Java

Compilation | Exécution

* le ***JRE*** (Java Runtime Environment) est l'environnement d'exécution java qui contient la machine virtuelle **JVM**
* Le ***compilateur Java*** produit un code dans un format propre à Java et qui s'appelle le ***byte code***
* Ce code est interprétable par la ***JVM***
* Le point de départ de tout programme Java est la méthode public static void main(s[] args)

Les types primitifs

* Les types primitifs sont les seuls types du langage qui ne sont pas des classes.
* Ces types sont regroupés en 4 catégories :
  * Nombres entiers
  * Nombres flottants
  * Caractères
  * Booléens

Nombres entiers

Type	Taille	Valeur minimale	Valeur maximale
byte	1 octet	-128	127
short	2 octets	-32 768	32 767
int	4 octets	-2 147 483 648	2 147 483 647
long	8 octets	-9 223 372 036 854 775 808	9 223 372 036 854 775 807

- ***Division par 0*** sur les entiers conduit à une erreur d'exécution
- ***Dépassement de capacité*** d'un entier conduit à la conservation des bits de signe les moins significatifs

Nombres flottants

Type	Taille	Valeur minimale	Valeur maximale
float	4 octets	$1.40239846^{E}-45$	$3.40282347^{E}38$
double	8 octets	$4.94065645841\ 24654^{E}-324$	$1.797693134862\ 316^{E}308$

Particularité

float x = 12.5 ;

Incorrect car par défaut toutes les constantes flottantes créées par le compilateur sont de type double, il faut dans ce cas l'expliciter comme ceci : **x = 12.5f**

- Chez les flottants il existe les limites : ***POSITIVE_INFINITY*** et ***NEGATIVE_INFINITY***
- Quand un résultat en virgule flottante devient plus petit qu'une des valeurs minimales, il est mis à 0.
- Quand il dépasse la plus grande valeur possible, il prend la valeur **POSITIVE_INFINITY** ou **NEGATIVE_INFINITY** suivant qu'il est positif ou négatif respectivement.
- En particulier, la division par 0 ne génère pas d'erreur en flottant : elle donne un résultat qui est soit POSITIVE_INFINITY soit NEGATIVE_INFINITY.
- Quand le résultat a une valeur indéfinie, il prend la valeur **NaN** (*Not a Number*). Cela arrive dans quatre cas :
  - la division de 0 par 0 ;
  - la soustraction de *_INFINITY à lui-même ;
  - la division de *_INFINITY par lui-même ;
  - la multiplication *_INFINITY par 0.

Conversions des nombres

* Il est possible d'écrire des expressions mixtes dans lesquelles interviennent des opérateurs de types différents
* Il y a conversion dans le plus grand des types
  * int->long->float->double
* Exemple :

```
float  f = 2.5f ;   int i = 3 ;   long l = 4 ;
double d = i * l + f  ;
// i est convertit en long et ensuite multiplié, le résultat est convertit en float et ensuite additionné et le résultat final convertit en double
```

* Promotion numérique : **char**, **byte** et **short** seront d'abord convertit en **int**

Type caractère

* Les caractères sont représentés en mémoire par 2 octets en utilisant le code universel « Unicode ».

Type	Taille	Notation
char	2 octets	'e' '\u2554'

Type booléen

Type	Valeur
boolean	true / false

```
boolean b = false ;
int n = 5 ;
boolean c =  n>3 ; // c faudra true
```

L'instruction **if**

```
if (CONDITION)
    //UNE instruction ;
else
    //UNE instruction ;

if (CONDITION) {
        //1 ou plusieurs d'instructions
} else if (CONDITION){
        //1 ou plusieurs d'instructions
} else {
        //1 ou plusieurs d'instructions
}
```

L'instruction **switch**

```
switch ( CHOIX ){
case 1:  …
                    break;
case 2:  …
                    break;
default:            … // Optionnel
                    break;

}
```

* Limitation à tester des types **char** ou des **entiers**. Depuis Java **7**, on peut tester des chaines de caractères **String.**

L'instruction **while** et **for**

```
while (CONDITION){
        // Instructions
}
for (int i = 0 ; i < 5 ; i++) {
        // Instructions
}
```

NB : L'instruction break permet de sortir d'une boucle.

```
while (i<3) {
        if(a==b) break ;
        i++ ;
}
```

L'instruction continue permet de passer prématurément au tour de boucle suivant.

```
while (i<3) {
        if(a==b) continue;
        i++ ;
}
```

Opérateur conditionnel

* **? :** forme abrégée de **if else.**
* *condition* **?** *resultat1* **:** *resultat2***;**

```
int nb = i < 4 ? 7 : 8 ;
// Si i<4 alors nb vaut 7 sinon nb vaut 8
```

Qu'est ce qu'une classe ?

