Université des Sciences et de la Technologie Houari Boumediene

Faculté d'Électronique et d'Informatique

Département d'Informatique

Programmation Orientée Objets

Dr. Ilhem BOUSSAÏD

iboussaid@usthb.dz

Ce cours s'inspire très largement du livre "Le langage java" de Henri Garreta et du cours écrit par Jacques Bapst, de l'École d'Ingénieurs et d'Architectes de Fribourg.

Licence 2 académique

Année universitaire 2014/2015

Table des matières

1 INTRODUCTION 1
1.1 Les paradigmes de programmation . . . 1
1.1.1 La programmation procédurale . . . 1
1.1.2 La programmation Orientée Objets (POO) . . . 4

2 FONDEMENTS DE LA PROGRAMMATION ORIENTÉE OBJETS 6
2.1 La notion d'objet . . . 6
2.2 La notion de classe . . . 7
2.2.1 Les objets, instanciation des classes . . . 9
2.2.2 Différence entre les objets et les types fondamentaux . . . 9
2.2.3 Attributs et méthodes d'instance . . . 11
2.2.4 Attributs et méthodes de classe . . . 12
2.3 Surcharge . . . 14
2.4 Contrôle de l'accessibilité . . . 15
2.4.1 L'encapsulation . . . 16
2.5 Constructeurs . . . 17
2.5.1 Constructeur par défaut . . . 18
2.5.2 Surcharge des constructeurs . . . 18
2.5.3 Désigner un constructeur par this(...) . . . 19
2.6 Membres constants (final) . . . 19
2.7 Les paquets (*packages*) . . . 21
2.7.1 Déclaration de package . . . 21
2.7.2 Importation de package . . . 22
2.7.3 Import statique . . . 24

3 HÉRITAGE ET POLYMORPHISME 26
3.1 Héritage . . . 26
3.1.1 Masquage des variables . . . 28
3.1.2 Redéfinition des méthodes . . . 29
3.1.3 Héritage et constructeurs . . . 31
3.1.4 Membres protégés . . . 33
3.1.5 Généralisation : conversion classe fille → classe mère . . . 33
3.1.6 Particularisation : conversion classe mère → classe fille . . . 33
3.1.7 Modificateur final . . . 34
3.2 Le polymorphisme . . . 34
3.3 Relations entre classes . . . 36
3.4 La classe racine Object en Java . . . 37
3.4.1 Redéfinition de méthodes clés de Object . . . 37

©I. Boussaïd, 2014–2015

4 Les classes abstraites et interfaces **45**
4.1 Classes abstraites . . . 45
4.2 Interface . . . 47
4.2.1 Implémentation d'interfaces . . . 48
4.2.2 Utilisation des interfaces . . . 49
4.2.3 Exemple . . . 49
4.2.4 Interface de marquage . . . 52
4.2.5 Problème . . . 53
4.2.6 Interface de typage . . . 57
4.2.7 Interface et évolution . . . 58
4.2.8 Méthodes par défaut et conflit . . . 59

5 Les exceptions **60**
5.1 Exceptions . . . 60
5.2 Avantages des exceptions . . . 60
5.3 Gestion des exceptions . . . 61
5.3.1 Générer (lever) une exception (throw) . . . 62
5.3.2 Capturer et traiter une exception (try/catch/finally) . . . 64
5.3.3 Propagation des exceptions (throws) . . . 65
5.3.4 Créer de nouveaux types d'exceptions . . . 66
5.4 Les exceptions par l'exemple . . . 67
5.4.1 Capturer et traiter l'exception . . . 67
5.4.2 Propager l'exception . . . 68
5.4.3 Lever une exception . . . 69

6 Les collections **70**
6.1 Qu'est ce qu'une Collection . . . 70
6.2 Les interfaces de Collection . . . 71
6.3 Interface Collection . . . 72
6.3.1 Méthodes principales de Collection<E> . . . 72
6.4 Les listes . . . 72
6.5 Les itérateurs . . . 74
6.6 Les ensembles . . . 75
6.7 Map<K,V> . . . 76
6.7.1 Méthodes principales de Map<K,V> . . . 77
6.8 Les méthodes de la classe Collections . . . 78
6.9 Généricité . . . 78
6.9.1 Pourquoi la généricité ? . . . 78

Chapitre Un

Introduction

1.1 Les paradigmes de programmation

Pour des raisons fondamentales, tous les langages de programmation offrent la même expressivité : Tout problème qui peut être résolu par un programme écrit dans le langage A peut aussi l'être par un programme rédigé en langage B.

Il existe cependant :

– différentes approches à la résolution algorithmique de problèmes,

– différents mécanismes de programmation,

– différents styles de programmation.

Chacun de ces éléments peut influencer les qualités d'un programme : Correction, efficacité, modularité, concision, lisibilité, ergonomie, ...

Nous présentons dans ce chapitre deux principales approches de programmation :

1. **Programmation Procédurale** : Centrée sur les procédures (ou fonctions)
2. **Programmation Orientée-Objet** : Centrée sur les données

Il faut parler de la démarche de développement par **découpe fonctionnelle descendante et programmation procédurale** pour comprendre l'intérêt de la démarche de **développement orientée objets**.

1.1.1 La programmation procédurale

La forme la plus immédiate pour décrire un travail à effectuer est de lister les **actions à réaliser**. On découpe une tâche complexe à effectuer en une **hiérarchie d'actions** à réaliser de plus en plus simples, petites et précises (pour décrire, on utilise le **verbe**).

C'est l'approche que vous connaissez. Le langage C est adapté à la programmation procédurale.

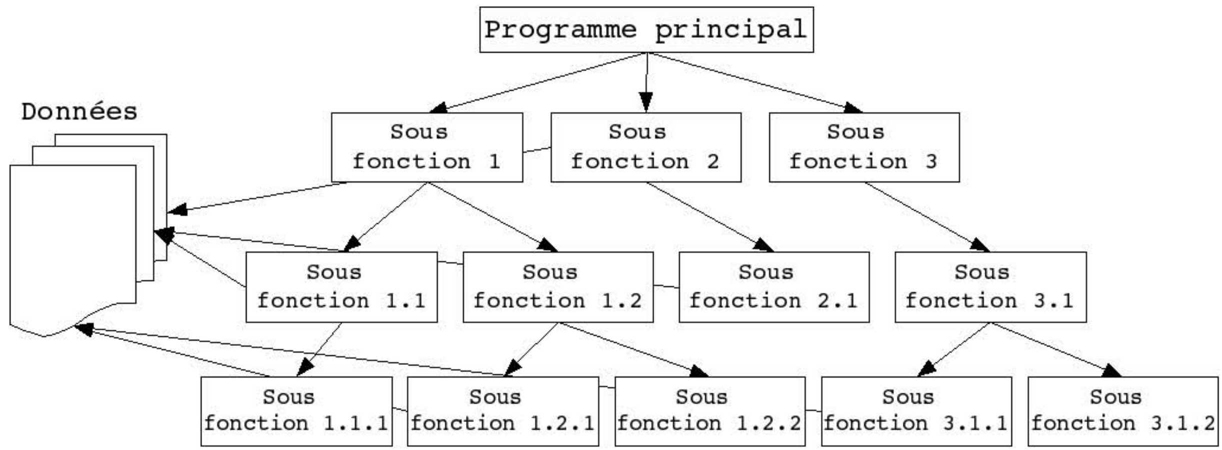


FIGURE 1.1: Modélisation par décomposition fonctionnelle

Dispersion données/fonctions

Dans une programmation procédurale issue d'une découpe fonctionnelle descendante, les données et les fonctions travaillant sur ces données sont **dispersées** dans des modules différents. On utilise ainsi des fonctions auxquelles on fournit des arguments et qui retournent des résultats. Ces fonctions peuvent éventuellement être rangées dans des bibliothèques, pour que l'on puisse les réutiliser. **On retrouve ici les notions de modules, et de compilation séparée**.

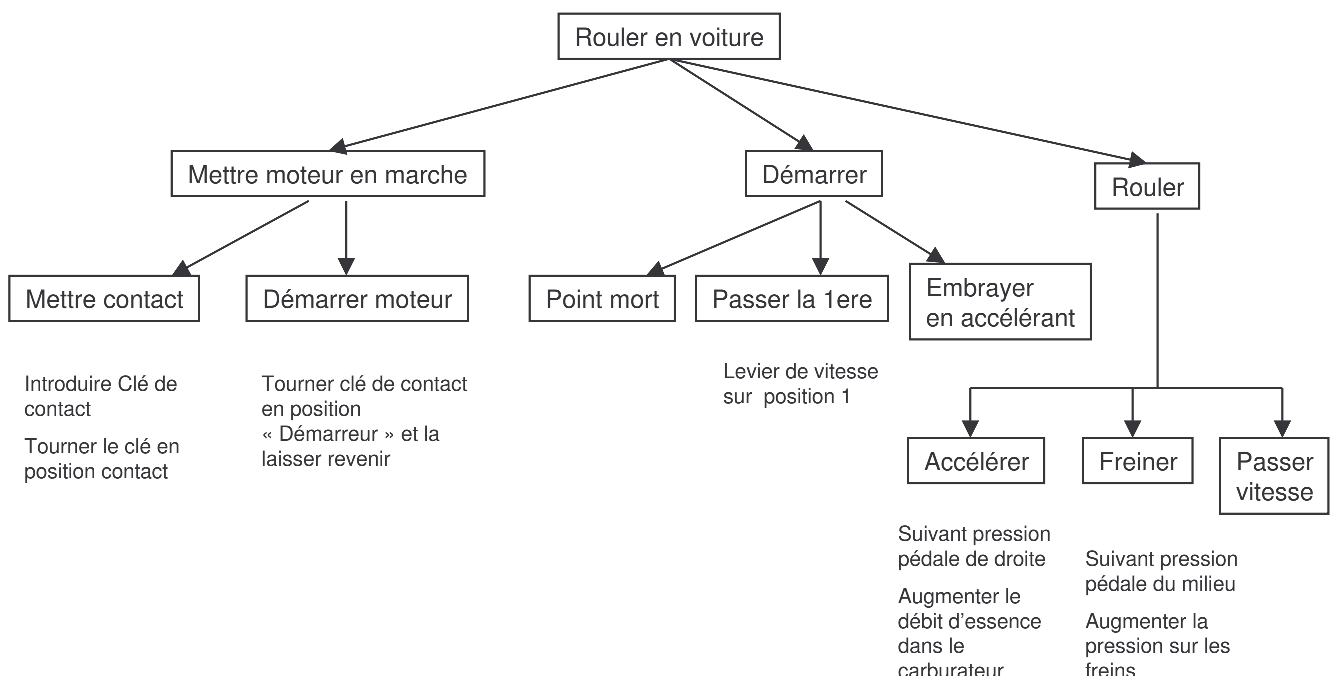
Lorsque le logiciel évolue, il faut **faire évoluer les structures de données et les fonctions en parallèle** (probablement dans des modules différents).

Maintenir cette cohérence est laborieux parce que données et fonctions sont dispersées.

Exemple 1.1. *Rouler en voiture :*

1. ***Mettre*** *moteur en marche :*
   (a) ***Mettre*** *Contact*
   (b) ***Démarrer*** *Moteur*
2. ***Démarrer*** *Voiture :*
   (a) ***Mettre*** *Point Mort*
   (b) ***Passer*** *La Première*
   (c) ***Embrayer*** *En Accélérant*
3. ***Rouler*** *:*
   (a) ***Accélérer***
   (b) ***Freiner***
   (c) ***Passer*** *Vitesse*

Exemple 1.2. *Vous devrez réaliser un programme permettant la gestion de prêts de livres dans une bibliothèque. Le modèle à mettre en œuvre contiendra deux structures différentes, dont les champs contiennent à minima les informations données ci-dessous.*



ADHERENT

```
typedef struct SAdherent {
int adh_Index;
char adh_Nom[CMAX];
int adh_NbEmprunts;
} Adherent;
```

LIVRE

```
typedef struct SLivre {
 int liv_Index;
 char liv_Titre[CMAX];
 char liv_Auteur[CMAX];
 int liv_Emprunteur;
} Livre;
```

- `adh_Index` (resp. `liv_Index`) identifie de manière unique un adhérent (resp. un livre).
- `liv_Emprunteur` contient la valeur de `adh_Index` de l'emprunteur du livre.
- Le nombre de livres empruntés par un adhérent est stocké dans `adh_NbEmprunts` (un adhérent peut donc emprunter plusieurs livres).

CONTRAINTES DE PROGRAMMATION

La fonction `main()` devra contenir un tableau de structures `Adherent` et un tableau de structure `Livre` (ce ne seront pas des variables globales).

Approche procédurale : "Que doit faire mon programme ?"

Votre programme devra proposer un menu comme celui-ci :

1 Gestion des adhérents
   - Ajouter, modifier ou supprimer un adhérent
   - afficher la liste des adhérents par ordre alphabétique
2 Gestion des livres
   - Ajouter, modifier ou supprimer un livre
   - afficher la liste des livres par ordre alphabétique (titre)
3 Gestion des emprunts
   - Emprunter un livre
   - Afficher la liste des livres empruntés
   - Rendre un livre
   - Afficher la liste des emprunteurs de livres
4 Quitter le programme

Approche orientée-objet : "De quoi doit être composé mon programme ?"

Des objets similaires peuvent être regroupés dans une même classe. Ainsi on peut identifier 3 classes :

- Classe `Bibliothèque` qui comprend un ensemble de documents et un ensemble d'adhérents ;
- Classe `Livre` identifié par le titre du livre et le nom de l'auteur ;
- Classe `Adhérent` identifié par le nom et prénom de l'adhérent. Les adhérents peuvent emprunter des livres.

1.1.2 La programmation Orientée Objets (POO)

? **On se pose d'abord la question : "de quoi parle-t-on ?" et non pas la question "que veut-on faire ?"**

Principe

- Accent mis sur **ce qu'est le système** (ses composants) → identification des composants du système : les objets
- **Centrée sur les données**. On ne raisonne pas uniquement en verbes, mais davantage en noms.
   - Une personne est caractérisée par son prénom, son nom de famille, son sexe, sa date de naissance, son lieu de naissance, sa profession : ce sont ses propriétés.
   - Une personne peut exécuter les opérations suivantes : se réveiller, se lever, marcher, courir, sauter, s'asseoir, dormir, manger, lire, écrire, parler (dire son âge), se taire, ...

- **Approche ascendante** : on peut partir des objets du domaine (briques de base) et remonter vers le système global
- **Données** et **opérations traitant les données** ne sont pas séparées, mais **réunies au sein d'un même module** (l'objet).

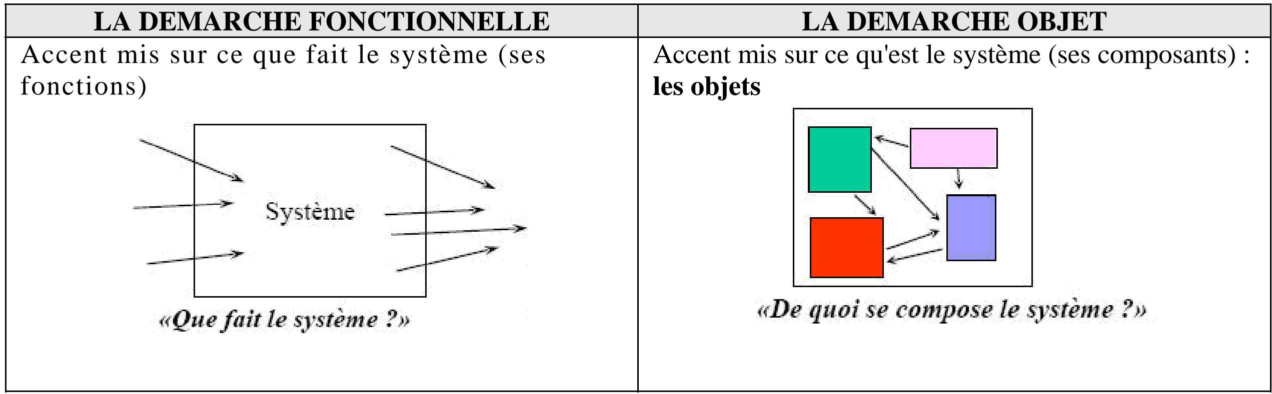


Figure 1.2: Modélisation fonctionnelle versus orientée objets

Difficulté

Déterminer les objets fondamentaux du système.

POO vs. programmation procédurale

1. **Programmation procédurale** : détermination d'abord des procédures pour manipuler des données, puis de la structure à imposer aux données pour faciliter leur manipulation.
2. **POO réalise l'inverse** : détermination d'abord des données de façon cohérente dans des entités structurantes (objets), puis des opérations sur ces données.

   ⇒ POO est de ce point de vue plus **modulaire**, **réutilisable**, **flexible** et plus **facile à maintenir**

Chapitre Deux

Fondements de la Programmation orientée objets

2.1 La notion d'objet

Un objet est une abstraction du monde réel c.-à-d. des données informatiques regroupant des caractéristiques du monde réel. Exemple une personne, une voiture, une maison, ...

Un objet est une entité qui contient à la fois des données et des traitements (les données sont appelées attributs et les traitements opérations ou méthodes).

Un objet possède :

- Une **identité unique** : permet de distinguer un objet d'un autre (par exemple un produit pourra être repéré par un code, une voiture par un numéro de série, ...)
- Un **état interne** donné par des valeurs de variables (ou attributs) à un instant donné. Les propriétés sont définies dans la classe d'appartenance de l'objet. L'état d'un objet à un moment donné est la conséquence des comportements passés.
  - Ex : patient mesure 1,82m et pèse 75 Kg
  - Les attributs sont typés et nommés (ex : float hauteur ; float poids ;)
- Un **comportement** (capacités d'action de l'objet) donné par l'ensemble des opérations qu'il peut exécuter en réaction aux **messages** provenant des autres objets. Les opérations sont définies dans la classe d'appartenance de l'objet.

