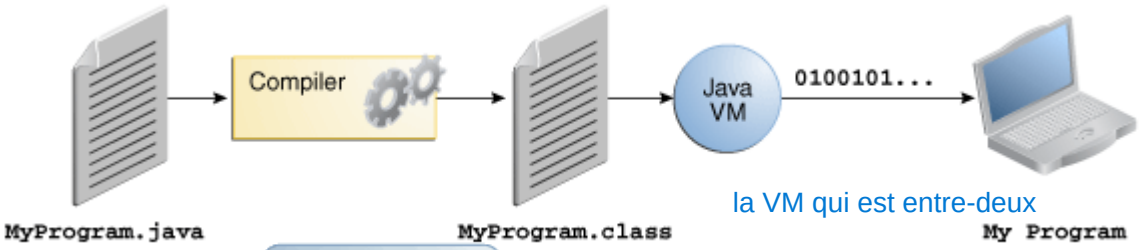
La taille des types primitifs est indépendante de la plateforme. Java supporte un code source écrit en Unicode. (Code Universel). Java est accompagné d'une librairie standard, l'API Java, téléchargeable sur le site de Oracle.

Java VS Python

- *Java est un langage **compilé** à **typage statique** (statically typed),*
- *tandis que Python est un langage **interprété** à **typage dynamique**.*
- *Cela signifie notamment que Java nécessite une compilation préalable du code et impose de préciser le type de chaque variable, alors que Python exécute le code directement et détermine les types à l'exécution*

il faut définir pour chaque variable quel type on veut. dans python, on a pas besoin de dire que a = int(5) il sait que c'est int, Java pas.

Introduction : compilation, exécution



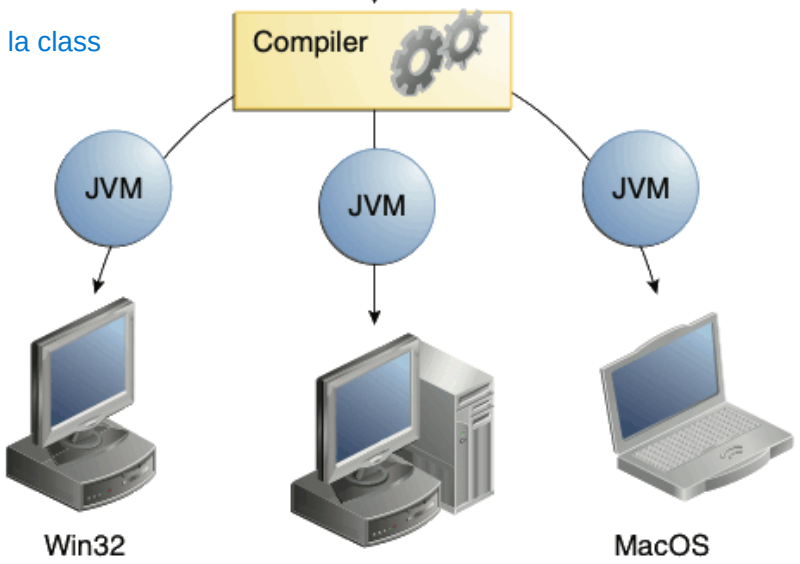
la VM qui est entre-deux

Java Program

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

HelloWorldApp.java

JDK est entre le Java Program et la class



Introduction : premier exemple

```
public class Exemple00
{
  public static void main(String args[])
  {
   String s = "Hello World\n";
   System.out.println(s);
  }
}
```

Version python:

```
print(" Hello World\n!")
```

on a créer un modèle. A partir de ce modèle, je peux créer des objets.

P.ex new_Exemple00

on peut faire un test pour voir si on a le JDK installé :
/programmes/java/jdk....
le jdl c'est le runtime
Javac = Java Compiler

Introduction : premier exemple

- Compiler le programme :
  - javac Exemple00.java
- Exécuter le programme :
  - java Exemple00

Introduction : exercice 1

- Éditer
- Compiler
- Exécuter

le programme *Exemple00*

Série 1

Introduction : caractéristiques

- Caractéristiques
- Présentations
  - JRE Machine virtuelle - il va executer le programme en code machine
  - JDK
  - Versions de Java

1.0 = 1990 ~
21.0 = aujourd'hui

La syntaxe : les bases

- Les identificateurs
- Les mots réservés
- Les commentaires
  - Les commentaires explicatifs
  - Les commentaires Javadoc
  - Les tags

La syntaxe : types primitifs et définitions de variables

tout ce qui est minuscule est objet dans Java

- Les types de base (primitifs)
  - Déclaration et initialisation
  - Les entiers
  - Les nombres flottants
  - Les caractères
- Les types références
- Les wrappers
- Les méthodes
- La méthode *main(…)*

Important : c'est tout ce qui n'est pas object . Dans Java tout est objet, sauf les primitifs

Int - Bool -

enveloppe un Int par ex. On va voir dans les exercices

La syntaxe : les instructions

ça ressemble à Python - il y a des différences

- Instructions simples
- Expressions conditionnelles
  - *if / else*
  - *switch*
- Itérations
  - L'itérateur *while*
  - L'itérateur *do / while*
  - L'itérateur *for*

La syntaxe : les instructions

– https://docs.oracle.com/javase/tutorial/java/nutsandbolts/if.html

La syntaxe : les opérateurs

- https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html

Java VS Python

Java : Déclaration avec types explicites et affectations initiales *:*

```
int age = 20;
double prix = 4.99;
String salut = "Bonjour";
age = 21;          // OK, 'age' reste un entier
// age = "vingt";  // Erreur de compilation : type incompatible
```

.

Python : Création de variables par affectation libre :

```
age = 20
prix = 4.99
salut = "Bonjour"
age = 21        # OK, 'age' référence maintenant un nouvel
                  entier
age = "vingt"   # OK en syntaxe, 'age' référence maintenant
                  une chaîne
```

La syntaxe : les opérateurs

- Les différentes catégories d'opérateurs
  - Opérateurs d'affectation
  - Opérateurs logiques
  - Opérateurs de comparaison
  - Opérateurs arithmétiques
  - Opérateurs binaires
  - L'opérateur conditionnel ternaire

Séries 2, 3

La syntaxe : exercices 2 et 3

- Complément pour les séries d'exercices 2 et 3 :

Dans le dossier *Corrigés\SimpleDataInput*
vous trouverez les deux fichiers :

SimpleDataInput.java
et
SimpleDataInput.class

Copiez le fichier .class dans votre dossier d'exercice.
Vous pouvez alors utiliser les méthodes statiques suivantes :

Java VS Python

- **Java :** On utilise le mot-clé `if`, suivi de la condition entre parenthèses `(...)`. Le bloc d'instructions à exécuter si la condition est vraie est placé entre accolades `{`

```
if (x > 0) {
    System.out.println("x est positif");
} else if (x < 0) {
    System.out.println("x est négatif");
} else {
    System.out.println("x vaut 0");
}
```

Explication : En Java, les parenthèses autour de `x > 0` sont obligatoires, tout comme les accolades délimitant chaque bloc. Le mot-clé `else if` (en deux mots) permet de chaîner une seconde condition si la première est fausse. Remarquons qu'en Java, la condition dans un `if` **doit** être de type booléen explicite – on ne peut pas mettre par exemple un entier tout seul (en C, 0 ou non-0 pourrait servir de faux/vrai, mais pas en Java). Seul un `boolean` évalué à true ou false est accepté.

Java VS Python

- **Python :** On utilise le mot-clé `if` suivi de la condition sans parenthèses, puis on termine la ligne par `:`. Le bloc conditionnel est indenté (généralement par 4 espaces). La clause sinon s'écrit avec le mot-clé `else:` (et `elif:` pour "else if"). Exemple équivalent en Python :

```
if x > 0:
    print("x est positif")
elif x < 0:
    print("x est négatif")
else:
    print("x vaut 0")
```

- *Explication :* En Python, l'indentation remplace les accolades pour englober le bloc de chaque branche de la condition. On note que le mot-clé est `elif` (contracté) au lieu de `else if`. Il n'y a pas de point-virgule en fin de ligne. La condition peut être toute expression évaluant à un booléen. (Techniquement, Python permet aussi d'utiliser n'importe quel objet dans une condition – par exemple un nombre, une liste, etc.

La syntaxe : SimpleDataInput

```
public static double readDouble()
public static int readInt()
public static long readLong()
public static float readFloat()
public static char readChar()
public static String readString()
```

La syntaxe : les tableaux

Ils sont dérivés de la classe Object : il faut utiliser des méthodes pour y accéder dont font partie des messages de la classe Object tel que equals() ou getClass().

https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html

Le premier élément d'un tableau possède l'indice 0.

comme dans Python, l'index commence toujours par 0

La déclaration des tableaux

Java permet de placer les crochets après ou avant le nom du tableau dans la déclaration.

La syntaxe : exemple

```
// déclaration et allocation
int tableau[] = new int[50];
```

Ou — on veut un tableur de 50 int

```
int[] tableau = new int[50];
```

Ou — on déclare un tableau et voilà son nom

```
int tab[];   // déclaration
tab = new int[50]; //allocation
```

on a pas les méthodes pour

tab25 = valeur de la case 25. on ne peut pas mélanger les types dans un tableau

La syntaxe : les tableaux

Java ne supporte pas directement les tableaux à plusieurs dimensions : il faut déclarer un tableau de tableau.

Exemple :

```
float tableau[][] = new float[10][10];
```

La taille des tableaux de la seconde dimension peut ne pas être identique pour chaque occurrence.

Exemple :

```
int dim1[][] = new int[3][];
dim1[0] = new int[4];
dim1[1] = new int[9];
dim1[2] = new int[2];
```

La syntaxe : les tableaux

Chaque élément du tableau est initialisé selon son type par l'instruction **`new`** :

0	pour les numériques,
'\0'	pour les caractères,
false	pour les booléens et
null	pour les chaînes de caractères et les autres objets.

La syntaxe : les tableaux

L'initialisation explicite d'un tableau

Exemple :

```
int tableau[5] = {10,20,30,40,50};
int tableau[3][2] = {{5,1},{6,2},{7,3}};
```

La taille du tableau n'est pas obligatoire si le tableau est initialisé à sa création.

Exemple :

```
int tableau[] = {10,20,30,40,50};
```

La syntaxe : les tableaux

Le nombre d'éléments de chaque ligne peut ne pas être identique :

Exemple :

si on laisse les crochets vide, on est pas limité dans le nombre de valeur par ligne

```
int[][] tabEntiers = {
        {1,2,3,4,5,6},
        {1,2,3,4},
        {1,2,3,4,5,6,7,8,9}
              };
```

La syntaxe : les tableaux

Le parcours d'un tableau

Exemple :

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

La variable **length** retourne le nombre d'éléments du tableau.

Pour passer un tableau à une méthode, il suffit de déclarer les paramètres dans l'en tête de la méthode

Exemple :

```
public void printArray(String texte[])
{ ...}
```

La syntaxe : les tableaux

Les tableaux sont toujours transmis par référence puisque ce sont des objets.

Un accès à un élément d'un tableau qui dépasse sa capacité, lève une exception du type

si on veut atteindre un chiffre qui n'est pas dans le tableau, par ex . l'index 21 alors qu'il n'y a que 20 éléments

```
java.lang.arrayIndexOutOfBoundsException.
```

La syntaxe : exemple

```
try
{
   j = tab_int[tab_int.length];
    // lève IndexOutOfBoundsException
}
catch (ArrayIndexOutOfBoundsException e)
{
   System.out.println
         ("Indice incorrect " + e);
}
```

exception est la mère des toutes les exceptions

c'est comme except as dans python

Séries 4, 5

Java VS Python

- **Java :**

```
int[] notes = {10, 12, 15};
System.out.println(notes[0]);        // affiche 10
notes[1] = 14;                       // modifie le deuxième
                                      élément
System.out.println(notes.length);    // affiche 3 (taille
                                      du tableau)
for (int note : notes) {
    System.out.println(note);        // parcourt et affiche
                                      chaque note
}
```

Java VS Python

- **Python :**

```
notes = [10, 12, 15]
print(notes[0])          # affiche 10
notes[1] = 14            # modifie le deuxième élément
print(len(notes))        # affiche 3 (taille de la liste)
for note in notes:
    print(note)          # parcourt et affiche chaque note
```

La syntaxe : chaînes de caractères

La définition d'un caractère se fait grâce au type char :

Exemple :

```
char touche = '%';
```

La définition d'une chaîne se fait grâce à l'objet String :

Exemple :

```
String texte = "bonjour";
```

Les variables de type String sont des objets. Partout ou des constantes chaînes de caractères figurent entre guillemets, le compilateur Java génère un objet de type String avec le contenu spécifié.

La syntaxe : chaînes de caractères

Les chaînes de caractères ne sont pas des tableaux : il faut utiliser les méthodes de la classe **`String`** d'un objet instancié pour effectuer des manipulations.

un String est immuable

Il est impossible de modifier le contenu d'un objet **`String`** construit à partir d'une constante. Cependant, il est possible d'utiliser les méthodes qui renvoient une chaîne pour modifier le contenu de la chaîne.

Exemple :

```
String texte = "Java Java Java";
texte = texte.replace('a','o');
```

La syntaxe : chaînes de caractères

Java utilise Unicode, un standard universel de codage des caractères. Les types char et String représentent des caractères Unicode selon l'encodage UTF-16 :

- Un char occupe 2 octets (16 bits).
- Il peut représenter un caractère Unicode dans la plage de base (BMP).
- Certains caractères spéciaux (émoticônes, idéogrammes rares, etc.) nécessitent deux char (appelés surrogate pairs).
- Les caractères Unicode 0 à 127 sont identiques à ceux du standard ASCII.
- Les codes **128 à 255** ne correspondent pas directement à un “ASCII étendu” universel :
  - Ils varient selon les anciens encodages (ISO-8859-1, Windows-1252, etc.).
  - Unicode leur attribue des correspondances spécifiques, mais **pas équivalentes à tous les jeux ANSI**.

La syntaxe : caractères spéciaux

Les caractères spéciaux dans les chaînes

Caractères spéciaux	Affichage
\'	Apostrophe
\"	Guillemet
\\	anti slash
\t	Tabulation
\b	retour arrière (backspace)
\r	retour chariot
\f	saut de page (form feed)
\n	saut de ligne (newline)
\xxx	caractère ASCII xxx (octal)(obsolète)
\uxxxx	caractère Unicode en hexadécimal (valide sur 4 chiffres)

La syntaxe : l'addition de chaînes

Java admet l'opérateur + comme opérateur de concaténation de chaînes de caractères.

L'opérateur + permet de concaténer plusieurs chaînes. Il est possible d'utiliser l'opérateur +=

Exemple :

```
String texte = "";
texte += "Hello";
texte += "World3";
```

avec le + ça permet d'ajouter les éléments à la variable

La syntaxe : l'addition de chaînes

L'opérateur + sert aussi à concaténer des chaînes avec tous les types de bases. La variable ou constante est alors convertie en chaîne et ajoutée à la précédente. La condition préalable est d'avoir au moins une chaîne dans l'expression sinon le signe + est évalué comme opérateur mathématique.

Exemple :

```
System.out.println
      ("La valeur de Pi est : "+Math.PI);
int duree = 121;
System.out.println("durée = " + duree);
```

tout ça devient une chaine de caractère. il n'y a plus la valeur

La syntaxe : la comparaison de deux chaînes

Il faut utiliser la méthode equals()

Exemple :

```
String texte1 = "texte 1";
String texte2 = "texte 2";
if (texte1.equals(texte2)) ...
```

equals permet de comparer des valeurs. == va comparer les adressages. Dans tous les langages c'est mieux d'utiliser equals

La syntaxe : longueur d'une chaîne

La méthode **`length()`** permet de déterminer la longueur d'une chaîne.

Exemple :

```
String texte = "texte";
int longueur = texte.length();
```

longueur ← 5

La syntaxe : casse d'une chaîne

Les méthodes Java **`toUpperCase()`** et **`toLowerCase()`** permettent respectivement d'obtenir une chaîne tout en majuscule ou

tout en minuscule.

Exemple :

```
String texte = "texte";
String
textemaj=texte.toUpperCase();
```

La syntaxe : autres méthodes

Autres méthodes utiles, en plus de length() et equals(String)

String toUpperCase()
renvoie la chaîne de caractères convertie en majuscules
String toLowerCase()
renvoie la chaîne de caractères convertie en minuscules
char charAt(int)
renvoie le nième caractère de la chaîne
int compareTo(String)
compare la chaîne avec l'argument
boolean endsWith(String)
vérifie si la chaîne se termine par l'argument
int indexOf(String)
renvoie la position à laquelle se trouve l'argument dans la chaîne
String replace(char, char)
renvoie la chaîne dont les occurrences d'un caractère sont remplacées
String substring(int, int)
renvoie une partie de la chaîne
String trim()
enlève les caractères non significatifs de la chaîne (code >32)

La syntaxe : déclaration d'une classe

soit il est public soit il est privé. on parle de dans le projet.

```
[ClassModifiers] class
ClassName
   [extends SuperClass]
   [implemenents Interfaces]
{
 // insérer ici les champs et
les
 // méthodes
}
```

hérite d'une autre classe

implémenter une interface

interface est un modèle. ça n'existe pas

en java il n'y a pas d'héritage multiple

je peux hériter d'une classe et implémenter plusieurs interfaces

ClassModifiers

abstract abstrait, qui n'existe pas - proche de l'interface, contient des méthodes abstraites. On doit les redéfinir, c'est juste un modèle.

la classe contient une ou des méthodes abstraites, qui n'ont pas de définition explicite. Une classe déclarée abstract ne peut pas être instanciée : il faut définir une classe qui hérite de cette classe et qui implémente les méthodes nécessaires pour ne plus être abstraite.

final c'est quelque chose qui ne se modifie pas. on ne peut pas hériter de cette classe

la classe ne peut pas être modifiée, sa redéfinition grâce à l'héritage est interdite. Les classes déclarées final ne peuvent donc pas avoir de classes filles.

private

la classe n'est accessible qu'à partir du fichier où elle est définie

public

La classe est accessible partout

ClassModifiers

Les modificateurs **`abstract`** et **`final`** ainsi que **`public`** et **`private`** sont mutuellement exclusifs.

Le mot clé **`extends`** permet de spécifier une superclasse éventuelle : ce mot clé permet de préciser la classe mère dans une relation d'héritage.

Le mot clé **`implements`** permet de spécifier une ou des interfaces que la classe implémente. Cela permet de récupérer quelques avantages de l'héritage multiple.

L'ordre des méthodes dans une classe n'a pas d'importance. Si dans une classe, on rencontre d'abord la méthode A puis la méthode B, B peut être appelée sans problème dans A.

Exemple :

```
public class Cercle
{
    // Propriété, champ, attribut ou variable membre
    double rayon = 1;
    // Accesseur : méthode permettant de lire une propriété
    double getRayon()
    {
        return rayon;
    }
    // Modificateur : méthode pour modifier une propriété
    void setRayon(double r)
    {
        rayon = (r>0) ? r : -r;
    }
    double surface()
    {
        return 3.141592653589793*rayon*rayon;
    }
}
```

classe publique

UML

Cette classe peut être représentée de la façon suivante (formalisme visuel UML) :

pour modéliser une classe sans mettre de code
il est capable de faire la carte du projet en ne mettant que les éléments importants.

Cercle
rayon : double = 1
getRayon():double setRayon(double) surface() : double

← Nom de la classe

← Propriétés (ici 1 est la valeur par défaut)

← Méthodes

ça donne un schéma de tout le projet

L'intérêt de cette démarche est d'encapsuler les données et les traitements de façon à pouvoir les réutiliser dans des contextes différents.

La création d'un objet : instancier une classe

Exemple :

```
Cercle m;
String chaine;
```

L'opérateur new se charge de créer une instance de la classe et de l'associer à la variable

Exemple :

```
m = new Cercle();
```

je suis obligé de faire un new pour créer l'objet

Il est possible de tout réunir en une seule déclaration

Exemple :

```
Cercle m = new Cercle();
```

Instancier une classe

Chaque instance d'une classe nécessite sa propre variable. Plusieurs variables peuvent désigner un même objet.

En Java, tous les objets sont instanciés par allocation dynamique. Dans l'exemple, la variable m contient une référence sur l'objet instancié (contient l'adresse de l'objet qu'elle désigne : attention toutefois, il n'est pas possible de manipuler ou d'effectuer des opérations directement sur cette adresse comme en C).

Si m2 désigne un objet de type `Cercle`,

l'instruction

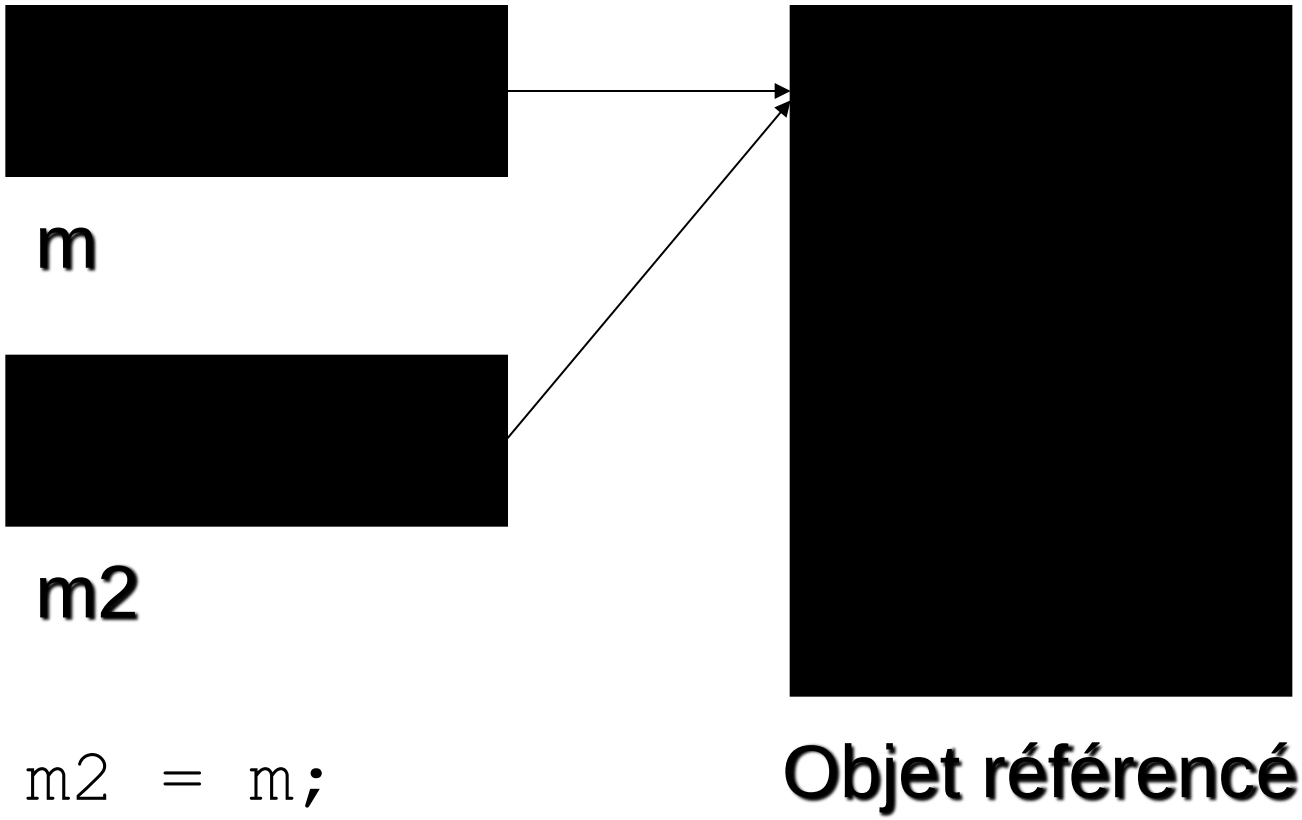
```
m2 = m;
```

ne définit pas un nouvel objet mais m et m2 désignent tous les deux le même objet.

n'est pas une copie de cet objet, mais est vraiment le même obje

Instancier une classe

```
m = new Cercle();
```




```
m2 = m;
```

L'opérateur new

par défaut -> A()si on met des paramètres dans

L'opérateur **`new`** est un opérateur de haute priorité qui permet d'instancier des objets et d'appeler une méthode particulière de cet objet : ***le constructeur***. Il fait appel à la machine virtuelle pour obtenir l'espace mémoire nécessaire à la représentation de l'objet puis appelle le constructeur pour initialiser l'objet dans l'emplacement obtenu. Il renvoie une valeur qui référence l'objet instancié.

Si l'opérateur **`new`** n'obtient pas l'allocation mémoire nécessaire, il lève l'exception **`OutOfMemoryError`**.

Remarque sur les objets de type String

Un objet String est automatiquement créé lors de l'utilisation d'une constante chaîne de caractères sauf si celle ci est déjà utilisée dans la classe.

Ceci permet une simplification lors de la compilation de la classe.

Exemple :

```
public class TestChaines1
{
   public static void main(String[] args)
   {
   String chaine1 = "bonjour";
   String chaine2 = "bonjour";
   System.out.println
         ("(chaine1 == chaine2) = " +
      (chaine1 == chaine2) );
   }
}
```

Résultat :

```
(chaine1 == chaine2) = true
```

Autre exemple :

Pour obtenir une seconde instance de la chaîne, il faut explicitement demander sa création en utilisant l'opérateur new.

Exemple :

```
public class TestChaines2
{
   public static void main(String[] args)
   {
    String chaine1 = "bonjour";
    String chaine2 = new String("bonjour");
    System.out.println("(chaine1 == chaine2) = " +
            (chaine1 == chaine2) );
   }
}
```

Résultat :

```
(chaine1 == chaine2) = false
```

Remarque :

Les tests réalisés dans ces deux exemples sont réalisés sur les références des instances. Pour tester l'égalité de la valeur des chaînes, il faut utiliser la méthode **`equals()`** de la classe String.

Utiliser des objets

Voyons comment utiliser l'objet cercle1 pour calculer l'aire d'un cercle de rayon 5 et stocker le résultat dans une variable :

```
public
static void main (String args[])
{
   ...
   double resultat;
   cercle1.rayon = 5;
   resultat = cercle1.surface();
   ...
}
```

Utiliser des objets

Ici, on accède directement à l'attribut « rayon » de notre objet cercle1. En faisant cela on viole le principe d'**encapsulation** qui est un des piliers de la POO. Ce principe dit qu'une classe doit fournir les méthodes nécessaires pour lire ou modifier les attributs d'un objet.

get et set - agit sur les attribus - set rayon ou get rayon par exemple

Une méthode permettant de lire un attribut est appelée **accesseur** et son nom commence par **get** suivi du nom de l'attribut à lire.