* Une ***classe*** représente une ***catégorie d'objets***, elle définit la ***structure*** de l'***objet***
* A chaque ***classe*** correspond un ***type***.
* Un **objet** est une **instance** de sa classe

Portée d'une classe

* **public** : accessible par **tout** objet
    * **public class** MaClasse
* (par défaut) : accessibilité niveau **package**
    * **class** Maclasse

Variables

* Les noms de variables sont sensibles à la casse
* Par convention, un nom de variable est écrit en minuscules, sans accent
* Le nom d'une variable ne peut pas commencer par un chiffre
* Par convention, lorsque le nom de la variable est composé de plusieurs mots, on peut utiliser une majuscule pour l'initiale de chaque nouveau mot

Déclaration d'une variable

* **int** maVar1;
* **int** maVar2, maVar3;
* **int** maVar=5, maVarBis;

Variables d'instance

* Ce sont les variables définies dans la classe
* Elles sont accessibles partout dans cette même classe
* Les variables d'instance définiront les caractéristiques d'un objet
* A noter : Les variables définies dans des blocs d'instruction et dans les méthodes ne sont accessibles que dans le bloc ou la méthode concernés, ce sont des variables locales

Méthodes de classe

* Par convention, le nom de la méthode est en minuscule.
* Il faut indiquer la valeur de retour ou ***void***, le nom de la méthode suivi des éventuels **arguments** entre **()**. La valeur de retour est renvoyée par le mot clé ***return***.

```
void maMethode (int arg1, String arg2) {

}
```

```
String direBonjour(){
        return  "Bonjour" ;
}
```

Portée d'une variable d'instance, méthode de classe

* **private :** ne sont accessibles que par la classe dans laquelle elles sont définies
* **public :** sont accessibles par tout objet
* **protected :** définies comme protégées ne sont accessibles que par les classes dérivées et classes du même package
* **(par défaut) :** (sans modificateur) sont accessibles par toute classe appartenant au même package

Encapsulation

* En ***POO*** pure on réalise ce que l'on nomme l'***encapsulation*** des données, on ne peut pas accéder aux champs d'un objet autrement qu'en recourant aux méthodes prévues à cet effet
* L'encapsulation permet de garantir **l'intégrité** des données contenues dans l'objet
* L'accès aux données se fera via les méthodes d'accesseurs (**getters**) et mutateurs (**setters**).
* Un accesseur est une méthode qui va permettre d'accéder aux variables d'un objet en lecture, et un mutateur permettra de les modifier

Mot clé : **this**

* Le mot-clé ***this*** désigne, dans une classe, l'instance courante de la classe elle-même
* Il peut être utilisé pour rendre le code explicite et non ambigu.
  * Variable d'instance et variable locale portant le même nom

Constructeur

* Un constructeur est une méthode d'instance qui va se charger de **créer l'objet** et, le cas échéant, d'initialiser ses variables de classe.
* Une classe ne peut disposer d'aucun constructeur, Il existe un constructeur **par défaut** sans arguments.
* Dés qu'une classe possède un constructeur, le constructeur par défaut ne peut plus être utilisé.
* Le constructeur est une méthode, sans valeur de retour (pas de void), portant le **même nom** que la classe et pouvant disposer d'un nombre quelconque d'arguments.

* Une même classe peut définir **0, 1 ou plusieurs** constructeurs (avec des arguments différents).
* Un constructeur peut appeler un autre constructeur de la même classe via la méthode « **this(** [arguments éventuels] **)** »
* Un constructeur peut être déclaré « **private** » et dans ce cas ne pourra pas être appelé de l'extérieur (mais bien en **interne**)
* La méthode ***super()*** fait appel au constructeur de sa superclasse (toute classe dérive de la classe **Object**) (est implicite)

Construction et initialisation

* La création d'un objet entraîne, dans l'ordre :
  * Une initialisation par défaut des tous les champs de l'objet.

