Java

Éléments de base :

Opérations :

Affectation (=)
Addition | Soustraction | Multiplication | Division (**+** | **-** | ***** | **/**)
Modulo (**mod**)

Opérateurs logiques :

égale (**==**)
non égal (**!=**)
inférieur/ supérieur ou égal (**<=** | **>=**)
ET logique (**&&**)
OU logique (**||**)
Inverse (**!**)

Raccourcis :

a = a + b => a **+=** b
a = a - b => a **-=** b)
a = a * b => a ***=** b)
a = a / b => a **/=** b)

Déclarer une variable :

niveau type nom_variable **;** (Aucune valeur à assigner pour l'instant) (**niveau** réfère au niveau de contrôle)
niveau type nom_variable **=** valeur **;** (Déclarer la variable avec une valeur)
final type nom_variable **=** valeur **;** (On utilise le mot 'final' pour déclarer une constante)

Types de données :

String (avec un S majuscule) => **String** text = "Hi World !" ;
int => **int** age = 19 ;
double => **double** prix = 10.56854 ;
float (pareil que double mais moins précis) => **float** prix = 10.56 ;
boolean (prend true ou false) => boolean condition = false ; (on utilise le '**!**' pour le not => **!**condition)

Les String :

Concaténation :

String favoriteCity = "Buenos Aires" ;
int numberOfTrips = 5 ;
String story = "I've traveled to " + favoriteCity + " " + numberOfTrips + " times!" ;
=> "I've traveled to Buenos Aires 5 times!"

Commentaires :

Commenter une ligne : **//**

Commenter un paragraphe :
/** ligne1
***** ligne2
***** ligne 3 ***/**

Portée des variables :

Variable globale : disponible dans tout le bloc du code entre les { } du code.
Variable locale : n'est disponible que dans le contexte où il a été déclaré dans un sous bloc de { }.

Niveaux de contrôle :

public : visible pour tous
protected (protégé) **:** visible pour le package et l'ensemble de ses sous-classes.
package-protected (protégé par paquet) **:** généralement visible uniquement par le package dans lequel il se trouve (paramètres par défaut).
private (privé) **:** accessible uniquement dans le contexte dans lequel les variables sont définies (à l'intérieur de la classe dans laquelle il est situé).

Boucles :

boucle for : (exemple)
```
for (int i = 0 ; i < 5 ; i++) {
// Instructions ;
}
```

boucle while : (exemple)
```
while (Conditions) {
// Instructions ;
}
```

boucle do … while : (exemple) => (Cette boucle est exécutée au moins une fois)
```
do {
// instructions
} while (Conditions);
```

Ignorer une étape dans la boucle : (exemple)
```
while (Conditions) {
// Instructions ;
  if (i == 2 || i == 5) {
    continue ;  (<= utilisé pour ignorer une étape)
  }
}
```

Interrompre la boucle : (exemple)
```
while (Conditions) {
// Instructions ;
  if (i == 2 || i == 5) {
    break ;  (<= utilisé pour interrompre et sortir de la boucle)
  }
}
```

if / else if / else :

```
if (condition1) {
// instructions
}
else if (condition2) {
// instructions
}
else {
// instructions
}
```

switch .. case :

```
switch (variable) {
        case 0 :
            Instructions ;
            break;
        case 1 :
            Instructions ;
            break;
        case 2 :
            Instructions ;
            break;
        default :
            Instructions ;
}
```

Type énumération :

```
enum Direction {
north, east, south, west;
}
```

Afficher un élément avec println :

```
System.out.println ("Hello World !") ;
```

Programmation orienté objet :

Les classes :

Creation : (exemple)

```
class Telephone {
            int pixels ;  // => Attributs
            String nom ;
            Marque marque ;

            public Telephone (int pixels, String nom, Marque marque) {  //  => Constructeur
                        this.pixels = pixels ;
                        this.nom = nom ;
                        this.marque = marque ;
            }
}
```

```
class Marque {
	String nom ;

	public Marque (String nom) {
		this.nom = nom ;
	}
}
```

<u>Utilisation</u> : (exemple)

```
public class OC {
	public static void main(String[] args) {
		Marque apple = new Marque (“apple”) ;
		Telephone iphone = new Telephone (34000, “iphone”, apple)
		System.out.println (iphone.nom) ; // => affiche iphone
		System.out.println (apple.nom) ; // => affiche apple
		}
}
```

<u>Héritage</u> : (exemple)

- si Class1 hérite de Class2, alors on ne peut pas changer le type d’une variable p de type Class2 en type Class1
- Tout les méthodes de Class2 sont valables pour Class1
- Si une méthode de Class2 est précédée du mot ‘final’, Class1 ne peut donc pas hériter cette méthode et l’utiliser. (de même si le mot ‘final’ précède Class2, alors Class1 ne peut plus hériter de cette classe)

```
public class OC {
	public static void main(String[] args) {
		Voiture voiture = New Voiture () ;
		Bateau bateau = New Bateau () ;
		voiture.start() ;  // => Affiche VROOOOOM
		bateau.start() ;  // => Affiche VROOOOOM
	}
}

class Vehicule {
	void start () {
		System.out.println (“VROOOOOM”) ;  // => Méthode
	}
}

class Voiture extends Vehicule {  // => la classe Voiture hérite les fonctionnalités de Vehicule
}

class Bateau extends Vehicule {  // => héritage avec ‘extends’
}
```

<u>Polymorphisme (surcharger les fonctionnalités de la classe mère)</u> : (exemple)

```
public class OC {
	public static void main(String[] args) {
		Voiture voiture = New Voiture () ;
		Bateau bateau = New Bateau () ;
		voiture.start() ;  // => Affiche VROOOOOM et Allumage feux.
```

```
                bateau.start() ;  // => Affiche VROOOOOM
        }
}
class Vehicule {
        void start () {
                System.out.println ("VROOOOOM") ;  // => Méthode
        }
}
class Voiture extends Vehicule {
        @Override  //  => montrer que c'est une surcharge d'une methode de la classe mère
        void start() {
                super.start() ;  // => Appeler le start() de la classe mère en utilisant 'super'
                allumerFeux () ;  // => remplacer le comportement de start de voiture par allumerFeux()
        }
}

        void allumerFeux() {
                System.out.println ("Allumage feux") ;
        }
}

class Bateau extends Vehicule {  // => héritage grâce à extends
}
```

Les tableaux : (Taille fixe)

Déclarer le tableau :
```
Type_Element[ ] Nom_Tableau = New Type_Element[Nombre_Element] ;
```
(les indices du tableau vont de **0** à **n-1**)

exemple 1 :
```
String[ ] tableau = New String[10] ;  // => tableau contenant 10 élément de type string
```

exemple 2 : *On peut le déclarer l'instancier en 2 étapes*
```
int[ ] cupsOfCoffeePerDayOfTheWeek ; // => Déclaration
cupsOfCoffeePerDayOfTheWeek = New int[7] ; // => Instanciation
```

exemple 3 : *Tableau multidimensionnelle*
```
String[ ][ ] myTheatreSeats = New String[n][m] ; // tableau de n lignes et m colonnes
```

Changer et accéder à une valeur du tableau :
```
String[ ] tableau = New String[10] ;
tableau[3] = "index 3" ;  // => on change la valeur d'indice 3 par "index 3"
```

Taille du tableau : taille = tableau.length() ; // taille contient la taille du tableau

Les listes : (Taille variable)

Déclarer une liste :
```
List<Type_Element> Nom_List = new ArrayList<Type_Element>() ;
```
!!! Type_Element => Integer / Double / Boolean / Float (doivent être écrit de cette manière)

exemple 1 : `List<String> myList = new ArrayList<String>() ;`
exemple 2 : `List<Integer> myList = new ArrayList<Integer>() ;`

Fonctions sur liste : `List<Type> myList = new ArrayList<Type>() ;`

Ajouter élément => myList.add(**Element**) ; // ajouter un élément avec add(**Element**)
Insérer élément => myList.add(**Indice**, **Element**) ; // insérer **Element** à la position **Indice** (décale la liste)
Remplacer élément => myList.set(**Indice**, **Element**); //Change l'ancien valeur a la position **Indice** par **Element**
Supprimer élément => myList.remove(**Indice**) ; // Supprime l'élément à la position **Indice**
Taille de la liste => taille = myList.size() ; // taille comporte la taille de la liste myList

Les dictionnaires :

Déclarer dictionnaire :
Map<Type_cle, Type_valeur> Nom_Dic = new Hashmap<Type_cle, Type_valeur>() ;
!!! les types => Integer / Double / Boolean / Float (doivent être écrit de cette manière).