Une méthode permettant de modifier un attribut est appelée **modificateur** et commence par **set** suivi du nom de l'attribut à modifier.

L'intérêt de respecter ce principe permet de rendre le code plus fiable, car les modificateurs sont chargés de vérifier la validité de la valeur avant de modifier un attribut.

Dans notre exemple, le fait d'utiliser la méthode setRayon() nous garantit que l'attribut rayon de notre classe aura bien une valeur positive.

Constructeurs et this

A chaque fois qu'un objet est instancié avec new, le constructeur (méthode particulière) de la classe est automatiquement appelé. Son rôle est d'allouer la mémoire et d'initialiser les attributs de l'objet.

Le constructeur est une méthode qui porte le même nom que la classe, et qui ne possède pas de type (pas même void). Il n'est jamais appelé explicitement, mais uniquement par l'intermédiaire de l'opérateur new.

Si nous reprenons l’exemple de la classe cercle, on peut la définir cette fois en ajoutant un constructeur qui permet d’initialiser la valeur de l’attribut rayon à l’instanciation :

Exemple :

```
public class Cercle
{
    private double rayon = 1;
    public Cercle(double r)
    {
        rayon = r ;
    }
    public double getRayon()
    {
        return rayon;
    }
    public void setRayon(double r)
    {
     rayon = (r>0) ? r : -r;
    }
    public double surface()
    {
        return Math.PI*rayon*rayon
    }
}
```

Constructeurs

Lorsque le constructeur n'a pas été défini par le programmeur, comme dans la première définition de la classe Cercle, Java fournit un ***constructeur par défaut***.

Si le programmeur définit un constructeur, le constructeur par défaut fourni par Java ***n'est plus disponible***.

On peut définir ***plusieurs constructeurs*** pour une même classe, à condition que leur signature soit différente. Ce mécanisme peut être appliqué à n'importe quelle méthode, on dit qu'on ***surcharge*** la méthode.

Initialisation des attributs

- Les attributs d'un objet peuvent être initialisés :
  - A la ***création de l'objet*** : les attributs qui sont de type primitif sont initialisés à 0 et les attributs objets à null.
  - Avec une ***valeur par défaut*** dans la déclaration.
  - Dans le ***constructeur*** de l'objet.

this

this est une variable qui change selon le contexte. Elle contient toujours une ***référence à l'objet courant***. Son utilisation n'est pas nécessaire, mais elle peut être utile dans les cas suivants :

A l'intérieur d'une méthode (ou d'un constructeur, qui est une méthode particulière), lorsqu'une variable locale ou un paramètre porte le même nom qu'un attribut de l'objet, on peut lever l'ambiguïté en précisant explicitement la référence de l'objet courant avec this. Dans ce cas on dit que le paramètre **masque** l'attribut (ou en général qu'un identificateur en masque un autre).

Exemple :

Ainsi, si une méthode reçoit un paramètre hauteur qui est également un attribut de l'objet, l'expression ***this.hauteur*** désignera l'attribut, et l'expression ***hauteur*** le paramètre.

Exemple:

```
public class Rectangle
{
   public int x, y, largeur, hauteur;
   public int volume (int hauteur)
   {
    return hauteur * largeur * this.hauteur;
   }  //    ^ argument              ^ attribut
}
```

this (2)

Le mot réservé this a une **deuxième** fonction lorsqu'il est suivi de parenthèses, il permet d'appeler un constructeur pour l'objet en cours. Le plus souvent on l'utilise dans une classe qui dispose de ***plusieurs constructeurs*** pour appeler un constructeur depuis un autre.

Exemple :

```
public class Rectangle
{
  public int x, y, largeur, hauteur;
  public Rectangle (int l, int h)
  {            // Premier constructeur
    largeur = l;
    hauteur = h;
  }
  public Rectangle ()
  {            // Deuxième constructeur
    this(2,3);    // appel du premier constructeur
  }
}
```

Attributs et méthodes statiques

Lorsqu'on instancie des objets à partir d'une classe, chacun de ces objets possède les mêmes attributs, mais chaque objet possède ses propres valeurs pour ces attributs.

attributs d'instance

Ainsi, un même attribut pour 2 objets différents pourra avoir deux valeurs différentes. C'est pour cette raison que ces attributs sont appelés ***attributs d'instance*** : ils dépendent de l'instance.

Attributs et méthodes de classe

Dans certains cas, des attributs peuvent être ***communs*** à tous les objets d'une classe et exister indépendamment de tout objet de cette classe.

Par exemple, pour une classe d'objet Cours, nous pouvons manipuler un attribut « ***noteEliminatoire*** » qui est commun à tous les cours. On peut aussi penser à un attribut qui définit la durée d'une période d'un cours. Ces attributs ***ne sont pas dépendants d'un cours particulier***, mais plutôt de la classe Cours.

attributs de classe

Ces attributs sont appelés des ***attributs de classe*** (associés à une classe et non à une instance) ou encore des ***attributs statiques***. La déclaration de tels attributs sera précédée par le modificateur ***static***.

Constantes

Les constantes seront la plupart du temps des attributs statiques. Il est cependant possible de déclarer des constantes d'instance, et dans ce cas, elles devront être initialisées au plus tard dans le constructeur de l'objet.

Exemple :

```
public class Cours
{
  public static int duree;
  ...
}
dureeTotale=nbPeriode*Cours.duree;
```

Dans cet exemple, seul le respect des ***conventions*** de codage nous permet de penser que ***Cours*** est une classe, et d'en déduire que ***duree*** est donc un attribut ***statique***.

Exemple 2 : classe *Math*

Java fournit 2 constantes statiques dans la classe Math :

Math.PI et Math.E

Ici encore, les conventions de codage nous renseignent sur la nature des éléments : PI et E sont des **constantes** (majuscules) **statiques** (car le préfixe est un nom de classe et qu'on n'a pas eu besoin d'instancier d'objet de la classe Math).

Méthodes statiques

Ces méthodes ne s'appliquent à ***aucun objet***, ce sont des fonctions au sens classique du terme (comme en C). Elles permettent de fournir des ***services*** à d'autres classes sans passer obligatoirement par une instanciation. Comme pour les attributs statiques, ces méthodes sont définies en utilisant le modificateur ***static*** et sont appelées en préfixant leur nom par le nom de la **classe** au lieu de l'objet.

Exemple:

Math.sqrt(double)

Math.pow(double, double)

Restriction

Ces fonctions de classe n'ont ***pas accès aux attributs d'instance*** (car elles ne s'appliquent pas à un objet), mais elles peuvent accéder aux attributs statiques de la classe.

Exemple :

```
public class A
{
  public int i;
  public static int j;

  public void methode1()
  {
    System.out.println(i + " " + j);  // OK
    methode2();                       // OK
  }

  public static void methode2() // Méthode statique
  {
    System.out.println(i);     // Faux (i variable d'instance)
    System.out.println(j);     // OK (j variable statique)
    methode1();                // Faux (methode1 d'instance)
    methode2();                // OK (récursivité infinie)
    A v = new A();
    v.methode1();              // OK
  }
}
```

Java VS Python

- **Java :**

```
class UtilMath {
// Méthode statique qui ajoute deux entiers et retourne le résultat
    public static int addition(int a, int b) {
            return a + b;
        }
    }

class Main {
     public static void main(String[] args) {
         int res = UtilMath.addition(3, 5);
         System.out.println("Résultat = " + res);
        }
    }
```

Java VS Python

- **Java :**

```
def addition(a, b):
    # Retourne la somme de a et b
    return a + b


# Appel de la fonction
res = addition(3, 5)
print("Résultat =", res)
```

Java VS Python

- **Java :**

```
class Person {
    // Attribut d'instance
    private String name;
    // Constructeur pour initialiser l'objet
    public Person(String name) {
        this.name = name;
    }
    // Méthode d'instance
    public void sePresenter() {
        System.out.println("Bonjour, je m'appelle " + this.name);
    }
}
// Utilisation de la classe Person
class Main {
    public static void main(String[] args) {
        Person p = new Person("Alice");  // création d'un objet Person
        p.sePresenter();  // appel de la méthode sur l'objet
    }
}
```

Java VS Python

- **Python :**

```
class Person:
    def __init__(self, name):
        # Attribut d'instance
        self.name = name

    def se_presenter(self):
        # Méthode d'instance
        print(f"Bonjour, je m'appelle {self.name}")

# Utilisation de la classe Person
p = Person("Alice")          # création d'un objet Person
p.se_presenter()             # appel de la méthode sur
                               l'objet
```

Portée des attributs et des méthodes

Dans l'idéal visé par la POO, un objet est ***indépendant*** s'il est capable de gérer tout seul ses données.