Type	Valeur par défaut
Primitifs numériques	O
boolean	false
char	\uoooo (null unicode)
Object	null

  * Une initialisation explicite lors de la déclaration du champ
  * L'exécution des instructions du corps du constructeur

Héritage

* L'héritage permet de définir une nouvelle classe à partir d'une classe existante en **héritant** de ses **propriétés** et **aptitudes** et en y ajoutant de nouvelles méthodes et données
* En java, une classe peut hériter au **maximum d'une seule autre classe**, c'est pourquoi il existe une notion d'interface qui permet, dans un certain sens, de pallier à cette limitation
* Une classe hérite d'une autre classe par le biais du mot clé ***extends***
* L'héritage peut se faire **sur plusieurs niveaux** (superclasse, sous-classes, sous-classes de sous-classes, …). Les relations d'héritage sont à considérer comme une **arborescence** où chaque classe hérite non seulement de son parent immédiat mais aussi de tous les ancêtres de ce parent.

Construction d'un objet dérivé

* Supposons : **class** Enfant **extends** Parent
* La création de l'objet *Enfant* se fait, dans l'ordre :
  * **Allocation de mémoire** pour un objet de type ***Enfant***
  * **Initialisation** par **défaut** de tous les **champs** de ***Enfant***
  * **Initialisation** des **champs** hérités de ***Parent*** (et blocs d'initialisation)
  * Exécution du **constructeur** de ***Parent***
  * **Initialisation explicite** des **champs** de ***Enfant*** (et blocs d'initialisation)
  * Exécution du **constructeur** de ***Enfant***

Classes et méthodes **abstraites**

* Une classe abstraite est définie par le biais du mot clé **abstract**
* Une classe abstraite est une classe qui **ne permet pas d'instancier des objets**, elle ne sert que de classe de base pour une dérivation
* Dés qu'une classe comporte une ou plusieurs méthodes abstraites, elle est abstraite
* Une **méthode abstraite** doit être déclarée ***public* ou *protected***
* Une classe dérivée d'une classe abstraite peut ne pas redéfinir les méthodes abstraites de sa classe de base. Dans ce cas elle est elle même abstraite.

Interface

* Une interface **définit** le **comportement** qui doit être implémenté par une classe
* Une interface sert à définir un super-type et à utiliser le **polymorphisme**
  * Le polymorphisme permet d'ajouter de nouveaux objets dans un scénario préétabli en utilisant le type de base.
  * On utilise chaque objet comme objet de sa classe de base mais son comportement effectif dépend de sa classe effective (de la classe dérivée)

* Pour définir une interface on utilise simplement le mot clé ***interface*** à la place de *class*
* Une interface **définit les en-têtes** d'un certains nombre de **méthodes** et peut définir des constantes
* Les interfaces peuvent **se dériver entre-elles** via le mot clé **extends**
* On peut utiliser des variables de type interface
* Une classe peut **implémenter plusieurs interfaces** via le mot ***implements***

Redéfinition/Overriding de méthodes

* Une classe dérivée peut redéfinir les méthodes de la classe de base.
* La méthode redéfinie doit avoir la **même signature** (même nom et même nombre et types d'arguments)
* Les valeurs de **retour** des 2 méthodes doivent être de **même type** (ou être covariantes)
* La redéfinition d'une méthode **ne doit pas diminuer les droits d'accès** à cette méthode, en revanche elle peut les augmenter.
* Il est recommandé d'utiliser l'annotation ***@Override***. En plaçant cette annotation, le compilateur vérifiera que la méthode est correctement utilisée

Cas particulier : valeurs de retour covariantes

* Pour qu'il y ait redéfinition de méthode le **type** de **retour** doit être **identique**
* Exception : Depuis java 5, la méthode qui redéfinit une autre peut renvoyé un type identique ou **dérivé**
* Exemple **:**

```
class Parent{
        public Parent methodUntel(){}
}
class Enfant extends Parent{
        public Enfant methodUntel(){} //redéfinition
}
```

Surcharge/Overloading de méthodes

* La méthode surchargée est une méthode portant le **même nom** mais avec **des arguments différents** (en type et/ou en nombre) qu'une autre dans la même classe ou une sous-classe
* **La valeur de retour peut différer**, par contre une méthode ayant le même nom et les mêmes arguments mais ayant un type de retour différent aboutirait à une erreur de compilation.