Objet = identité + état (attributs) + comportement (méthodes)

L'objet informatique :

Un objet modélise toute entité identifiable, concrète ou abstraite, manipulée par l'application logicielle

- **une chose tangible** (ex : ville, véhicule, étudiant, un bouton sur l'écran, porte, ascenseur, bouton, clavier, souris, avion, ...)
- **une chose conceptuelle** (ex : date, réunion, planning de réservation, compte en banque, équation mathématique, facture, commande, marché, . . .)

L'état et le comportement sont liés : le comportement dépend de l'état et l'état est modifié par le comportement.

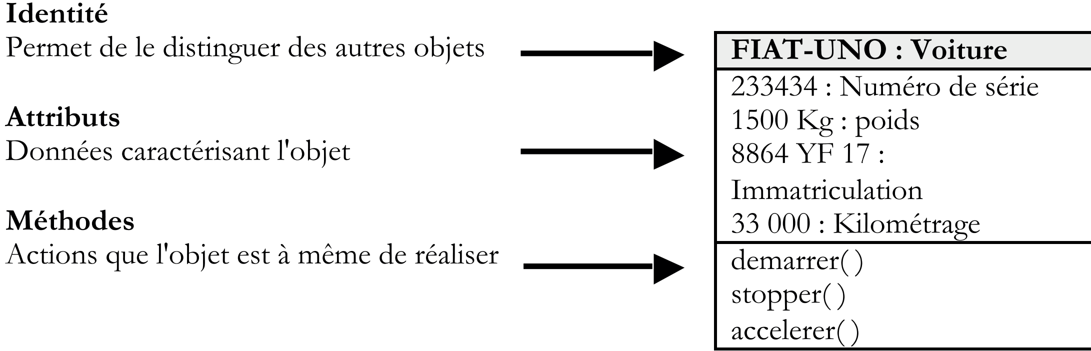


Figure 2.1: Représentation d'un objet Voiture

2.2 La notion de classe

Une classe est une description d'une **famille d'objet** ayant **même structure** et **même comportement**.

On dit qu'un objet est une **instance de classe**, c'est à dire une valeur particulière de la classe (notion d'**instanciation**).

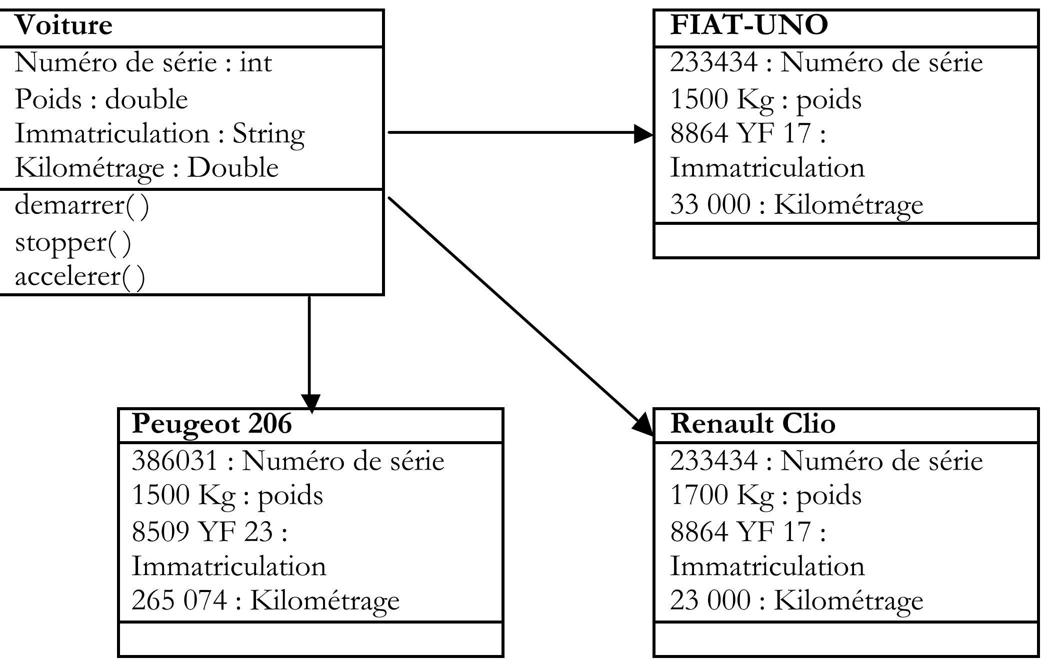


Figure 2.2: Représentation de la classe Voiture et de ses instances

Fondamentalement, une classe est constituée par un ensemble de membres, qui représentent :

- **Les données** : on les appelle alors **attributs**, **variables**, **champs**, ou encore **données membres**, dont les valeurs représentent l'état de l'objet
- **Les traitements** : on les appelle alors **méthodes** ou **fonctions** membres applicables aux objets.

En java, la définition d'une classe s'effectue par l'utilisation du mot clé `class`.

La syntaxe de la déclaration d'une classe

```
visibilité class identificateur {
        ....
}
```

où visibilité est soit rien, soit le mot `public` (la classe est alors dite publique, voir section 2.4).

Les membres des classes se déclarent comme les variables et les fonctions en C, sauf qu'on peut les préefixer par certains qualifieurs, `private`, `public`, etc., expliqués plus loin.

Par exemple, une application gérant des points du plan pourrait être décrite par une classe et déclarée de la manière suivante :

Exemple : La classe Point

```
public class Point {
        int x ; // abscisse
        int y ; // ordonnee

        void affiche() {
             System.out.println("(" + x + "," + y + ")");
         }
         double distance(Point p) {
             int dx = x - p.x;
             int dy = y - p.y;
             return Math.sqrt(dx * dx + dy * dy);
        }
        public static void main( String args[] ){
                ...
        }
}
```

Retenons déjà que la définition d'une classe publique doit impérativement être enregistrée dans un fichier du même nom que la classe suivi de l'extension `.java` (ici, `Point.java` ; attention au respect de la casse : `point.java` n'est pas valable !).

Remarque 2.1. *La plupart des compilateurs imposent que chaque fichier source java contienne **au plus** une classe publique.*

Comme en C/C++, on retrouve la méthode `main` qui sera appelée en premier lieu lors de l'exécution du code ; ici, cette méthode ne renvoie rien, ce qui est spécifié par le mot clé `void`.

Contrairement au C, en Java, l'ordre des champs (ou attributs) en mémoire n'est pas l'ordre des champs lors de la déclaration.

En Java, le code est **toujours** dans une méthode.

En java le **passage des paramètres** des méthodes se fait **par valeur**.

2.2.1 Les objets, instanciation des classes

Une classe est un **type de données** : à la suite d'une déclaration comme la précédente, on pourra déclarer des variables (objets) de type `Point`, c'est-à-dire des variables dont les valeurs sont des **instances de la classe** `Point` et ont la structure définie par cette dernière.

```
Point p; // déclaration d'une variable Point
```

Contrairement à la déclaration d'une variable d'un type primitif (comme `int n ;`), elle ne réserve pas d'emplacement pour un objet de type `Point`, mais seulement un emplacement pour une référence à un objet de type `Point`.

On obtient que `p` soit une référence valide sur un objet soit en lui affectant une instance existante, soit en lui affectant une instance nouvellement créée, ce qui s'écrit :

```
p = new Point();
// création d'une nouvelle instance de la classe Point
//et place sa référence dans p
```

Attention

Nous avions pu écrire plus haut `new Point()` sans avoir défini de constructeur. En effet, toute classe possède un **constructeur par défaut, sans argument**. Mais une fois qu'un constructeur est ajouté à la classe `Point`, le constructeur implicite sans argument disparaît (voir section 2.5).

2.2.2 Différence entre les objets et les types fondamentaux

En Java, il existe deux sortes de types

1. Les types primitifs qui sont manipulés par leur valeur. Il existe huit types fondamentaux : `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char`. À partir de ces types fondamentaux, on peut créer par combinaison une infinité de nouveaux types objets.
2. Les types objets qui sont manipulés par leur référence : `Object`, `String`, `int[]`, `StringBuilder`, etc.

La principale différence entre les objets et les types fondamentaux est la façon dont ils sont manipulés par la machine virtuelle Java :

– Les variables de type fondamental sont manipulées par **valeur**
– Les variables de type objet sont manipulées par adresse que l'on appel **référence** (pas un pointeur car pas d'arithmétique).

Pour simplifier on peut dire que la mémoire de l'ordinateur est divisée en deux parties :

1. **La pile** : tout ce qui se trouve dans la pile est **accessible directement**.
2. **Le tas** : tout ce qui se trouve dans le tas **n'est accessible que indirectement en utilisant des références** (adresse mémoire) contenues dans la pile.

Lors de la création d'une variable, deux cas se présentent :

1. si la variable est de **type fondamental** alors sa valeur est stockée directement dans la **pile**
2. si la variable est de **type objet** alors son contenu est stocké dans le tas et seule une référence vers le contenu est stockée dans la pile.

Exemple 2.1.

```
int n=7 ;
Point z=new Point(1,2) ;
int[] tab=new int[3] ;
```

La mémoire sera organisée comme indiqué sur la figure 2.3 (en Java les tableaux sont des objets).

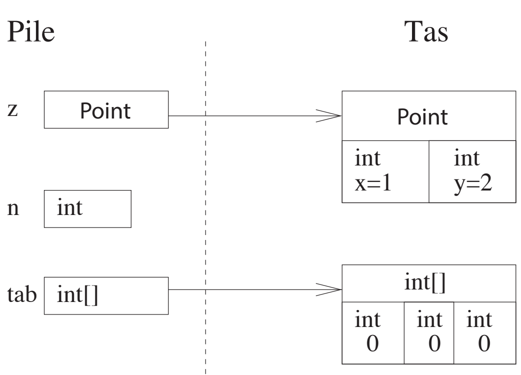


Figure 2.3: Stockage des types fondamentaux et des objets

2.2.3 Attributs et méthodes d'instance

La déclaration donnée en exemple introduit une classe `Point`, comportant pour le moment deux variables d'instance, `x` et `y`, et deux méthodes d'instance, `affiche` et `distance`.

En disant que ces éléments sont des variables et des méthodes d'instance de la classe `Point` on veut dire :

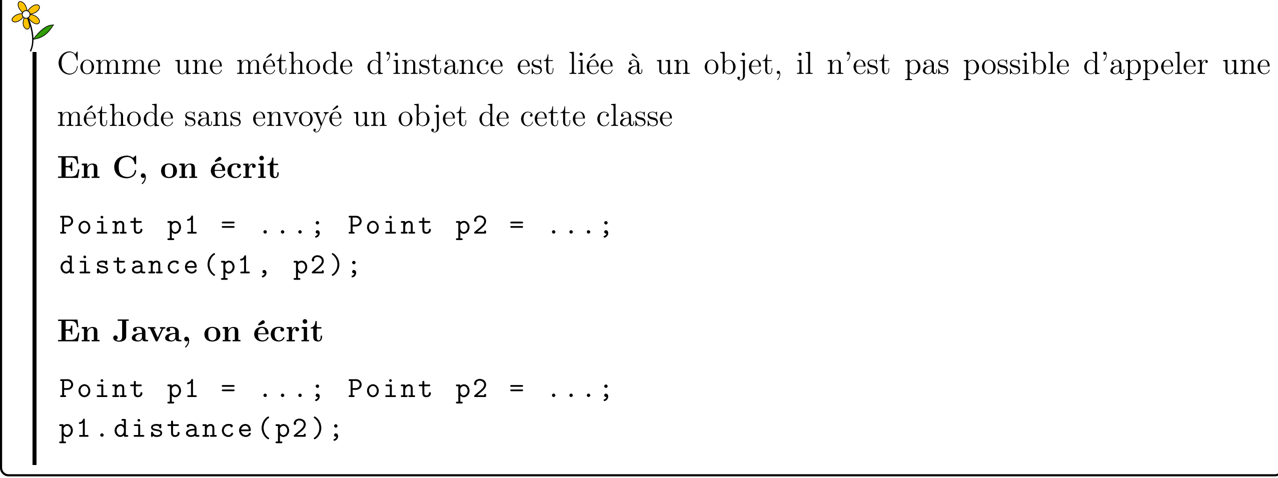
- **Pour les variables**, qu'il en existe un jeu différent dans chaque instance, disjoint de ceux des autres instances : chaque objet `Point` a sa propre valeur de `x` et sa propre valeur de `y`,
- **Pour les méthodes**, qu'elles sont liées à un objet particulier : un appel de la méthode `affiche` n'a de sens que s'il est adressé à un objet `Point` (celui qu'il s'agit d'afficher).

Ainsi, **une méthode d'instance est nécessairement appelée à travers un objet** :

`p.x` : un accès à la variable d'instance `x` de l'objet `p`

`p.affiche()` : un appel de la méthode `affiche` sur l'objet `p`

`a.distance(b);` (`a` et `b` sont des variables de type `Point`).



Comme une méthode d'instance est liée à un objet, il n'est pas possible d'appeler une méthode sans envoyé un objet de cette classe

En C, on écrit

```
Point p1 = ...; Point p2 = ...;
distance(p1, p2);
```

En Java, on écrit

```
Point p1 = ...; Point p2 = ...;
p1.distance(p2);
```

Exemple 2.2. *Par exemple, dans la méthode distance de la classe* `Point` *:*

```
...
double distance(Point p) {
        int dx = x - p.x;
        int dy = y - p.y;
        return Math.sqrt(dx * dx + dy * dy);
}
...
```

- les variables `x` et `y` qui apparaissent seules désignent les coordonnées de l'objet à travers lequel on aura appelé la méthode distance.

– les expressions `p.x` et `p.y` désignent les coordonnées de l'objet mis comme argument de cet appel.

? Passage par référence/par valeur ?

– on appel passage par valeur un appel de fonction qui recopie les arguments lors de l'appel de fonction
– on appel passage par référence un appel de fonction qui copie l'adresse sur la pile des arguments lors de l'appel de fonction

En Java, il n'y a pas de passage par référence, le **passage se fait uniquement par valeur**. Comme un objet est manipulé par son adresse (sa réference) donc sa référence est recopié comme valeur

2.2.3.1 L'identificateur this

À l'intérieur d'une méthode d'instance, l'identificateur `this` désigne l'objet à travers lequel la méthode a été appelée. Par exemple, la méthode distance précédente peut aussi s'écrire comme ceci :

L'identificateur this

```
  ...
 double distance(Point p) {
     int dx = this.x - p.x;
     int dy = this.y - p.y;
     return Math.sqrt(dx * dx + dy * dy);
} ...
```

2.2.4 Attributs et méthodes de classe

Le mot clé `static` peut être utilisé pour déclarer des **variables de classe** (qui seront alors partagées par toutes les instances de la classe) ou des **méthodes de classe**. De telles méthodes spécifient que l'action réalisée n'est pas liée à une instance particulière de cette classe (la méthode de classe n'a pas besoin d'un objet pour être appelée).

©I. Boussaïd, 2014–2015

Variable et méthode de classe

```
class Point {
    int x, y; // variables d'instance
    static int nombre; // variable de classe
    void afficher() { // méthode d'instance
        System.out.println("(" + x + "," + y + ")");
    }
    static int distanceSymetrique(Point p, Point q) {
        // méthode de classe
        return Math.abs(p.x - q.x) + Math.abs(p.y - q.y);
    }
}
```

Contrairement aux membres d'instance, les membres de classe ne sont pas attachés aux instances. Ce qui signifie :

– **Pour une variable**, qu'il n'y en a qu'une partagée par toutes les instances d'une classe. Alors que de chaque variable d'instance il y a un exemplaire pour chaque objet effectivement créé. L'expression `Point.nombre` désigne un accès à la variable de classe `nombre`

– **Pour une méthode**, qu'on peut l'appeler sans devoir passer par un objet. L'expression `Point.distanceSymetrique(p, q)` est un appel correct de la méthode de classe `distanceSymetrique`.

Attention

Ce qui n'est pas statique est appelé **dynamique**. Dans une méthode statique, l'emploi de `this` n'est pas autorisé. En effet, il n'existe pas nécessairement d'objet particulier ayant été utilisé pour appeler cette méthode. Pour la même raison, une méthode statique ne peut pas appeler une méthode dynamique. À l'inverse, en revanche, une méthode dynamique peut parfaitement appeler une méthode statique. Enfin, on note que le point d'entrée d'un programme Java, à savoir sa méthode `main`, est une méthode statique :

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

Remarque 2.2. *Pour bien mettre en évidence les différences entre les membres de classe et les membres d'instance, examinons une erreur qu'on peut faire quand on débute :*

```
public class Essai {
        void test() { ...      }
        public static void main(String[] args) {
            test();    // *** ERREUR ***
        }
}
```

Sur l'appel de `test` on obtient une erreur

Cannot make a static reference to the non-static method test

Deux solutions :

1. Créer une instance permettant d'appeler `test` en passant par un objet :

```
public class Essai {
        void test() {
        ...
        }
      public static void main(String[] args) {
          Essai unEssai = new Essai();
          unEssai.test();    // OK
        }
}
```

2. Rendre `test` statique :

```
public class Essai {
        static void test() {
                ...
        }
      public static void main(String[] args) {
          test();    // OK
        }
}
```

2.3 Surcharge

Plusieurs méthodes d'une même classe peuvent porter le même nom, pourvu qu'elles aient des arguments en nombre et/ou en nature différents ; c'est ce que l'on appelle la **surcharge** (en anglais *overloading*).

Ainsi on peut écrire dans la classe `Point` deux méthodes `deplace`, la première à deux arguments de type `int`, la deuxième à un seul argument de type `int` :

Surcharge

```
public class Point{
        private int x, y;
        public void deplace (int dx, int dy) {
        // deplace (int, int)
                x+=dx;
                y+=dy;
        }
        public void deplace (int dx){ // deplace (int)
                x+=dx;
        }
}
```

Remarque

Il ne faut pas confondre la **surcharge** (*overloading*) avec la **redéfinition** (*overriding*) des méthodes :

1. **La surcharge des méthodes** consiste dans le fait que deux méthodes, souvent membres de la **même classe**, ont le **même nom** mais des **signatures différentes**.
2. **La redéfinition des méthodes** consiste dans le fait que deux méthodes ayant le **même nom** et la **même signature** sont définies dans **deux classes dont l'une est sous-classe**, directe ou indirecte, de l'autre (voir section 3.1.2).