Ce qui implique qu'il soit ***le seul*** à pouvoir accéder à l'intégralité de ses données.

Pour interdire l'accès à certains attributs par les méthodes d'autres objets, on les déclare comme attributs ***privés***.

public et *private*

On peut ainsi définir 2 catégories de propriétés :

Les attributs ***publics*** : accessibles par tous, ils font partie de l'**interface** de la classe.

Les attributs ***privés*** : seul l'objet peut y accéder, c'est une sécurité supplémentaire pour garantir leur intégrité.

Une tentative d'accès à une variable privée depuis un autre objet génère une ***erreur*** à la compilation.

Le même mécanisme est disponible pour les méthodes : une méthode qui n'est utile qu'à l'objet pourra être déclarée privée, interdisant ainsi son appel par d'autres objets. Ce type de méthode est parfois appelé ***méthode de service ou méthode interne***.

Constructeurs, accesseurs et modificateurs

Certaines méthodes doivent toujours être publiques : les constructeurs, accesseurs et modificateurs.

On recommande généralement de déclarer les ***attributs privés*** (sauf s'il y a une bonne raison de ne pas le faire), et de les rendre accessibles au travers de méthodes publiques (accesseur et modificateur).

Lorsqu'un membre est déclaré ***sans modificateur de portée***, il ne sera accessible que par les objets qui font partie du même ***paquetage***. La notion de paquetage sera définie au chapitre suivant, pour l'instant on admettra que 2 objets font partie du même paquetage si les classes qui ont servi à les instancier sont stockées dans le même dossier.

Résumé

En résumé, voici les différentes portées possibles pour les membres d'une classe :

Portée	Modificateur	Accessible depuis une méthode de :
Publique	public	n'importe quel objet
De paquetage	aucun (par défaut)	objets des classes dans le même dossier
Privée	private	l'objet lui-même

NB : le concept d'héritage introduira un modificateur de portée supplémentaire. Il sera présenté au chapitre suivant.

Diagrammes de classe UML

Dans un diagramme UML, la ***portée*** d'un membre est indiquée en préfixant le membre par le signe + s'il est public et par le signe – s'il est privé. Le soulignement indique qu'il s'agit d'un membre statique.

Notre classe Cercle deviendrait alors :

Cercle
-rayon : double = 1 +PI : double = 3.14159
+Cercle(double) +getRayon():double +setRayon(double) +surface() : double

← Nom de la classe

← Propriété privée (valeur 1 par défaut)

← Attribut de classe (statique) public

← Constructeur

← Autres méthodes publiques

Diagrammes de classe UML

En UML, l'absence de préfixe devant un membre indique que sa portée ***n'est pas connue*** et non qu'il s'agit d'une portée par défaut.

Pour un diagramme Java, la ***portée par défaut*** (de paquetage) est parfois représentée par le signe ~ (tilde).

Voyons à présent l'implémentation de la classe décrite ci-dessus :

Exemple :

```
public class Cercle
{
 private double rayon;     //
attribut ou variable membre
 public static final double PI
            = 3.14159;
 public Cercle(double r)
 {
  rayon = r ;
 }
```

Exemple (suite) :

```
  public double getRayon() // Accesseur
  {
    return rayon;
  }

  // Modificateur
  public void setRayon(double r)
  {
    rayon = (r>0) ? r : -r;
  }

  public double surface()
  {
    return PI*rayon*rayon;
  }
}
```

Déclaration de constantes

Pour une donnée, **final** indique qu'il s'agit d'une **constante**, d'instance s'il n'y a pas simultanément le modificateur **static**, et de classe si la donnée est **final static**. Une donnée **final** ne pourra être affectée qu'une seule fois, au moment de sa déclaration, ou, pour le cas d'un attribut non statique, dans tout constructeur qui ne fait pas appel à un autre constructeur de la même classe valuant cet attribut.

```
public [static] final double PI=3.14159;
```

On recommande fortement l'utilisation de constantes nommées plutôt que de valeurs littérales : le code y gagnera en lisibilité.

Héritage

L'héritage est un des piliers fondateurs de la POO. Il permet de créer de nouvelles classes qui possèdent d'office les membres d'une classe existante. Cette classe existante est appelée classe de base, classe mère, parente ou encore superclasse, et la nouvelle classe créée sera appelée classe fille, sous-classe ou classe dérivée.

Une **sous-classe** hérite de tous les membres (attributs et méthodes) de sa **superclasse**.

On peut ensuite y ajouter des membres (attributs et méthodes supplémentaires) ou redéfinir des membres hérités.

Exemple :

Imaginons que nous ayons besoin d'une classe Cylindre, qui aurait les mêmes caractéristiques que la classe Cercle, mais avec un attribut longueur en plus :

```
public class Cylindre extends Cercle
// Cylindre hérite de (étend) Cercle
{
  private double longueur;

  public void afficher()
  {
    System.out.println(
    "Je suis un cylindre de rayon :"
    + getRayon() + " et de longueur : "
    + longueur);
  }
}
```

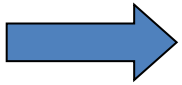
Héritage

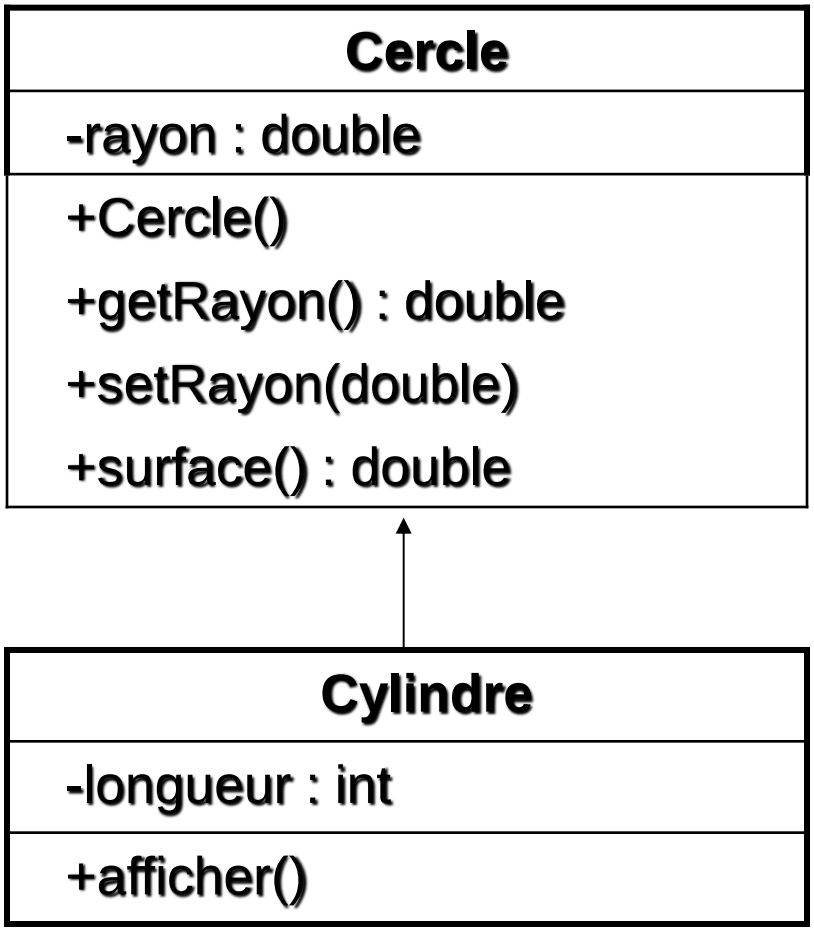
Nous venons de créer une classe Cylindre héritant de Cercle, et nous lui avons ajouté 2 membres : longueur et afficher.

L'héritage peut être schématisé de la façon suivante :

Héritage

Attention si *private*





Attention si *private*

On notera que la flèche pointe vers la superclasse et que les membres de la superclasse ne sont pas représentés dans la classe dérivée.

Héritage et redéfinition de membres

Les attributs et méthodes hérités d'une superclasse peuvent être redéfinis dans la classe dérivée.

Pour redéfinir un attribut, il suffit de le déclarer, dans la classe fille, d'un type différent de celui de la classe mère.

Pour redéfinir une méthode, il faut veiller à ce que la signature de la méthode redéfinie soit la même que celle de la superclasse, sinon, il s'agira d'une surcharge (ajout d'une méthode supplémentaire portant le même nom, mais une signature différente).

Exemple :

La méthode surface() héritée de Cercle ne convient pas pour un cylindre, il est donc nécessaire de la redéfinir si on souhaite obtenir une valeur cohérente pour un cylindre.

Lorsqu'une méthode a été redéfinie, la méthode de la superclasse peut toujours être appelée avec le mot-clé **super**.

