Antisèche

POO avec Java

Références

- https://fr.wikibooks.org/wiki/Programmation_Java/Introduction
- cheatography.com/43384/cs/12920/
- http://www-inf.int-evry.fr/cours/CSC4002/Diapos/coursjava-diapos.pdf

Types

TYPES	VALEURS	OPÉRATEURS	EXEMPLES
byte, short, int, long	entiers	+ - * / %	99 -12 458735
float, double	nombres à virgule	+ - * /	3.14 -2.5 6.098e7
boolean	vrai/faux	&& || !	true false
char	caractère (lettre, chiffre, symbole…)		‘A’ ‘1’ ‘\n’
String	chaîne de caractères	+	"AB" "Salut" "2.5"

Créer des variables

On crée toujours une variable en entrant son type, son nom puis sa valeur :

```
int age; // age ne vaut rien
age = 16; // age vaut 16
int age; // age ne vaut rien à nouveau

float prix = 74.99f; // prix vaut 74.99
int prix2 = (int) prix; // prix2 vaut 74

String prenom = “Alice”;
```

Modifier des variables

On peut modifier des variables en utilisant des opérateurs, tant que la nouvelle valeur qu'on veut leur attribuer est compatible avec le type de ces variables :

```
int age = 10;
age += 1; // équivaut à age = age + 1
prix /= 2;
prenom += " au pays des merveilles"
```

Tableaux

Les tableaux sont des objets de taille fixe. Ils peuvent stocker plusieurs valeurs (variables ou objets de même type), mais on ne peut pas en ajouter de nouvelles

```
int[] maListe = {2,4,6,8};
int[] maListe2 = new int[4]; // tableau vide de taille 4
```

Les objets de classe `ArrayList` sont également des tableaux. Ils peuvent changer de taille de façon dynamique

```
ArrayList maListe = new ArrayList(); // collection vide
maListe.add(2); // on ajoute une valeur à la collection
```

Comparer des variables

OPÉRATEURS	SIGNIFICATION	VRAI	FAUX
==	égal à	2 == 2	2 == 3
!=	différent de	2 != 3	2 != 2
<	inférieur à	2 < 3	2 < 1
<=	inférieur ou égal à	2 <= 2	2 <= 1

Répéter une opération

Les deux boucles ci-dessous affichent le même résultat :

```
int i = 0;
do
{
  System.out.println(i);
  i += 1;
} while (i < 9);
```

```
int i = 0;
while (i < 10)
{
  System.out.println(i);
  i += 1;
}
```

Répéter une opération

Les deux boucles ci-dessous affichent le même résultat :

```
for (int i = 0; i < 10; i++)
{
  System.out.println(i);
}
```

La boucle `for-each` passe en revue les éléments d'une liste :

```
int[] liste = {0,1,2,3,4,5,6,7,8,9};
for (int i : liste)
{
  System.out.println(i);
}
```

Faire des choix

```
int a = 7; // que va afficher le programme ci-dessous ?

if (a > 0) {
 if (a == 5)
   System.out.println(“a vaut 5”);
 else
   System.out.println(“a ne vaut pas 5 mais est positif”);
}
else if (a < 0)
 System.out.println(“a est négatif”);
else
 System.out.println(“a est nul”);
```

Afficher du texte

Le raccourci `sysout` permet d'afficher une chaîne de caractères avant de sauter une ligne :

```
System.out.println("Voici une ligne");
```

L'instruction suivante fait la même chose sans sauter de ligne :

```
System.out.print("Voici une ligne. ");
System.out.print("Je suis sur la même ligne");
```

On peut **contaténer** un `String` avec d'autres types de variable :

```
int age = 5;
System.out.print("Alice a "+ age +" ans");
```

Faire une fonction

type renvoyé | nom | type de l'argument | argument

signature

```
public float Moyenne(int[] liste)
{
  float somme = 0f;
  float compteur = 0f;
  for (int i : liste){
    somme += i;
    compteur += 1;
  }
  return (somme/compteur);
}
```

corps

variables locales

valeur renvoyée

Classes et objets

nom de classe

méthode main()

```
public class Principale
{
 public static void main(String[] args)
 {
  String prenom = “Alice”;
  System.out.println(“Bonjour ” + prenom);
 }
}
```

instructions

corps

Classes et objets

2 étapes pour créer un objet :