La **signature d'une méthode** se compose de trois éléments :

- le nom de la méthode,
- la suite des types de ses arguments,
- le fait que la méthode soit ordinaire (d'instance) ou `static`.

Par exemple, si la définition d'une méthode commence par la ligne

```
public double moyenne(double[] tableau, int nombre, String titre) {
```

alors sa signature est le triplet :

```
< moyenne, (double[], int, String), non statique >
```

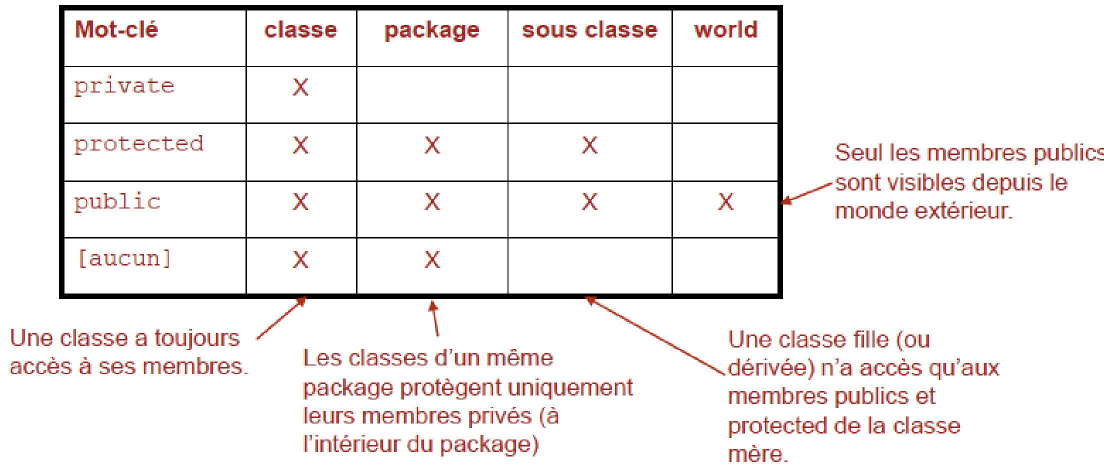
2.4 Contrôle de l'accessibilité

La déclaration d'un membre d'une classe (variable ou méthode, d'instance ou de classe) peut être qualifiée par un des mots réservés `private`, `protected` ou `public`, ou bien ne comporter aucun de ces qualifieurs.

- **Membres publics** : un membre de la classe C dont la déclaration commence par le qualifieur `public` est accessible partout où C est accessible ; c'est le contrôle d'accès le

plus permissif.

– **Membres privés** : un membre d'une classe `C` dont la déclaration commence par le qualifieur `private` n'est accessible que depuis [les méthodes de] la classe `C` ; c'est le contrôle d'accès le plus restrictif,
– **Membres protégés** : un membre d'une classe `C` dont la déclaration commence par le qualifieur `protected` n'est accessible que depuis [les méthodes de] la classe C, les autres classes du paquet auquel `C` appartient et les sous-classes, directes ou indirectes, de `C` (le concept de membre protégé est lié à la notion d'héritage. Pour cette raison, ces membres sont expliqués plus en détail à la section 3.1).
– Un membre d'une classe `C` dont la déclaration ne commence pas par un des mots `private`, `protected` ou `public` est dit avoir l'**accessibilité par défaut** (ou **package**) ; il n'est accessible que depuis [les méthodes de] la classe `C` et les autres classes du paquet auquel `C` appartient.



Mot-clé	classe	package	sous classe	world
private	X			
protected	X	X	X	
public	X	X	X	X
[aucun]	X	X		

2.4.1 L'encapsulation

On dit que les informations (données) et les comportements (traitements) d'un objet sont encapsulés, c-à.-d. qu'ils sont à l'intérieur d'une entité (objet).

Les données d'un objet doivent être accessibles (et surtout modifiables) qu'au travers de méthodes prévues à cet effet.

la seule façon de modifier l'état d'un objet est d'utiliser les méthodes de celui-ci

Remarque 2.3. *Il est recommandé de rendre systématiquement inaccessible les propriétés des instances d'une classe par les instances des autres classes en spécifiant l'accès* `private`*. On dit que les propriétés sont encapsulées, comme si elles étaient protégées par une capsule.*

©I. Boussaïd, 2014–2015

Exemple 2.3.

Considérons la classe `Point`, en déclarant les champs `x` et `y` privés, ils ne sont plus visibles à l'extérieur de la classe `Point`.

```
public class Point {
        private int x, y;
        ....
}
```

Si on cherche à accéder au champ `y` depuis une autre classe, en écrivant par exemple `p.y = - p.y;` pour transformer un certain objet `p` de la classe `Point` en son symétrique par rapport à l'axe des abscisses, on obtient un message d'erreur du compilateur :

'File.java :19 : y has private access in Point

La valeur du champ `y` peut néanmoins être fournie par l'intermédiaire d'une méthode, c'est-à-dire d'une fonction fournie par la classe `Point` et applicable à tout objet de cette classe.

```
class Point {
         private int x, y;
         public int getX() { return x; }
         public int getY() { return y; }
         public void setPos(int a, int b) {
                x = a; y = b; }
        ... }
```

avec cette définition, pour transformer un point `p` en son symétrique par rapport à l'axe des abscisses il faut écrire : `p.setPos(p.getX(), - p.getY());`

L'interface d'un objet

L'interface d'un objet correspond à l'ensemble des points d'entrée d'un objet qui sont visibles depuis l'extérieur. donc en Java, l'interface d'un objet est l'**ensemble des méthodes publiques** de celui-ci

2.5 Constructeurs

Un constructeur d'une classe est une méthode qui a **le même nom que la classe** et **pas de type de retour** (même pas `void`). Si on spécifie un type de retour, le compilateur croit que c'est une méthode.

Exemple 2.4.

```
class Point {
         private int x, y;
```

```
        public Point(int a, int b) {// Constructeur
                this.x = a;
                this.y = b;
        }
        ...
}
```

Un constructeur est toujours appelé de la même manière : lors de la création d'un objet, en utilisant l'opérateur `new` :

```
Point p = new Point(200, 150);
```

Dans un constructeur on ne doit pas trouver d'instruction de la forme `return expression ;`

2.5.1 Constructeur par défaut

Lorsque le code d'une classe ne comporte pas de constructeur, un constructeur sera automatiquement ajouté par Java. Pour la classe `Point`, ce constructeur par défaut sera :

```
[public] Point() { }
```

Même accessibilité que la classe (`public` ou non)

2.5.2 Surcharge des constructeurs

Une classe peut en effet avoir plusieurs constructeurs, qui devront alors différer par leurs signatures (constructeurs avec des arguments en nombre ou en nature différents). On parle de **surcharge**.

Exemple 2.5.

```
class Point {
        private int x, y;
        public Point(int x, int y) {
                this.x = x;
                this.y = y;
        }
        public Point(int x) {
                this.x = x;
                this.y = 0;
        }
        ...
}
```

Lorsqu'une classe a un ou plusieurs constructeurs, l'un de ces constructeurs doit être **obligatoirement** utilisé pour créer un objet de cette classe.

Dans l'exemple ci-dessus, si on tente d'écrire maintenant `new Point()`, on obtient un message d'erreur du compilateur :

The constructor Point() is undefined

Rien ne nous empêche cependant de réintroduire un constructeur sans argument.

2.5.3 Désigner un constructeur par this(...)

À l'intérieur d'un constructeur, l'identificateur `this`, utilisé comme un nom de méthode, désigne un autre constructeur de la même classe (celui qui correspond aux arguments de l'appel).

```
class Point {
        private int x, y;
        public Point(int x, int y) {
            this.x = x;
            this.y = y;
            ...
        }
        public Point(int x) {
                this(x, 0);
                ...
       }
       ...
}
```

Lorsqu'un tel appel de `this(...)` apparaît, il doit être **la première instruction du constructeur**.

2.6 Membres constants (final)

La déclaration d'un membre d'une classe peut commencer par le qualifieur `final`.

- **Pour une variable** cela signifie qu'une fois initialisée (dans la déclaration ou dans le constructeur), elle ne pourra plus changer de valeur ; en particulier, toute apparition de cette variable à gauche d'une affectation sera refusée par le compilateur. Une telle variable est donc plutôt une **constante**.
- **Pour une méthode** : cela veut dire qu'une méthode ne sera pas redéfinie dans les éventuelles sous-classes (voir section 3.1.7)

Exemple 2.6. *Voici comment déclarer une classe dont les instances sont des points immuables :*

```
public class Point {
        final int x, y;
        Point(int a, int b) {
                x = a;
                y = b;
        }
        ...
}
```

Les affectations de `x` et `y` qui apparaissent **dans le constructeur sont acceptées** par le compilateur, car il les reconnaît comme des initialisations ; toute autre affectation de ces variables sera refusée.

Le programmeur a deux manières d'assurer que des **variables d'instance ne seront jamais modifiées une fois les instances crées** :

1. Qualifier ces variables `final`
2. Qualifier ces variables `private` et ne pas définir de méthode qui permettrait de les modifier (des *setters*).

La première manière est préférable, car

– la qualification `private` n'empêche pas qu'une variable soit modifiée depuis une méthode de la même classe,

Constantes de classe

Les variables de classe peuvent elles aussi être qualifiées `final`. Par convention, le nom de ces variables est tout en majuscules.

Exemple 2.7.

```
public class Point {
    public static final int XMAX = 600;
    public static final int YMAX = 400;
    private int x, y;
    public Point(int x, int y) throws Exception {
        if (x < 0 || x >= XMAX || y < 0 || y > YMAX)
          throw new Exception("Coordonnées␣invalides");
          this.x = x;
          this.y = y;
        }
        ...
}
```

2.7 Les paquets (*packages*)

En Java, l'objectif de réutilisabilité du code conduit à une organisation des classes en paquets (*packages*).

- Chaque package regroupe de manière cohérente un certain nombre de classes apparentées
- Les packages permettent de **définir un espace de désignation** (*Namespace*) pour les classes qu'ils contiennent.
- Ils servent également à **gérer les droits d'accès** (visibilité) des classes les unes par rapport aux autres.
- Les packages sont organisés de manière hiérarchique (structure arborescente avec racines multiples).
- Dans le nom des packages, le point ('.') est utilisé comme séparateur entre les différents niveaux hiérarchiques.
- Exemple : `java.lang.String` : La classe `String` se trouve dans le sous-package `lang` du package `java`

2.7.1 Déclaration de package

Le mot clé `package` est utilisé pour indiquer le paquet auquel appartiennent la ou les classes :

```
package nom_du_paquet;
```

Exemple 2.8.

```
package poo.exemple;
public class Point {
  ...
}
```

- La classe `Point` fait partie du package `poo.exemple`
- Le nom complet de la classe est " `poo.exemple.Point`"

Remarque 2.4. *Les noms des packages sont habituellement écrits en utilisant uniquement des minuscules (convention de codage).*

2.7.2 Importation de package

En vue de **réutiliser une classe prédéfinie**, on peut importer des packages en début de code. Il existe deux formes :

1. Importation d'une classe individuelle : `import poo.exemple.Point;`
2. Importation de toutes les classes contenues dans un package : `import poo.exemple.*;`

L'instruction `import` permet d'éviter de nommer une classe avec son package.
L'importation d'un paquetage entier (`....*`) **ne rend pas visible le contenu des éventuels sous-packages**. Si nécessaire, les sous-packages doivent être importés explicitement.

- En java, les classes sont regroupées en deux ensembles : les packages de noyau (commençant par `java.*`) et les packages d'extensions (commençant par `javax.*`). Par exemple, pour importer les sous-packages du package `swing`, nous ajouterons en début de code la ligne suivante :

  ```
  import javax.swing.*;
  ```

- **Par défaut**, un package de base (`java.lang`) est **importé implicitement** lors de chaque compilation même si le programmeur ne le spécifie pas. On peut donc utiliser ses classes par leurs noms simples (par exemple : `String`, `Math`, `System`, ...) sans avoir à les importer.

Tableau 2.1: Librairie de classes Java

Package	Description
java.lang	Le noyau des classes du langage, comprenant notamment String, Math, System, Thread et Exception. Ce package est implicitement importé.
java.io	Classes et interfaces d'entrée/sorties. Bien que certaines des classes de ce paquetage soient conçues pour travailler directement avec les fichiers, la plupart d'entre-elles permet de travailler avec des flux d'octets ou des flux de caractères.
java.math	Petit paquetage qui contient des classes destinées à l'arithmétique entière de précision arbitraire et à l'arithmétique décimale.
java.net	Classes et interfaces destinées à l'interconnexion avec d'autres systèmes (réseau).
java.security	Classes et interfaces de contrôle d'accès et d'authentification. Plusieurs sous-paquetages.
java.util	Diverses classes utilitaires y compris un cadre de collections puissant, à utiliser avec les collections d'objets
java.applet	Définit la classe Applet qui est la super-classe de toutes les applets.
javax.accessibility	Ensemble de classes et d'interfaces permettant la réalisation d'applications adaptées à des personnes handicapées (par exemple malvoyants)
java.awt	Définit le cadre de l'interface utilisateur graphique de base AWT (*Abstract Windowing Toolkit*). Contient également les classes permettant de gérer des graphiques 2D.
java.awt.event	Définit les classes et les interfaces utilisées pour la gestion des événements en AWT et Swing.
java.awt.image	Ensemble de classes et d'interfaces servant à manipuler les images.
java.awt.print	Ensemble de classes et d'interfaces destinées à l'impression de documents.
javax.swing	Grand paquetage (comportant de nombreux sous-paquetages) destiné à étendre AWT pour la création d'interfaces utilisateur graphiques (GUI) sophistiquées. Introduit de nouveaux composants graphiques avec des propriétés beaucoup plus étendues qu'AWT.
javax.swing.event	Complète et améliore java.awt.event avec des classes spécialisées pour les composants Swing.

Import * et ambiguité

Si deux paquetages possèdent une classe de même nom, il y a ambiguïté !

```
import java.util.*;
import java.awt.*;
public class ImportClash {
   public static void main(String[] args) {
         List list=... // ***** oups
   }
}

import java.util.*;
import java.awt.*;
import java.util.List;
public class ImportClash {
     public static void main(String[] args) {
           List list=... // ok
     }
}
```

Attention

`import *` pose un problème de maintenance si des classes peuvent être ajoutées dans les paquetages utilisés

Règle de programmation : éviter d'utiliser des `import *`.

2.7.3 Import statique

Permet d'accéder aux membres statiques d'une classe dans une autre sans utiliser la notation '.'

notation : `import static chemin.classe.*;`

```
import java.util.Scanner;
import static java.lang.Math.*;
public class StaticImport {
   public static void main(String[] args) {
        Scanner in=new Scanner(System.in);
        System.out.println("donner un nombre :");
        double value=sin(in.nextDouble());
        System.out.printf("son sinus est %f\n",value);
  }
}
```

Lors de la résolution des membres, les membres (même hérités) sont prioritaires sur le scope

```
import static java.util.Arrays.*;
```

```
public class WeirdStaticImport {
        public static void main(String[] args) {
            java.util.Arrays.toString(args); // ok
            toString(args); // toString() in java.lang.Object
                // cannot be applied to (java.lang.String[])
        }
  }
```

Règle de programmation : utiliser l'import statique avec parcimonie

F

Chapitre Trois

Héritage et Polymorphisme

3.1 Héritage

La notion d'héritage (*inheritence*) permet la création de nouvelles classes à partir d'anciennes (qui seront alors appelées *superclasses*), en vue d'y ajouter de nouvelles fonctionnalités ou de redéfinir des fonctionnalités existantes. Un héritage peut être spécifié par le mot clé `extends` de la manière suivante :

```
public class SousClasse extends SuperClasse
```

- Lorsque l'on déclare que la classe fille hérite de la classe mère, il est possible d'enrichir la classe fille avec des attributs et des méthodes supplémentaires.
- On parle alors d'**enrichissement** ou d'**extension** modulaire.
- Il est également possible de redéfinir des méthodes héritées en donnant une nouvelle implémentation de ces méthodes.
- On parle alors de **redéfinition** ou de **substitution**. Enrichissement et redéfinition ne sont pas exclusifs.

Attention

En Java, chaque classe possède au plus une super-classe directe. On parle dans ce cas d'**héritage simple** (par opposition à l'**héritage multiple**, comme pour le C++, qui permet à une classe d'hériter de plusieurs classes parentes). L'utilisation des **interfaces** (section 4.2) permet de ne pas être limité par cette contrainte.

Exemple 3.1. *Définissons la classe* `Pixel` *comme sous-classe de la classe* `Point` *qui nous a déjà servi d'exemple. Un* `Pixel` *est un* `Point`*, étendu par l'ajout d'une variable d'instance qui représente une couleur (*`Color` *est une classe définie dans le paquet* `java.awt`*) :*

```
class Point {
        int x, y;
        ...
}
class Pixel extends Point {
        java.awt.Color couleur;
        ...
}
```

Avec cette définition, un `Pixel` se compose de trois variables d'instance : `x` et `y`, héritées de la classe `Point`, et `couleur`, une variable d'instance spécifique.