Le mot clé super

Ainsi dans notre exemple précédent, on peut toujours appeler la méthode surface() de Cercle depuis un objet instancié à partir de Cylindre avec la syntaxe suivante :

super. surface();

On pourrait donc redéfinir la méthode surface de notre classe Cylindre de la façon suivante :

```
public double surface()
{ return 2*super.surface()+
      2*Math.PI*getRayon()*longueur; }
```

Attention si rayon est private !

On peut interdire la redéfinition d'une méthode avec le modificateur ***final***.

Héritage et portée des membres

Une classe dérivée hérite de tous les membres publics et de package (portée par défaut, sans modificateur).

Les membres **privés** sont hérités mais inaccessibles par un objet de la classe dérivée. Seuls les accesseurs/modificateurs peuvent les atteindre (à condition qu'eux aient été déclarés publics).

Dans notre exemple, ***l'attribut rayon ne sera pas accessible*** directement par un objet Cylindre, mais uniquement par l'intermédiaire des accesseurs ***getRayon()*** et setRayon() qui, eux, sont publics.

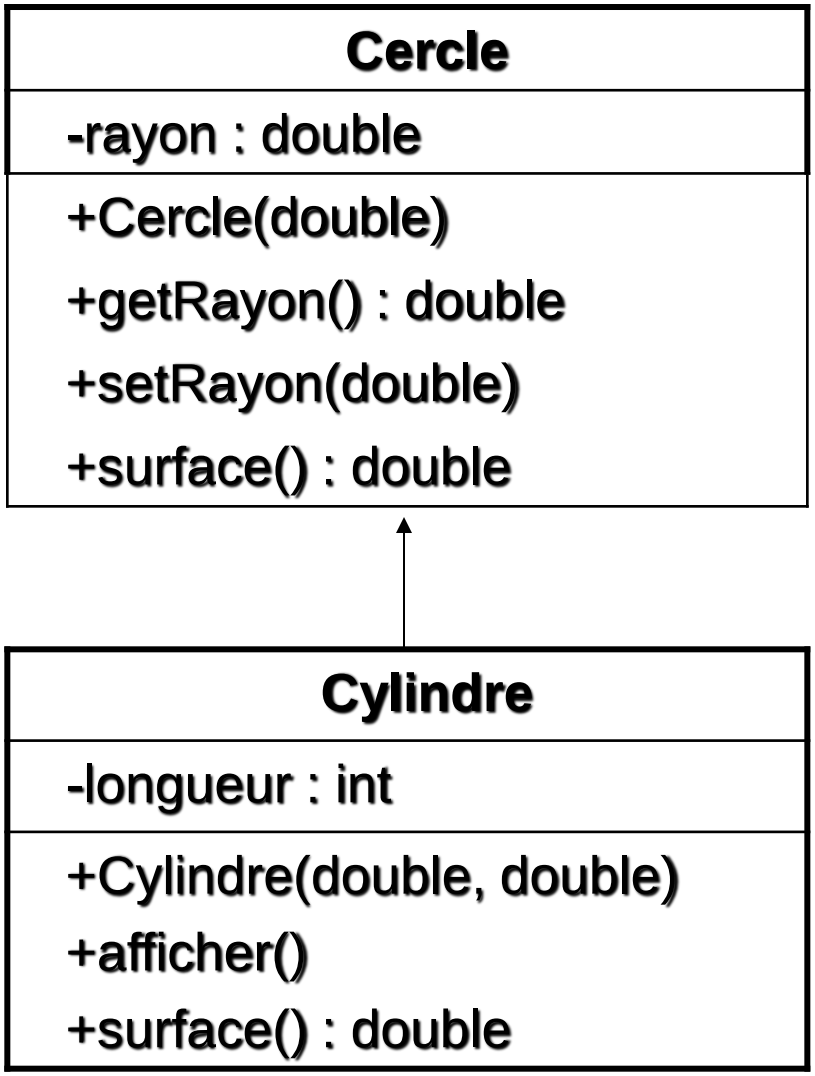
Un nouveau modificateur de portée apparaît avec l'héritage : **protected**.

Un membre protégé de ***Cercle*** est accessible de partout dans le paquetage de Cercle et, si Cercle est public, dans les classes héritant de Cercle dans les autres paquetages.

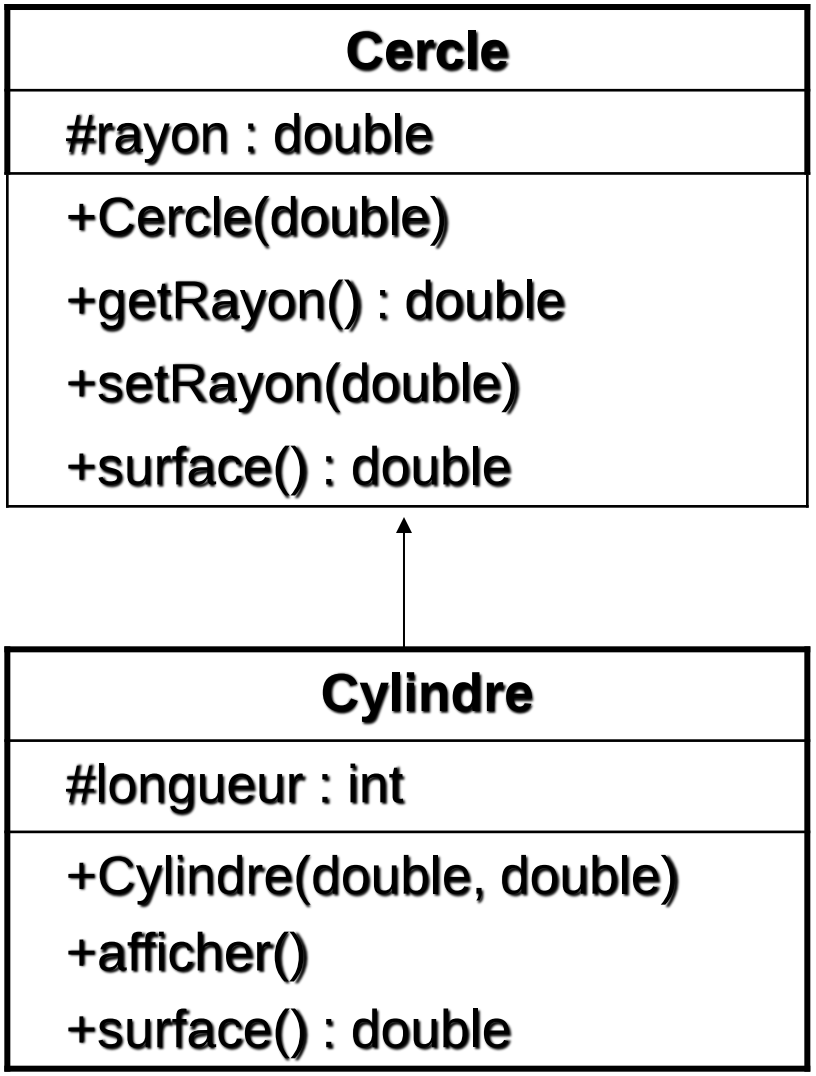
En UML, les membres protégés sont préfixés par un # (dièse).

UML

Avec *private*



Avec *protected*



protected

```
public class Cercle
{
  protected double rayon;
  ...
}

public class Cylindre extends Cercle
{
   protected double longueur;
  ...
}
```

Constructeur et *super()*

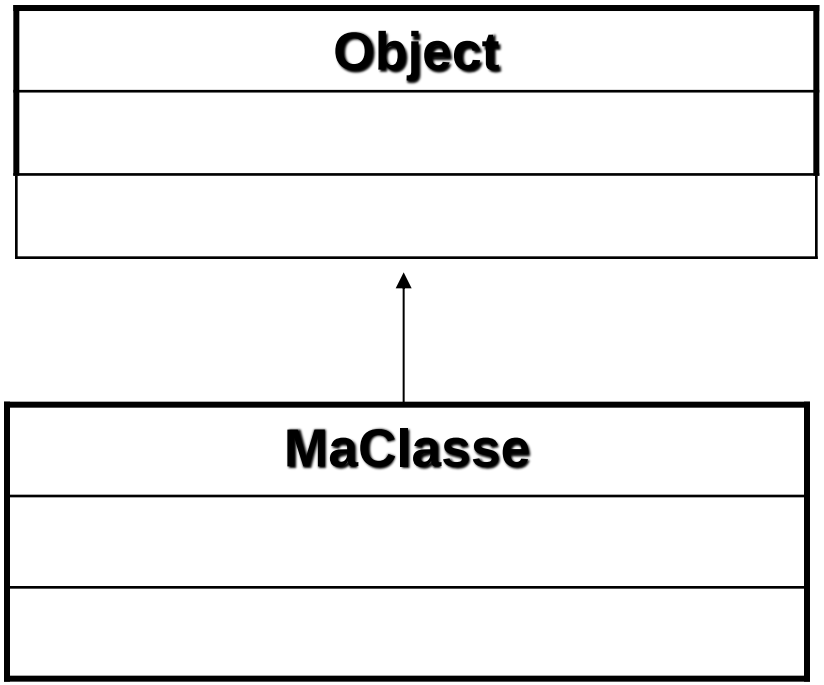
Constructeur de la classe Cylindre :

```
public Cylindre(double r, double l)
{
 super(r);          // Obligatoire !
 longueur = l;
}
```

L'appel à super(r) est obligatoire si la classe Cercle ne possède pas de constructeur sans argument !