- Disposer d'une **référence** : `Eleve e;`
- **Instancier** l'objet : `new` + appel à un constructeur

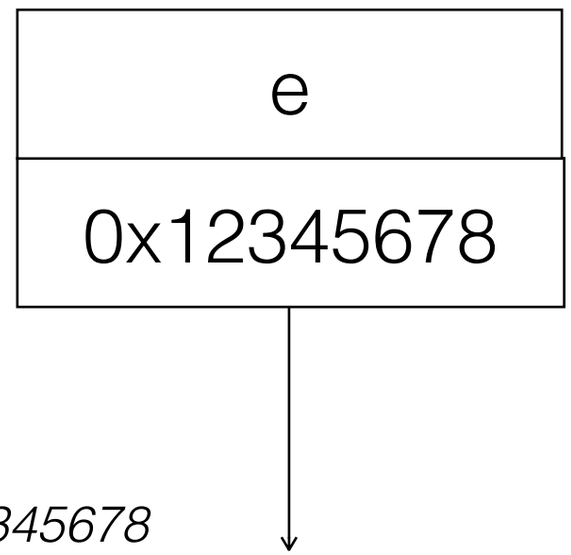
```
e = new Eleve("Carroll", "Alice", [10.5]);
```

Le tout peut s'écrire en une ligne :

```
Eleve e = new Eleve("Carroll", "Alice", [10.5]);
```

Classes et objets

(objet de classe `Eleve` *stockant une référence mémoire, par exemple 0x12345678)*



(espace mémoire référencé)

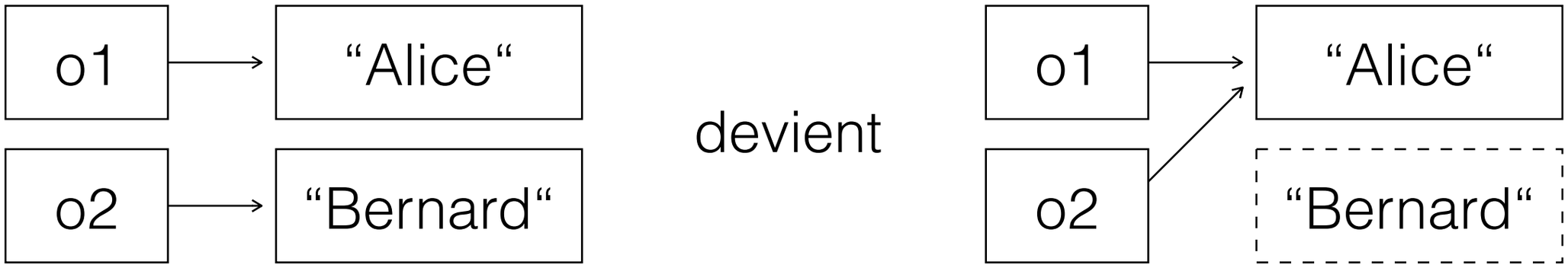
“Alice“, “Carroll”, [10.5]

```
System.out.println(e); // Affiche 0x12345678
System.out.println(e.getNom()); // Affiche “Carroll”
```