Fonctions sur dictionnaire :
Map<Type_cle, Type_valeur> myMap = new Hashmap<Type_cle, Type_valeur>() ;
Ajouter (**cle**, **valeur**) => myMap.put(**cle**, **valeur**) ;
Accéder à la valeur associé à **cle** => valeur = myMap.get(**cle**) ;
Supprimer élément associé à **cle** => myMap.remove(**cle**) ;
Taille de dictionnaire => taille = myMap.size() ;

Les fonctions :

Déclarer une fonction :

exemple :

```
public static int sum(int a, int b)
{
  int calc = a + b;
  return calc;
}
```

les type valeur et les type référence :

Type valeur : (exemple) => normal

```
int a = 10
int b = a
System.out.println(a);  // => 10
System.out.println(b);  // => 10
a = 15;
System.out.println(a);  // => 15
System.out.println(b);  // => 10
```

Type référence : (exemple) => pointeur

```
public class Car {
  String color = "red"
}
```

```
Car car = new Car();
Car carToPaint = car;
System.out.println(car.color); // -> red
System.out.println(carToPaint.color); // -> red
carToPaint.color = "yellow";
System.out.println(car.color); // -> yellow
System.out.println(carToPaint.color); // -> yellow
```

La récursivité : *(exemple)*

```
public static int factorial(int n) {
      if (n == 1) return 1;  //  Condition d'arrêt
      else return n * factorial (n-1);  // Appel de la récursivité
   }
}
```

Les exceptions :

```
try {
// un peu de code
// une fonction à essayer pouvant générer une erreur
// encore du code
}
catch (ExceptionName e) {
// code à exécuter au cas où l'essai ne fonctionnerait pas et qu'une erreur se produirait
}
```

quelques nom d'exception :

1. NullPointerException : levée lorsqu'une référence null est utilisée où une référence d'objet valide est requise.
2. ArrayIndexOutOfBoundsException : levée lorsqu'un indice de tableau hors limites est utilisé.
3. ClassCastException : levée lorsqu'un objet est converti en une classe incompatible.
4. IllegalArgumentException : levée lorsqu'un argument illégal est passé à une méthode.
5. NumberFormatException : levée lorsqu'une chaîne de caractères ne peut pas être convertie en un nombre.
6. IOException : levée lorsqu'une erreur d'entrée/sortie se produit.
7. IllegalStateException : levée lorsqu'un objet se trouve dans un état incompatible avec la méthode appelée.
8. UnsupportedOperationException : levée lorsqu'une méthode non prise en charge est appelée.

Les fichiers :

Lire un fichier ligne par ligne :

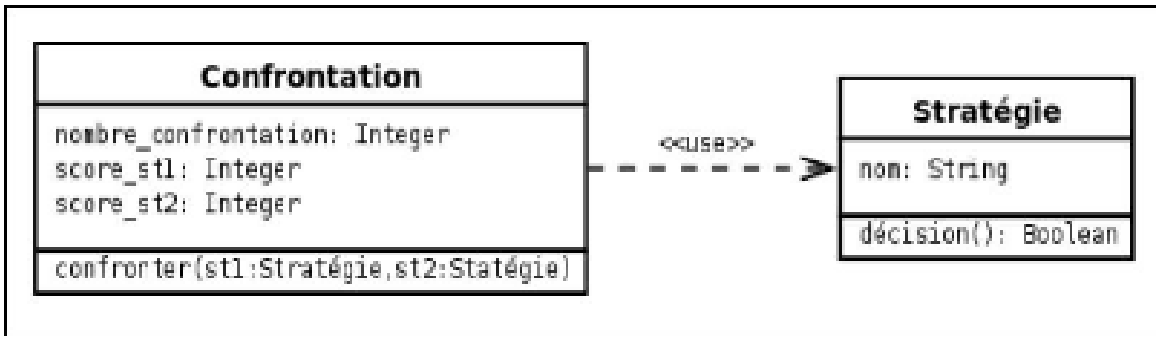
```
public static void main(String[] args) {
// Création d'un fileReader pour lire le fichier
FileReader fileReader = new FileReader("/path/to/the/file");
// Création d'un bufferedReader qui utilise le fileReader
BufferedReader reader = new BufferedReader (fileReader);
   try {
      // une fonction à essayer pouvant générer une erreur
      line = reader.readLine();
      while(line != null) {
            // affichage de la ligne
            System.out.println(line);
            // lecture de la prochaine ligne
            line = reader.readLine();
   }
catch (IOException e) {
    e.printStackTrace();
}
reader.close();
}
```

Ecrire dans un fichier :

```
public static void main(String[] args) {
// Création d'un fileWriter pour écrire dans un fichier
FileWriter fileWriter = new FileWriter("/path/to/the/file", false);
// Création d'un bufferedWriter qui utilise le fileWriter
BufferedWriter writer = new BufferedWriter (fileWriter);
   try {
      // ajout d'un texte à notre fichier
      writer.write("preferenceNewsletter=false");
      // Retour à la ligne
      writer.newLine();
      writer.write("preferenceColor=#425384");
catch (IOException e) {
      e.printStackTrace();
}
writer.close();
}
```

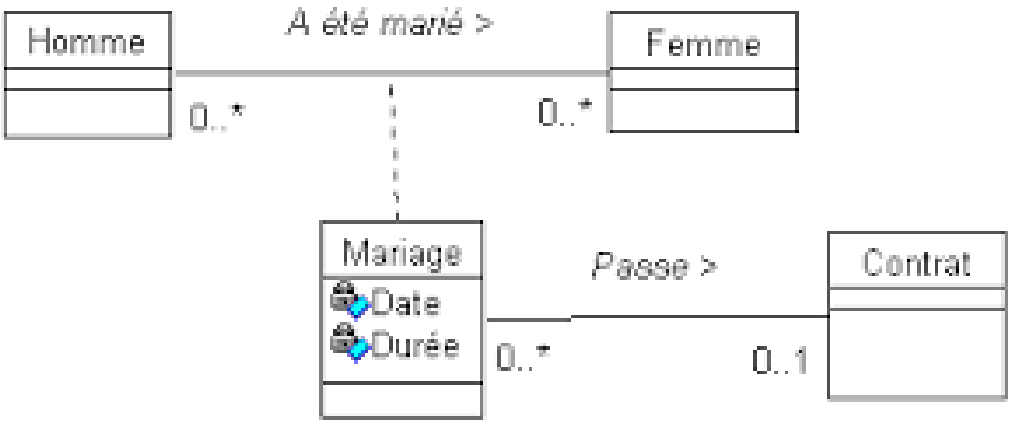
Relations entre les classes :

Relation de dépendance :
On dit qu'il y a relation de dépendance entre une classe A et une classe B si la classe A fait référence à la classe B dans son texte. On dit que A dépend de B.



Relation structurelle :
Une relation de dépendance est dite relation structurelle si elle dure dans le temps. C'est par exemple le cas quand B est le type d'attribut de la classe A.

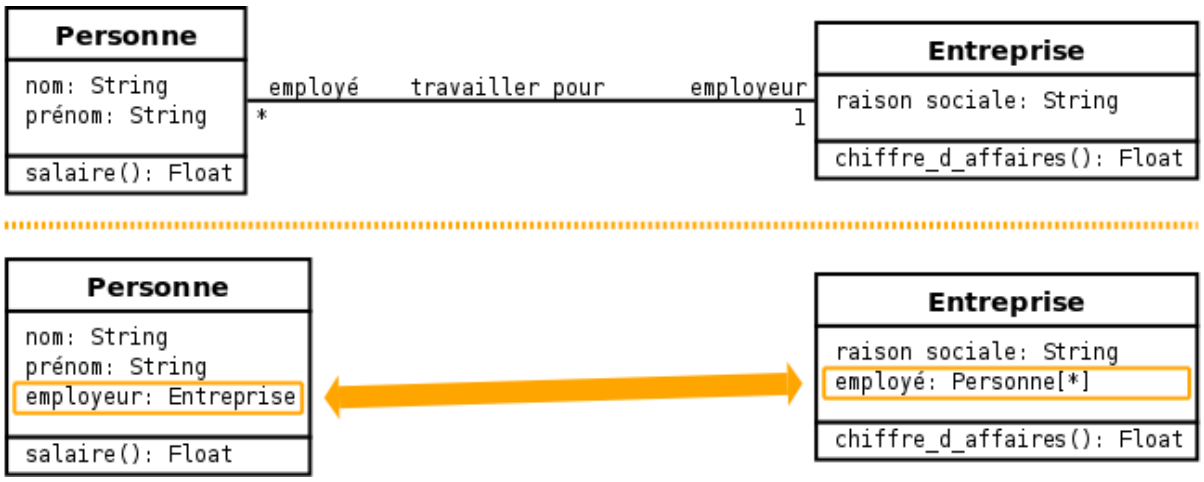
Relation d'association :
Définition : Une relation d'association est une relation entre deux classes qui traduit un couplage faible : chaque classe pourrait être considérée indépendamment de l'autre.
Propriété : Les objets des deux classes sont relativement indépendants et leurs durées de vie ne sont pas liées.



Expliquer une relation : nom et rôle
Il est essentiel de savoir ce que représente une relation et donc de préciser sa signification.

On peut nommer la relation :
- C'est un verbe (travaille-pour, emploie, etc.).
- Le nom est placé au milieu de la relation.
- La lecture se fait de gauche à droite ou de haut en bas. Une direction peut indiquer le sens de lecture quand il n'est pas naturel.

On peut donner le rôle joué par les objets dans la relation (extrémités de la relation).



Multiplicité (cardinalité) :

Objectif : indiquer combien d'objets peuvent/doivent prendre part à la relation :
forme générale : min..max
exemple : 1..4 le nombre d'objet est compris entre min et max inclus la valeur de max peut être « * » : nombre quelconque (« illimité ») un entier (ex : 4) : le nombre exact d'objets (simplification d'écriture de 4..4) 0..1 : optionnel * ou 0..* : 0 ou plusieurs (un nombre quelconque y compris 0) 1..* : au moins 1 (un nombre quelconque mais au moins 1) une multiplicité omise crée une ambigüité (souvent considérée comme 1..1)

Navigation :
Définition : La navigation est le fait de traverser une relation (en fait un lien) : à partir d'une objet d'une extrémité, on peut atteindre les objets de l'autre extrémité.
Exemple : Depuis une personne, Xavier, on obtient l'entreprise où Xavier travaille (N7).

Propriété : La relation d'association est bidirectionnelle : navigable dans les deux sens.
Exemple : Partant de Xavier, on trouve son employeur, N7, et inversement, de N7 on trouve Xavier parmi les employés.

Principe : Rôles (et noms) d'association sont utilisés pour la navigation (termes en bleu)
Sens de navigation : Expliciter dans quel sens une relation peut être traversée.
Flèche à l'extrémité d'une relation : la relation peut être traversée dans le sens de la flèche
Croix à une extrémité : interdit le sens de navigation qui mène à la croix Train Personne 0..1 0..1 conducteur passager * d'un objet train on peut obtenir le conducteur ou les passagers d'une personne on ne peut pas retrouver le train (Train n'apparaît pas dans Personne)

Relation d'agregation :

Définition : La relation d'agrégation est un cas particulier d'association où une classe est prépondérante par rapport à l'autre : ses objets (le tout) agrégent d'autres objets (les parties).
Notation : On utilise un losange vide à l'extrémité de la relation, côté tout. A B tout partie Exemple : Un département de formation s'appuie sur des gestionnaires et des enseignants.
losange du coté de departement