Héritage implicite

En Java, toutes les classes définies sans héritage explicite (avec le mot-clé extends) héritent implicitement de la classe Object. Celles qui sont définies avec un héritage explicite héritent indirectement de cette même classe Object.



Héritage et constructeurs : super()

Par défaut, lorsque le constructeur d'une classe est invoqué, Java exécute automatiquement le constructeur sans argument de la super-classe avant d'exécuter la première instruction du constructeur.

S'il n'existe pas, une erreur sera générée à la compilation (c'est une bonne raison pour toujours définir un constructeur sans argument).

On peut appeler un autre constructeur de la super-classe avec le mot-clé **super()**, si on respecte ces 2 conditions :

- Seul le constructeur de la classe peut appeler le constructeur de la superclasse ET
- l'appel DOIT être la première instruction du constructeur.

Rappel : un constructeur sans paramètres est fourni par Java tant qu'aucun autre n'a été défini.

Ce constructeur n'est plus accessible dès qu'un autre a été défini.

Différences entre super et new : new alloue la mémoire, puis exécute les instructions du constructeur. Alors que super exécute seulement les instructions, l'allocation mémoire a été prise en charge par le constructeur de la classe dérivée.

Séries 09, 10

Classes abstraites

Le mot clé abstract s'applique aux méthodes et aux classes.

Abstract indique que la classe ne pourra être instanciée telle quelle. De plus, toutes les méthodes de cette classe abstraite ne sont pas implémentées et devront être redéfinies par des méthodes complètes dans ses sous-classes.

Abstract permet de créer une classe qui sera une sorte de moule. Toutes les classes dérivées pourront profiter des méthodes héritées et n'auront à implémenter que les méthodes déclarées **abstract**.

Exemple :

```
public abstract class ClasseAbstraite
{
  ClasseAbstraite() { ... //code du constructeur }
  void méthode() { ...  // code partagé par tous
                  // les descendants}
  abstract void méthodeAbstraite();
}

public class ClasseComplete extends ClasseAbstraite
{
  ClasseComplete() { super(); ... }
  void méthodeAbstraite() { ... // code de la
                          // méthode }
  // void méthode est héritée
}
```

Méthodes abstraites

Une méthode abstraite est une méthode déclarée avec le modificateur **abstract** et sans corps. Elle correspond à une méthode dont on veut forcer l'implémentation dans une sous-classe. L'abstraction permet une validation du codage.

Une sous-classe sans le modificateur abstract et sans définition explicite d'une ou des méthodes abstraites génère une erreur de compilation.

Une classe est automatiquement abstraite dès qu'une de ses méthodes est déclarée abstraite. Il est possible de définir une classe abstraite sans méthodes abstraites.

Interfaces

Une classe dont toutes les méthodes sont abstraites porte le nom d'interface : l'ensemble des méthodes abstraites forme l'interface qui permettra d'utiliser les objets de cette classe.

Une interface ne pouvant pas être instanciée, ses attributs sont implicitement constants (public, static et final) et ses méthodes sont implicitement abstraites et publiques (public et abstract).

Il n'est pas nécessaire de rajouter les mots-clés abstract et final dans la déclaration d'une interface, car le mot-clé interface au début de la déclaration de la classe suffit :

Interfaces : Syntaxe

```
[public] interface Nom_interface
                [extends Autre_interface]
{
  type nom_attr [, nom_attr... ];
  ... // autant de déclarations de membres
      // données (propriété, attribut)
      // qu'on le désire
  type nom_meth ([param, param, ... ]);
  ... // autant de déclarations de membres
      // fonctions (méthodes) abstraites
      // qu'on le désire
}
```

Le mot clé *implements*

Lorsqu'on déclare une classe et qu'on veut qu'elle satisfasse aux exigences d'une interface, on dit qu'elle met en oeuvre cette interface ou qu'elle implémente les méthodes déclarées dans cette interface, et on l'indique comme suit :

```
class Classe extends Parent
   implements Interface1 [,Interface2 ... ]
```

Une classe peut satisfaire plusieurs interfaces. C'est ce mécanisme qui permet de remplacer l'héritage multiple qui n'existe pas en Java.

Une interface peut hériter d'une autre interface : cela permet de lui ajouter des attributs constants ou des méthodes abstraites.

L'héritage en résumé

L'héritage permet donc de réutiliser aux mieux des modules existants, mais également une meilleure modularisation d'un programme : les attributs/méthodes communs appartiennent à la superclasse, et les membres spécifiques seront ajoutés aux classes dérivées.

L'héritage crée un réseau de dépendances entre classes. Ce réseau est organisé sous forme arborescente et porte le nom de **hiérarchie de classes**.

Une classe peut avoir plusieurs sous-classes, mais ne peut avoir qu'une seule superclasse : la notion d'héritage multiple n'existe pas en Java (contrairement au C++), mais elle est remplacée par la notion d'interface (une classe peut implémenter plusieurs interfaces).

Rappel des modificateurs de classe

Modificateur	Rôle
abstract	La classe contient une ou des méthodes abstraites qui n'ont pas de définition explicite. Une classe déclarée abstract ne peut pas être instanciée : il faut définir une classe qui hérite de cette classe et qui implémente les méthodes nécessaires pour ne plus être abstraite.
final	La classe ne peut pas être modifiée, sa redéfinition grâce à l'héritage est interdite. Les classes déclarées final ne peuvent donc pas avoir de classes filles.
private	La classe n'est accessible qu'à partir du fichier où elle est définie.
public	La classe est accessible partout.

Rappel des différentes portées possibles pour les variables d'une classe

Portée	Modificateur	Accessible depuis une méthode de :
Publique	public	N'importe quel objet.
De paquetage	aucun (par défaut)	Objets des classes dans le même dossier.
Privée	private	L'objet lui-même.
Protégée	protected	Objets des classes dans le même dossier. Objets des classes héritées (partout).

Variables d'instance, de classe et constantes

Mot clé	Accessible depuis une méthode de :
	Variable d'instance, chaque instance de la classe a accès à sa propre occurrence de la variable.
static	Variable de classe, chaque instance de la classe partage la même variable.
final	Les constantes sont définies avec le mot clé final : leur valeur ne peut pas être modifiée une fois qu'elles sont initialisées.

Rappel des modificateurs de méthodes

Modificateur	Rôle
public	La méthode est accessible aux méthodes des autres classes.
private	L'usage de la méthode est réservé aux autres méthodes de la même classe.
protected	La méthode ne peut être invoquée que par des méthodes de la classe dans le même package ou de ses sous classes.
final	La méthode ne peut pas être modifiée (redéfinition lors de l'héritage interdite).
static	La méthode appartient simultanément à tous les objets de la classe (comme une constante déclarée à l'intérieur de la classe). Il est inutile d'instancier la classe pour appeler la méthode mais la méthode ne peut pas manipuler de variable d'instance. Elle ne peut utiliser que des variables de classes.
synchronized	La méthode fait partie d'un thread. Lorsqu'elle est appelée, elle barre l'accès à son instance. L'instance est à nouveau libérée à la fin de son exécution.
native	Le code source de la méthode est écrit dans un autre langage.
	Sans modificateur, la méthode peut être appelée par toutes autres méthodes des classes du package auquel appartient la classe.

Polymorphisme

L'héritage définit un transtypage implicite de la sous-classe vers la superclasse : on peut affecter à une référence d'une classe n'importe quel objet d'une de ses sous-classes.

En poussant ce raisonnement, en Java, une variable déclarée comme Object pourra recevoir la référence d'un objet instancié à partir de n'importe quelle classe.

En général, le polymorphisme désigne la capacité d'un objet à prendre plusieurs formes.

En programmation, ce terme signifie qu'une même variable d'une classe donnée, peut être utilisée pour référencer des objets de différentes classes (qui héritent de la première) et automatiquement appeler la méthode spécifique à l'objet référencé (celle de la classe dérivée).

Polymorphisme

Un langage orienté objet est dit polymorphique, s'il offre la possibilité de pouvoir percevoir un objet en tant qu'instance de classes variées, selon le contexte.

Si nous appliquons ces principes à nos classes Cercle et Cylindre, on pourrait faire ceci :

Exemple :

```
Cercle c;     // Déclaration d'un Cercle

Cylindre c2=new Cylindre();   // Déclaration et
  // instanciation d'un Cylindre (hérite de Cercle)

c=c2;      // c référence un Cylindre (même
           // s'il a été déclaré comme Cercle)

c.getSurface();  // c'est bien la méthode de
                 // Cylindre qui est appelée :
                 // c'est elle qui est la plus
                 // spécifique à l'objet même si
                 // la référence est celle d'un
                 // Cercle.
```

L'intérêt de ce mécanisme est de pouvoir manipuler avec une même variable des objets instanciés à partir de classes différentes (à condition qu'elles héritent toutes de la classe qui a servi à déclarer la variable).