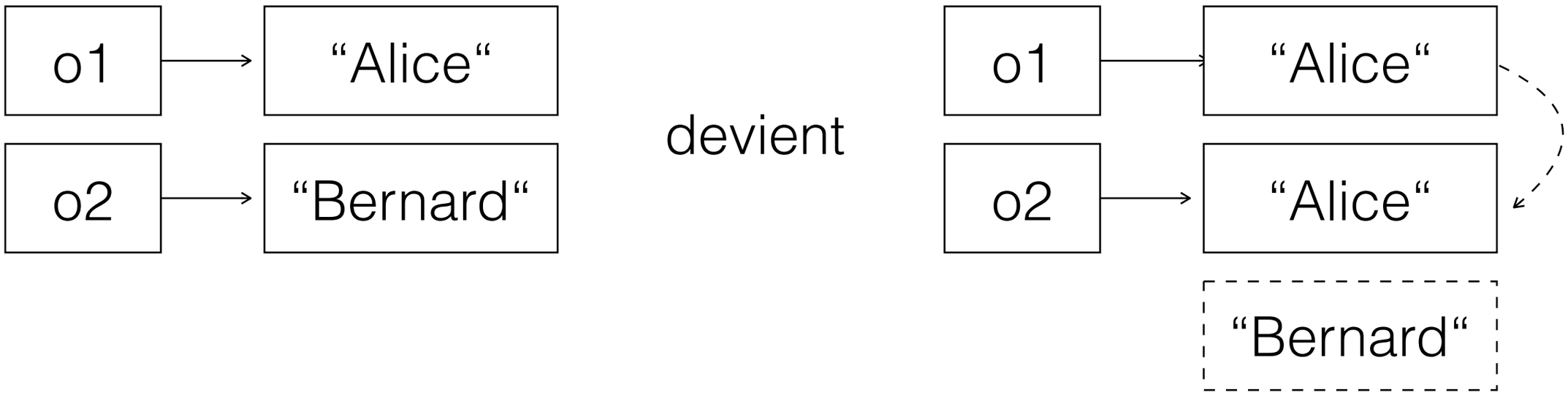
Copier un objet

Les objets sont manipulés par des références

`o2 = o1` : copie la référence o1 dans o2



`o2 = o1.clone()` : o2 référence un nouvel objet (copie de o1)



Comparer des objets

`o1 == o2`