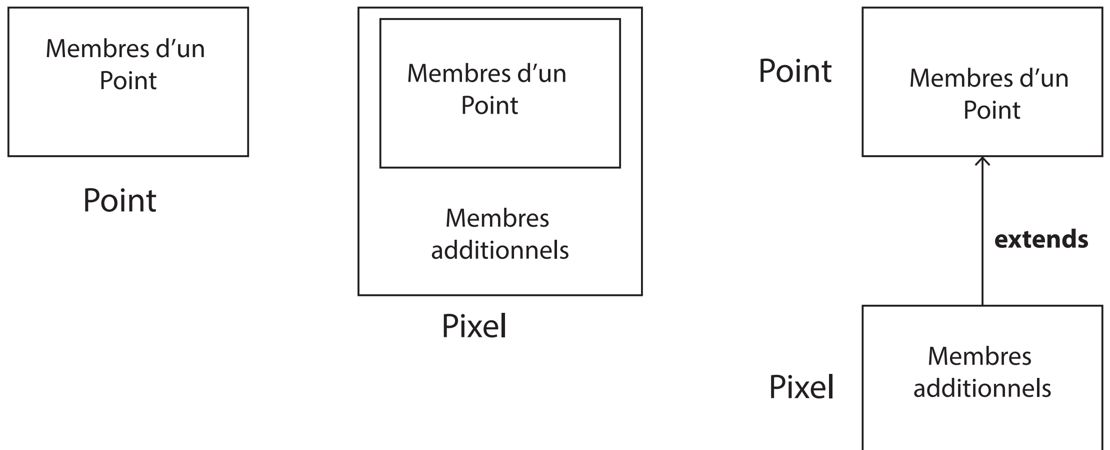


Figure 3.1: Héritage

- `Point` est la classe mère et `Pixel` la classe fille.
- la classe `Pixel` hérite de la classe `Point`
- la classe `Pixel` est une sous-classe de la classe `Point`
- la classe `Point` est la super-classe de la classe `Pixel`

La classe Object

En Java, il y a une classe particulière, nommée `Object`, qui n'a pas de super-classe, et toutes les autres classes héritent de `Object`. Par conséquent, toute classe hérite, directement ou indirectement, de la classe `Object` (cf. section 3.4 pour plus de détails).

Parmi les méthodes de la classe `Object`, dont toute classe hérite donc, on peut citer notamment les deux méthodes :

```
public boolean equals(Object o);
```

Renvoie vrai si deux objets sont égaux structurellement (si leurs champs sont égaux Voir section 2.7.2).

```
public String toString();
```

Chaîne permettant un affichage d'un objet

```
public int hashCode();
```

Renvoie un "résumé" d'un objet sous forme d'un entier

En particulier, certaines classes peuvent (doivent) redéfinir ces méthodes de façon appropriée.

La classe `Object` constitue la généralisation ultime :

Tous les objets sont des `Object` !

Chacune de ces méthodes possèdent une implantation par défaut :

```
Point p = new Point(15, 23);
p.toString();   // Point@3dfeca64
p.equals("hello"); // false, car équivalent à ==
p.equals(p); // true, car équivalent à ==
p.hashCode(); //1040108132 , valeur aléatoire calculé une fois (sur 24bits)
```

3.1.1 Masquage des variables

Si une sous-classe définit une variable avec le même nom qu'une variable de sa classe parente (***** une pratique à éviter *****), alors la variable de la super-classe est **masquée** dans le corps de la sous-classe.

Dans ce cas, on peut, dans le corps de la sous-classe accéder à la variable de la classe parente en utilisant le mot-clé `super` suivi d'un point et du nom de la variable.

Exemple 3.2. *Imaginons qu'on a introduit une mesure de qualité dans les points, et aussi une notion de qualité dans les pixels, ces deux notions n'ayant pas de rapport entre elles*

```
public class Point {
        int x, y;
```

```
        int qualite;
        ...
}
public class Pixel extends Point {
        Color couleur;
        int qualite;
        ...
}
```

À l'intérieur des méthodes de la classe `Pixel`, on peut utiliser :

```
qualite          //Champ qualite de la classe Pixel
this.qualite     //Champ qualite de la classe Pixel
super.qualite    //Champ qualite de la classe Point
((Point)this).qualite    //Champ qualite de la classe Point
                         //(par transtypage)
```

Attention
La notation `super.super.f()` n'est pas autorisée et il est impossible d'accéder à une méthode masquée d'une classe ancêtre (autre que la classe parente). Seul le **casting** permet d'accéder à un champ masqué d'une classe ancêtre.

Remarque 3.1. *Les **champs statiques** peuvent également être masqués mais ils restent **accessibles en les préfixant avec le nom de la classe**.*

3.1.2 Redéfinition des méthodes

Une sous-classe peut redéfinir (*override*) les méthodes dont elle hérite et fournir ainsi des implémentations spécialisées pour celles-ci.

- La **redéfinition d'une méthode** c'est le fait de donner une **nouvelle implémentation en conservant la signature** (la signature d'une méthode est composée de son nom et des types de ses arguments).

  Le type du résultat de la redéfinition doit être soit identique, soit une sous-classe du type du résultat de la méthode héritée (depuis la version 5 de java).

- Lorsqu'une méthode redéfinie par une classe est invoquée pour un objet de cette classe, c'est la nouvelle définition et non pas celle de la super-classe qui est invoquée.
- En redéfinissant une méthode, il est possible d'**augmenter sa zone de visibilité** (`private`→package→`protected`→`public`) mais **non de la restreindre** (ex. `public` ↛ `private`).

Attention

Ne pas confondre **redéfinition** (*overriding*) avec **surcharge** (*overloading*).

Voir section 2.3.

Exemple 3.3. *Redéfinition de la méthode* `toString` *de la classe* `Object` *:*

```
public class Point {
        int x, y;
        ...
        public String toString() {
            return "(" + x + "," + y + ")";
        }
 }
public class Pixel extends Point {
        Color couleur;
        ...
        public String toString() {
            return "[" + super.toString() + ";" + couleur + "]";
        }
}
```

- Lorsqu'une méthode est redéfinie dans une sousclasse, il est malgré tout possible de faire appel à la méthode originale de la superclasse à l'aide du mot clé `super`. Par exemple, `super.toString()` permettra de faire appel à la méthode `toString` de la classe `Point`.
- À l'extérieur de la classe de déclaration, il n'est, par contre, pas possible pour un objet de type `Pixel`, d'invoquer la méthode `toString` de la classe `Point`.
- La méthode invoquée par un appel `obj.toString()` dépendra du type de l'objet référencé par la variable `obj` lors de l'exécution de cette instruction (**liaison dynamique**, voir section 3.2).
- C'est toujours la méthode associée au type effectif de l'objet référencé qui est exécutée, même si l'objet est enregistré dans une variable déclarée avec le type d'une classe parente :

```
Pixel r = new Pixel(2, 5, red);
Point p = r; // Conversion élargissante (Upcasting)
String s = p.toString(); // Invoque r.toString()
// et non pas toString() de la
// classe Point car la variable p
// référence un objet de type Pixel
```

L'annotation `@Override` peut être utilisée pour indiquer explicitement l'intention de redéfinir une méthode.

Bien qu'étant facultative, elle apporte deux avantages :

– Si on se trompe en écrivant le nom de la méthode lors de la redéfinition, le compilateur peut alors informer le programmeur de son erreur.

– La redéfinition étant explicite, le code est plus clair.

```
@Override
public void affiche() { ... }
```

L'annotation @Override n'est pas obligatoire mais fortement recommandée.

Les **méthodes statiques** (méthodes de classe) peuvent être masquées, dans des sous-classes, par des méthodes statiques possédant le même nom (*** **une pratique à éviter** ***). Ces méthodes masquées restent cependant accessibles en les préfixant avec le nom de la classe dans laquelle elles sont définies (comme il est recommandé de le faire avec toutes les méthodes statiques).

3.1.3 Héritage et constructeurs

Un constructeur d'une sous-classe peut faire appel à un constructeur de la classe parente en utilisant le mot réservé `super` selon la syntaxe suivante :

```
super(arguments);
```

Si un constructeur d'une sous-classe invoque explicitement un constructeur de la classe parente, l'instruction `super(...)` doit être **la première instruction du constructeur**.

Exemple 3.4. *Si la classe Point n'a qu'un constructeur, à deux arguments :*

```
public class Point {
        int x, y;
        Point(int x, int y) {
                this.x = x;
                this.y = y;
        }
        ...
}
```

Le constructeur pour la classe `Pixel` doit **obligatoirement** être comme ceci :

```
public class Pixel extends Point {
```

```
    Color couleur;
    Pixel(int x, int y, Color couleur) {
        super(x, y);
        this.couleur = couleur;
    }
        ...
}
```

Attention

Il n'est pas possible d'utiliser à la fois un autre constructeur de la classe et un constructeur de sa classe mère dans la définition d'un de ses constructeurs.

```
public class A extends B{
  public A( int x ) {
    super ( ) ;
    this ( ) ; //*****ERREUR******
  }
}
```

Exemple 3.5. *Appel implicite du constructeur de la classe mère*

```
class A {
public int x;
public A() {x=5; }
}

class B extends A {
 public B() {x++;}
 public B(int i){this(); x=x+i; }
public B(String s){super(); x- -; }
}
```

qu'affichera le code suivant ?

```
B b1=new B(); B b2 =new B(2003); B b3= new B("Bonjou");
System.out.println(b1.x + " et " + b2.x + " et encore " + b3.x );
```

- 6 et 2009 et encore 4
- 1 et 2004 et encore 4
- 1 et 2004 et encore 2003
- autre chose

Le constructeur `B()` n'appelle explicitement ni `this()`, ni `super()`. Donc, par convention, le constructeur de la super-classe `A` est appelé (implicitement) avant de procéder. Ceci donne `b1.x=6`. Le constructeur `B(2003)` appelle le constructeur précédent avec le `this()`, ce qui donne `b2.x=6`. Ensuite on y ajoute `2003`, ce qui donne finalement `b2.x=2009`. Le constructeur

`B("Bonjour")` appelle le constructeur de la super-classe `A` avec le `super()`. Ceci donne `b3.x=5`. Ensuite on le décrémente et on a finalement `b3.x=4`.

3.1.4 Membres protégés

Les membres privés de la superclasse ne sont pas accessibles par les sousclasses. Même s'il ne sont pas accessibles dans les sous-classes, les champs privés sont malgré tout hérités dans les sous-classes (une zone mémoire leur est allouée).

Le qualifieur `protected` permet d'indiquer qu'un membre d'une classe `C` est accessible depuis les méthodes de la classe `C`, celles des autres classes du paquet auquel `C` appartient, ainsi que celles des **sous-classes, directes ou indirectes, de** `C` (indépendamment du paquet).

Il s'agit d'un accès plus restrictif que `public` mais - contrairement à ce que son nom pourrait laisser croire - **moins restrictif que l'accès par défaut** (package).

3.1.5 Généralisation : conversion classe fille → classe mère

La relation d'héritage ("**Est un...**") permet de traiter les objets des classes filles comme s'ils étaient des objets de leur classe mère (par **généralisation**).

Si nécessaire, le système effectue donc une **conversion élargissante** (automatique) de la classe fille vers la classe mère (**Upcasting**).

3.1.6 Particularisation : conversion classe mère → classe fille

Une **conversion explicite** (**transtypage**, **casting**) d'un objet de la classe mère vers un objet de la classe fille (**Downcasting**) est possible si l'instance à convertir référence effectivement (au moment de l'exécution) un objet de la classe fille considérée (sinon, il y aura une erreur **ClassCastException** à l'exécution).

Exemple 3.6.

```
Point unPoint;
Pixel unPixel = new Pixel(a, b, c);
...
unPoint = unPixel; // OK. Généralisation : un pixel est un point
...
unPixel = unPoint; //  ERREUR un Point n'est pas forcément un pixel
...
unPixel = (Pixel) unPoint; // OK. Particularisation :
                       // unPoint référence effectivement un Pixel
```

Downcasting : utilisation de instanceof

L'opérateur `instanceof` permet de tester (à l'exécution) l'appartenance d'un objet à une classe (ou une interface) donnée.

Dans l'exemple précédent, on aurait pu écrire :

```
if (unPoint instanceof Pixel) {
  unPixel = (Pixel) unPoint;
}
else {
... // unPoint ne référence pas un Pixel }
```

Attention

Le downcasting ne permet pas de convertir une instance d'une classe donnée en une instance d'une sous-classe !

3.1.7 Modificateur final

- **Dans la déclaration d'une classe**, le modificateur `final` indique que la classe ne peut pas être sous-classée (on ne peut pas créer de classes dérivées). Les classes `java.lang.String` et `java.lang.Integer`, etc. sont finales.
- **Dans la déclaration d'une variable**, le modificateur `final` indique que **la valeur de la variable ne peut pas être modifiée après l'affectation initiale** (dans la déclaration ou dans le constructeur). Permet de définir des **valeurs constantes**.
- **Dans la déclaration d'une méthode**, le modificateur `final` indique que **la méthode ne peut pas être redéfinie dans une sous-classe**.
- **Dans la déclaration des paramètres d'une méthode**, le modificateur `final` indique que la valeur de ces paramètres ne peut pas être modifiée (il s'agit de paramètres d'entrée de la méthode).

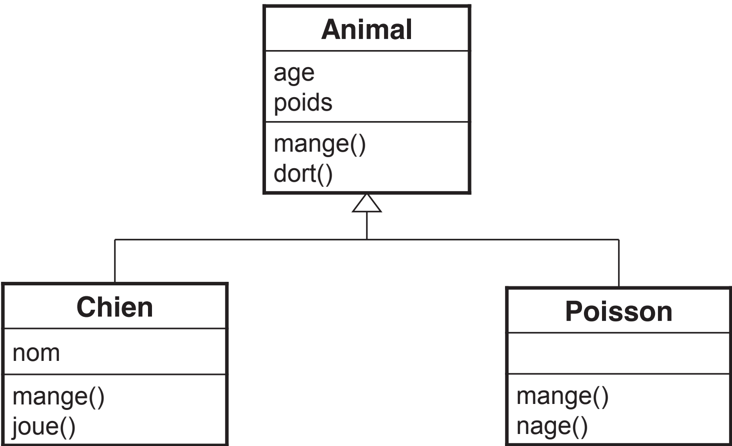
3.2 Le polymorphisme

Le **polymorphisme** peut être vu comme **la capacité de choisir dynamiquement la méthode qui correspond au type réel de l'objet**.

- En Java, dans la plupart des situations où il y a des relations d'héritage, la détermination de la méthode à invoquer n'est pas effectuée lors de la compilation.
- C'est seulement à l'exécution que la machine virtuelle déterminera la méthode à invoquer selon le type effectif de l'objet référencé à ce moment là.

– Ce mécanisme s'appelle "**Recherche dynamique de méthode**" (*Late Binding* ou *Dynamic Binding*).
– Ce mécanisme de recherche dynamique (durant l'exécution de l'application) sert de base à la mise en œuvre de la propriété appelée **polymorphisme**.

Exemple 3.7. *On considère le diagramme de classes suivant :*



Si l'on souhaite enregistrer et manipuler une collection d'animaux (une ménagerie) on peut créer et alimenter le tableau suivant :

```
// Déclaration et création du tableau
 Animal[] menagerie = new Animal[6];
// Alimentation du tableau
 menagerie[0] = new Poisson(...);
 menagerie[1] = new Chien(...);
 menagerie[2] = new Chien(...);
 menagerie[3] = new Animal(...);
 menagerie[4] = new Poisson(...);
 menagerie[5] = new Chien(...);
```

Le polymorphisme nous permet d'écrire une méthode `nourrir` dont la fonction est de donner à manger à chaque animal contenu dans le tableau passé en paramètre en appelant successivement la méthode `mange()` pour chacun d'eux.

```
//-----------------------------------------------------
// Appelle la méthode mange() pour chaque animal
// contenu dans le tableau passé en paramètre
//-----------------------------------------------------
public static void nourrir(Animal[] tabAnimaux) {
        if (tabAnimaux == null) return;
        for (int i=0; i<tabAnimaux.length; i++) {
          if (tabAnimaux[i] != null) {
            tabAnimaux[i].mange();
```

```
} }
}
```

On peut ensuite appeler la méthode `nourrir()` en lui passant en paramètre la ménagerie précédemment créée :

```
nourrir(menagerie);
```

Sur la base du contenu de `menagerie`, la méthode `nourrir()` appellera successivement :

- `mange()` de la classe `Poisson`
- `mange()` de la classe `Chien`
- `mange()` de la classe `Chien`
- `mange()` de la classe `Animal`
- `mange()` de la classe `Poisson`
- `mange()` de la classe `Chien`

La méthode `nourrir()` n'a pas besoin de déterminer elle-même quelle méthode doit être appelée pour chaque animal. **Le polymorphisme fera en sorte que le message mange() soit interprété (à l'exécution) de manière appropriée selon les objets qui le reçoivent** (ainsi chaque animal mangera selon ses goûts !).

Remarque 3.2. *Avec l'encapsulation et l'héritage, le polymorphisme est une des propriétés essentielles de la programmation orientée objet.*

La surcharge de méthodes peut également être considérée comme une forme de polymorphisme. Le choix de la méthode à invoquer est cependant déterminé à la compilation.

3.3 Relations entre classes

Il existe une autre relation importante qui peut exister entre deux classes. Il s'agit de la relation de **composition** (ou **agrégation**) qui est caractérisée par une relation de type :

A un...", "**Possède un...**", "**Est composé de...**" ou "**Contient...**".

Exemple 3.8. *Une voiture* ***possède*** *quatre roues*
Une voiture ***a un*** *propriétaire*

En Java, les relations de composition sont réalisées en créant dans la classe "contenant" une **référence vers un objet** de la classe "contenu".

La distinction sémantique entre composition et agrégation ne se traduit pas (en Java) par des différences d'implémentation.

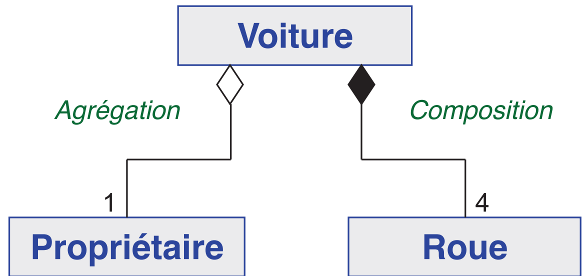


Figure 3.2: Composition / agrégation

```
public class Voiture {
   ...
  private Propriétaire leProprietaire;
  private Roue[ ] lesRoues;
   ...
}
```

3.4 La classe racine Object en Java

En Java, toutes les classes héritent par défaut de la classe `java.lang.Object`

La déclaration :

```
public class Point { ... }
```

est équivalent (implicitement) à :

```
public class Point extends Object { ... }
```

3.4.1 Redéfinition de méthodes clés de Object

La classe racine `Java.lang.Object` contient quelques méthodes, qu'on peut réutiliser telles quelles ou redéfinir.

Parmi elles, 3 sont particulièrement importantes à redéfinir :

1. `String toString()`
2. `boolean equals(Object obj)`
3. `int hashCode()`

3.4.1.1 Redéfinition de toString

la méthode `String java.lang.Object::toString()` fournit la représentation sous forme de chaîne de caractères d'information sur l'état actuel de l'objet sollicité. Son implémentation par défaut dans `Object` fournit une chaîne contenant :

– le nom de la classe concernée

– l'adresse de l'objet en mémoire, en hexadecimal, précédée de @

Cette représentation n'est pas satisfaisante pour la majorité des objets dans les applications utilisateurs

Ex. : Afficher un objet de type Employe signifie bien plus que le nom de la classe et l'adresse de l'objet !

Exemple 3.9. ***Absence de redéfinition de*** `String toString()`

```
public class Employe {
 // Pas de redéfinition de String toString()
 }
public class StartUp {
  public static void main(String[] args) {
    Employe e = new Employe("Sandrine", "Durand");
    System.out.println(e);
    /* La méthode toString() de l'objet e est automatiquement
    invoquée. Affiche par exemple : "Employe@580ab5cc" */
 }
 }
```

Exemple 3.10. ***Redéfinition de*** `String toString()`

```
public class Employe {
 // Redéfinition de String toString()
@Override
public String toString(){ return prenom + "␣" + nom; }
}
public class StartUp {
   public static void main(String[] args) {
     Employe e = new Employe("Sandrine", "Durand");
     System.out.println(e); // Affiche : "Sandrine Durand"
     /* On peut aussi : System.out.println(e.toString());
      On peut  egalement : System.out.println("Employ e : " + e);
      affichera : "Employ e : Sandrine Durand" */
}
}
```

3.4.1.2 Redéfinition de equals

Comparaison des objets

Si `a` et `b` sont d'un type classe ou tableau, la condition `a == b` ne traduit pas l'égalité des valeurs de `a` et `b`, mais l'**égalité des références**.

Autrement dit, la condition (`a == b`) est vraie non pas lorsque `a` et `b` sont des objets égaux, mais **lorsque `a` et `b` sont le même objet**.

La méthode `boolean java.lang.Object::equals(Object o)` renvoie vraie si l'objet courant < est égal > à l'objet passé en argument. Son implémentation par défaut dans `Object` n'est rien de plus que l'égalité des références.

```
class Object {
    ...
    public boolean equals(Object o) {
        return this == o;
    }
}
```

Exemple 3.11. *Absence de redéfinition de* `boolean equals(Object o)`

```
Point p = new Point(10, 20);
Point q = new Point(10, 20);
...
System.out.println(p == q); // ceci affiche false
System.out.println(p.equals(q));
// ceci affiche false car c'est la méthode equals de Object qui a été invoquée.
```

Tous les objets sont censés posséder la méthode `equals` qui implémente une notion d'égalité plus utile.

Chaque classe peut redéfinir la méthode `equals` pour en donner une version adaptée au rôle et aux détails internes de ses instances. Pour la classe `Point` voici une première version de la méthode `equals`, pas très juste :

```
class Point {
    ...
    public boolean equals(Point o) {
        return x == o.x && y == o.y; // VERSION ERRONÉE !!!
    }
}
```

Bien que répondant en partie aux besoins, cette version de `Point.equals` est erronée car, à cause du type de l'argument, **elle ne constitue pas une redéfinition de la méthode** `Object.equals(Object o)`. Voici une version plus correcte :

```
class Point {
    ...
    public boolean equals(Object o) {
        return o instanceof Point
                   && x == ((Point) o).x
                   && y == ((Point) o).y;
    }
}
```

Remarque 3.3. *la méthode* `instanceof` *est présentée en section 3.1.6.*

3.4.1.3 Redéfinition de hashCode

La sémantique de `equals` représente une relation d'équivalence sur les objets non `null` :

- **réflexive** : `x.equals(x)` doit renvoyer vrai
- **symétrique** : si `x.equals(y)` alors `y.equals(x)`
- **transitive** : si `x.equals(y)` et `y.equals(z)` alors `x.equals(z)`
- **cohérente** : l'invocation répétée de `x.equals(y)` doit renvoyer la même valeur, pourvu que l'implémentation de `equals` n'a pas changé la sémantique de l'égalité
- `x.equals(null)` doit renvoyer faux

hashcode et equals

Il en résulte que deux objets égaux par `equals` **doivent renvoyer la même signature numérique ou hash code**. Mais l'inverse n'est pas forcément vrai !

☞ Tout objet redéfinissant `equals` doit redéfinir `hashcode` si l'objet doit être utilisé dans les `Collection` (donc tout le temps)

La méthode `int hashCode()` :

- Renvoie un entier qui peut être utilisé comme valeur de hachage de l'objet
- Permet au objet d'être utiliser dans les tables de hachage. Comme `hashCode()` est utilisé par l'API des collections basé sur des tables de hachage (`java.util.HashMap` et `java.util.HashSet`), il est important que la valeur de hachage couvre l'ensemble des entiers possibles, sinon la table de hachage se transforme en liste chainée (O(1) -> O(n))

```
public class Point {
   public Point(int x,int y) {
        this.x=x;
        this.y=y;
   }
  @Override
  public int hashCode() {
```

```
        return x ^ Integer.rotateLeft(y,16);
    }
  @Override
  public boolean equals(Object o) {
        ...
    }
```

- `x.equals(y)` implique `x.hashcode()==y.hashcode()`
- Les valeurs de `hashcode` doivent de préférence être différentes pour les objets du programme
- `hashcode` doit être rapide à calculer (éventuellement précalculée).

```
/**
 * Simple Java Class to represent Person with name, id and date of birth.
 */
public class Person implements Comparable<Person>{
   private String name;
   private int id;
   private Date dob;

    public Person(String name, int id, Date dob) {
        this.name = name;
        this.id = id;
        this.dob = dob;
    }

   @Override
   public boolean equals(Object other){
       if(this == other) return true;

       if(other == null || (this.getClass() != other.getClass())){
           return false;
       }

       Person guest = (Person) other;
       return (this.id == guest.id) &&
              (this.name != null && name.equals(guest.name)) &&
              (this.dob != null && dob.equals(guest.dob));
   }

    @Override
   public int hashCode(){
       int result = 0;
       result = 31*result + id;
       result = 31*result + (name !=null ? name.hashCode() : 0);
       result = 31*result + (dob  !=null ? dob.hashCode() : 0);
```

```java
        return result;
    }

    @Override
    public int compareTo(Person o) {
        return this.id - o.id;
    }
}
```

3.4.1.4 Copie des objets - redéfinition de la méthode clone()

Une conséquence du fait que **les objets** et les tableaux **sont toujours manipulés à travers des références** est la suivante :

– **l'affectation d'une expression de type objet ou tableau ne fait pas une vraie duplication de la valeur de l'expression,** mais uniquement une **duplication de la référence** (ce qu'on appelle parfois une copie superficielle).

Exemple 3.12. *considérons :*

```java
Point p = Point(10, 20);
Point q = p;
```

À la suite de l'affectation précédente on peut penser qu'on a dans `q` un duplicata de `p`. **FAUX**, **il n'y a pas deux points, mais un seul point référencé par deux variables distinctes**, comme le montre la figure 3.3.

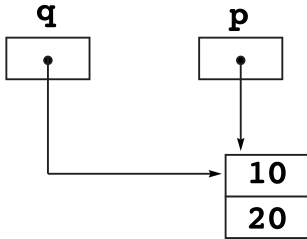


Figure 3.3: Copie de la référence sans duplication de l'objet (copie superficielle)

Inconvénient

Dans certaines situations cette manière de "copier" est suffisante, mais elle présente un inconvénient évident : **chaque modification de la valeur de p se répercute automatiquement sur celle de q**.

DUPLICATION EFFECTIVE D'UN OBJET

Pour obtenir la **duplication effective d'un objet**, il faut appeler sa méthode `clone`.

La méthode clone()

Tous les objets sont censés posséder une telle méthode car elle est déclarée dans la classe `Object`, la super-classe de toutes les classes. Il appartient à chaque classe de redéfinir la méthode `clone`, pour en donner une version adaptée au rôle et aux détails internes de ses instances.

La méthode `clone` rend un résultat de type `Object`, il faut donc l'utiliser comme suit (la nature même de la méthode `clone` garantit que la conversion du résultat de `p.clone()` vers le type `Point` est légitime) :

```
Point q = (Point) p.clone();
```

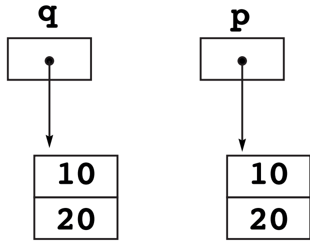


FIGURE 3.4: Duplication effective d'un objet (copie profonde)

DÉFINIR LA MÉTHODE CLONE

La méthode clone doit être redéfinie dans chaque classe où cela est utile ; elle doit être public.

Exemple 3.13.

```
class Point {
    int x, y;
    public Point(int a, int b) {
        x = a; y = b;
    }
    public Object clone() {
```

```
            return new Point(x, y);
        }
    }
```

Class< ?> getClass()

? Permet d'obtenir un objet `Class` représentant la classe d'un objet particulier ? Un objet `Class` est un objet qui correspond à la classe de l'objet à l'exécution

```
String s="toto";
Object o="tutu";
s.getClass()==o.getClass(); // true
```

? Cette méthode est `final` ? Il existe une règle spéciale du compilateur indiquant le type de retour de `getClass()` (cf cours Reflection & Types Paramétrés)

Chapitre Quatre

Les classes abstraites et interfaces

4.1 Classes abstraites

Une **classe abstraite** est une classe qui **contient une ou plusieurs méthodes abstraites**. Elle peut malgré tout contenir d'autres méthodes (concrètes).

- **Une méthode abstraite** est une méthode qui **ne contient pas de corps**. Elle possède simplement une signature de définition suivie du caractère point-virgule (pas de bloc d'instructions).
- Une méthode abstraite doit obligatoirement être déclarée avec le modificateur `abstract`.
- Il est obligatoire d'employer le qualifieur `abstract` devant toute classe qui possède des méthodes abstraites, propres ou héritées.

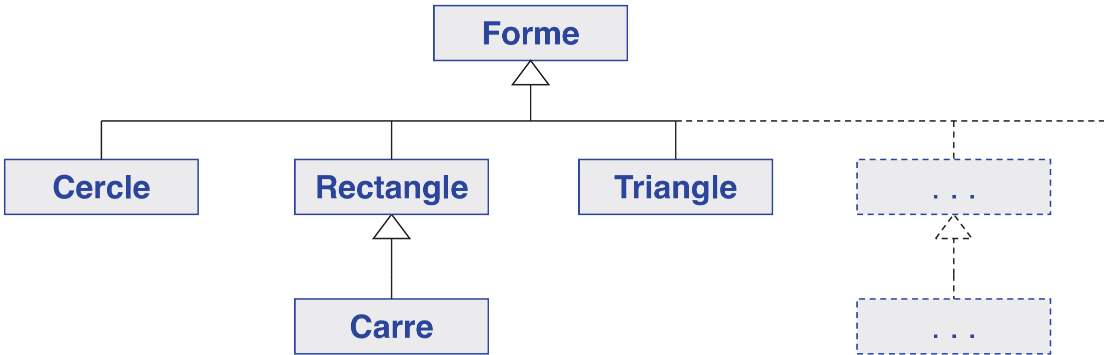
Règles

Les règles suivantes s'appliquent aux classes abstraites :

- Une classe abstraite **ne peut pas être instanciée** : on ne peut pas créer d'objet en utilisant l'opérateur `new`.
- Une sous-classe d'une classe abstraite **ne peut être instanciée que si elle redéfinit chaque méthode abstraite de sa classe parente** et qu'elle fournit une implémentation (un corps) pour chacune des méthodes abstraites.
- Si une sous-classe d'une classe abstraite n'implémente pas toutes les méthodes abstraites dont elle hérite, cette sous-classe est elle-même abstraite (et ne peut donc pas être instanciée).
- Les méthodes déclarées avec l'un des modificateurs suivants : `static`, `private` ou `final` ne peuvent pas être abstraites étant donné qu'elles ne peuvent pas être redéfinies dans une sous-classe.

Exemple 4.1. *On souhaite créer une hiérarchie de classes qui décrive des formes géométriques à deux dimensions.*

- *Sachant que toutes les formes possèdent les propriétés périmètre et surface, il est judicieux de placer les méthodes définissant ces propriétés (*`perimetre()` *et* `surface()`*) dans la classe qui est à la racine de l'arborescence (*`Forme`*).*
- *La classe* `Forme` *ne peut cependant pas implémenter ces deux méthodes car on ne peut pas calculer le périmètre et la surface sans connaître le détail de la forme.*



- *On pourrait naturellement implémenter ces méthodes dans chacune des sous-classes de* Forme *mais il n'y a aucune garantie que toutes les sous-classes les possèdent (ce serait laissé au bon vouloir des programmeurs des sous-classes).*
- *Il est possible de résoudre ce problème en déclarant dans la classe* Forme *deux méthodes abstraites* perimetre() *et* surface() *ce qui rend la classe* Forme *elle-même abstraite et impose aux créateurs de sous-classes de les implémenter.*

```
//-----------------------------------------------------------
// Forme
//-----------------------------------------------------------
public abstract class Forme { // Classe abstraite
  public abstract double perimetre(); // Méthode abstraite
  public abstract double surface(); // Méthode abstraite
}
//-----------------------------------------------------------
// Cercle
//-----------------------------------------------------------
public class Cercle extends Forme {
  public static final double PI = 3.14159265358979;
  protected double rayon;
  public Cercle(double rayon) {
        this.rayon = rayon;
  }
  public double getRayon()  { return rayon; }
  public double perimetre() { return 2*PI*rayon; }
  public double surface()   { return PI*rayon*rayon; }
}

//-----------------------------------------------------------
// Rectangle
//-----------------------------------------------------------
public class Rectangle extends Forme {
  protected double longueur, largeur;
  public Rectangle(double longueur, double largeur) {
    this.longueur = longueur;
```

```
    this.largeur  = largeur;
}
  public double getLongueur() { return longueur; }
  public double getLargeur() { return largeur; }
  public double perimetre() { return 2*(longueur + largeur);
}
  public double surface() {
    return longueur * largeur;
} }
```

Même si la classe `Forme` est abstraite, il est tout de même possible de déclarer des variables de ce type qui pourront recevoir des objets créés à partir des sous-classes concrètes :

```
Forme[] dessin = new Forme[3];
dessin[0] = new Rectangle(2.5, 6.8); // Conversion élargissante
dessin[1] = new Cercle(4.66); // Conversion élargissante
dessin[2] = new Carre(0.125); // Conversion élargissante
```

Le polymorphisme permet ensuite d'invoquer une méthode commune sur chacun des objets :

```
double surfaceTotale = 0.0;
for (int i=0; i<dessin.length; i++) {
   // Calcule la somme des surfaces
   surfaceTotale += dessin[i].surface();
}
```

4.2 Interface

Une interface est une classe entièrement faite de membres publics qui sont :

- des méthodes abstraites (ou non depuis Java 8),
- variables statiques finales (c'est-à-dire des constantes de classe). Les variables sont implicitement `static` et `final` même si les modificateurs correspondants sont omis.

Une interface se déclare avec le mot-clée `interface` au lieu de `class`.

Exemple 4.2.

```
public interface Imprimable {
  public void print();
}
```

Exemple 4.3. *Un autre exemple, extrait de la bibliothèque Java (plus précisément du paquet* `java.util`*) :*

```
     interface Collection {
         boolean isEmpty();
```

```
        int size();
        boolean contains(Object o);
        boolean add(Object o);
        Iterator iterator();
       ...
   }
```

Une déclaration d'interface définit un nouveau type référence (comme le fait une déclaration de classe ou de classe abstraite).

Exemple 4.4.

```
Imprimable document;
Imprimable[] aImprimer;
```

Règles

Les règles suivantes s'appliquent aux interfaces :

- **Une interface ne peut définir que des méthodes abstraites** qui sont implicitement publiques. Il n'y a alors pas besoin d'écrire les qualifieurs `public` et `abstract` devant les méthodes.
- Une interface ne peut pas contenir de méthodes statiques.
- Une interface ne définit **pas de constructeur** (on ne peut pas l'instancier).

4.2.1 Implémentation d'interfaces

Une interface est une **spécification** : elle fixe la liste des méthodes qu'on est certain de trouver dans toute classe qui déclare être conforme à cette spécification. Cela s'appelle une **implémentation de l'interface**, et s'indique avec le qualifieur `implements` :

```
public class Rapport implements Imprimable {
  public void print(){
    // implémentation de la méthode print
  }
  ...}

class Tas implements Collection {
   public boolean isEmpty() {
        // implémentation de la méthode isEmpty
        }
   public int size() {
        // implémentation de la méthode size
        }
        ...
}
```

Remarque 4.1. *Notez que les méthodes des interfaces sont toujours publiques (implicitement) ; par conséquent, leurs définitions dans des sous-classes doivent être explicitement qualifiées public, sinon une erreur sera signalée.*

4.2.2 Utilisation des interfaces

- **Deux interfaces peuvent hériter l'une de l'autre**. Par exemple, dans le paquet `java.util` on trouve l'interface `SortedSet` (ensemble trié) qui est une sous-interface de `Set`, elle-même une sous-interface de `Collection`, etc.

  ```
  public interface Zoomable extends Observable {...}
  ```

  Une sous-interface hérite de toutes les méthodes abstraites et de toutes les constantes de son interface parente et peut définir de nouvelles méthodes abstraites ainsi que de nouvelles constantes.
- En Java, une classe ne peut hériter que d'une et d'une seule classe parente (héritage simple). **Une classe peut**, par contre, **implémenter plusieurs interfaces** (une sorte d'héritage multiple) :

  ```
  UneClasse implements interface1, interface2, ...
  ```

- L'**implémentation d'une ou de plusieurs interfaces** (`implements`) **peut être combinée avec l'héritage simple** (`extends`). La clause `implements` doit suivre la clause `extends`.

  ```
  public class Rapport implements Imprimable {...}
  public class Livre implements Imprimable, Zoomable {...}
  public class Cercle extends Forme implements Imprimable {...}
  public class Carre extends Rectangle
                        implements Imprimable, Zoomable {...}
  ```

- Contrairement aux classes, une interface peut posséder plusieurs interfaces parentes (héritage multiple).

  ```
  public interface Transformable
  extends Scalable, Translatable, Rotatable {...}
  ```

4.2.3 Exemple

Papiers, bouteilles, piles électriques, Cageots, etc. sont des objets différents, ayant des comportements différents (déchirer du papier, remplir une bouteille, ...), mais sont tous recyclables (tous peuvent être recyclés même si le processus est différent)

```
public interface Recyclable {
public void recycle();
} // Recyclable
```

```
public class Paper implements Recyclable {
public void dechire() { ... }
public void recycle() {
System.out.prilntln("recyclage␣papier");
}
} // Paper

public class Bottle implements Recyclable {
...
public void recycle() {
System.out.println("recyclage␣bouteille");
}
} // Bottle
...
Recyclable[] trashcan = new Recyclable[2];
trashcan[0] = new Paper(); // projection des instances
trashcan[1] = new Bottle(); // sur le ??type'' Recyclable
for (int i = 0; i< trashcan.length; i++) {
trashcan[i].recycle(); // message indifférencié
} // mais traitements différents

+---trace--------------
| recyclage papier
| recyclage bouteille
+----------------------
```

Important

La référence n'est pas l'objet !

- Le type de la référence définit les envois de message autorisés.
- La classe de l'objet définit le traitement exécuté.

```
Paper p = new Paper();
p.dechire(); // ok
p.recycle(); // ok
Recycable r = p; // ok : p est aussi de type Recyclable : 2 références sur le m
r.recycle(); // ok : code de recycle dans classe Paper exécuté
r.dechire(); // NON, envoi de message interdit sur type Recycable
```

on peut utiliser une interface là où on utilise une classe (sauf création d'instances) tous les deux sont des ?types ?

```
public void aMethod(Recyclable r) {
...
//traitement en n'invoquant sur r que des méthodes de Recyclable
}
```

```
...
Recyclable aRecyclableObject = new Paper();
someObject.aMethod(aRecyclableObject);
someObject.aMethod(new Bottle()); // projection implicite sur l'interface
```

une classe peut implémenter plusieurs interfaces, elle doit dans ce cas fournir un comportement pour chacune des méthodes de chaque interface.

```
public interface Flammable {
public void burn();
} // Flammable
...
public class Paper implements Recyclable, Flammable {
public void recycle() { ... traitement ... }
public void burn() { ... traitement ... }
} // Paper
```

Quels types possibles pour une instance de `Paper` ?

```
Recyclable[] trashcan = new Recyclable[2];
trashcan[0] = new Paper(); // projection des instances
trashcan[1] = new Bottle(); // sur le ??type'' Recyclable
for (int i = 0; i< trashcan.length; i++) {
trashcan[i].recycle(); // traitement indifférencié

+--------------------
 | recyclage papier
 | recyclage bouteille
+-------------------
```

Comment la ?bonne ? méthode est-elle appelée ? Late-binding et early binding early binding compilateur gén'ere un appel à une fonction en particulier et le code appelé est précisément déterminé lors de l'édition de liens late binding (POO) le code appelé lors de l'envoi d'un message à un objet n'est déterminé qu'au moment de l'exécution (run time) le compilateur ne vérifie « que » l'acceptation du message et la validité des types d'arguments et de retour.

Late-binding

Mécanisme fondamental de la programmation objet

```
public class Recycleur {
/** applique le processus de recyclage à r
* @param r l'objet à recycler */
public void doIt(Recyclable obj) {
obj.recycle();
}
}
public class RecycleurMain {
```

```
public static void main(String[] args) {
Recycleur usine = new Recycleur();
Recyclable ref;
if (args[0].equals("papier")) {
ref = new Paper() ;
}
else { ref = new Bottle(); }
usine.doIt(ref);
}
} }
```

Ce code compile, résultat dépend de args[0]. ...> java Recycleur papier recyclage papier ...> java Recycleur autre recyclage bottle méthode recycle invoquée non connue a priori.

4.2.4 Interface de marquage

Il est parfois utile de définir une **interface entièrement vide** appelée **interface de marquage**. Les interfaces `java.lang.Cloneable` et `java.io.Serializable` constituent deux exemples d'interfaces de marquage de la plateforme Java.

- Une classe peut implémenter cette interface en la nommant dans la clause `implements` sans avoir à implémenter de méthodes.
- Il est donc possible de tester si un objet implémente l'interface de marquage à l'aide de l'opérateur `instanceof` :

  ```
  if (obj instanceof Cloneable) {...}
  ```

- Une interface de marquage sert à communiquer aux instances qui l'implémentent des informations supplémentaires concernant l'objet, son comportement, son implémentation, ...

? Classe abstraite ou interface ?

Lors de la conception d'une application ou d'une librairie, le choix conceptuel entre une classe abstraite et une interface n'est pas toujours facile. Les points à prendre en considération sont les suivants :

– Une interface est une pure spécification, si elle contient un grand nombre de méthodes, il peut être fastidieux de toutes les implémenter (il n'est pas possible de définir des comportements par défaut).
– Une classe abstraite peut contenir des méthodes concrètes.
– Une interface peut être implémentée par une classe sans qu'il y ait de rapports étroits entre les deux. Une sous-classe est liée par une relation plus forte ("Est un ...").
– Si l'on ajoute de nouvelles méthodes à une interface, toutes les classes qui l'implémentent doivent implémenter les nouvelles méthodes. Avec une classe abstraite, il est possible de définir une nouvelle méthode non-abstraite avec une implémentation par défaut.

4.2.5 Problème

On s'intéresse à la modélisation d'un bricoleur qui peut effectuer certaines tâches telles que visser, couper, casser. Chacune de ces tâches s'accomplit à l'aide d'un outil adapté.

Par exemple, un tournevis est un outil adapté pour visser, on pourrait donc avoir quelque chose ressemblant à :

```
public class Bricoleur {
    public void visse(Tournevis t) {
        t.visse();
    }
    ...
}

public class Tournevis {
    public void visse() {
        System.out.println("Tournevis visse");
    }
}

public class Tournevis {
    public void visse() {
        System.out.println("Tournevis visse");
    }
}

public class Marteau {
    public void casse() {
```

```
        System.out.println("Marteau casse");
    }
}

public class Scie {
    public void coupe() {
        System.out.println("Scie coupe");
    }
}

public class Bricoleur {
    public void visse(Tournevis t) {
        t.visse();
    }
    public void casse(Marteau m) {
        m.casse();
    }
    public void coupe(Scie s) {
        s.coupe();
    }
    ...
}
```

Prise en compte d'un cutter ? d'une masse ?

On ajoute :

```
public class Masse {
    public void casse() {
        System.out.println("Masse casse");
    }
}

public class Cutter {
    public void coupe() {
        System.out.println("Cutter coupe");
    }
}
```

? NON !

Pas de généralisation possible, on est obligé de modifier le code de `Bricoleur` pour ajouter un nouvel outil

```
public class Bricoleur {
    public void visse(Tournevis t) {
        t.visse();
    }
    public void casse(Marteau m) {
        m.casse();
```

```
    }
    public void coupe(Scie s) {
        s.coupe();
    }
    public void coupe(Cutter c) {
        c.coupe();
    }
    public void casse(Masse m) {
        m.casse();
    }
    ...
}
```

Utiliser les interfaces

Définir une interface pour les outils sachant couper, visser, casser (définir des abstractions pour ces notions).

```
public interface PeutVisser {
        public void visse();
}

public interface PeutCouper {
        public void coupe();
}

public interface PeutCasser {
        public void casse();
}
```

Ce qui donne :

```
public class Tournevis implements PeutVisser {
    public void visse() {
        System.out.println("Tournevis visse");
    }
}

public class Scie implements PeutCouper {
    public void coupe() {
        System.out.println("Scie coupe");
    }
}

public class Marteau implements PeutCasser {
    public void casse() {
        System.out.println("Marteau casse");
    }
}
```

et donc

```
public class Bricoleur {
    public void visse(PeutVisser visseur) {
        visseur.visse();
    }
    public void casse(PeutCasser cassant) {
        cassant.casse();
    }
    public void coupe(PeutCouper coupant) {
        coupant.coupe();
    }
    ...
}

Bricoleur bob = new Bricoleur();
bob.coupe(new Scie());
bob.casse(new Marteau());
+--trace-------------
+ Scie coupe
+ Marteau casse
+--------------------
```

si maintenant on ajoute :

```
public class Masse implements PeutCasser {
    public void casse() {
        System.out.println("Masse casse");
    }
}

public class Cutter implements PeutCouper {
    public void coupe() {
        System.out.println("Cutter coupe");
    }
}
```

Sans rien modifier on peut écrire :

```
Bricoleur bob = new Bricoleur();
bob.coupe(new Scie());
bob.coupe(new Cutter());
bob.casse(new Marteau());
bob.casse(new Masse());
```

qui produit :

```
+--trace-------------
+ Scie coupe
+ Cutter coupe
```

```
+ Marteau casse
+ Masse casse
+--------------------
```

Multi-Implémentation

```
public class CouteauSuisse implements PeutCouper, PeutVisser, PeutCasser {
    public void coupe() {
        System.out.println("CouteauSuisse␣coupe");
    }
    public void visse() {
        System.out.println("CouteauSuisse␣visse");
    }
    public void casse() {
        System.out.println("CouteauSuisse␣casse");
    }
}
```

```
Bricoleur mcGyver = new Bricoleur();
CouteauSuisse swissKnife = new CouteauSuisse();
mcGyver.coupe(swissKnife);
mcGyver.casse(swissKnife);
mcGyver.visse(swissKnife);
+--trace-----------------
+ CouteauSuisse coupe
+ CouteauSuisse casse
+ CouteauSuisse visse
+------------------------
```

```
CouteauSuisse swissKnife = new CouteauSuisse();
PeutCouper coupant = swissKnife; // Upcast de CouteauSuisse --> PeutCouper
coupant.coupe(); // pas de pb
swissKnife.casse(); // pas de pb
coupant.casse(); // !!! INTERDIT !!!
// (détecté à la compilation)
((CouteauSuisse) coupant).casse(); // ok : Downcast licite de
                        // PeutCouper --> CouteauSuisse
((Marteau) coupant).casse(); // compile mais Downcast illicite de
                        // Marteau --> CouteauSuisse
```

4.2.6 Interface de typage

On veut pouvoir ranger les différents outils dans une boîte à outils représentée par un tableau.

Solution : avoir une interface `Tool` qui sert uniquement à repérer les outils (typer)

```
public interface Tool { }
public class Scie implements PeutCouper, Tool { ...}
public class Marteau implements PeutCasser, Tool { ...}
Tool[] ToolBox = new Tool[5];
ToolBox[0] = new Scie();
ToolBox[1] = new Marteau();
...
```

4.2.7 Interface et évolution

Avant la version 8 de Java, il n'est pas possible de faire évoluer une interface en ajoutant des méthodes a postériori.

Une interface demande à ce que les classes qui l'implante implante toutes ses méthodes

La version 8 de Java introduit la notion de **méthode par défaut**.

Une méthode par défaut est une méthode dont l'implantation est utilisée si la classe ne fournit pas elle même d'implantation

```
interface StringComparator {
    public abstract int compare(String s1, String s2);
    public default boolean lessThan(String s1, String s2) {
        return compare(s1, s2) < 0;
    }
}
```

Si une classe qui implante `StringComparator` ne déclare pas la méthode `lessThan`, l'implantation par défaut sera choisie (utilisation du mot clé `default`).

Les méthodes par défaut sont virtuelles : elle peuvent être redéfinies dans les classes qui implémente l'interface.

Conceptuellement, une interface qui possède des méthodes qui ont du code est appelée un **trait**.

En Java, il n'y a pas un mot-clé spécifique pour ces interfaces, ce sont juste **des interfaces avec des méthodes par défaut**.

Remarque 4.2. *Comme* `java.lang.Object` *fourni toujours les méthodes* `toString`, `equals` *et* `hashCode`, *il est inutile d'écrire une méthode par défaut* `toString`, `equals` *ou* `hashCode` *dans une interface.*

La version de `java.lang.Object` *sera toujours préférée à celle de l'interface*

4.2.8 Méthodes par défaut et conflit

Si deux méthodes par défaut sont disponibles, celle du sous-type est choisie si il existe un sous-type, sinon le compilateur plante

```
public interface Bag {
public abstract int size();
public default boolean isEmpty() {
return size() == 0;
}
}

public class HashBag implements Bag {
private int size;
...
public int size() {
return size;
}
// isEmpty de Bag est utilisé
}

public interface Empty {
public default boolean isEmpty() {
return true;
}
}
public class EmptyBag implements Bag, Empty {
// problème, 2 méthodes isEmpty() par défaut
}
```

Résoudre le conflit

```
public interface Empty {
public default boolean isEmpty() {
...
}}
public interface Bag {
...
public default boolean isEmpty() {
...
} }
public class EmptyBag implements Bag, Empty {
public boolean isEmpty() {
return Empty.super.isEmpty();
} }
```

`SuperInterface.super` permet d'accéder à l'implantation par défaut dans une interface

Chapitre Cinq

Les exceptions

5.1 Exceptions

Les exceptions représentent des **événements** qui peuvent survenir durant l'exécution d'un programme et qui **perturbent le déroulement normal des instructions**.

1. Les exceptions sont principalement utilisées pour représenter des **erreurs** de différents types :
   - des erreurs matérielles (crash du disque, ...)
   - des erreurs de programmation (indice d'un tableau hors limites, ...)
   - des erreurs liées à l'environnement d'exécution (mémoire insuffisante, ...),
   - des erreurs spécifiques à une librairie ou à une application (numéro d'article non-défini),
   - etc.
2. Les exceptions peuvent également représenter des événements qui en sont pas à proprement parler des erreurs, mais qui correspondent à des **situations exceptionnelles** (prévues) qui doivent être traitées de manières différentes du flux normal des opérations. Par exemple :
   - la fin d'un fichier,
   - un mot de passe incorrect,
   - des ressources momentanément non-disponibles,
   - etc.

5.2 Avantages des exceptions

- Les exceptions augmentent la **lisibilité** du code en **séparant** les instructions qui traitent le cas normal des instructions nécessaires au traitement des erreurs ou des événements exceptionnels.
- Elles permettent la **déclaration explicite des exceptions** qu'une méthode peut lever (fait partie de la signature).
- Forcent le programmeur à prendre en compte (traiter ou propager) les cas exceptionnels (erreurs ou autres événements) qui sont déclarés dans les méthodes qu'il invoque.

– Offrent un mécanisme de propagation automatique ce qui permet au programmeur de choisir à quel niveau il souhaite traiter l'exception (celui auquel il est à même de prendre les mesures adéquates).

5.3 Gestion des exceptions

En Java, les exceptions sont représentées par des objets de type `Throwable`.

La figure 5.1 montre la hiérarchie des classes d'exception. La classe de base pour tous les objets d'exception est `java.lang.Throwable`, avec ses deux sous-classes `java.lang.Exception` et `java.lang.Error`.

– La classe `Error` décrit les erreurs internes du système (par exemple, `VirtualMachineError`, `LinkageError`).

– La classe `Exception` décrit l'erreur causée par le programme (par exemple `FileNotFoundException`, `IOException`).

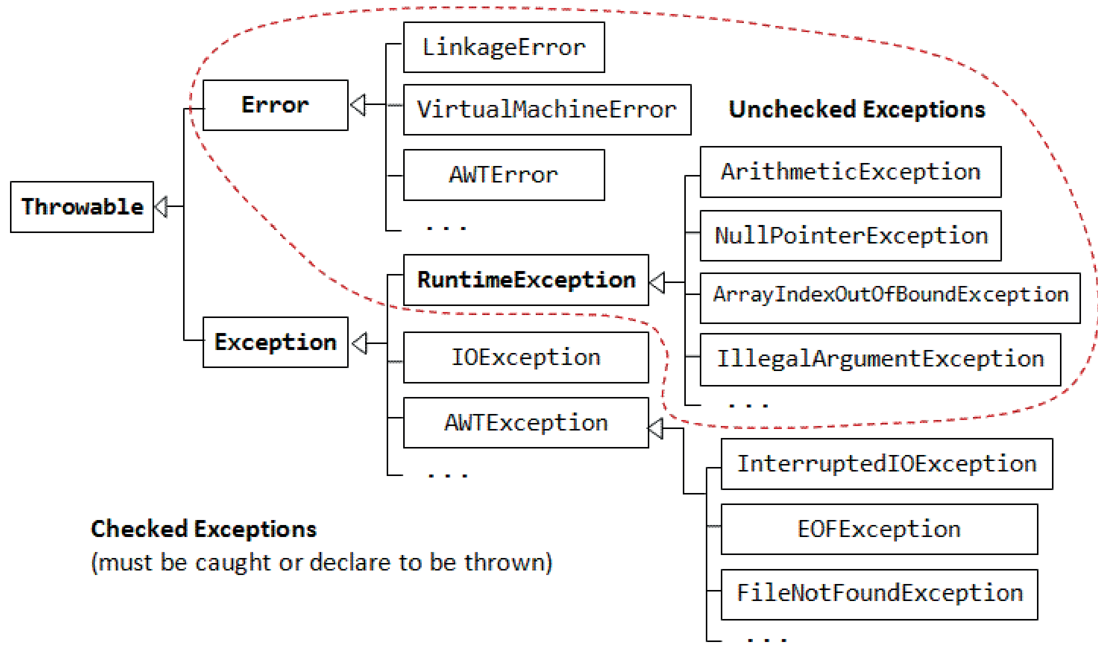


Figure 5.1: Hiérarchie d'exceptions

Différentes instructions permettent de gérer les exceptions.

– L'instruction `throw` sert à générer une exception (on dit également **lever une exception, lancer une exception**, ...).

– L'instruction `try / catch / finally` constitue le cadre dans lequel les exceptions peuvent être **détectées (capturées) et traitées**.

– Le mot-clé `throws` (à ne pas confondre avec `throw`) est utilisé dans la déclaration de méthode (**signature**) pour annoncer **la liste des exceptions** que la méthode peut générer. L'utilisateur de la méthode est ainsi informé des exceptions qui peuvent survenir lors de son invocation et peut prendre les mesures nécessaires pour gérer ces événements exceptionnels (c'est-à-dire les **traiter** ou les **propager**).

Checked/Unchecked Exceptions

Les exceptions contrôlées (checked) [du type `Exception`] doivent être soit traitées (dans une clause `catch`), soit propagées pour être traitées à un niveau supérieur. Les exceptions contrôlées sont celles qu'on est obligé de déclarer dans l'en-tête de toute méthode susceptible de les lancer (annoncée avec `throws` dans l'en-tête).
  – Elle ne peuvent pas être ignorées (le compilateur génère une erreur dans ce cas).

Les exceptions non-contrôlées (unchecked) [du type RuntimeException ou Error], le programmeur a le choix de les traiter ou de les ignorer (sans erreur à la compilation).
  – Si une telle exception survient et qu'elle n'est traitée nulle part, elle sera alors propagée et "remontera" jusqu'à la méthode `main()` interrompant ainsi brutalement l'application.

5.3.1 Générer (lever) une exception (throw)

Les exceptions peuvent être générées soit :

1. **par le système, lors de l'exécution de certaines instructions**. Parmi les instructions qui génèrent des exceptions, on peut citer :
   – La division entière par zéro qui génère
   `ArithmeticException`
   – L'indexation d'un tableau hors de ses limites qui génère
   `ArrayIndexOutOfBoundsException`
   – L'utilisation d'un tableau ou objet dont la référence vaut `null` qui génère
   `NullPointerException`
   – La création d'un tableau avec une taille négative qui génère
   `NegativeArraySizeException`
   – La conversion d'un objet dans un type non compatible qui génère
   `ClassCastException`
   – ...

2. **par l'exécution de l'instruction** `throw` (dans les classes pré-définies de la plate-forme Java [librairies] ou dans le code que l'on écrit soi-même, voir section 5.3.4)

   Pour **générer explicitement une exception**, on utilise l'instruction `throw` :

   ```
   throw exceptionObject;
   ```

   – L'expression qui suit l'instruction `throw` doit être un objet qui représente une exception (un objet de type `Throwable`).
   – Lors de la création de l'objet exception, on peut généralement lui associer un message (`String`) qui décrit l'événement.

l'instruction `throw` **instancie un objet exception**, arrête l'exécution normale des instructions et sauf `try-catch` propage l'exception.

Exemple 5.1. *Une bonne manière de traiter l'invalidité des arguments de la construction d'un point consiste à « lancer » une exception :*

```
public class Point {
        static final int XMAX = 1024, YMAX = 768;
        int x, y;
        Point(int a, int b) throws Exception {
            if (a < 0 || a >= XMAX || b < 0 || b >= YMAX)
               throw new Exception("Coordonnées illégales");
               x = a;
               y = b;
        }
        ...
}
```

Quelques méthodes que l'on peut utiliser avec les objets exception

– `getMessage()` : retourne un `String` contenant le message (texte) associé à l'exception
– `toString()` : retourne un `String` contenant le nom de l'exception suivi du message associé
– `printStackTrace()` : affiche sur la console de sortie le nom de l'exception, le message associé ainsi que l'état de la pile des appels qui ont conduit au traitement de l'exception (Stack-Trace)
– `getStackTrace()` : retourne les informations de la Stack-Trace sous forme d'un tableau de `StackTraceElement`

5.3.2 Capturer et traiter une exception (try/catch/finally)

Les instructions susceptibles de lever des exceptions peuvent être insérées dans un bloc `try / catch` qui se présente de la manière suivante :

```
try {
// Bloc contenant des instructions pouvant générer des exceptions.
    }
catch (ExceptionType1 e1) {
        // Bloc contenant les instructions qui traitent
        // (capturent) les exceptions du type
        // ExceptionType1 (ou d'une de ses sous-classes)
        // On peut référencer l'objet exception à l'aide de
        // la variable e1.
    }
catch (ExceptionType2 e2) {
        // Bloc contenant les instructions qui traitent
        // (capturent) les exceptions du type
        // ExceptionType2 (ou d'une de ses sous-classes).
        // On peut référencer l'objet exception à l'aide de
        // la variable e2.
}
catch (...) {
        ...
        // Et ainsi de suite...
}
finally {
        // block facultatif contenant des instructions qui vont
        // être exécutées qu'il y ait ou non une exception.
}
```

Les blocs `catch` sont en nombre quelconque, le bloc `finally` est optionnel.

- Si l'exécution des instructions contenues dans le bloc `try` ne provoque le lancement d'aucune exception, les bouts de code contenus dans les blocs `catch` sont ignorés.
- Si une des instructions contenues dans le bloc `try`, lève une exception, le contrôle est passé au premier bloc `catch` dont le type d'exception correspond à l'exception qui a été levée (même classe ou classe parente de l'exception levée).
- Après l'exécution de la dernière instruction du bloc `catch` considéré, le contrôle est passé à l'instruction qui suit le dernier bloc `catch` (ou à la clause `finally` s'il y en a une).
- Si aucun bloc `catch` ne correspond au type d'exception qui a été levée, **l'exception est propagée** au niveau supérieur c'est-à-dire que le contrôle est transféré au traitement d'exception de la méthode invoquante ou du bloc englobant.
  - Si aucun traitement n'existe au niveau supérieur pour ce type d'exception, la propaga-

tion se poursuit jusqu'à trouver un bloc `catch` traitant cette exception.

– Si ce n'est pas le cas, le programme se termine avec un message d'erreur sur la console de sortie (`Stack Trace`).

L'ordre des clauses catch est important !

Les divers types d'exception peuvent appartenir à différentes **familles** (voir figure 5.1). Une exception spécialisée (par ex. `FileNotFoundException`) peut faire partie d'une famille plus vaste (`IOException`) qui elle-même fait partie d'une famille plus générale (`Exception`), etc.

Si plusieurs clause `catch` sont compatibles avec le type d'exception qui a été levée (font partie de la même famille), c'est la première qui capturera l'exception et se chargera du traitement.

```
try { ...
}
catch (FileNotFoundException notFound) {
... }
catch (IOException ioErr) { ...
}
catch (Exception genErr) {
... }
```

Lorsqu'un problème a été détecté, il y a différentes mesures que l'on peut envisager, selon les cas, pour régler la situation. Dans un traitement d'exception (bloc `catch`) on peut par exemple :

– Faire quelque chose d'autre à la place (algorithme de substitution).
– Sortir de l'application (`System.exit()`) après affichage d'un message.
– Retourner une valeur spéciale ou valeur par défaut (pour une fonction).
– ...

Un bloc `finally` est généralement utilisé pour effectuer des opérations de conclusion (fermeture de fichiers, de connexion réseau, de base de données, etc.) qui devraient être effectuées dans tous les cas de figure (l'utilisation des instructions `break`, `continue`, `return` ou `throw` (dans les blocs `try` ou `catch`) n'empêche pas l'exécution préalable du bloc `finally`).

5.3.3 Propagation des exceptions (throws)

Les méthodes d'un objet qui sont susceptibles d'envoyer une exception doivent le déclarer dans leur signature avec le mot clé `throws` (avec un `s` à la fin).

```
... uneMéthode(...) throws ClasseException {
```

C'est ce qui va permettre au compilateur de savoir que, lorsqu'on appel cette méthode, il faut nécessairement le faire dans un bloc `try`, et que la (ou les) exception(s) signalées par cette clause doivent être "catchés".

Donc une exception peut être **propagée** jusqu'à une méthode appelante qui la capture et la traite.

5.3.4 Créer de nouveaux types d'exceptions

Pour créer ses propres types d'exceptions, il suffit de dériver (spécialiser) une classe de type `Throwable` (choisir une des classes existantes proche de celle que l'on souhaite créer ou, sinon, dériver la classe générale `Exception`).

- Généralement, dans cette sous-classe, on crée uniquement deux constructeurs : un constructeur sans paramètre et un constructeur qui prend un message (`String`) en paramètre.
- On crée rarement de nouveaux champs ou de nouvelles méthodes.

Exemple 5.2.

```
public class ExceptionCoordonnesIllegales extends Exception {
         public ExceptionCoordonnesIllegales() {
                super("Coordonnées illégales"); }
        }

public class Point {
  public Point(int a, int b) throws ExceptionCoordonnesIllegales {
    if (a < 0 || a >= XMAX || b < 0 || b >= YMAX)
        throw new ExceptionCoordonnesIllegales();
        x = a;
        y = b;
        }
        ...
  public static void main(String[] args) {
     ...
     try {
       Point p = new Point(u, v);
    }
     catch (ExceptionCoordonnesIllegales e) {
        System.out.println("Problème avec les coordonnées du point"); }
        ...
   }
}
```

5.4 Les exceptions par l'exemple

ERREUR DE DIVISION

```
public class Equation{
    private int a, b;
    //Constructeur
    public Equation(int a, int b) {
       this.a=a;
      this.b=b;
    }
    public void afficher() {
        System.out.println(a+"*X="+b);
    }
    public int solution() {
        return b/a;
    }

    public static void main(String args[]) {
      int valeurA=Integer.valueOf(args[0]).intValue();
      int valeurB=Integer.valueOf(args[1]).intValue();
      Equation equa = new Equation(valeurA,valeurB);
      equa.afficher();
      int x = equa.solution();
      System.out.println("résultat : X = "+x);
      }
}
```

L'instruction division entière peut lever une exception : `ArithmeticException` (erreur d'exécution). **Une exception levée (non propagée et non capturée) finit par provoquer l'arrêt de la "méthode du programme principal"**.

5.4.1 Capturer et traiter l'exception

```
public class Equation {
   private int a, b;
   ...
   int solution() {
        int x;
        try {
          //instruction susceptible de lever une exception
          x = b/a;
        }
        catch (ArithmeticException e) {
          //capture et traitement de l'exception
          x = -1; }
```

```
        return x;
    }
      public static void main(String args[]) {
        int valeurA=Integer.valueOf(args[0]).intValue();
        int valeurB=Integer.valueOf(args[1]).intValue();
        Equation equa = new Equation(valeurA,valeurB);
        equa.afficher();
        int x = equa.solution();
        System.out.println("résultat : X = "+x);
     }
}
```

L'instruction "`try ... catch`" permet de capturer des exceptions :

- dès qu'une exception est levée dans le corps de `try`, le traitement de ce corps est terminé
- `catch` définit le traitement pour les `ArithmeticException` capturées.
- l'exécution continue normalement en reprenant après le bloc `try-catch`.
- Ce mécanisme permet de traiter les erreurs et d'empêcher qu'elle n'arrête l'application en cours.

L'exception a été capturée et traitée : `x =-1`.

5.4.2 Propager l'exception

```
public class Equation {
    private int a, b;
    ...
    // throws indique que la méthode est susceptible
    // de lever une exception de type ArithmeticException
    int solution() throws ArithmeticException {
        return b/a;
    }

    public static void main(String args[]) {
        int valeurA=Integer.valueOf(args[0]).intValue();
        int valeurB=Integer.valueOf(args[1]).intValue();
        Equation equa = new Equation(valeurA,valeurB);
        equa.afficher();
        try {
          int x = equa.solution();
          System.out.println("résultat : X = "+x);
        }
        catch (ArithmeticException e) {
          System.out.println("pas de solution");
        }
     }
```

```
}
```

La déclaration `throws` permet d'indiquer que la méthode est susceptible de lever une exception : ici `ArithmeticException`, qu'elle ne capture pas par un `try- catch`

- `ArithmeticException` sera propagée/transmise à la méthode appelante si elle est levée.
- Donc il faut que les méthodes qui appellent la méthode solution mettent éventuellement en place un mécanisme de capture `try-cath` ou qu'elles propagent elles aussi l'exception.

5.4.3 Lever une exception

```
public class Equation {
  private int a, b;
  ...
  int solution() throws ArithmeticException {
  if (a==0)
      // l'instruction throw permet de générer/lever une exception
      throw new ArithmeticException("division entière par zéro");
  else
      return b/a;
  }
  public static void main(String args[]) {
    int valeurA=Integer.valueOf(args[0]).intValue();
    int valeurB=Integer.valueOf(args[1]).intValue();
    Equation equa = new Equation(valeurA,valeurB);
    equa.afficher();
    try {
        int x = equa.solution();
        System.out.println("résultat : X = "+x);
    }
    catch (ArithmeticException e) {
        System.out.println(e.getMessage());
    }
} }
```

L'instruction `throw` (sans s) permet de générer/lever une exception, ici `ArithmeticException`.

- De plus, un message d'erreur est ajouté dans les informations véhiculées par l'exception.
- À la capture d'exception, il est possible d'afficher le message associé en utilisant la méthode `getMessage` sur l'objet exception (`getMessage` est une des méthodes de l'objet `Exception` qui donne des informations sur l'exception).

Les collections

6.1 Qu'est ce qu'une Collection

Les collections en Java sont des classes permettant de manipuler un regroupement d'objets (ses éléments) : listes, ensembles, etc.

Dans les premières versions de Java, les collections étaient représentées par les `Vector` (tableaux dynamiques), `Hashtable` (tables associatives), etc. Puis avec Java 1.2, est apparu le *framework* pour la gestion de collections (dans le package `java.util`). Il a apporté des modifications dans la manière avec laquelle ces collections ont été réalisées et hiérarchisées.

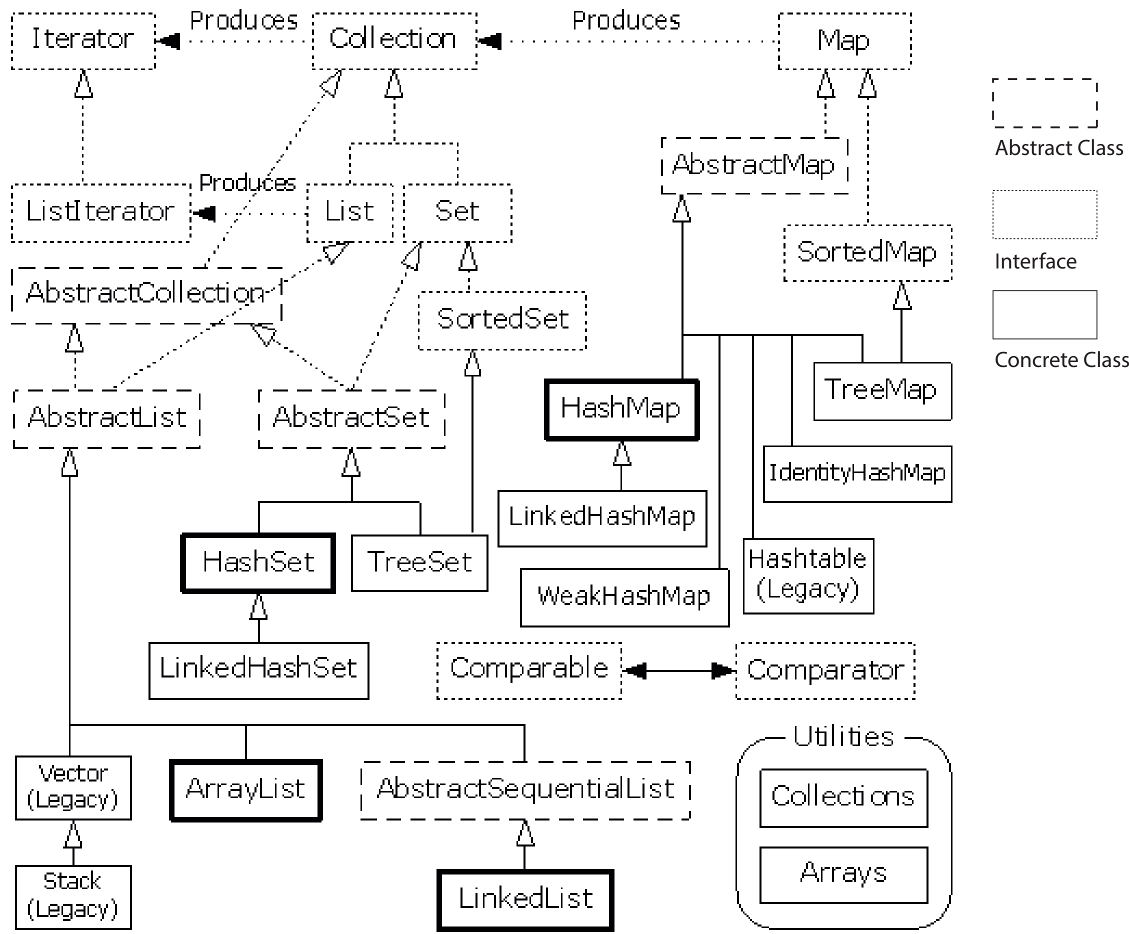


Figure 6.1: Taxonomy of interfaces, abstract classes, and concrete classes in the Java Collections Framework.

6.2 Les interfaces de Collection

Les interfaces en relation directe avec les collections (au nombre de 6) sont regroupées dans deux arborescences (cf. figure 6.2) :

1. **Map** : collections indexées par des clés (eg. Entrées d'un dictionnaire)
2. **Collection** : groupe d'objets, connu par ses éléments.

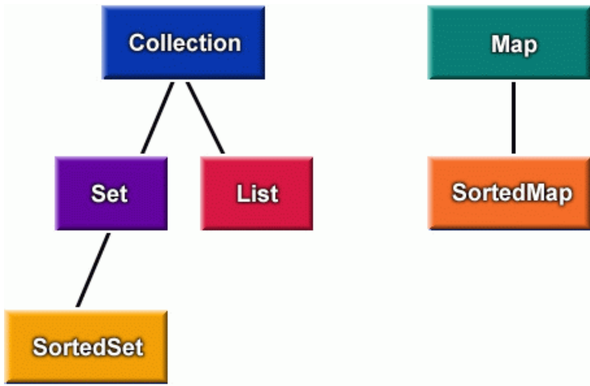


Figure 6.2: Les interfaces de Collection

Toutes les collections en JAVA implémentent l'interface `Collection` par le biais de sous interfaces comme `Set`, `Map` ou `List` :

1. `Collection` : un groupe d'objets où la duplication peut-être autorisée.
2. `Set` : un ensemble ne contenant que des valeurs et ces valeurs ne sont pas dupliquées.
3. `SortedSet` est un `Set` trié.
4. `List` : hérite de collection, mais autorise la duplication. Dans cette interface, un ensemble ordonné d'éléments est accessibles par l'indexage ou l'itération.
5. `Map` : organise un ensemble de couples clé/valeur. Chaque clé va permettre de retrouver sa valeur associée.
6. `SortedMap` est un `Map` trié.

Remarque

À partir de java 1.5, les collections sont typées. `Collection<E>` où `E` représente le type des éléments de la collection.

Le type de données que l'on souhaite placer dans la structure est défini lors de la création de cette dernière. Par exemple, pour une liste de `String`, la création se fera par exemple comme suit :

```
List<String> liste = new LinkedList<String>();
```

Si l'on veut stocker des éléments de types différents, on utilise le super-type commun, voir `Object`

6.3 Interface Collection

L'interface `Collection<E>` définit la notion de collection d'objets d'une façon assez générale. À ce niveau d'abstraction, les opérations garanties sont : obtenir le nombre d'éléments de la collection, savoir si un objet donné s'y trouve, ajouter (sans contrainte sur la position) et enlever un objet, etc.

Elle est la super interface de plusieurs interfaces du framework. Plusieurs classes qui gèrent une collection implémentent une interface qui hérite de l'interface `Collection`.

6.3.1 Méthodes principales de Collection<E>

- `boolean add(E e)` : Ensures that this collection contains the specified element (optional operation).
- `boolean contains(Object o)` Returns true if this collection contains the specified element,
- `boolean isEmpty()` Returns true if this collection contains no elements.
- `Iterator<E> iterator()` Returns an iterator over the elements in this collection.
- `boolean remove(Object o)` Removes a single instance of the specified element from this collection, if it is present (optional operation).
- `int size()` Returns the number of elements in this collection.
- `addAll`, `removeAll`, `toArray`, etc.

6.4 Les listes

L'interface `List<E>` hérite de `Collection`, mais autorise la duplication des éléments. Dans cette interface, un système d'indexation a été introduit pour permettre l'accès (rapide) aux

éléments de la liste.

- `List<E>` est une collection ordonnée (groupe d'objets repérés par des numéros (rang), allant de de 0 à `size()-1`), de taille non bornée.
- Elle permet de travailler sur des sous listes.
- En plus des méthodes de `Collection`, définie des méthodes travaillant sur les index
  - `add(int index, E element)` ajout de l'élément à l'index-ième position
  - `E get(int index)` fournit l'index-ième élément de la liste.
    `IndexOutOfBoundsException` si (`index < 0 || index >= size()`)
  - `E remove(int index)` supprime l'index-ième élément de la liste. (même exception)
  - `int indexOf(Object element)` indice de la première occurrence `element` dans la liste, -1 si absent
  - `ListIterator<E>` itérateur pour listes doublement chaînées

Deux implémentations possibles :

1. `LinkedList<E>` (liste doubelement chaînée) : Une implémentation complète de l'interface `List<E>`
2. `ArrayList<E>` : la même chose qu'un `Vector`, sauf que ce n'est pas synchronisé (un `Vector` est une liste implémentée par un tableau à taille variable. De plus, un objet `Vector` est synchronisé [1])

Règle générale

On utilise le plus souvent `ArrayList<E>` (si ajout et accès "direct" (indicé)) mais `LinkedList<E>` est utile si il y a beaucoup d'opérations d'insertions/suppressions pour éviter ainsi les décalages.

Test :

```
.../java/test$ java TestCollection2 20000 20000
*** insertion en tete LinkedList
20000 insertions ds LinkedList : 16 ms
*** insertion en tete ArrayList
20000 insertions dans ArrayList : 403 ms
*** remove LinkedList
20000 suppressions dans LinkedList : 8 ms
*** remove ArrayList
20000 suppressions dans ArrayList : 398 ms
```

En cas de "non obligation" (ou de doute) sur le choix : utiliser l'upcast vers l'interface associée à la collection pour faciliter le changement de choix d'implémentation

1. des dispositions sont prises pour interdire que deux threads accèdent de manière concurrente à un même `Vector`

```
List<Livre> aList = new ArrayList<Livre>();
//traitements avec uniquement des méthodes de l'interface List
```

si besoin ultérieurement on peut changer en :

```
List<Livre> aList = new LinkedList<Livre>();
//mêmes traitements sans autre changement
```

6.5 Les itérateurs

Pour parcourir les éléments d'une collection on utilise un itérateur. L'API `Java` définit une interface `java.util.Iterator<E>` (extraits) :

- `boolean hasNext()` Returns true if the iteration has more elements.
- `E next()` Returns the next element in the iteration.
- `void remove()` Removes from the underlying collection the last element returned by the iterator (optional operation).
- `ListIterator<E>` parcours avant/arrière (`previous()`, `hasPrevious()`)
- + `add(E e)`, `set(E e)`

Exemple 6.1.

```
Collection<Recyclable> trashcan = new ArrayList<Recyclable>();
trashcan.add(new Paper()); // upcast vers Recyclable
trashcan.add(new Battery()); // implicite
// itérateur sur la collection
Iterator<Recyclable> it = trashcan.iterator();
while(it.hasNext()) {
  Recyclable ref = it.next(); // it.next() du type Recyclable
  ref.recycle();
}
```

Les `Iterator` sont **fail-fast** (échec rapide) : si, après que l'itérateur ait été créé, la collection attachée est modifiée autrement que par un `remove` (ou `add` pour `ListeIterator`) de l'itérateur alors l'itérateur lance une `ConcurrentModificationException`.

```
List<Livre> l = ...;
for(int i = 0 ; i < 5; i++) {
   l.add(new Livre(...));
}
Iterator<Livre> itLivre = l.iterator();
Livre l = itLivre.next(); // ok
l.add(new Livre(...)); // modification de la liste
                        // => corruption de l'itérateur
l = itLivre.next(); // -> ConcurrentModificationException levée
```

Attention

Il ne faut pas parcourir une liste en utilisant get(int idx). Il faut utiliser les itérateurs.

Pourquoi ne faut-il pas écrire :

```
List<...> l = ...;
for(int i = 0; i < l.size(); i ++) {
 //utilisation de l.get(i)
}
```

```
.../java/test$ java TestCollection 20000
*** parcours LinkedList avec itérateur
pacours 20000 éléments : 7 ms
*** parcours LinkedList avec get(i)
parcours 20000 éléments : 480 ms
```

Possibilité d'utiliser la syntaxe "à la for-each" pour itérer sur les collections :

```
for(Reclyclable r : trashcan){
 r.recycle();
}
```

Remarque 6.1. *Cette syntaxe est possible sur les tableaux et toutes les classes qui implémentent l'interface* `Iterable<T>`.

Important !

Les collections ne peuvent contenir que des objets (et donc pas de valeurs primitives) :

`List<int>` n'est pas possible, il faut utiliser `List<Integer>`.

Depuis java 1.5, existe l'autoboxing ce qui signifie que les conversions type primitif → classe associée sont gérées par le compilateur. Ainsi on peut écrire :

```
List<Integer> l = new ArrayList<Integer>();
l.add(12); //correspond à l.add(new Integer(12));
int i = l.get(0); //correspond à int i = l.get(0).intValue();
```

6.6 Les ensembles

Un `Set<E>` est une collection sans répétition : un `Set<E>` ne peut pas contenir deux éléments `e1` et `e2` vérifiant `e1.equals(e2)`.

L'interface `Set<E>` contient uniquement les méthodes héritées de `Collection` (Pas de méthode supplémentaire par rapport à `Collection`).

Aucune indication n'est donnée a priori sur l'ordre dans lequel les éléments d'un `Set<E>` sont visités lors d'un parcours de l'ensemble.

Deux implémentations possibles :

1. `HashSet<E>` : les éléments sont rangés suivant une méthode de hachage (sans ordre de tri particulier), ce qui garantit un temps constant pour les opérations de base (`set`, `size`, `add`, `remove`).
2. `TreeSet<E>` : cette classe est un arbre binaire de recherche qui représente un ensemble trié d'éléments (les éléments sont rangés de manière ascendante).

Exemple 6.2.

```
import java.util.*;
public class SetExample {
   public static void main(String args[]) {
         Set<String> set = new HashSet<String>(); // Une table de Hachage
         set.add("Banane");
         set.add("Eclair");
         set.add("Grenade");
         set.add("Eclair");
         set.add("Champignon");
         System.out.println(set);
         Set SetTrie = new TreeSet(set); // Un Set trié
         System.out.println(sortedSet);
   }
}
[Grenade, Champignon, Banane, Eclair]
[Banane, Champignon, Eclair, Grenade]
```

6.7 Map<K,V>

Dans la bibliothèque standard Java, les tables associatives sont représentées par l'interface `java.util.Map`. Une **table associative** ou dictionnaire (map, dictionary) est une structure de données associant des **valeurs** à des **clefs**.

Cette interface n'hérite ni de `Set` ni de `Collection`. La raison est que `Collection` traite des données simples alors que `Map` des données composées (clé,valeur). **L'unicité des clés** est garantie : un objet `Map` ne peut pas contenir deux associations `(c1, v1)` et `(c2, v2)` telles que `c1.equals(c2)`.

Exemples de tables associatives :

Les opérations fondamentales sont l'ajout et la suppression d'une association et la recherche d'une association à partir de la valeur d'une clé.

- permet un accès rapide à la valeur à partir de la clé (comme un tableau)
- permet l'insertion rapide (comme dans une liste)
- accepte `null` comme clé ou valeur

Table	Clefs	Valeurs
annuaire	commune, nom	n° de téléphone
dictionnaire	mot	définition
index	mot	liste de pages
programme TV	chaîne, date	émission

- ne permet pas les accès concurrents
- teste si un objet existe ou non par rapport à la méthode equals() de l'objet

Deux implémentations possibles :

On retrouve principalement deux classes qui implémentent l'interface Map<K,V> : HashMap<K,V> et TreeMap<K,V>.

1. HashMap<K,V> : table de hachage (table d'adressage dispersé ou hashcode) avec garantie d'un accès en temps constant.
2. TreeMap<K,V> : arbre ordonné [2] suivant les valeurs des clés, i.e., clés triées (accès en log(n)).

6.7.1 Méthodes principales de Map<K,V>

- V get(K key) récupère la valeur associée à une clé
- void put(K key, V value) ajoute un couple (clé, valeur)
- V remove(Object key) supprime le couple associé à la clé key
- boolean containsKey(Object key) teste l'existence d'une clé (equals)
- boolean containsValue(Object value) teste l'existence d'une valeur (equals)
- Collection<V> values() renvoie la collection des valeurs
- Set<K> keySet() renvoie l'ensemble des clés
- Set<Map.Entry<K,V>> entrySet() renvoie l'ensemble des couples (clé,valeurs) (objets Map.Entry<K,V>)

Attention : pas d'itérateur

Les Maps n'implémentant pas l'interface Iterable, ils ne permettent pas d'être parcourus à l'aide d'un itérateur. Toutefois, il sera possible de le faire en convertissant chaque Map, à l'aide de la méthode values, en une Collection équivalente (qui, elle, implémente l'interface Iterable) .

2. ordre naturel ou une instance de Comparator

6.8 Les méthodes de la classe Collections

La classe `Collections` (**notez le pluriel !**) définit des méthodes statiques qui seront utilisées pour réaliser différents traitements sur les collections. Certaines de ces méthodes ne s'appliquent que sur des listes ; c'est le cas de :

- `sort` : trie les éléments de la liste dans l'ordre naturel ; il est possible de spécifier un objet `Comparator` permettant "d'expliquer" à `sort` comment trier des éléments ;
- `shuffle` : ordonne de manière aléatoire les éléments de la liste ;
- `reverse` : inverse l'ordre des éléments de la liste ;
- `fill` : réinitalise la liste avec une valeur définie ;
- `copy` : copie une liste dans une autre.

Remarquons que le tri d'éléments est naturel pour des valeurs numériques mais ne l'est pas forcément pour des objets plus complexes. En effet, la fonction de tri doit savoir comment comparer deux objets en vue de les classer. Dans ce but, on souhaitera implémenter l'interface `Comparable<T>` définissant une méthode `compareTo`. Cette dernière retournera 1, 0 ou 1 suivant que l'argument passé à la méthode `compareTo` est "plus petit", "équivalent" ou "plus grand", au sens de l'ordonnancement que l'on souhaite introduire.

6.9 Généricité

Il est parfois nécessaire de gérer des classes ou des méthodes pour lesquelles on ne connaît pas à l'avance le type traité. On peut alors spécifier un type indéfini dans la déclaration de la classe, comme dans l'exemple suivant :

```
public class Pile<E>
```

Le type `E` peut ensuite être utilisé au sein de la classe qui est appelée classe générique.

Il est également possible de définir des **méthodes génériques** ; on peut alors soit utiliser le caractère ? ou faire précéder le type de retour de la méthode par le type générique :

```
public static void method(? el)
public static <E> void method(E el)
```

On peut également obliger le type `E` à être une sous-classe d'une classe déterminée ou à implémenter une interface déterminée :

```
public static <E extends Pile> void method(E el)
```

6.9.1 Pourquoi la généricité ?

Exemple : Soit la classe suivante :

```
public class Paire {
   Object premier ;
   Object second;
   public Paire (Object a, Object b){
      premier= a; second = b;
   }

   public object getPremier(){
     return premier;
   }

   public object getSecond(){
      return second;
   }
```

Les deux inconvénients sont :

```
Paire  p = new Paire ("abc", "xyz");
String x =  (String)p.getPremier(); // le casting est obligatoire
Double y =  (Double)p.getSecond();
        // Il faut attendre l'exécution pour avoir
        // une levée d'exception(ClassCastException)
```

On définit alors une classe paramétrée :

```
public class Paire<T> {
   T premier ;
   T second;
   public Paire (T a, T b){
      premier a; second = b;
   }
   public T getPremier(){
      return  premier;
   }

   public getSecond(){
      return second;
   }
}
```

Le programme est alors plus simple et plus sur :

```
  Paire<String>  p = new Paire<String>  ("abc", "xyz");
  String x =  p.getPremier(); // pas de cast
  Double y =  p.getSecond();  // erreur de compilation (type mismatch)
```

Supposons que nous voulions ajouter à la classe `Paire<T>` la méthode suivante :

```
public T min(){
   if(premier.compareTo(second)<=0) return premier;
```

```
    else return second;
}
```

Le compilateur signale alors que la méthode `compareTo` n'est pas définie pour le type `T`. Il faut restreindre `T` à une classe qui a cette méthode, et définir la classe `Paire` de la façon suivante :

```
public class Paire<T extends Comparable> {
    T premier ;
    T second;
    public Paire (T a, T b){
        premier a; second = b;
    }
    public T getPremier(){
        return  premier;
    }

    public getSecond(){
        return second;
    }

    public T min(){
        if(premier.compareTo(second)<=0) return premier;
        else return second;
    }
}
```

Exemple : Créer un type générique :

```
public class Value<T> {
        private T v;
        public Value (T v){
        this.v = v;
        }
        public T getValue() {
            return this.v;
        }
}// Value
```

Usage :

```
Value<Integer> v1 = new Value<Integer>(12);
Value<List<Chou>> v2 = new Value<List<Chou>>(new ArrayList<Chou>());
Integer i = v1.getValue();
```

Exemple d'une classe à plusieurs variables de type :

Écrire une classe générique `TripletH` permettant de manipuler des triplets d'objets pouvant être chacun d'un type différent.

```java
class TripletH<T, U, V> {
        private T x;
        private U y;
        private V z; // les trois éléments du triplet

        public TripletH(T premier, U second, V troisieme) {
                x = premier;
                y = second;
                z = troisieme;
        }

        public T getPremier() {
                return x;
        }

        public U getSecond() {
                return y;
        }

        public V getTroisieme() {
                return z;
        }

        public void affiche() {
          System.out.println("premiere valeur : " + x +
          " - deuxieme valeur : " + y + " - troisieme valeur : " + z);
        }
}
```

Et en voici un petit programme d'utilisation :

```java
public class TstTripletH {
  public static void main(String args[]) {
      Integer oi = 3;
      Double od = 5.25;
      String os = "hello";
      TripletH<Integer, Double, String> tids =
          new TripletH<Integer, Double, String>(oi, od, os);
      tids.affiche();
      Integer n = tids.getPremier();
      System.out.println("premier element du triplet ti = " + n);
      Double d = tids.getSecond();
      System.out.println("second element du triplet td = " + d);
        }
}
```

Instanciation avec joker

Problème : on voudrait qu'une méthode écrite pour un type générique T puisse fonctionner avec toute instance des sous-classes de T

Solution : on peut spécifier qu'un paramètre de type est toute sous-classe ou sur-classe d'une classe donnée

- <?> désigne un type inconnu
- <? extends C> désigne un type inconnu qui est soit C soit un sous-type de C
- <? super C> désigne un type inconnu qui est soit C soit un sur-type de C

C peut être une classe, une interface ou un paramètre de type