Paquetages (définition)

Un paquetage (package) est un regroupement logique de classes.

Toutes les classes d'un paquetage sont enregistrées dans un dossier portant le nom du paquetage.

Un paquetage peut contenir des classes et d'autres paquetages.

On constate dès lors une analogie entre la hiérarchie des paquetages et celles des dossiers dans le système de fichiers.

Paquetages (définition)

La bibliothèque des classes standard Java est organisée de cette manière.

On peut le constater en explorant les sources de ces classes dans le fichier **src.zip** se trouvant dans le dossier d'installation du JDK.

Paquetages (définition)

NB : Le paquetage **java.lang** contient les classes de base de java et est toujours disponible sans qu'on ait à spécifier explicitement son utilisation.

A chaque nouvelle version de Java , le nombre de packages et de classes s'accroît :

Version Java	Date de sortie	Nombre approximatif de packages
J2SE5.0	2004	166+
Java SE6	2006	175+
Java SE7	2011	180+
Java SE8	2014	200+
Java SE9	2017	~95 (visibles sans modules)
Java SE11 (LTS)	2018	~90 (standards)
Java SE17 (LTS)	2021	~91
Java SE24	2025	~91

Paquetages (Utiliser des classes d'un paquetage)

Pour utiliser des classes d'un package existant, il existe 3 possibilités :

- On peut utiliser le "nom complet" de la classe, le nom de la classe est alors précédé des packages qui la contiennent.

```
monpackage.MaClasse objet = new monpackage.MaClasse();
```

- On peut importer la classe et l'utiliser avec son "nom court" par la suite :

```
import monpackage.MaClasse;
MaClasse objet = new MaClasse();
```

Paquetages (Utiliser des classes d'un paquetage)

- Enfin on peut importer toutes les classes du package (plus long à compiler)

```
import monpackage.*;
MaClasse objet = new MaClasse();
```

- Attention toutefois, cette syntaxe n'importe que les classes et pas les sous-paquetages.

Ainsi la ligne suivante :

```
import java.*;
```

n'importera que les classes du paquetage java, mais pas ses sous-paquetages (java.awt, java.appplet, … ne seront pas importés).

Paquetages (Utiliser des classes d'un paquetage)

Le paquetage java.lang est importé implicitement.

Il n'est pas nécessaire de l'importer pour utiliser ses classes.

On peut distinguer 3 types de paquetages :

- Le paquetage par défaut : classes stockées dans le répertoire courant.
- Les paquetages standard : classes fournies par Sun, se trouvent dans le dossier du JDK.
- Les paquetages utilisateur : autres classes, que l'on a développées ou téléchargées.

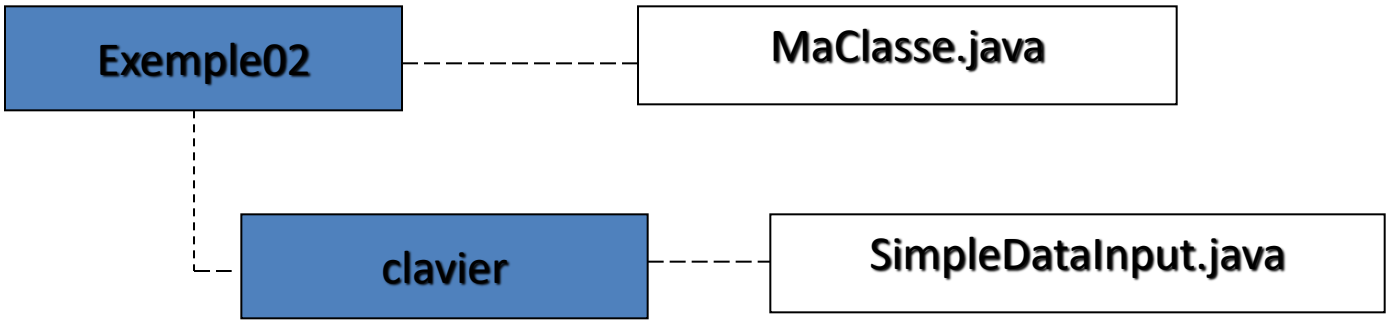
Utilisation des packages

- Premier cas :

  Le paquetage est dans le même répertoire que la classe qui l'utilise

Le nom du paquetage est : **clavier**

Le nom long de la classe est :
clavier.SimpleDataInput



Utilisation des packages

– Dans MaClasse.java, on trouvera :

```
import clavier .*; // ou

import clavier.SimpleDataInput;
```

– Dans SimpleDataInput.java, on trouvera :

```
package clavier;
```

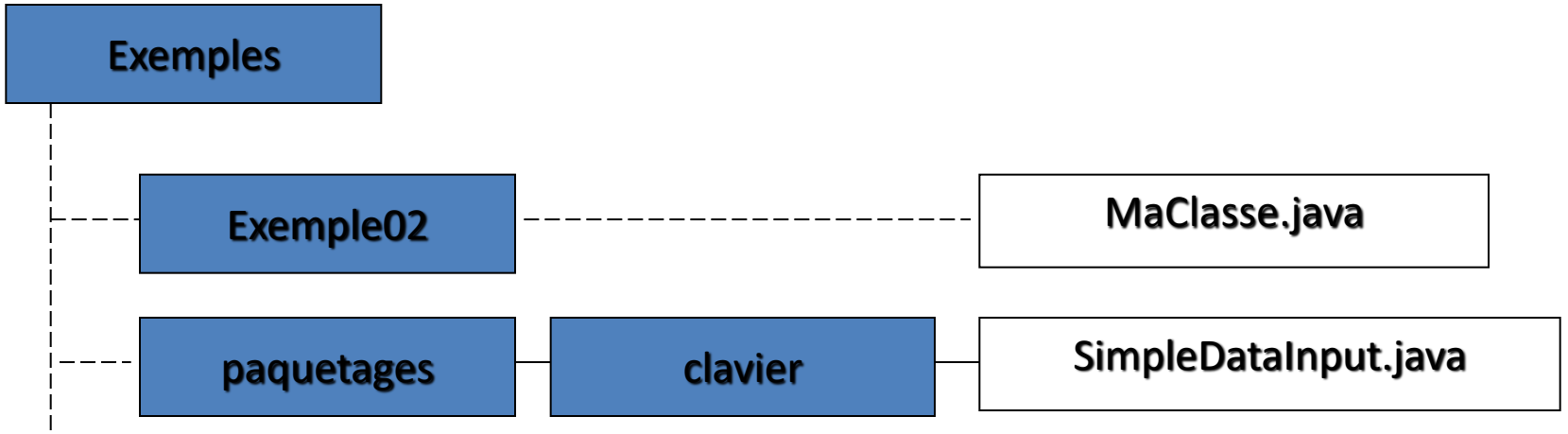
Utilisation des packages

- Deuxième cas :

  Le paquetage n'est pas dans le même répertoire que la classe qui l'utilise

Le nom du paquetage est : **paquetages.clavier**

Le nom long de la classe est :
paquetages.clavier.SimpleDataInput



Utilisation des packages

– Dans MaClasse.java, on trouvera :

```
import paquetages.clavier .*; // ou
import paquetages.clavier.SimpleDataInput;
```

– Dans SimpleDataInput.java, on trouvera :

```
package paquetages.clavier;
```

Utilisation des packages

– Pour la compilation on écrira :

```
javac -classpath ..\;. MaClasse.java
```

– Pour l'exécution :

```
java -classpath ..\;. MaClasse
```

Garbage Collector

Rôle :

Contrairement au C++, Java ne possède pas de destructeur.

Le garbage collector (ramasse-miette) s'occupe de détruire les instances qui ne sont plus référencées.

Le garbage collector surveille toutes les instances, détecte celles qui sont inutiles et les retire de la mémoire.

Cependant, juste avant de détruire les instances, il va appeler la méthode ***finalize()***, si elle existe.

Garbage Collector

Méthode ***finalize()*** :

La méthode ***finalize*** est donc appelée par le garbage collector juste avant la destruction d'une instance.

Cela permet, par exemple, de libérer les ressources que l'instance utilise ou encore d'exécuter certaines tâches à la destruction de l'instance.

La méthode doit être écrite par le développeur, il faut donc la définir dans la classe.

Garbage Collector

Exemple :

```
public class Exemple
{
  protected void finalize() throws Throwable     {
    // Instructions de finalisation
    // Par exemple rendre les ressources
    // au système.
  }
```

Garbage Collector

Contrôle du garbage collector :

Le garbage collector est entièrement intégré à la JVM (Java Virtual Machine).

Il intervient lorsqu'il y a un manque de mémoire.

Il est impossible de contrôler son passage.

Cependant, on peut forcer son intervention grâce à la méthode statique ***gc()*** de la classe ***System***.

```
System.gc();
```