Teste si les références sont égales (c-à-d si les deux objets pointent vers la même adresse en mémoire)

`o1.equals(o2)`

Teste si le contenu de l'objet référencé par o2 est identique au contenu de l'objet référencé par o1

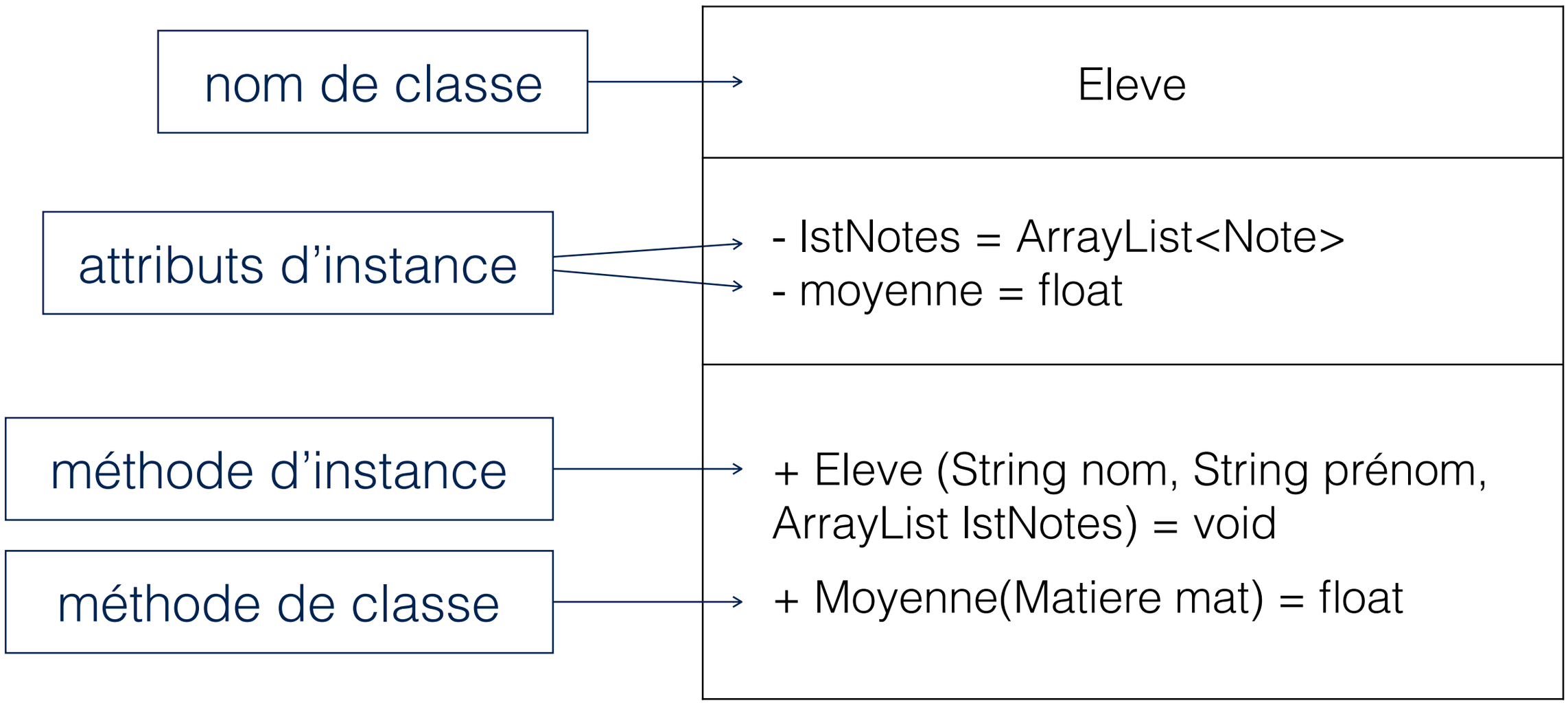
UML (*Unified Modeling Language*)

Il est plus facile de modéliser une classe et ses méthodes graphiquement qu'avec un bloc de lignes

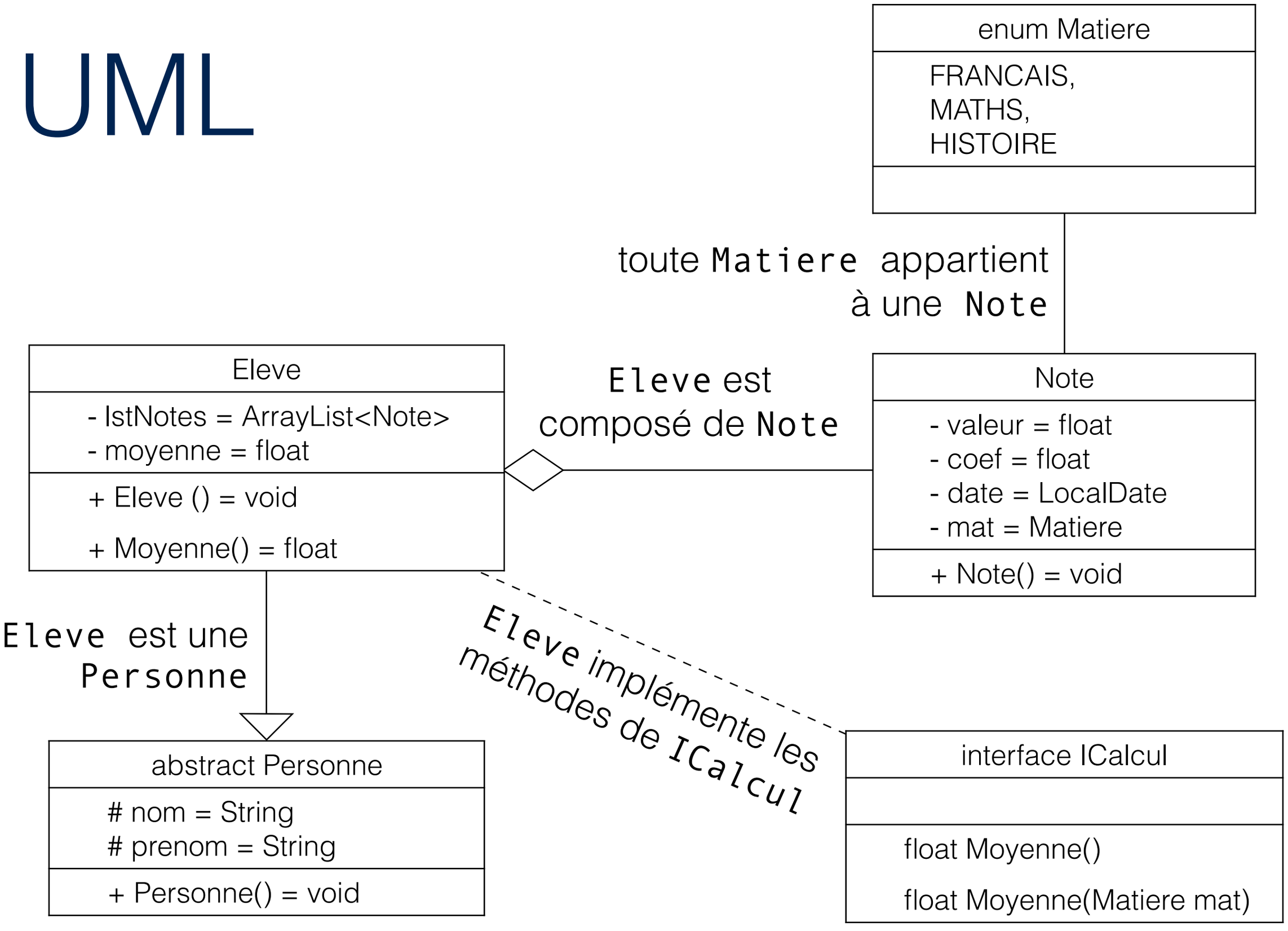
UML permet de concevoir des diagrammes modélisant toutes les étapes du développement d'une application informatique

