Programmation orientée-objet

Langage Java

Christina Boura, Stéphane Lopes

christina.boura@prism.uvsq.fr, stephane.lopes@prism.uvsq.fr

2014–2015

Université de Versailles St-Quentin-en-Yvelines
UFR des Sciences

Table des matières

Table des figures

Listings

Liste des exercices

Chapitre 1

Préambule

Sommaire

1.1 Objectifs et prérequis

Objectifs du cours

Se focaliser sur les concepts de la POO
Aborder la programmation objet en se focalisant sur les concepts (définition, application) et non pas sur la syntaxe spécifique à un langage.

- Compréhension des concepts objets
- Mise en œuvre de ces concepts avec le langage Java
- Montrer l'importance des *API* (*Application Programming Interfaces*) et des bibliothèques

1

Prérequis

- Notions de base en algorithmique
- Connaissance de base d'un langage impératif (C, . . .)

2

1.2 Plan

Plan général

- Introduction
- Vue d'ensemble des concepts objets
- Classe
- Héritage
- Module

3

Chapitre 2

Introduction

Sommaire

2.1 Caractéristiques des langages de programmation

Langage compilé vs. interprété

— Avec un *langage compilé*, le code source du programme est transformé en code machine par le compilateur
— Dans un *langage interprété*, le code source du programme est exécuté « à la volée » par l'interpréteur
— Certains langages sont à la fois compilés et interprétés

4

Langage impératif vs. déclaratif

Langage impératif

— Un *langage impératif* représente un programme comme une séquence d'instructions qui modifient son état au cours de son exécution
— Un programme décrit **comment** aboutir à la solution du problème
— Proche de l'architecture matérielle des ordinateurs (*architecture de von Neumann*)

Langage déclaratif

— Un *langage déclaratif* permet de décrire ce que le programme doit faire (le **quoi**) et non pas comment il doit le faire (le **comment**)
— Un programme respectant ce style décrit le problème à traiter.

5

Système de typage

— Un *système de typage* attribue des types aux éléments du langage
— Le typage est

explicite si les types apparaissent dans le code source

implicite si les types sont déterminés par le compilateur (*inférence de type*)

fort si les manipulations entre données de types différents sont interdites

faible si les possibilités de « trans-typage » sont nombreuses

statique si la vérification des types est réalisée à la compilation

dynamique si la vérification se fait à l'exécution

6

Support des paradigmes de programmation

Paradigme

Un paradigme de programmation représente la façon d'aborder un problème et d'en concevoir la solution.

— Programmation impérative
  — Programmation structurée
  — Programmation modulaire
  — Programmation par abstraction de données
  — Programmation objet
— Programmation fonctionnelle
— Programmation logique
— ...

Support d'un paradigme

Un langage supporte un paradigme quand il fournit les fonctionnalités pour utiliser ce style (de façon simple, sécurisée et efficace)

7

Exemple

Programmation logique avec Prolog

— Prolog permet de définir et d'interroger une *base de faits*
— Prolog est un langage déclaratif
— Un *fait* est une assertion simple (*Idéfix est un chien.*)
— Une *règle* décrit une inférence à partir des faits (*Les chiens aiment les arbres*)
— Une *requête* est une question sur la *base de connaissance* (*Idéfix aime-t'il les arbres ?*)

8

Exemple - Solveur de Sudoku 4×4 en Prolog

Requête

```
- requête

| ?- sudoku([_, _, 2, 3,
             _, _, _, _,
             _, _, _, _,
             3, 4, _, _],
             Solution).
```

Listing 2.1 – Requête en Prolog (Sudoku)

9

Exemple - Solveur de Sudoku 4×4 en Prolog

Résolution (partie 1)

```
- la solution doit être unifiée avec le problème
- le problème comporte 16 chiffres
- chaque chiffre est compris entre 1 et 4 (fd_domain)

sudoku(Puzzle, Solution) :-
  Solution = Puzzle,
  Puzzle = [A1, A2, A3, A4,
            B1, B2, B3, B4,
            C1, C2, C3, C4,
            D1, D2, D3, D4],
  fd_domain(Puzzle, 1, 4),
```

Listing 2.2 – Résolution du Sudoku (partie 1)

10

Exemple - Solveur de Sudoku 4×4 en Prolog

Résolution (partie 2)

```
- les blocs (lignes, colonnes et carrés) sont définis

  Row1 = [A1, A2, A3, A4],
  Row2 = [B1, B2, B3, B4],
  Row3 = [C1, C2, C3, C4],
  Row4 = [D1, D2, D3, D4],

  Col1 = [A1, B1, C1, D1],
  Col2 = [A2, B2, C2, D2],
  Col3 = [A3, B3, C3, D3],
  Col4 = [A4, B4, C4, D4],

  Square1 = [A1, A2, B1, B2],
  Square2 = [A3, A4, B3, B4],
  Square3 = [C1, C2, D1, D2],
  Square4 = [C3, C4, D3, D4],
```

Listing 2.3 – Résolution du Sudoku (partie 2)

11

Exemple - Solveur de Sudoku 4×4 en Prolog

Résolution (partie 3)

```
- le prédicat valid reçoit une liste de 12 blocs
- la liste vide est valide
- la tête de la liste ne comporte pas de doublons (fd_all_different)
- le reste de la liste doit être valide

valid([]).
valid([Head|Tail]) :-
  fd_all_different(Head),
  valid(Tail).
- une solution possède des blocs valides

  valid([Row1, Row2, Row3, Row4,
         Col1, Col2, Col3, Col4,
         Square1, Square2, Square3, Square4]).
```

Listing 2.4 – Résolution du Sudoku (partie 3)

12

Exemple

Programmation fonctionnelle avec Haskell

- Haskell est un langage fonctionnel
- Possède un système de typage statique, fort et principalement implicite (inférence de types)

13

Exemple - Haskell (partie 1)

```
-- Calcul de la fonction factorielle

-- Récursive
fact x = if x == 0 then 1 else fact (x - 1) * x

-- Pattern matching
fact 0 = 1
fact x = x * fact (x - 1)

-- Gardes
fact x
   | x > 1 = x * fact (x - 1)
   | otherwise = 1

-- Liste et intervalle
fac x = product [1..x]
```

Listing 2.5 – Factoriel avec Haskell

14

Exemple - Haskell (partie 2)

```haskell
-- Fonctions d'ordre supérieure
mapList f [] = []
mapList f (x:xs) = f x : mapList f xs

-- Listes en compréhension et évaluation paresseuse
take 10 [ (i,j) | i <- [1..], j <- [1..], i < j ]
```

Listing 2.6 – Exemples en Haskell

15

2.2 Évolution vers les objets

Motivations

- Un logiciel est difficile à développer, à modifier et à maintenir
- La plupart des logiciels sont livrés en retard et dépassent leur budget
- Les programmeurs « réinventent souvent la roue » car il y a peu de *réutilisation* de code

⇒ *Nécessité de trouver une approche plus efficace*

16

2.2.1 Évolution des paradigmes impératifs

Programmation structurée

Definition 1. *Choisissez les procédures. Utiliser les meilleurs algorithmes que vous pourrez trouver.*

Caractéristiques

- L'accent est mis sur les *traitements*
  - recherche des meilleurs algorithmes
- Approche très utilisée depuis de nombreuses années
- Approche de haut en bas (*top-down*)
  - un programme est décomposé en sous-programmes jusqu'à ne plus pouvoir décomposer
- A beaucoup amélioré la qualité du logiciel
  - permet une approche plus méthodique : il est naturel de décomposer un problème de cette façon

Limitations

Les données et les traitements restent indépendants.

- c'est un gros problème car on constate que les données changent moins que les traitements

⇒ maintenance plus difficile

- difficulté à assurer la cohérence entre les structures de données et les traitements qui les utilisent
- or la maintenance représente une grosse part du coût d'un logiciel

17

Programmation modulaire

Definition 2. *Choisissez vos modules. Découpez le programme de telle sorte que les données soient masquées par ces modules.*

Caractéristiques

- Un module est un ensemble de procédures connexes avec les données qu'elles manipulent
- L'élément primordial passe de la conception des procédures à l'organisation des données
- Principe de *masquage de l'information*
- Permet la *compilation séparée*

Limitations

Les types ainsi définis diffèrent dans leur utilisation (création, ...) des types de base.

par exemple, l'initialisation d'une variable d'un type défini par l'utilisateur nécessite l'appel d'une fonction particulière du type (laissé à la charge du programmeur)

18

Programmation par abstraction de données

Definition 3. *Choisissez les types dont vous avez besoin. Fournissez un ensemble complet d'opérations pour chaque type.*

Caractéristiques
- Utilise les *types abstraits de données* (TAD)
- L'*interface* d'un type abstrait isole complètement l'utilisateur des détails d'implémentation
  - par opposition à type concret (l'implémentation est protégée mais visible)

Limitations
Il est nécessaire de modifier un type pour l'adapter.

19

Programmation orientée objet

Definition 4. *Choisissez vos classes. Fournissez un ensemble complet d'opérations pour chaque classe. Rendez toute similitude explicite à l'aide de l'héritage.*

Caractéristiques
- Consiste à définir les classes puis à préciser les relations entre elles (notamment l'*héritage*)
  - définir des classes = définir des types utilisateurs
  - factoriser des comportements en créant des hiérarchies d'héritage
  - permet d'adapter un type existant sans le modifier

20

2.2.2 Difficultés de l'approche objet

Approche objet vs. approche structurée
- *L'approche objet est moins intuitive*
  - décomposer un problème en une hiérarchie de fonctions atomiques et de données est plus naturel
- *La modélisation objet est difficile*
  - Rien dans les concepts de base de l'approche objet ne dicte comment modéliser la structure objet d'un système de manière pertinente.
  - Comment mener une analyse qui respecte les concepts objet ?
  - Sans un cadre méthodologique approprié, la dérive fonctionnelle de la conception est inévitable.
- *L'application des concepts objet nécessite de la rigueur*
  - Le vocabulaire précis est un facteur d'incompréhensions.

21

Approche objet vs. langage de programmation
- Beaucoup de développeurs ne pensent objet qu'à travers un langage de programmation
- Les langages ne sont que des outils implémentant certains concepts objet d'une certaine façon
- Les langages ne garantissent en rien l'utilisation adéquat de ces moyens techniques

Programmer en Java ou en C# n'est pas concevoir objet !
- Seule une analyse objet conduit à une solution objet, i.e. qui respecte les concepts de base de l'approche objet.
- Le langage de programmation est un moyen d'implémentation
- Il ne garantit pas le respect des concepts objet

22

Conception objet

Definition 5. Concevoir objet, c'est d'abord concevoir un *modèle* qui respecte les concepts objet.

- Pour penser et concevoir objet, il faut savoir « prendre de la hauteur », jongler avec des concepts abstraits, indépendants des langages d'implémentation et des contraintes purement techniques
- Les langages de programmation ne sont pas un support adéquat pour cela
- Pour conduire une analyse objet cohérente, il ne faut pas directement penser en terme de pointeurs, d'attributs et de tableaux, mais en terme d'association, de propriétés et de cardinalités

23

2.2.3 Quelques langages orientés-objets

Quelques langages objets

- Simula, Ole-Johan Dahl et Kristen Nygaard (années 1960)
  - langage de simulation
  - les objets simulent les entités du monde réel
  - son développement débute en 1961, Simula I est terminé en 1965 (langage de simulation), Simula 67 en 1967 (langage plus généraliste, concepts objets)
- SmallTalk, Alan Kay et Dan Ingalls (début des années 1970)
  - Entièrement basé sur les notions d'objet et de message
  - A l'origine, conçu pour être utilisé par des enfants
  - Début des interfaces graphiques au Xerox Parc
  - Le plus «pur» des langages objets
- Eiffel, Bertrand Meyer (1985)
  - Langage OO pur
  - Conçu pour la sécurité du logiciel
  - Implémente la persistance, les pré et post-conditions, ...
- C++, Bjarne Stroustrup (années 1980, Standard ISO en 1998)
  - Extension du C (classes, template, STL, ...)
  - ARM C++ en 1989
  - Standard ISO en 1998
- Java, James Gosling (1995)
  - Portabilité
  - Intégration avec le web (applet)
- C#, Anders Hejlsberg (2001)

24

2.3 La technologie Java

2.3.1 Généralités

Quelques chiffres

(source : SUN en 2008)

- Six millions de développeurs Java
- 100 milliards (USD) en vente annuelle et 110 milliards (USD) de dépenses liées
- Java équipe :
  - 2,5 milliards de cartes à puce
  - 800 millions de PC
  - 1,85 milliards de téléphones
- 180 opérateurs télécom déploient les technologies Java

25

2.3.2 Les plateformes Java

Les plateformes Java

- Java Platform Standard Edition (Java SE ou J2SE)
  - dédiée aux postes clients et aux stations de travail
- Java Platform Enterprise Edition (Java EE)
  - dédiée aux applications d'entreprises (serveur, postes clients, ...)
- Java Platform Micro Edition (Java ME)
  - dédiée aux systèmes embarqués (téléphone portable, PDA, ...)

(voir figure 2.1).

26

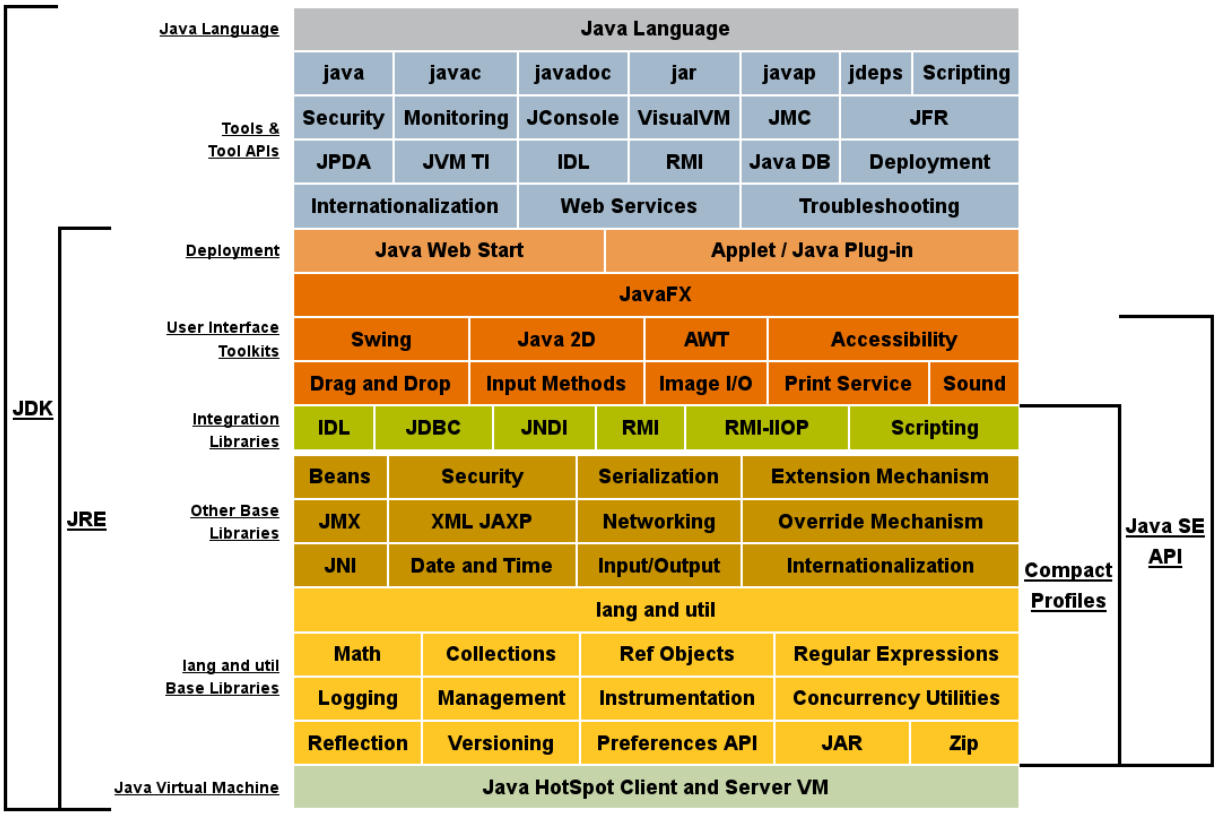


FIGURE 2.1 – Plateforme Java SE.

source : Java Platform SE 8 Documentation

27

Quelques composants de Java SE

JRE Le *Java Runtime Environment* (JRE) fournit la machine virtuelle, les bibliothèques et d'autres composants nécessaires pour l'exécution de programmes Java

JDK Le *Java Development Kit* (JDK) fournit le JRE ainsi qu'un ensemble d'outils pour le développement d'applications

java Le lanceur d'application est l'outil ligne de commande permettant l'exécution de programme Java

javac Le compilateur en ligne de commande

Plus d'informations

Java Platform Overview

28

2.4 Le langage Java

2.4.1 Caractéristiques du langage

Caractéristiques du langage

Simple un développeur peut être rapidement opérationnel
- dérivé de C++ en supprimant les constructions complexes (pointeurs, surcharge d'opérateurs, ...)

Orienté-objet langage relativement «pur»

Interprété la compilation génère un code intermédiaire qui est interprété

Portable un programme compilé fonctionne sans modification sur différentes plateformes

Robuste vérifications à la compilation et à l'exécution
- le système d'exécution gère et vérifie l'utilisation de la mémoire

Multithread supporte la programmation concurrente

Adaptable supporte le chargement dynamique de code

Sécurisé le langage intègre un modèle de sécurité sophistiqué

Ces caractéristiques sont discutées en détail dans un article de 1996 de James Gosling. 29

Compilation et interprétation

- Le langage Java est à la fois *interprété* et *compilé*
- Un fichier source (`.java`) est compilé en un langage intermédiaire appelé *bytecode* (`.class`)
- Ce bytecode est ensuite interprété par la *machine virtuelle Java*
  - c'est la clé de la portabilité du langage.

(voir figure 2.2).

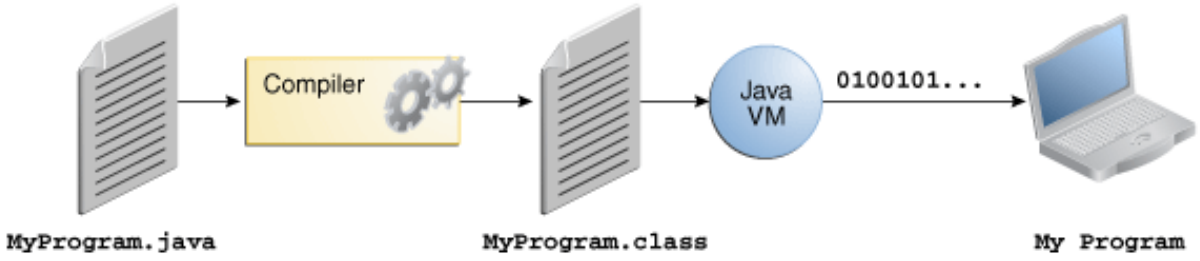


FIGURE 2.2 – Compilation et exécution.

source : About the Java Technology 30

Compilation (JDK)

```
$ javac <options> <fichiers source>
```

`-g|g:none` gère les informations pour le débogage

`-classpath|-cp` fixe le chemin de recherche des classes compilées

`-source` précise la version des fichiers sources (`1.6`, ..., `1.8`)

`-sourcepath` fixe le chemin de recherche des sources

`-encoding` précise l'encodage des fichiers sources ("UTF-8", ...)

-d fixe le répertoire de destination pour les classes compilées

-target précise la version de la VM cible

Compilation séparant les sources des fichiers compilés

```
$ javac -sourcepath src -source 1.7
        -d classes -classpath classes \
        -g src/MonApplication.java
```

31

Execution (JRE)

```
$ java [-options] class [args...]
$ java [-options] -jar jarfile [args...]
```

class est ici le nom de la classe (le `.class` doit pouvoir être trouvé dans le *CLASSPATH*)

-cp|-classpath fixe le chemin de recherche des classes compilées

-jar exécute un programme encapsulé dans un fichier `jar`

Exécution

```
$ java -cp classes MonApplication
```

32

2.4.2 Constructions de base du langage Java

Notions de base

Syntaxe

- Java différencie majuscules et minuscules
  - différence majuscule/minuscule même au niveau de l'interpréteur ⇒ le nom du fichier doit être correctement écrit
- Java possède une syntaxe proche du C
  - se retrouve à tous les niveaux (commentaires, types, opérateurs, ...)

Commentaires

/* ... */ le texte entre /* et */ est ignoré

// ... le texte jusqu'à la fin de la ligne est ignoré

33

Types primitifs

Type primitif

Un *type primitif* est un type de base du langage, i.e. non défini par l'utilisateur. En Java, les valeurs de ces types ne sont pas des objets.

boolean `true` ou `false`

byte entier de -128 à 127 (les types entiers sont signés)

short entier de $-32\,768$ à $32\,767$

int entier de -2^{31} à $2^{31}-1$

long entier de -2^{63} à $2^{63}-1$

char caractère Unicode sur 16 bits de `'\u0000'` à `'\uffff'`

float nombre en virgule flottante simple précision (32 bits IEEE 754)

double nombre en virgule flottante double précision (64 bits IEEE 754)

34

Littéraux

Littéral
Un *littéral* est la représentation dans le code source d'une valeur d'un type.

Entiers `123` de type `int`, `123L` de type `long`, `0x123` en hexadécimal, `0b101` en binaire (*Java SE 7*)

Flottants `1.23E-4` de type `double`, `1.23E-4F` de type `float`

Booléens `true` ou `false`

Caractères `'a'`, `'\t'` ou `'\u0000'`

Chaînes `"texte"`

Null `null` (valeur des références non initialisées)

35

Exemple
Déclarations et initialisations de variables

```
// Exemples de declaration avec initialisation
// (pas indispensable mais conseille)

byte aByte = 12;              // Un entier sur 8 bits
short aShort = 130;           // Un entier sur 16 bits
int anInteger = -153456;      // Un entier sur 32 bits

// Remarquer le L pour le litteral de type long
// (sinon erreur a la compilation: entier trop grand)
long aLong = 987654321234L; // Un entier sur 64 bits

// Remarquer le F pour le litteral de type float
// (sinon erreur a la compilation: perte de precision)
float aFloat = 1.3F;          // Un reel simple precision
double aDouble = -1.5E-4;     // Un reel double precision

char aChar = 'S';             // Un caractere
boolean aBoolean = true;      // Un booleen

// La constante est introduite par le mot-cle final
final int aConst = 0;         // Une constante
```

Listing 2.7 – Déclarations et initialisations de variables

36

Références 1/2
- Les variables de type tableau, énumération, objet ou interface sont en fait des *références*
- La valeur d'une telle variable est une référence vers (l'adresse de) une donnée
  - pas d'arithmétique des pointeurs en Java
  - les références assurent une meilleure sécurité (moins d'erreurs de programmation)
- Dans d'autres langages, une référence est appelée *pointeur* ou *adresse mémoire*
- En Java, la différence réside dans le fait qu'on ne manipule pas directement l'adresse mémoire : le nom de la variable est utilisé à la place
- L'association (l'affectation) d'une donnée à une variable *lie* l'identificateur et la donnée

37

Références 2/2
(voir figure 2.3).

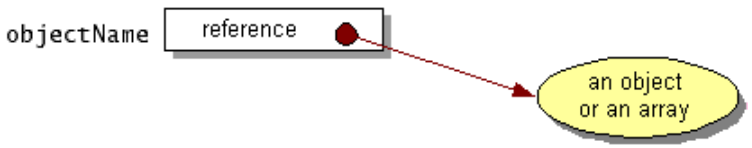


FIGURE 2.3 – Référence.

source : The Java Tutorial

38

Référence vs. pointeur

— Dans les deux cas, ce sont des variables (ou des constantes) dont la valeur (le contenu) est une adresse mémoire
— Un pointeur est un concept de bas-niveau permettant une manipulation précise de l'adresse (arithmétique des pointeurs, pointeur de fonction, ...)
— Une référence est une *abstraction* de plus haut niveau qui fournit une interface plus simple mais plus limitée pour manipuler l'adresse

Fonctionnalités	Référence	Pointeur
Simplicité	Oui	Non
Notation spécifique	Non	Oui
Arithmétique des adresses	Non	Oui
Adresse de fonctions	Non	Oui

39

Tableaux

— Un *tableau* est une structure de données regroupant plusieurs valeurs de même type
— La taille d'un tableau est déterminée lors de sa création (à l'exécution)
— La taille d'un tableau ne varie pas par la suite
— Un tableau peut contenir des références
  — tableau d'objets ou tableau de tableaux
  — permet de simuler des tableaux à plusieurs dimensions

(voir figure 2.4).

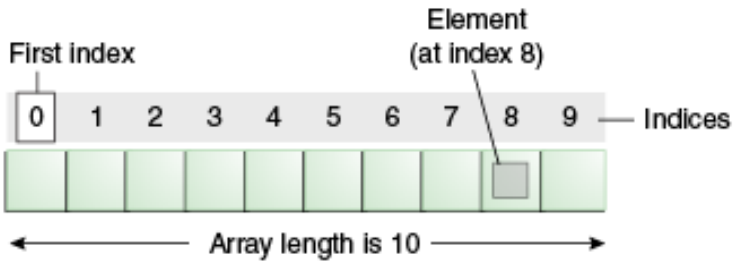


FIGURE 2.4 – Un tableau de 10 éléments.

source : The Java Tutorials: Arrays

40

Déclaration et création de tableaux

— La déclaration d'une variable de type tableau se fait en ajoutant [] au type des éléments

```
int[] unTableau;
```

— La création du tableau se fait en utilisant l'opérateur `new` suivi du type des éléments du tableau et de sa taille entre []

```
new int[10];
```

— La référence retournée par new peut être liée à une variable

```
int[] unTableau = new int[10];
```

— Il est possible de créer et d'initialiser un tableau en une seule étape

```
int[] unTableau = { 1, 5, 10 };
```

41

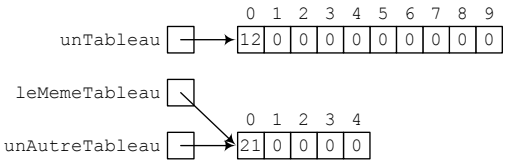
Manipulation de tableaux

- L'accès aux éléments d'un tableau se fait en utilisant le nom du tableau suivi de l'indice entre [] (exemple : $unTableau[2]$)
- La taille d'un tableau peut être obtenue en utilisant la propriété $length$ (exemple : $unTableau.length$)
- La méthode de classe `arraycopy` de `System` permet de copier efficacement un tableau

42

Tableaux et références

```
int [] unTableau = new int[10];
int [] leMemeTableau = unTableau;
leMemeTableau[0] = 12;
int [] unAutreTableau = new int[5];
leMemeTableau = unAutreTableau;
leMemeTableau[0] = 21;
```



43

Chapitre 3

Vue d'ensemble des concepts objets

Sommaire

3.1 Objet

3.1.1 Système orienté objet

Système orienté objet

- Lors de son exécution, un *système OO* est **un ensemble d'objets qui interagissent**
- Les objets forment donc l'*aspect dynamique* (à l'exécution) d'un système OO
- Ces objets représentent soit
  - des entités du monde réel (⇒ ce sont donc des modèles d'entités réelles), soit
  - des objets « techniques » nécessaires durant l'exécution du programme.

44

3.1.2 Objet

Objet

- Un *objet* est formé de deux composants indissociables
  - son *état*, i.e. les valeurs prises par des variables le décrivant (*propriétés*)
  - son *comportement*, i.e. les opérations qui lui sont applicables
- Un objet est une *instance* d'une *classe*

— Un objet peut avoir plusieurs types, i.e. supporter plusieurs interfaces

Exemple
Des points et des cercles
(voir figure 3.1).

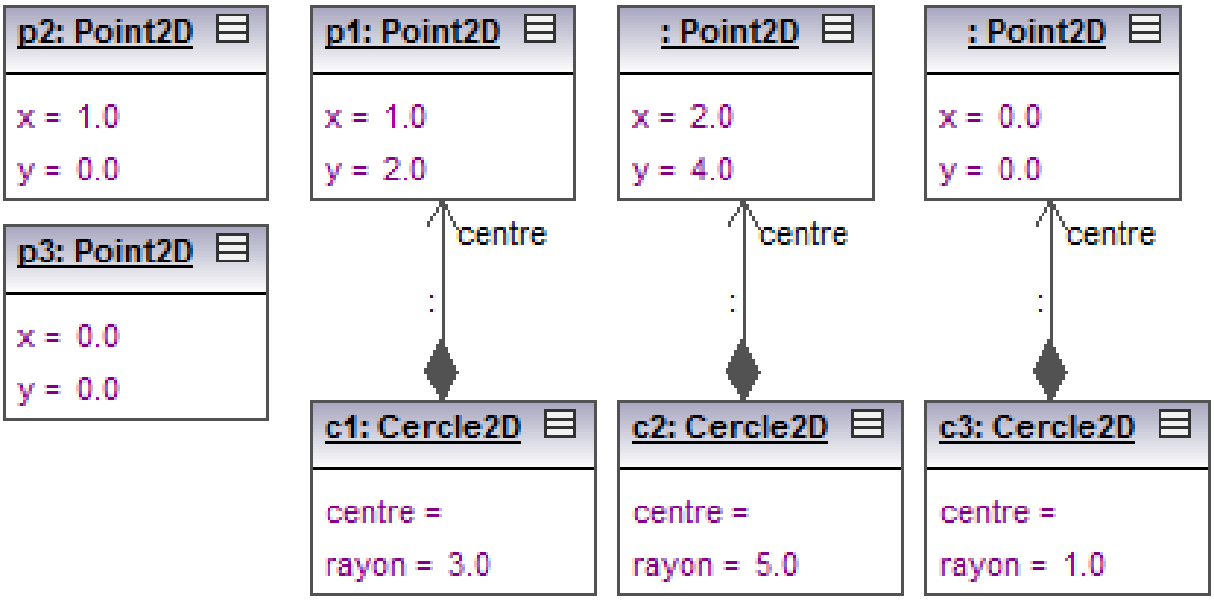


FIGURE 3.1 – Diagramme d'objets UML.

— Les objets `p1`, `p2` et `p3` sont des instances de la classe `Point2D` et deux autres objets `Point2D` sont anonymes
— Les objets `c1`, `c2` et `c3` sont des instances de la classe `Cercle2D`
— L'état de chaque objet est représenté par la valeur de ses propriétés
— L'attribut `centre` est une référence sur un objet `Point2D`
— Le comportement n'est pas représenté au niveau des objets
  — une opération est invoquée par rapport à un objet
  — mais elle est rattachée à la classe car le code est partagé par tous les objets d'une classe
— L'objet `p3` et le point anonyme de coordonnée (0, 0) sont égaux mais pas identiques

Exemple
Des points et des cercles en Java

```
Point2D p1 = new Point2D(1.0, 2.0);
Point2D p2 = new Point2D(1.0);
Point2D p3 = new Point2D();
Point2D unAutreP3 = p3;
      assert p3 == unAutreP3; // 2 points identiques

Cercle2D c1 = new Cercle2D(p1, 3.0);
Cercle2D c2 = new Cercle2D(new Point2D(2.0, 4.0), 5.0);
Cercle2D c3 = new Cercle2D();
```

Listing 3.1 – Instanciation de cercles et de points en Java

— Pour `p1, p2, p3` : déclaration de la variable, création de l'objet et initialisation de la variable en « une seule étape »
— Pour `unAutreP3` : déclaration de variable, pas de création d'objet, initialisation à partir de la référence sur `P3`

3.1.3 Message

Communication entre objets

— Un objet solitaire n'a que peu d'intérêt ⇒ différents objets doivent pouvoir interagir
— Un *message* est un moyen de communication (d'interaction) entre objets
— *Les messages sont les seuls moyens d'interaction entre objets* ⇒ l'état interne ne doit pas être manipulé directement
— Le (ou les) type(s) d'un objet détermine les messages qu'il peut recevoir

48

Message

— Un message est une requête envoyée à un objet pour demander l'exécution d'une opération
— Un message comporte trois composants :
  — *l'objet auquel il est envoyé* (le destinataire du message),
  — le nom de l'opération à invoquer,
  — les paramètres effectifs.

49

Exemple

Échange de messages

(voir figure 3.2).

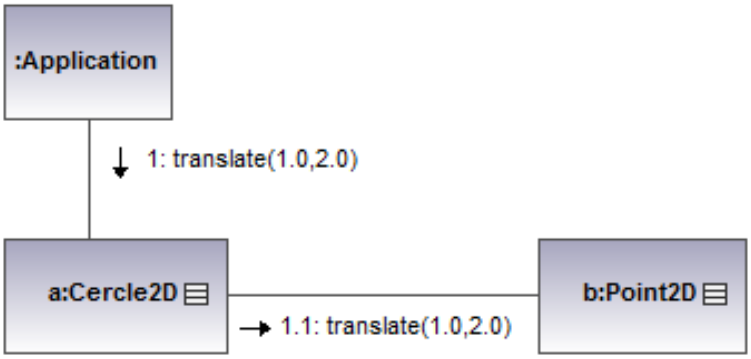


FIGURE 3.2 – Diagramme de communication UML.

— L'objet `Application` envoie un message à une instance `a` de `Cercle2D`
— Le message se traduit par l'exécution de l'opération `translate(1.0, 2.0)` par l'objet `a`
— Durant cette exécution, `a` envoie un message à l'objet `b` de classe `Point2D` (translater le cercle revient à translater son centre)
— La numérotation des messages reflète leur ordre et leur niveau d'imbrication

50

Exemple

Échange de messages en Java

— lors de l'exécution d'une opération de l'application, envoi de `a.translate(1.0, 2.0)`
  — lors de l'exécution de `translate` du cercle `a`, envoi de `b.translate(1.0, 2.0)`
    — exécution de `translate` du point `b`

51

3.2 Classe

3.2.1 Classe

Classe

— Une *classe* est un « modèle » (un « moule ») pour une catégorie d'objets structurellement identiques
  — Par exemple, la *voiture immatriculée 123456* est une instance de la classe *Voiture*
  — L'état et le comportement des voitures sont communs à toutes les voitures mais l'état courant de chaque voiture est indépendant des autres

  - Chaque instance aura ses propres copies des variables d'instances
- Une classe définit donc l'implémentation d'un objet (son état interne et le codage de ses opérations)
- L'ensemble des classes décrit l'*aspect statique* d'un système OO

52

Composition d'une classe

- Une classe comporte :
  - la définition des *attributs* (ou *variables d'instance*),
  - la *signature* des opérations (ou *méthodes*),
  - la *réalisation* (ou *définition*) des méthodes.
- Chaque instance aura sa propre copie des attributs
- La signature d'une opération englobe son nom et le type de ses paramètres
- L'ensemble des signatures de méthodes représente l'interface de la classe (publique)
- L'ensemble des définitions d'attributs et de méthodes forme l'implémentation de la classe (privé)

53

Exemple

Les classes `Cercle2D` *et* `Point2D` *(mode abrégé)*

(voir figure 3.3).

FIGURE 3.3 – Diagramme de classes UML (mode abrégé).

54

Exemple

Les classes `Cercle2D` *et* `Point2D`

(voir figure 3.4).

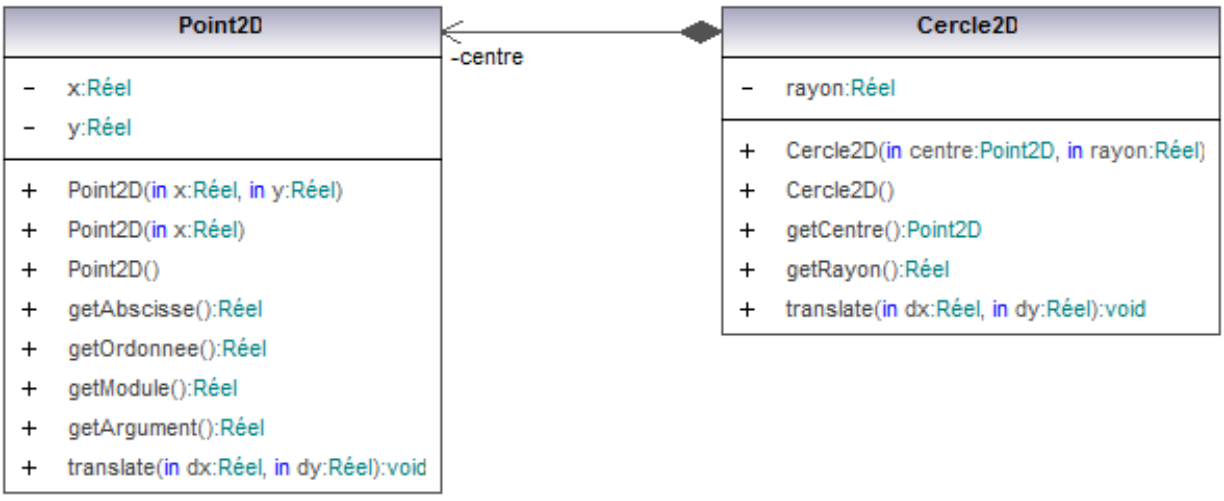


FIGURE 3.4 – Diagramme de classes UML.

- Un rectangle représente une classe
  - 1erpavé : nom de la classe
  - 2epavé : attributs
  - 3epavé : signature des méthodes
- Accès aux membres

- + = public, # = protégé, – = privé, ~ = paquetage
- en général, les attributs sont privés et les méthodes publiques

— `Point2D` et `Cercle2D` sont des constructeurs (surcharge), `getXXX` sont des accesseurs, `translate` est un mutateur.

— Exemple d'invariant : « le rayon du cercle doit toujours être positif »

Exemple

La classes `Cercle2D` en Java

```
class Cercle2D implements Deplacable {
    /** Le centre du cercle. */
    private Point2D centre;

    /** