Notes de cours
Programmation II
INF2120

Version 1.0
10 septembre 2013

L'héritage en Java

Mélanie Lord
UQAM

Table des matières

1. Notions de base

1.1 Généralités

L'héritage est un principe fondamental en programmation orientée objet (POO).

L'héritage permet de créer des classes qui héritent de toutes les caractéristiques (attributs/données et méthodes/comportements) d'une autre classe qu'on appelle **classe de base**, **classe mère (ou "parent")** ou **superclasse**. On peut construire ainsi une **hiérarchie de classes :**

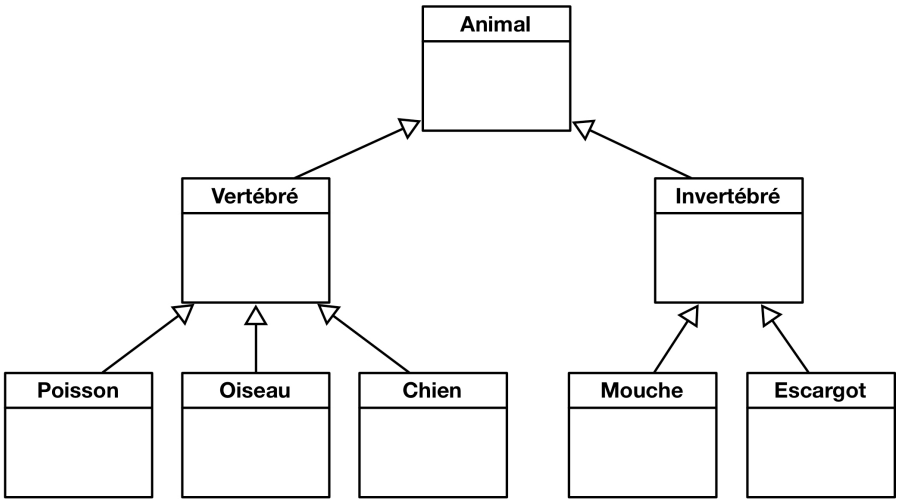


Figure 1.1 Hiérarchie de classes sur les animaux.

Dans une hiérarchie :

- Vers le bas : **spécialisation**
- Vers le haut : **généralisation**

On appelle **classes dérivées** ou **classes enfants** ou **sous-classes** les classes qui héritent d'une classe de base.

Pour spécialiser (enrichir) la classe de base, une **classe dérivée** peut définir des **caractéristiques supplémentaires** (attributs et méthodes). Ainsi, une classe dérivée est composée des attributs et méthodes de sa classe de base ET de ses propres attributs et méthodes.

L'héritage est transitif. Par exemple, dans la figure 1 ci-dessous, la classe `Escargot` hérite de la classe `Invertébré` qui elle-même hérite de la classe `Animal`. Par transitivité, la classe `Escargot` hérite donc aussi de la classe `Animal`.

Un lien d'héritage entre la classe de base et la classe dérivée signifie "est un" : un objet de la classe dérivée "EST UN" objet de la classe de base. L'inverse n'est pas vrai : un objet de la classe de base "N'EST PAS UN" objet de la classe dérivée.

Exemple 1.1

Dans la figure 1.1 :

Les classes `Vertébré` et `Invertébré` sont des classes dérivées de leur classe de base `Animal` :

- un `Vertébré` **est un** `Animal`
- un `Invertébré` **est un** `Animal`
- ~~un `Animal` est un `vertébré`~~ (non, pas toujours)
- ~~un `Animal` est un `Invertébré`~~ (non, pas toujours)

Les classes `Poisson`, `Oiseau` et `Chien` sont des classes dérivées de leur classe de base `Vertébré`.

- un `Poisson` **est un** `Vertébré`
- un `Oiseau` **est un** `Vertébré`
- un `Chien` **est un** `Vertébré`
- ~~un `Vertebré` est un `Poisson`~~ (non, pas toujours)
- ~~un `Vertebré` est un `Oiseau`~~ (non, pas toujours)
- ~~un `Vertebré` est un `Chien`~~ (non, pas toujours)

Les classes `Mouche` et `Escargot` sont des classes dérivées de leur classe de base `Invertébré`.

- une `Mouche` **est un** `Invertébré`
- un `Escargot` **est un** `invertébré`
- ~~un `Invertébré` est une `Mouche`~~ (non, pas toujours)
- ~~un `Invertébré` est un `Escargot`~~ (non, pas toujours)

Exemple 1.2

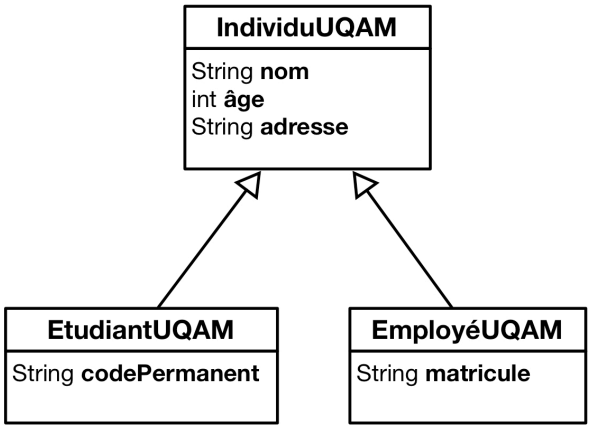


Figure 1.2 Hiérarchie de classes sur les types d'individus à l'UQAM et spécialisation (attributs).

Dans la figure 1.2 :

- Un **EtudiantUQAM** (classe dérivée) **est un** IndividuUQAM (classe de base).
  - **`ÉtudiantUQAM`** possède les caractéristiques de sa classe de base soit un **`nom`**, un **`âge`** et une **`adresse`** PLUS un **`codePermanent.`**
- Un **EmployeUQAM** (classe dérivée) est un IndividuUQAM (classe de base).
  - **`EmployéUQAM`** possède les caractéristiques de sa classe de base soit un **`nom`**, un **`âge`** et une **`adresse`** PLUS un **`matricule.`**

Mélanie Lord
Lord.melanie@uqam.ca

Exemple 1.3

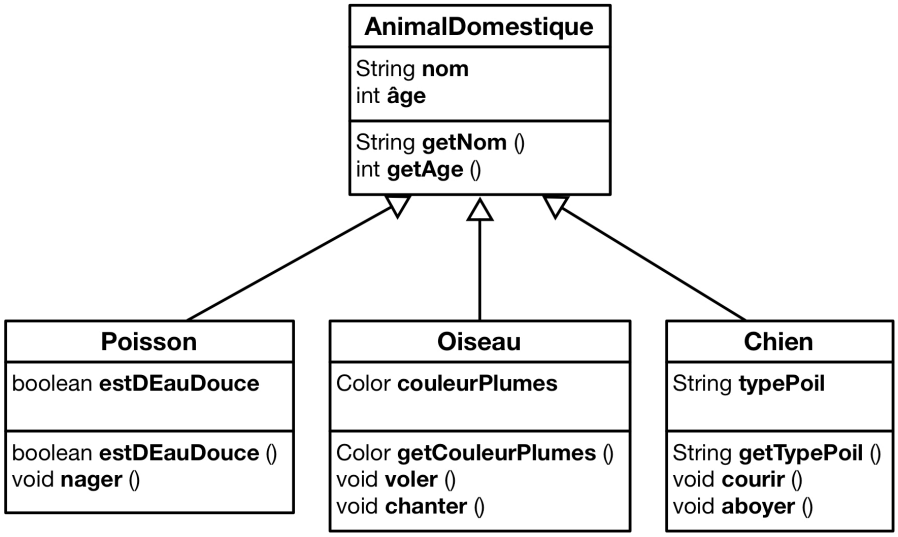


Figure 1.3 Hiérarchie de classes sur les Animaux domestiques et spécialisation (attributs et méthodes).

Dans la figure 1.3 :

- Un **`Poisson, un Oiseau ou un Chien`** (classes dérivées) **est un** AnimalDomestique (classe de base). Chacun possède donc les attributs de sa classe de base (**`nom`** et **`âge`**) ainsi que les méthodes de sa classe de base (**`getNom()`** et **`getAge()`**).

DE PLUS

- **`Poisson`** possède ses propres attributs (**`estDEauDouce`**) et ses propres méthodes (**`estDEauDouce()`**, **`nager()`**).
- **`Oiseau`** possède ses propres attributs (**`courleurPlumes`**) et ses propres méthodes (**`getCouleurPlumes()`**, **`voler()`**, **`chanter()`**).
- **`Chien`** possède ses propres attributs (**`typePoil`**) et ses propres méthodes (**`getTypePoil()`**, **`courir()`**, **`aboyer()`**).

1.2 Intérêt de l'héritage

L'héritage permet :

- de réutiliser intégralement ce qui existe déjà (réutilisation) et de pouvoir l'enrichir, le spécialiser (adaptabilité des classes). Par exemple, la classe de base `AnimalDomestique` étant définie, on peut la réutiliser pour construire la classe `Poisson`, la classe `Oiseau` et la classe `Chien`, etc.
  - Sauve du temps et $$$
- de regrouper dans une classe les caractéristiques communes (attributs et méthodes) à plusieurs classes.
  - Évite la répétition de code
  - Favorise la maintenance (modification à un seul endroit).

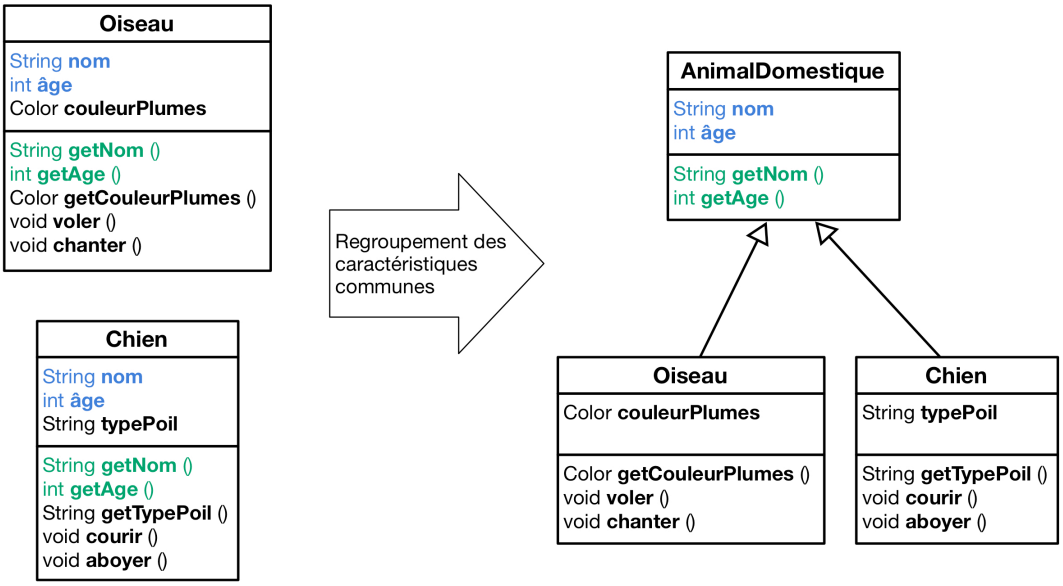


Figure 1.4 La partie de gauche montre la répétition de code (attributs = bleu et méthodes = vert) dans les classes `Oiseau` et `Chien`. La partie de droite illustre comment l'utilisation de l'héritage, avec une classe de base commune (`AnimalDomestique`), pallie ce problème. En ce qui concerne la maintenance, on voit bien que si l'on ajoute, par exemple, un attribut `nomDuPropriétaire`, on n'aura qu'à l'ajouter une seule fois à la classe de base `AnimalDomestique` (partie droite) plutôt que de répéter l'ajout dans chacune des deux classes `Oiseau` et `Chien` (partie gauche). Ainsi, des modifications effectuées dans une superclasse se transmettent automatiquement aux sous-classes.

1.3 La classe Object

Référence Java 6 : http://docs.oracle.com/javase/6/docs/api/

Dans le langage Java, la classe **`java.lang.Object`** est la **superclasse de toutes les autres classes**. Autrement dit, toute classe hérite implicitement de la classe `Object`.

- Lorsqu'une classe n'hérite pas explicitement d'une autre classe, celle-ci hérite **directement** de la classe Object.
- Si une classe `B` hérite explicitement d'une classe `A`, la classe `B` hérite **indirectement** (via la classe `A`) de la classe `Object` (l'héritage est transitif).

Cette classe contient entre autres les méthodes `toString()` et `equals()` que nous verrons éventuellement plus en détail.