Cet outil aide à la conception d'un algorithme

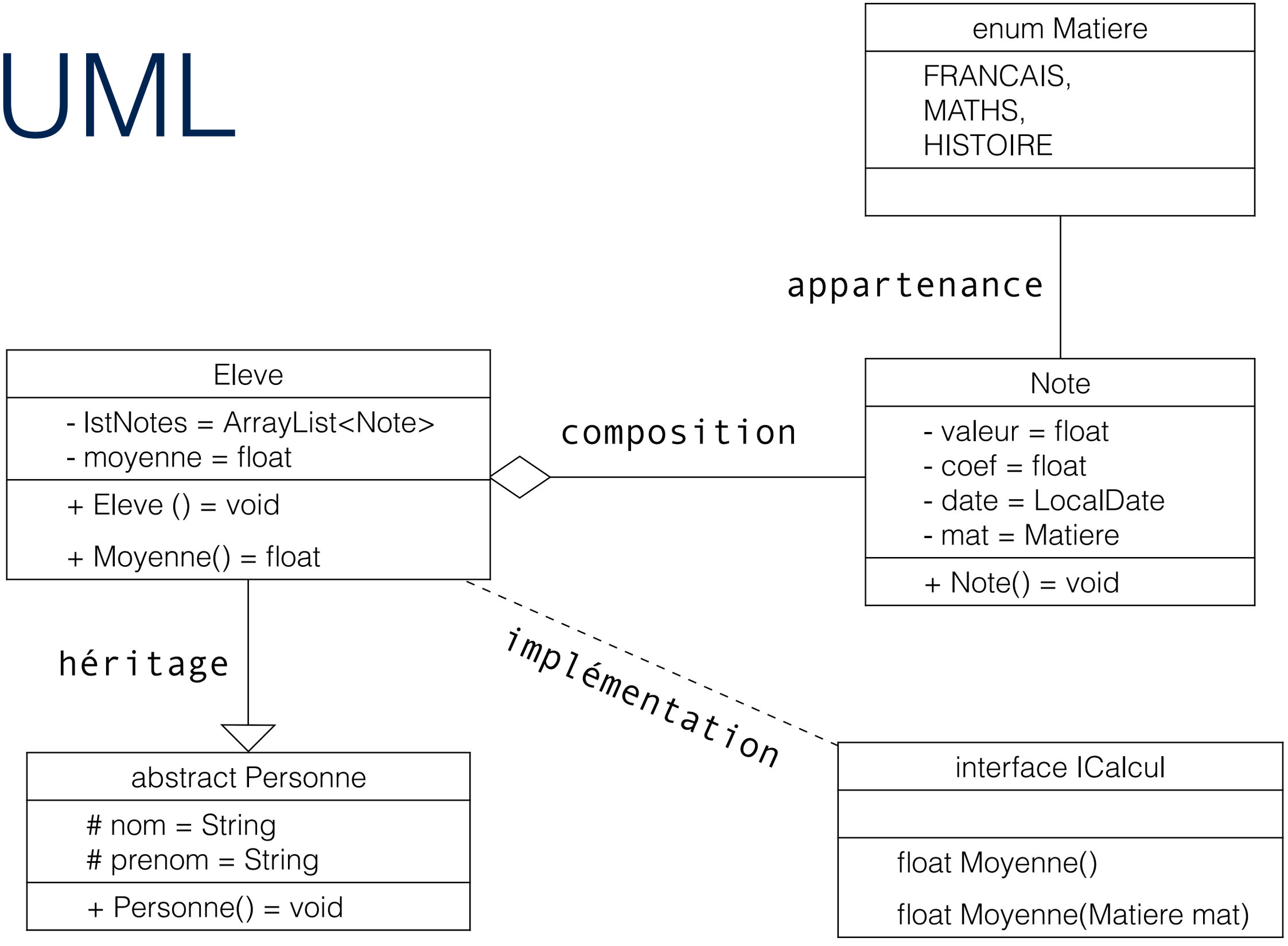
UML


nom de classe
Eleve
attributs d'instance
- lstNotes = ArrayList<Note>
- moyenne = float
méthode d'instance
+ Eleve (String nom, String prénom, ArrayList lstNotes) = void
méthode de classe
+ Moyenne(Matiere mat) = float

UML


enum Matiere
FRANCAIS,
MATHS,
HISTOIRE
toute Matiere appartient à une Note
Eleve
- lstNotes = ArrayList<Note>
- moyenne = float
+ Eleve () = void
+ Moyenne() = float
Eleve est composé de Note
Note
- valeur = float
- coef = float
- date = LocalDate
- mat = Matiere
+ Note() = void
Eleve est une Personne
Eleve implémente les méthodes de ICalcul
abstract Personne
nom = String
prenom = String
+ Personne() = void
interface ICalcul
float Moyenne()
float Moyenne(Matiere mat)

UML


enum Matiere
FRANCAIS,
MATHS,
HISTOIRE
appartenance
Eleve
- lstNotes = ArrayList<Note>
- moyenne = float
+ Eleve () = void
+ Moyenne() = float
composition
Note
- valeur = float
- coef = float
- date = LocalDate
- mat = Matiere
+ Note() = void
héritage
implémentation
abstract Personne
nom = String
prenom = String
+ Personne() = void
interface ICalcul
float Moyenne()
float Moyenne(Matiere mat)

Modificateurs d’accès

Les classes, les attributs et les méthodes bénéficient de **niveaux d'accessibilité**, qui indiquent dans quelles circonstances on peut accéder à ces éléments

Parmi ces niveaux, on distingue d’abord 4 **niveaux d’encapsulation**, dont 3 correspondent à un mot-clé :

- `private`
- `protected`
- `public`

Encapsulation

RESTRICTION	SYMBOLE	SIGNIFICATION
private	—	accessible seulement de l'intérieur de la classe
<default>		accessible aux classes du même package
protected	#	accessible aux classes du package et aux classes dérivées (même hors package)
public	+	accessible à toutes les classes