Exemple : polymorphisme, redéfinition et surdéfinition

```
public class Parent {

    public void method(int entier){
        System.out.println("Parent method entier"); }

    public void method(double d){
        System.out.println("Parent method double"); }

}
```

```
public class Enfant extends Parent{

    public void method(int entier){
        System.out.println("Enfant method entier"); }

    public void method(float f){
        System.out.println("Enfant method flottant"); }

}
```

Utiliser une méthode main et effectuer des tests

* **class** Parent
  * **public String** myMethod(**Parent** p)
* **class** Enfant **extends** Parent
  * **public String** myMethod(**Parent** p)
  * **public String** myMethod(**Enfant** p)
* **class** PetitEnfant **extends** Enfant
* Méthode **main** pour tests sur :
  * Parent parent = **new Parent();**
  * Enfant enfant = **new Enfant();**
  * Parent pEnfant = **new Enfant();**
  * Parent ptEnfant = **new PetitEnfant();**

Référence à l'objet

* Les variables de type classe contiennent non pas l'objet lui-même mais la **référence** vers cet objet
* Exemple :

  A a = new A();

  A aBis = a;

  // Recopie la référence contenue dans a dans aBis

  // a et aBis pointe sur le même objet
* Passage de paramètres :
  * Variable de type classe : par copie de référence
  * Variable de type primitif : par copie de valeur

Classe **Object**

* Toute classe **dérive** de la classe **Object**
* **5 méthodes** définies dans **Object** seront héritées par chaque classe et pourront être redéfinies
  * public String **toString**()
  * public boolean **equals**(Object obj)
  * public native int **hashCode**()
  * protected native Object **clone**()
  * protected void **finalize**()
    * (appelée juste avant la collecte par la garbage collector)

Méthode : **equals(Object o)**

* Dans la classe Objet, cette méthode vérifie que les deux objets possèdent la **même référence** mémoire et sont donc en fait le **même objet** : o1==o2
* Le rôle de cette méthode est de vérifier si deux **instances** sont **sémantiquement équivalentes** même si ce sont deux instances distinctes, c'est pourquoi cette méthode peut-être redéfinie
* La classe String redéfinit cette méthode et renvoie **true** si les chaînes de caractères sont identiques

* Contraintes à respecter lors de la redéfinition :
  * **Symétrie** : pour deux références a et b, si a.equals(b) alors il faut obligatoirement que b.equals(a)
  * **Réflexivité** : pour toute référence non null, a.equals(a) doit toujours renvoyer true
  * **Transitivité** : si a.equals(b) et b.equals(c) alors a.equals(c)
  * **Consistance avec la méthode hashCode()** : si deux objets sont égaux en invoquant la méthode equals() alors leur méthode hashCode() doit renvoyer la même valeur pour les deux objets (L'inverse n'est pas vrai, deux objets dont la méthode hashCode() renvoie la même valeur, n'implique pas obligatoirement que l'invocation de la méthode equals() sur les deux objets renvoie true.)
  * Pour toute **référence non null**, a.equals(null) doit toujours renvoyer **false**

Méthode : **hashCode()**

* Les méthodes **equals** et **hashCode**() sont étroitement liées, le **hashCode**() sert à **optimiser l'equals**
* La valeur du hash code est essentiellement utilisée par les **collections** de type **Hashxxx** qui utilisent la valeur du hash code pour améliorer leur performance, elle a pour objectif de fournir un code de hashage (valeur de type **int**)
* Ces classes sont amenées à tester l'égalité de 2 objets, elles comparent d'abord les valeurs des hash codes des 2 objets, si les hash codes sont différents, l'equals() n'est même pas appelé car les objets sont considérés comme différents
* Il est donc nécessaire de redéfinir les méthodes hashCode() et equals de manière coordonnée si l'une ou l'autre est redéfinie

* Contraintes à respecter lors de la redéfinition :
  * La valeur renvoyée doit être **constante** lors de plusieurs invocations sur un même objet durant la durée de vie de l'application. Cette valeur n'a cependant pas d'obligation à être constante entre plusieurs exécutions de l'application
  * **Deux objets égaux** (l'invocation de la méthode equals() sur une instance avec l'autre en paramètre renvoie true) doivent obligatoirement avoir le **même hash code.**
  * Si deux objets ne sont pas égaux en invoquant la méthode equals, alors l'invocation de la méthode hashCode() de chacun des objets n'a pas l'obligation de renvoyer des valeurs entières différentes

Méthode : **toString()**

* Cette méthode est utilisée par la machine Java toutes les fois où elle a besoin de représenter un objet sous forme d'une chaîne de caractères
* Par exemple, la méthode ***System.out.println*** prend en paramètre un objet de type ***String***. Lorsque l'objet passé en paramètre n'est pas de type **String,** la machine Java a besoin de convertir cet objet en un objet de type **String.** Cette conversion s'effectue par appel à la méthode ***toString*()**
* Les chaînes de caractères retournées par la méthode ***toString()*** de la classe ***Object*** ont toutes cette forme : le nom de la classe, suivie du caractère @, et son adresse en hexadécimal

Garbage Collector

* Lorsqu'il n'existe plus aucune référence à un objet, l'emplacement correspondant pourrait être libéré et être utilisé pour autre chose.
* Pour des soucis d'efficacité Java n'impose pas que cela se fasse immédiatement.
* L'objet est devenu candidat au ramasse-miettes (garbage collector). Le garbage collector est un mécanisme de gestion automatique de la mémoire, il libère la mémoire des objets qui ne sont plus utilisés et donc référencés

Mot clé : **final**

* Variable "**final**" : indique que cette variable est une constante et que sa valeur ne pourra plus être modifiée.
  * private **final** int limite = 10;
* Méthode "**final**" : indique que cette méthode ne pourra pas être redéfinie
* Classe : on ne peut pas dériver une classe déclarée "**final**"

Mot clé : **static**

* Le mot-clé **static** est utilisable pour des variables, méthodes, classes internes ou blocs de code
* Dépend de la **classe** et non d'une instance de classe

Variables de classe

* Les variables de classe sont **communes** à **toutes** les **instances** de votre classe, ceci est possible grâce au mot clé ***static***.
  * public **static** final String sigleEuro = "€";
* L'initialisation d'un champ usuel est faite à la création d'un objet de la classe, l'initialisation des champs ***static*** d'une classe est faite avant la première utilisation de la classe et donc cette variable sera communes aux différentes instances de la classe
* Une variable de classe ne dépend donc pas d'une instance particulière, donc (pour peu qu'elle soit accessible), je peux y accéder en la préfixant du nom de la classe : **MaClasse.maVariableStatique**

* **Exemple classe utilitaire java.lang.Math :**

```
public final class Math {

        private Math() {}

        public static final double E = 2.718281828459045235 4;

        public static final double PI = 3.14159265358979323846;
...
}
```

On peut directement utiliser la variable PI :

```
        int rayon = 7;
        double  circonference = 2*Math.PI*rayon;
```

Méthodes de classe

* Tout comme les variables de classe, on peut créer des méthodes de classe indépendante d'une instance précise
* Ce sont généralement des méthodes utilitaires
  * Exemple dans la classe java.lang.Math, méthode **sqrt** qui renvoie le carré du nombre donné
    * **public static double sqrt(double a)**

String

* La classe permet de **gérer** les **chaînes** de **caractères**
* Elle est déclarée **final**, on ne peut donc pas l'étendre
* L'objet String est stocké dans un **tableau** de **caractères**
* Cette classe est **immuable**, une fois la chaîne stockée on ne peut plus la modifier, dés que l'on modifie, concatène une chaîne il a création d'une nouvelle chaîne

Méthode	Explication
length()	Nombres de caractères de la chaîne
substring(int start) **substring(int start, int end)**	Extraction d'une sous-chaîne
compareTo(String s2) **compareToIgnoreCase(String s2)**	Comparaison de 2 chaînes
startsWith(String s) endsWith(String s)	Vérification du début/fin de la chaîne
equals(String s2) **equalsIgnoreCase(String s2)**	Vérification de l'égalité de 2 chaînes (+ sans tenir compte de la casse)
toUpperCase() **toLowerCase()**	Retourne une nouvelle chaîne avec les caractères de la chaîne initiale en majuscules/miniscules
toCharArray()	Retourne le tableau de caractères
indexOf(String s) **lastIndexOf(String s)**	Indice(dernier indice) de la sous-chaîne/caractère donné
String.valueOf(primitif/Objet/char[])	Retourne l'argument convertit en chaîne
replace(String old, String new)	Remplace les occurrences de la sous-chaîne *old* par *new*

StringBuilder

* Les objets de type **String** sont immuables, ne sont pas modifiables
* Dés que l'on modifie, concatène une chaîne il a création d'une nouvelle chaîne
* Cette gestion des chaînes génèrent une perte de temps dans les programmes qui manipule intensivement les chaînes
* La classe **StringBuilder** est dédiée à la manipulation de chaînes
* La méthode principale **append** permet d'ajouter une chaîne à la fin

Constructeurs StringBuilder

* **StringBuilder**()
  * Construit un StringBuilder sans caractères, avec la capacité initiale de 16 caractères
* **StringBuilder**(CharSequence seq)
  * Construit un StringBuilder qui contient la séquence spécifiée
  * CharSequence est une interface, String et StringBuilder implémente notamment cette interface
* **StringBuilder**(int capacity)
  * Construit un StringBuilder sans caractères avec la capacité initiale spécifiée
* **StringBuilder**(String str)
  * Construit un StringBuilder qui contient la chaîne de caractère spécifiée