2. Définition d'une classe dérivée

2.1 Propriétés générales

Une classe dérivée est une modélisation particulière, plus spécifique de la classe de base et est enrichie d'informations supplémentaires.

Une classe dérivée :

- contient les attributs de la classe de base.
- peut contenir d'autres attributs (enrichissement).
- possède (à priori) les méthodes de sa classe de base.
- peut redéfinir (ou masquer) des méthodes (voir section 5).
- peut définir de nouvelles méthodes.

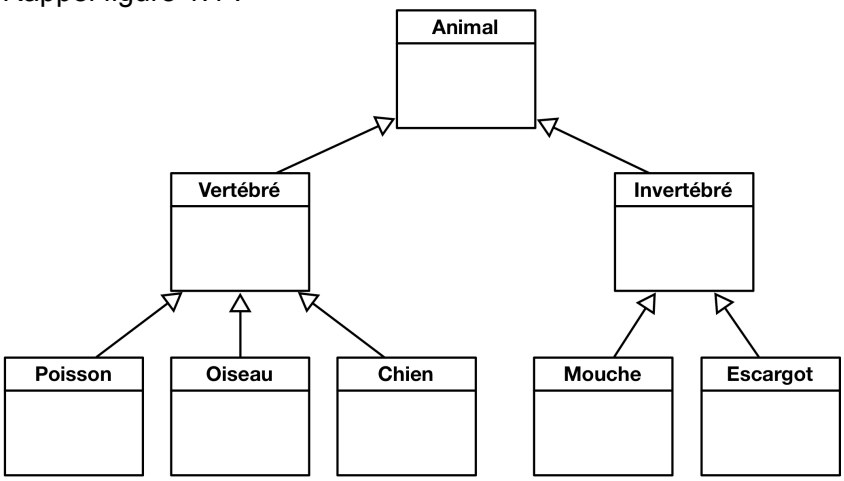
2.2 Syntaxe pour définir une relation d'héritage

L'héritage se réalise avec le mot **`extends`**.

```
[Modificateur d'accès] <class> NomClasseDérivée extends nomSuperClasse{
}
```

Exemple 2.1 : définition des relations d'héritage montrées à la figure 1.1

Rappel figure 1.1 :



Sans classe de base

```
public class Animal {
}
```

Classe de base = `Animal`

```
public class Vertebre extends Animal {
}
```

```
public class Invertebre extends Animal {
}
```

Classe de base = Vertebre

```
public class Poisson extends Vertebre {
}

public class Oiseau  extends Vertebre {
}

public class Chien extends Vertebre {
}
```

Classe de base = `Invertebre`

```
public class Mouche extends Invertebre {
}

public class Escargot  extends Invertebre {
}
```

Note *:*

- *En Java, une classe ne peut hériter que d'une seule superclasse (héritage simple).*

2.3 Les mots réservés `this` et `super`

`this` : fait référence à l'objet courant : *this object.*

- On peut l'utiliser pour :
  - désigner un attribut de "cet" objet.
  - pour appeler un constructeur de "cet" objet dans un autre constructeur de "cet" objet.

`super` : fait référence à la partie parente de l'objet courant (la superclasse).

- On peut l'utiliser pour :
  - accéder à un attribut visible de la superclasse (non recommandé pour l'encapsulation).
  - accéder à un constructeur de la superclasse.
  - accéder à une méthode de la superclasse.

2.4 Constructeurs d'une classe dérivée

Une classe dérivée est responsable de la construction de sa classe de base : pour pouvoir construire un `Oiseau`, il faut d'abord construire un `AnimalDomestique`, (voir fig. 1.3).

- Dans tous les constructeurs d'une classe dérivée, il faut **s'assurer qu'un constructeur de la classe de base est appelé**. L'appel du constructeur de la classe de base doit être la **première instruction** dans le constructeur de la classe dérivée.
- **Appel explicite** : on appelle explicitement un constructeur de la classe de base (superclasse) en utilisant le mot réservé **`super`**.
- **Appel implicite** : Lorsque le constructeur de la classe dérivée ne fait aucun appel explicite au constructeur de la classe de base, Java tente d'appeler implicitement (automatiquement) un constructeur sans argument (le pseudoconstructeur s'il existe ou un autre constructeur sans argument). Si aucun constructeur sans argument n'est trouvé, on obtient une erreur de compilation.
- Après l'appel (implicite ou explicite) du constructeur de la classe de base pour construire la partie parente, le constructeur de la classe dérivée peut contenir toutes les instructions nécessaires à sa propre construction.

Exemple 2.2

a)

```
public class A1 {

  private String s;

  public A1 (String s) {
    this.s = s;
  }
}
```

```
public class B1 extends A1 {

  private int nbr;
  //NE COMPILE PAS. Le
  //pseudoconstructeur de
  //cette classe ne peut pas
  //trouver un constructeur
  //sans arguments dans la
  //superclasse A1.
}
```

b)

```
public class A2 {
  private String s;

  //pseudoconstructeur
}
```

```
public class B2 extends A2 {
  private int nbr;
  //OK, le pseudoconstructeur
  //de cette classe peut
  //appeler le
  //pseudoconstructeur
  //de la superclasse A2
}
```

c)

```
public class A3 {
  private String s;

  public A3 () {
    s = "vide";
  }
}
```

```
public class B3 extends A3 {
  private int nbr;
  //OK, le pseudoconstructeur
  //de cette classe peut
  //appeler le constructeur
  //sans arguments
  //de la superclasse A3.
}
```

d)

```
public class A4 {
  private String s;

  public A4 () {
    s = "vide";
  }
}
```

```
public class B4 extends A4 {
  private int nbr;

  public B4 () {
    //OK. Appel explicite du
    //constructeur sans
    //arguments de la super-
    //classe. (non obligatoire,
    //car appel implicite si
    //absent)
    super();

    //initialisation de
    //l'attribut nbr
    //de cette classe
    nbr = 8;
  }
}
```

```
public class B4V2 extends A4 {
  private int nbr;

  public  B4V2 () {
    //NE COMPILE PAS.
    //Il n'existe aucun
    //constructeur dans la
    //superclasse qui prend
    //un argument de type
    //String.
    super("vert");

    //initialisation de
    //l'attribut nbr de cette
    //classe
    nbr = 8;
  }
}
```

e)

```
public class A5 {
  private String s;

  public A5 (String s) {
    this.s = s;
  }
}
```

```
public class B5 extends A5 {
  private int nbr;

  public B5 () {
    //OK. Appel explicite du
    //construteur qui prend un
    //argument de type String
    //de la superclasse.
    super("coucou");

    //init de l'attribut nbr
    //de cette classe
    nbr = 8;
  }
}
```

		`public class B5V2 extends A5 {`
		`  private int nbr;`
		
		`  public B5V2`
		`  (String s, int nbr) {`
		`  //OK. Appel du constructeur`
		`  //à un argument de type`
		`  //String de la superclasse`
		`  super(s);`
		
		`  //initialisation de`
		`  //l'attribut nbr`
		`  //de cette classe`
		`  this.nbr = nbr;`
		`  }`
		`}`

Étapes lors de la construction (`new`) d'un objet d'une sous-classe B qui hérite de la superclasse A :

1. **Allocation mémoire** pour l'objet B (donc A et B)
2. **Initialisation implicite** des attributs (A et B): initialisation avec les valeurs par défaut.
3. **Initialisation explicite des attributs hérités de A**: initialisation avec les valeurs données à la déclaration (s'il y a lieu).
4. **Exécution du constructeur de A**
5. **Initialisation explicite des attributs de B** : initialisation avec les valeurs données à la déclaration (s'il y a lieu).
6. **Exécution du constructeur de B**.

2.5 Appel de méthodes de la superclasse

Lors de la définition d'une classe dérivée, on peut faire **appel aux méthodes (visibles) de la classe de base**. Pour ce faire, on préfixe le nom de la méthode à appeler du mot clé **`super`**.

Exemple 2.3

- Montrer l'utilisation du mot réservé **`super`** pour appeler une méthode ou un constructeur de la superclasse.
- Montrer l'utilisation du mot réservé **`this`** pour appeler un constructeur dans un constructeur de la même classe ou pour désigner un attribut de la classe.

```
public class Personne {
  private String nom;
  private int age;
```

```
  public Personne (String nom, int age) {
    //this refere a un attribut de cette classe.
    this.nom = nom;
    this.age = age;
  }

  protected String getNom() {
    return nom;
  }

  public int getAge() {
    return age;
  }

  public String toString() {
    return "NOM       : " + getNom() + "\n" +
           "AGE       : " + getAge();
  }
}
```

```
public class Etudiant extends Personne {
  String codePermanent;
  int nbrCours;

  public Etudiant (String nom, int age) {
    //Appel du constructeur de la superclasse
    super (nom, age);
    codePermanent = "XXXX00000000";
    nbrCours= 0;
  }

  public Etudiant (String nom, int age,
              String codePermanent, int nbrCours) {

    //Appel du constructeur Etudiant(String, int)
    //dans cette classe (qui s'occupe deja de
    //la construction de la partie parente).
    this (nom, age);

    //this doit etre utiliser sinon conflit de noms
    //(parametres et attributs)
     this.codePermanent = codePermanent;
     this.nbrCours = nbrCours;
  }

public String toString () {
    //Puisque cette classe definit deja la methode
    //toString, on doit preciser que celle qu'on veut
    //appeler est celle de la superclasse, en prefixant le
    //nom de la methode avec le mot super.
    String s = super.toString() + "\n" +
               "CODE PERM : " + codePermanent + "\n" +
               "NBR COURS : " + nbrCours;
    return s;
  }
```

```
  public String toString2 () {
    //Ici, il n'est pas necessaire d'utilser le mot super
    //pour appeler les methodes getNom et getAge de la
    //classe de base, car il n'y a pas de conflit de noms.
    String s = "NOM       : " + getNom() + "\n" +
               "AGE       : " + getAge() + "\n" +
               "CODE PERM : " + codePermanent + "\n" +
               "NBR COURS : " + nbrCours;
    return s;
  }
}
```

3. Visibilité (droits d'accès)

La visibilité des membres d'une classe (attributs et méthodes) peut être déterminée à l'aide de quatre modificateurs d'accès en Java : `private`, `protected`, `<rien>` et `public`. Le tableau suivant décrit la visibilité permise par chaque type d'accès.

Type d'accès	Mot réservé Java	Visibilité
privée	`private`	Un membre privé n'est visible qu'à l'intérieur de la classe où il est déclaré.
protégé	`protected`	Un membre protégé est visible dans : ▪ la classe où il est déclaré ▪ les sous-classes de la classe où il est déclaré ▪ les classes du même paquetage que celle où il est déclaré.
paquetage	`<rien>`	Un membre sans modificateur d'accès est visible dans toutes les classes se trouvant dans le même paquetage que la classe où il est déclaré.
publique	`public`	Un membre public est visible dans toutes les classes (de partout).

Exemple 3.1 (tiré et adapté de [1])

** Les instructions en commentaires ne compilent pas*
*** S.o.p = System.out.println();*

```java
package p1;
public class A {
  public    static int a1;
            static int a2;
  protected static int a3;
  private   static int a4;
}
```

```java
package p1;
public class B {
  public static void main
                  (String[] args) {
    S.o.p(A.a1);
    S.o.p(A.a2);
    S.o.p(A.a3);
    //S.o.p(A.a4); //accès privé
  }
}
```

```java
package p1;
public class C extends A {
  public static void main
                   (String[] args) {
    S.o.p(A.a1);
    S.o.p(A.a2);
    S.o.p(A.a3);
    //S.o.p(A.a4); //accès privé
  }
}
```

```java
package p2;
public class D extends p1.A {
  public static void main
                    (String[] args) {
    S.o.p(p1.A.a1);
    //S.o.p(p1.A.a2); //accès paquetage
    S.o.p(p1.A.a3);
    //S.o.p(p1.A.a4); //accès privé
  }
}
```

```java
package p2;
public class E {
  public static void main
                    (String[] args) {
    S.o.p (p1.A.a1);
    //S.o.p(p1.A.a2); //accès paquetage
    //S.o.p(p1.A.a3); //accès protégé
    //S.o.p(p1.A.a4); //accès privé
  }
}
```

```java
package p2;
public class F extends D {
  public static void main
                     (String[] args) {
    S.o.p(p1.A.a1);
    //S.o.p(p1.A.a2); //accès paquetage
    S.o.p(p1.A.a3);
    //S.o.p(p1.A.a4); //accès privé
  }
}
```

4. Conversion de types entre les classes d'une hiérarchie

Notion de type statique et dynamique :

Type statique d'une variable : type à la compilation (type déclaré)
Type dynamique d'une variable : type à l'exécution (type en mémoire)

Dans l'expression `A a = new B();` le type statique de la variable a est `A` et son type dynamique est `B`.