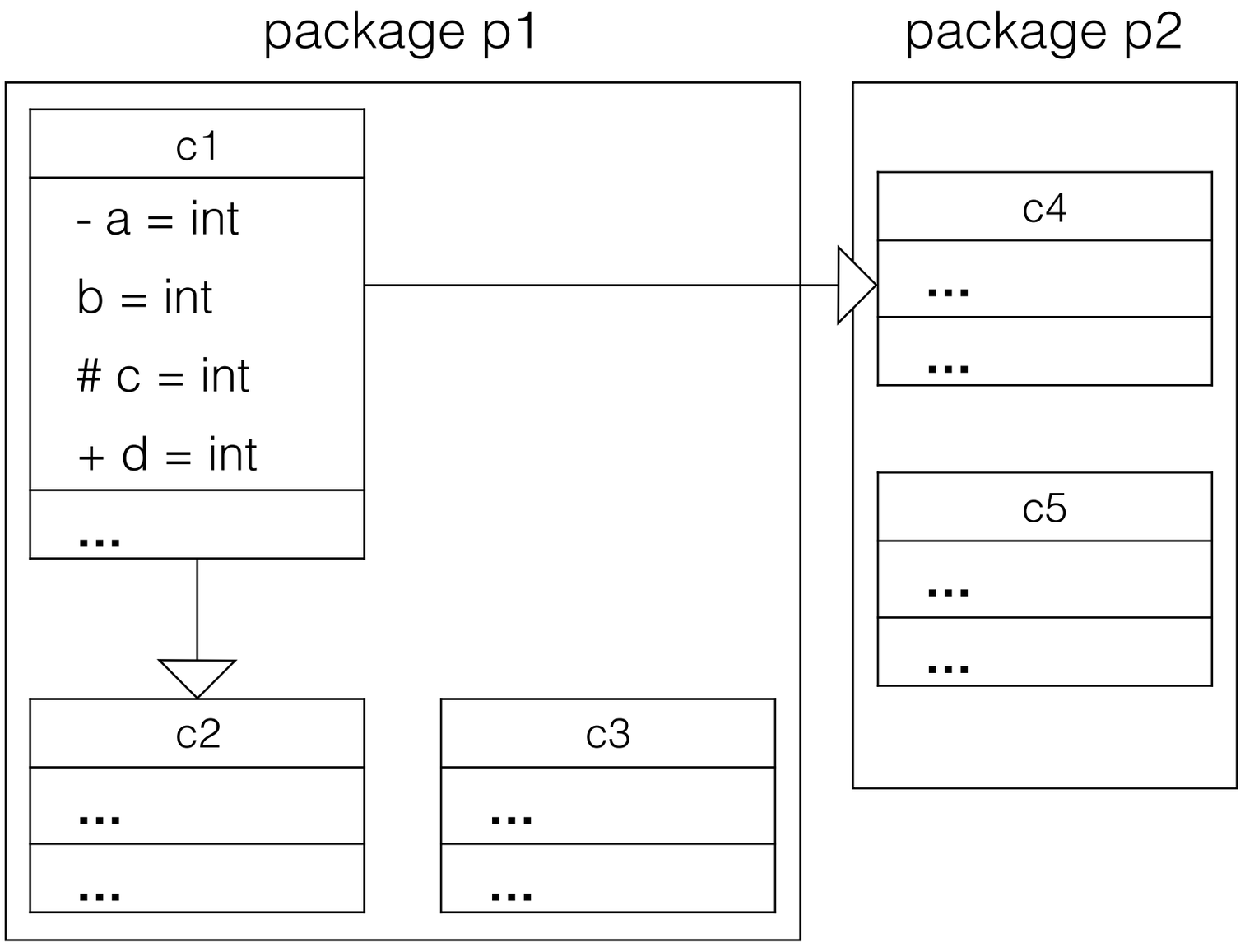
Encapsulation

```
package p1;
class c1 {
  private int a;
  int b;
  protected int c;
  public int d;
}

class c2 extends c1 {
  …
}

class c3 {
  …
}
```

	accessible de…			
	c2	c3	c4	c5
a	—	—	—	—
b	oui	oui	—	—
c	oui	oui	oui	—
d	oui	oui	oui	oui

```
package p2;
class c4 extends c1 {
  …
}

class c5 {
  …
}
```

Encapsulation (UML)



accessible de…	c2	c3	c4	c5
a	—	—	—	—
b	oui	oui	—	—
c	oui	oui	oui	—
d	oui	oui	oui	oui

Autres modificateurs

```
abstract … // classe ou méthode abstraite


final variable … // valeur figée
final objet … // référence figée
final Methode() … // ne peut pas être modifiée dans une classe dérivée

final Classe … // ne peut pas avoir de classe dérivée


static Methode() … // peut être appelée sans instancier sa classe
static attribut … // précise qu'il s'agit d'un attribut de classe
```

Constructeur

Il permet de définir un objet automatiquement :

- son nom est identique à la classe
- il ne renvoie aucune valeur
- on l'appelle à travers `new`

Si aucun constructeur n'est spécifié, le compilateur en fabrique un par défaut qui initialise les attributs à 0

La **surcharge** des méthodes permet à une classe d'avoir plusieurs constructeurs ayant une **signature** différente (nombre d'arguments, type des arguments)

Constructeur

```
public class Eleve extends Personne
{
 private ArrayList lstNotes =
  new ArrayList();
 protected float moyenne;

 public Eleve(String nom, String prenom, ArrayList
 lstNotes){
  super(nom, prenom);
  this.lstNotes = lstNotes;
 }
}
```

nom de classe

attributs de classe

constructeur

attributs hérités de la classe de base `Personne`

attribut défini lors de l'instanciation de l'objet

Détruire un objet

Un objet est automatiquement détruit par le *garbage collector* lorsqu'il n'est plus référencé (pas de technique particulière)

La destruction d'un objet n'est pas garantie (par ex. le programme peut se terminer avant que l'objet ne soit détruit)

Si la classe décrit une méthode appelée `finalize()`, celle-ci est appelée avant la libération de la mémoire de l'objet

Classe abstraite

Une classe abstraite **ne peut pas être instanciée**

```
ClasseAbstraite c1 = new ClasseDerivee(); // valide
ClasseAbstraite c2 = new ClasseAbstraite(); // invalide
```

Une classe abstraite est constituée de méthodes incomplètes qu'il faudra redéfinir dans la classe dérivée

```
abstract void Methode();
```

La même méthode peut donc avoir un comportement différent selon les situations (**polymorphisme**)

Classe abstraite

```
abstract class Personne
{
 protected nom;
 protected prenom;
 public Personne(nom, prénom) { // constructeur
  this.nom = nom;
  this.prenom = prenom;
 }
 abstract void Afficher(); // méthode abstraite
}

Personne a = new Eleve(“Carroll”, “Alice”);
Eleve b = new Eleve();
```

Héritage

L'héritage est la définition d'une classe par extension des caractéristiques d'une autre classe

```
public class Eleve extends Personne {}
```

Par exemple, la classe `Eleve` hérite de `Personne` et de ses attributs `nom` et `prenom`

La classe qui est héritée est qualifiée de super-classe, de classe mère ou de **classe de base**. Elle peut être abstraite ou non

Interface

En Java, on ne peut hériter que d'une seule classe de base. On a donc recourt aux interfaces pour permettre aux classes d'implémenter des méthodes prédéfinies

```
public class Eleve implements ICalcul {}
```

Contrat : toute classe qui implémente une interface doit être capable de faire ce qui est présenté dans cette interface

Une interface n'est **pas instanciable**. En implémentant une interface, on doit obligatoirement redéfinir ses méthodes

Les méthodes d'interface sont publiques et statiques par défaut

Interface

```
public interface ICalcul {
 void Moyenne(); // public static par défaut
 void Moyenne(Matiere mat); // surcharge
}

public class Eleve implements ICalcul {
 // il faut définir les méthodes d'interface
 public void Moyenne() {
  float somme = 0f, float compteur = 0f;
  for (int i : liste) {
   somme += i;
   compteur += 1;
  }
 System.out.println(somme/compteur);
}
```

Extensions (*packages*)

Les classes créées doivent être stockées dans un fichier appelé **package**. Pour inclure ma classe dans l'extension `m2i.formation.exemple`, il faut écrire au début du fichier :

```
package m2i.formation.exemple;
```

Si une classe est créée sans spécifier de package, elle sera stockée dans un package par défaut (plus difficile à trouver)

Il existe plusieurs packages déjà créés. Ils contiennent des classes qu'on peut **importer** au début de notre code, et dont on peut utiliser les méthodes

Extensions (*packages*)

```
// importer une classe permettant de créer et manipuler
des tableaux
import java.util.ArrayList;


// importer une classe permettant de créer et manipuler
des valeurs aléatoires
import java.util.Random;

// importer toutes les classes de l'extension java.util
import java.util.*;

// importer un package que j'ai créé moi-même
import mon.package.MaClasse;
```