Soit la hiérarchie de classes suivante :

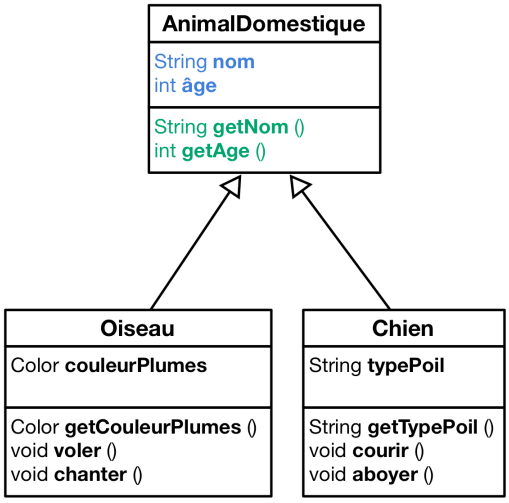


Figure 4.1

Supposons l'affectation suivante :
`a <- b` où `a` est l'objet affecté et `b`, l'objet qu'on affecte.

Conversion implicite

Il y a conversion implicite si le type (statique) de b EST_UN type (statique) de a.
Type (statique) de a < est un < Type (statique) de b

Dans la figure 4.1 :

- Un Oiseau EST_UN Oiseau (évidemment – ne demande pas de conversion)
- Un Chien EST_UN Chien (évidemment – ne demande pas de conversion)
- Un `Oiseau` EST_UN `AnimalDomestique` (mais pas le contraire).
- Un `Chien` EST_UN `AnimalDomestique` (mais pas le contraire).

Il y a donc conversion implicite lorsqu'on affecte un Oiseau à un AnimalDomestique (mais pas le contraire) et lorsqu'on affecte un Chien à un AnimalDomestique (mais pas le contraire). Lorsqu'un Chien ou un Oiseau est converti (implicitement) en AnimalDomestique, les données et méthodes de la partie "Chien" ou de la partie "Oiseau" de l'objet sont simplement ignorées à la compilation. À l'exécution cependant, le **type dynamique** sera bien Oiseau ou Chien.
On ne peut évidemment pas affecter un Oiseau à un Chien (et vice versa) puisqu'un Oiseau n'est pas un Chien (et vice versa).

Dit autrement, il faut que le type de `b` soit égale ou plus spécifique (plus spécialisé) que le type de `a` ; il faut que le type de `b` soit le type de `a` ou un sous-type (sous-classe) de `a`.

Conversion explicite

Pour qu'une conversion explicite ne produise pas d'erreur de compilation :

- le type de la conversion qu'on applique doit respecter la règle de la conversion implicite :

  Type (statique) de a < est un < (Type de conversion) de b

De plus, pour qu'une conversion explicite `(C)b` ne produise pas d'erreur à l'exécution (ClassCastException), il faut que le type dynamique de b SOIT_UN C :

Type (dynamique) de b < est un < C (type de conversion)

Exemple 4.1

Soit les trois classes de la hiérarchie de classes montrée à la figure 4.1
Voici une série d'instructions d'affectation (légales ou non) :

```
Oiseau o1 = new Oiseau();
Chien c1 = new Chien();
Chien c2 = new Chien();
AnimalDomestique a1 = new AnimalDomestique();
AnimalDomestique a2 = new Oiseau();
AnimalDomestique a3 = new Chien ();
//Oiseau o2 = new AnimalDomestique(); //NE COMPILE PAS
//o1 = c1; //NE COMPILE PAS
//c1 = o1; //NE COMPILE PAS
a1 = c1;
//c1 = a1; //NE COMPILE PAS
a2 = o1;
a3 = o1;
a3 = c1;
c1 = (Chien)a3;
c2 = (Chien)a3;
//c2 = (Oiseau)a3; //NE COMPILE PAS
//c2 = (Chien)a2;  // PLANTE À L'EXÉCUTION (CLASS CAST EXCEPTION)
a2 = c1; //Type dyn. de a2 est maintenant Chien
c2 = (Chien)a2; //OK
```

5. Surdéfinition et redéfinition

5.1 Signature d'une méthode

La signature d'une méthode (ou d'un constructeur) comprend :

- **Nom de la méthode** (ou du constructeur = nom de la classe)
- **Nombre de paramètres** et leur **type**, dans l'**ordre** où ils apparaissent (le nom des paramètres n'est pas considéré).

Note *:*

- Le type de retour (pour les méthodes) ne fait pas partie de la signature de la méthode.
- À l'intérieur d'une même classe, les méthodes (ou constructeurs) doivent posséder des signatures différentes.

Exemple 5.1 (tiré de [1])

	Entête	Signature
1	`void machin (int x)`	`machin (int)`
2	`void machin (int y)`	`machin (int)`
3	`void machin (String x)`	`machin (String)`
4	`String machin (String x)`	`machin (String)`
5	`void machin (int x, int y)`	`machin (int, int)`
6	`void chose (int x, int y)`	`chose (int, int)`

1 et 2 : même signature, même si le nom des paramètres est différent.

3 et 4 : même signature, même si le type de retour est différent.

1 (ou 2), 3 (ou 4), 5 et 6 : pourraient être déclarées dans la même classe puisqu'elles ont des signatures différentes.

5.2 Surdéfinition (ou surcharge) de méthodes

La surdéfinition consiste à définir des méthodes ayant le même nom, mais de signatures différentes.

Par exemple, dans l'exemple 5.1, les méthodes 1 (ou 2), 3 (ou 4) et 5 sont des méthodes surchargées (ou surdéfinies) qui peuvent coexister à l'intérieur d'une même classe.

Une sous-classe peut bien sûr surcharger une méthode de sa superclasse.

5.3 Redéfinition (ou masquage) de méthodes

Une sous-classe peut redéfinir (ou masquer) des méthodes de sa classe de base.

La redéfinition d'une méthode dans une sous-classe doit obéir aux principes suivants :

- La méthode redéfinie doit avoir une **signature identique** à la méthode de sa superclasse.
- La méthode redéfinie doit avoir le **même type de retour** que celle de sa classe mère.
- La méthode redéfinie doit avoir une **visibilité égale ou supérieure** à celle de sa classe de base. La redéfinition ne doit pas diminuer les droits d'accès.

Quand on appelle une méthode sur un objet, le compilateur cherche d'abord à trouver la définition de cette méthode dans la classe de l'objet en question. S'il ne la trouve pas, il cherche alors dans la superclasse. S'il ne la trouve pas, il cherche dans la superclasse de la superclasse, et ainsi de suite, jusqu'à ce qu'il trouve une définition qui correspond à la méthode cherchée. Si la méthode n'existe nulle part dans la hiérarchie, on obtient une erreur de compilation.

Exemple 5.2 (tiré de [1])

```
public class A {
   public void machin (int x) {...}                nouvelle définition
   public float machin (int x, int y) {...}        surdéfinition
   public int truc () {...}                        nouvelle définition
   public String toString () {...}                 redéfinition(de toString()
}                                                  de la classe Object)

public class B extends A {
   public void machin (String s) {...}             surdéfinition
   public int truc () {...}                        redéfinition
   public void chose (float f) {...}               nouvelle définition
}
```

6. Polymorphisme

Une entité polymorphe est une entité qui peut prendre plusieurs formes. Le polymorphisme est la propriété de ce qui est polymorphe.

On a vu précédemment que le type dynamique d'un objet (dont le type appartient à une hiérarchie de classe) pouvait changer en cours d'exécution. Par exemple (voir figure 4.1), les deux instructions suivantes montrent le changement du type dynamique de la variable a1:

```
AnimalDomestique a1 = new Oiseau(); //Type statique = AnimalDomestique
                                    //Type dynamique = Oiseau

a1 = new Chien(); //Type statique = AnimalDomestique (ne change pas)
                  //Type dynamique = Chien (a changé)
```

Une méthode exécutée sur un objet est choisie en fonction du type dynamique de l'objet sur lequel elle s'applique, au moment de l'exécution.

En Java, le polymorphisme est donc mis en oeuvre par l'intermédiaire de l'héritage et de la **redéfinition** de méthodes. Ainsi, lorsqu'on appelle une méthode redéfinie dans plusieurs classes d'une hiérarchie, celle qui sera effectivement exécutée sera choisie en fonction du type de l'objet au moment de l'exécution (type dynamique).

Exemple 6.1 (tiré de [1])

```
public class Animal {
  String nom;
  public Animal ( String nom ) {
    this.nom = nom;
  }
  public void dormir () {
    System.out.println ( "zzZZZzzZz ..." );
  }
  public String toString () {
    return "Je suis un animal et mon nom est " + nom;
  }
}
```

```
public class Chat extends Animal {
  String race;
  public Chat ( String nom, String race ) {
    super ( nom );
    this.race = race;
  }
  public void miauler () {
    System.out.println ( "miaou !!!" );
  }
  public String toString () {
    return super.toString() + ", je suis un chat de race " + race;
  }
}
```

** Les instructions suivantes qui sont en commentaires ne compilent pas ou bien plante à l'exécution.*

```
public class TestsAnimal {
  public static void main (String[] params) {

    Animal ulysse;      //Animal : type statique de ulysse
    Chat marcel;        //Chat : type statique de marcel

    //Animal : type dynamique de ulysse
    ulysse = new Animal ( "Ulysse" );

    ulysse.dormir();

    //miauler non définie dans Animal
    //ulysse.miauler();

    // Animal : type dynamique de ulysse
    //Conversion de type pour que ça compile mais
    //ClastCastException à l'execution
    // ((Chat)ulysse).miauler();

    System.out.println ( ulysse );

    //Chat : type dynamique de marcel
    marcel = new Chat ( "Marcel", "siamoise" );

    marcel.dormir();

    marcel.miauler();

    System.out.println ( marcel );

    //Chat : type dynamique de ulysse
    ulysse = new Chat ( "Ulysse", "persane" );

    ulysse.dormir();

    //miauler non définie dans Animal
    //ulysse.miauler();

    //Conversion de type pour que ça compile
    ((Chat)ulysse).miauler();
```

```
    System.out.println ( ulysse );

    //Animal est la superclasse de Chat
    //Animal N'EST PAS UN CHAT
    //marcel = new Animal ( "Marcel" );
  }
}
```

Ce qu'affiche la méthode main de la classe TestsAnimal :

```
zzZZZzzZz ...
Je suis un animal et mon nom est Ulysse
zzZZZzzZz ...
miaou !!!
Je suis un animal et mon nom est Marcel, je suis un chat de race siamoise
zzZZZzzZz ...
miaou !!!
Je suis un animal et mon nom est Ulysse, je suis un chat de race persane
```

7. Classes et méthodes finales (`final`)

On sait déjà que le modificateur `final`, lorsqu'utilisé dans la déclaration d'une variable, empêche la valeur de celle-ci d'être modifiée après sa première initialisation.

Appliqué à une méthode, le modificateur `final` empêche la redéfinition de cette méthode dans une sous-classe.

Exemple 7.1

```
public class Momo {
  public final void afficher () {
    System.out.println("Je suis Momo !");
  }
}

public class Mimi extends Momo {
  //Erreur de compilation : interdiction de
  //redéfinir la méthode afficher
  public void afficher () {
    System.out.println("Je suis Mimi !");
  }
}
```

Appliqué à une classe, le modificateur `final` empêche cette classe d'avoir des sous-classes, il arrête l'héritage. La classe `java.lang.Math` et la classe `java.lang.String` sont des exemples de classes finales.

Exemple 7.2

```
public final class Toto {…}

//Erreur de compilation : interdiction
//d'étendre la classe Toto
public class Titi extends Toto {…}
```

8. Exercices

8.1 Conception d'une hiérarchie de classes

1. Soit les classes suivantes :

DocPhoto
String Auteur String nomFichier String extensionFichier int resolutionDPI int ISO double ouverture double tempsExpo
String getAuteur () String getNomFichier () String getExtensionFichier () void ouvrirDocument (String extFichier) int getResolutionDPI () int getISO () double getOuverture () double getTempsExpo ()

DocAudio
String Auteur String nomFichier String extensionFichier double duréeEnMin int debitKBPS
String getAuteur () String getNomFichier () String getExtensionFichier () void ouvrirDocument (String extFichier) double getDuréeEnMin () void tronquer (double nbrMin) int getDebitKBPS ()

DocTexte
String Auteur String nomFichier String extensionFichier int nombrePages int nombreMots
String getAuteur () String getNomFichier () String getExtensionFichier () void ouvrirDocument (String extFichier) int getNombrePages () int getNombreMots () void ajouterAnnexe (Annexe a)

Créer une hiérarchie de documents numériques pour organiser l'information de manière à ce qu'il n'y ait aucune répétition de code (attributs et méthodes). Vous pouvez vous reporter à la figure 1.4 pour voir un exemple.

Vous pouvez (et devez) ajouter des classes pour réunir les données et méthodes communes.

2. Écrivez toutes les classes Java nécessaires pour créer la hiérarchie que vous avez conçues en A) en respectant le principe d'encapsulation des données. Ne tenez compte que des attributs des classes (oubliez les méthodes).

8.2 Constructeurs de sous-classes

Compléter les constructeurs des sous-classes VehiculeTerrestre et Camion dans le code qui suit :

```
public class Vehicule {

   //vitesse en KM/h
   private double vitesse;

   //nombre de passagers dans le vehicule
   private int nbrPassagers;

   /**
    * Constructeur d'initialisation.
    * @param vitesse la valeur d'initialisation
    * de la vitesse de ce vehicule.
    */
   public Vehicule (double vitesse) {
      this.vitesse = vitesse;
   }
```

```
   /**
    * Constructeur d'initialisation.
    * @param vitesse la valeur d'initialisation
    * de la vitesse de ce vehicule.
    * @param nbrPassagers la valeur d'initialisation
    * du nombre de passagers pour ce vehicule.
    */
   public Vehicule (double vitesse, int nbrPassagers) {
      this.vitesse = vitesse;
      this.nbrPassagers = nbrPassagers;
   }

   /**
    * Permet de modifier la valeur du nombre de
    * passagers.
    * @param nbrPassagers la nouvelle valeur pour
    * le nombre de passagers.
    */
   public void setNbrPassagers (int nbrPassagers) {
      this.nbrPassagers = nbrPassagers;
   }
}

public class VehiculeTerrestre extends Vehicule {
   private int nbrRoues = 4;

   /**
    * Constructeur d'initialisation
    * qui initialise :
    * - la vitesse de a 100 km/h
    * - le nombre de passagers a 2
    * - le nombre de roues à 2.
    */
   public VehiculeTerrestre () {
      //A COMPLETER
   }

   /**
    * Constructeur d'initialisation qui
    * initialise :
    * - la vitesse a 45.5 km/h
    * - le nombre de passagers a 3
    * - le nombre de roues (valeur en parametre).
    * @param nbrRoues la valeur d'initialisation
    * du nombre de roues.
    */
   public VehiculeTerrestre (int nbrRoues) {
      //A COMPLETER
   }

   /**
    * Permet de modifier le nombre de roues.
    * @param nbrRoues la nouvelle valeur pour
    * le nombre de roues.
    */
   public void setNbrRoues (int nbrRoues) {
      this.nbrRoues = nbrRoues;
   }
}
```

```
public class Camion extends VehiculeTerrestre {

   //si le camion possede une remorque ou non
   private boolean avecRemorque = false;

   /**
    * Constructeur d'initialisation qui
    * initialise :
    * - la vitesse a 100 km/h
    * - le nombre de passagers a 0
    * - le nombre de roues à 6
    * - si ce camion possede un remorque ou non
    *   (valeur en parametre).
    * @param avecRemorque true si ce
    * camion possede un remorque, false
    * sinon.
    */
   public Camion (boolean avecRemorque) {
      //A COMPLETER
   }
}
```

8.3 Polymorphisme et conversion de types

Soit la hiérarchie de classes suivante :

```
public class AnimalDomestique {
   private String nom = "anonyme";
   private int age;

   public void setNom (String nom) {
      this.nom = nom;
   }

   public String getNom () {
      return nom;
   }

   public void meDeplacer() {
      System.out.println("je suis " + nom + " et je me deplace !");
   }
}

public class Oiseau extends AnimalDomestique {
   private String couleurPlumes = "rouge";

   public void gazouiller () {
      System.out.println("Je suis " + getNom() + " Cui cui cui...");
   }

   public void meDeplacer() {
      System.out.println("Je suis "
              + getNom() + " et je vole !");
   }
}

public class Chien extends AnimalDomestique {
   private String typePoil = "long et soyeux";

   public String getTypePoil () {
      return typePoil;
   }
```

```
   public void meDeplacer() {
      System.out.println("Je suis " + getNom()
              + " et je cours apres la baballe...");
   }

   public void aboyer () {
      System.out.println("Je suis " + getNom() + ". Woof !");
   }
}
```

A. Dites si chacun des blocs d'instructions suivants compile ou non.

Instructions	Compile ?
`Oiseau o1 = new AnimalDomestique();`	
`AnimalDomestique a1 = new Chien();`	
`Chien c1 = new Chien();` `AnimalDomestique a2 = (AnimalDomestique)c1;`	
`AnimalDomestique a3 = new Oiseau();` `Chien c2 = (Chien)a3;`	
`Oiseau o2 = new Chien();`	
`AnimalDomestique a4 = new Chien();` `a4.meDeplacer();`	

`AnimalDomestique a5 = new Chien();`
`a5 = new Oiseau();`
`a5.aboyer();`	

`Oiseau o3 = new Oiseau();`
`Chien c3 = new Chien();`
`AnimalDomestique a6 = c3;`
`a6 = o3;`
`a6.gazouiller();`	

`new Oiseau().meDeplacer();`	

B. Toutes les instructions suivantes compilent. Dites celles qui provoqueront une erreur à l'exécution (`ClassCastException`).

```
AnimalDomestique bill = new AnimalDomestique();
AnimalDomestique bob = new Oiseau();
AnimalDomestique batman = new Chien();
Oiseau berta = new Oiseau();
berta.setNom("Jasmine");
```

Instructions	Erreur à l'exécution ?
`bob = (AnimalDomestique)bill;`	
`batman = (AnimalDomestique)bill;`	
`berta = (Oiseau)batman;`	
`((Chien)batman).meDeplacer();`	

`batman.meDeplacer();`	
`((Chien)batman).aboyer();`	
`((Chien)bob).aboyer();`	

C. Qu'affiche la méthode `main` suivante :

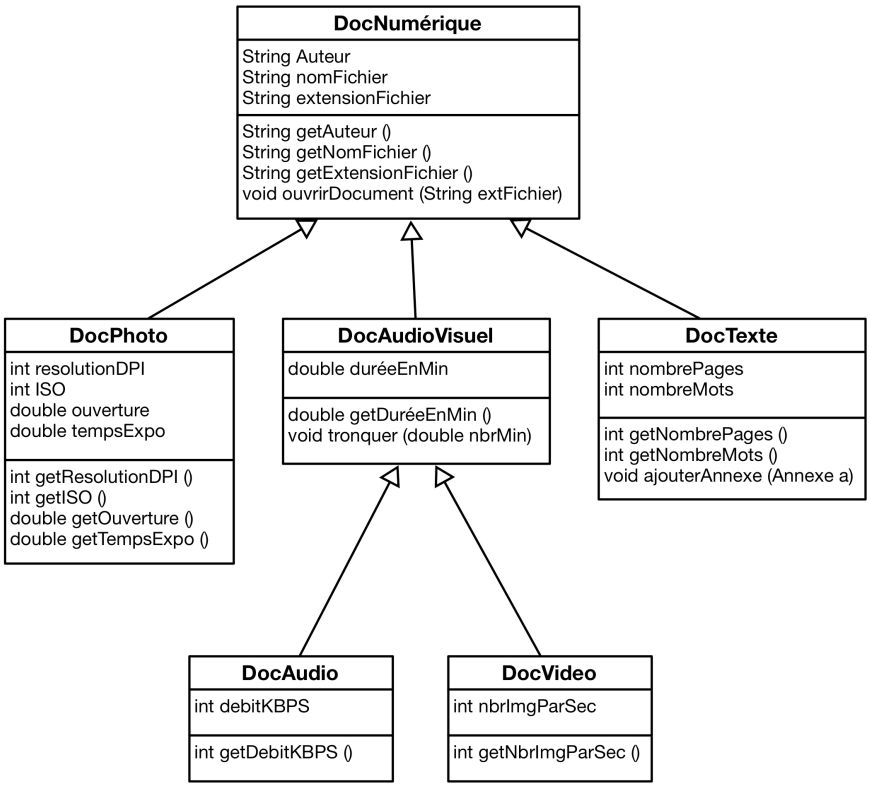
```
public static void main (String [] args) {

   AnimalDomestique bill = new AnimalDomestique();
   AnimalDomestique bob = new Oiseau();
   AnimalDomestique batman = new Chien();
   Oiseau berta = new Oiseau();
   berta.setNom("Jasmine");
   Chien bryan = new Chien();
   bryan.setNom("Marcel");
   bill.setNom("Horace");
   bryan.aboyer();
   ((Oiseau)bob).gazouiller();
   bryan.setNom("Rex");
   bob.meDeplacer();
   bryan.meDeplacer();
   batman.setNom(bob.getNom());
   ((Chien)batman).aboyer();
   batman = berta;
   ((AnimalDomestique)berta).meDeplacer();
   bill.meDeplacer();
   berta.gazouiller();
}
```

9. Solutions des exercices

Solution exercice 8.1

A.



B.

```
public class DocNumerique {
   private String Auteur;
   private String nomFichier;
   private String extensionFichier;
}
```

```
public class DocPhoto
             extends DocNumerique {
   private int resolutionDPI;
   private int ISO;
   private double ouverture;
   private double tempsExpo;
}
```

```
public class DocAudioVisuel
            extends DocNumerique {
   private double dureeEnMin;
}
```

```
public class DocTexte
            extends DocNumerique {
   private int nombrePages;
   private int nombreMots;
}
```

```
public class DocAudio
            extends DocAudioVisuel {
   private int debitKBPS;
}
```

```
public class DocVideo
            extends DocAudioVisuel {
   private int nbrImgParSec;
}
```

Solution exercice 8.2

```
public VehiculeTerrestre () {
   super(100);
   super.setNbrPassagers(2);
   nbrRoues = 2;
}

public VehiculeTerrestre (int nbrRoues) {
   super(45.5, 5);
   super.setNbrPassagers(3);
   this.nbrRoues = nbrRoues;
}

public Camion (boolean avecRemorque) {
   //appel implicite du constructeur sans arg.
   super.setNbrPassagers(0);
   super.setNbrRoues(6);
   this.avecRemorque = true;
}
```

Solution exercice 8.3

A. Dites si chacun des blocs d'instructions suivants compilent ou non.

Instructions	Compile ?
`Oiseau o1 = new AnimalDomestique();`	NON
`AnimalDomestique a1 = new Chien();`	OUI
`Chien c1 = new Chien();` `AnimalDomestique a2 = (AnimalDomestique)c1;`	OUI
`AnimalDomestique a3 = new Oiseau();` `Chien c2 = (Chien)a3;`	OUI (mais plante à l'exécution)
`Oiseau o2 = new Chien();`	NON
`AnimalDomestique a4 = new Chien();` `a4.meDeplacer();`	OUI

`AnimalDomestique a5 = new Chien();`
`a5 = new Oiseau();`
`a5.aboyer();`
	NON

`Oiseau o3 = new Oiseau();`
`Chien c3 = new Chien();`
`AnimalDomestique a6 = c3;`
`a6 = o3;`
`a6.gazouiller();`
	NON

`new Oiseau().meDeplacer();`	OUI

B. Toutes les instructions suivantes compilent. Dites celles qui provoqueront une erreur à l'exécution (`ClassCastException`).

```
AnimalDomestique bill = new AnimalDomestique();
AnimalDomestique bob = new Oiseau();
AnimalDomestique batman = new Chien();
Oiseau berta = new Oiseau();
berta.setNom("Jasmine");
```

Instructions	Erreur à l'exécution ?
`bob = (AnimalDomestique)bill;`	NON
`batman = (AnimalDomestique)bill;`	NON
`berta = (Oiseau)batman;`	OUI
`((Chien)batman).meDeplacer();`	NON
`batman.meDeplacer();`	NON
`((Chien)batman).aboyer();`	NON
`((Chien)bob).aboyer();`	OUI

C. Voici ce qu'affiche la méthode `main` :

```
Je suis Marcel. Woof !
Je suis anonyme Cui cui cui...
Je suis anonyme et je vole !
Je suis Rex et je cours apres la baballe...
Je suis anonyme. Woof !
Je suis Jasmine et je vole !
je suis Horace et je me deplace !
Je suis Jasmine Cui cui cui...
```

9. Référence

[1] Laforest, Louise. *INF2120 – Programmation II (version alpha)*, Notes cours.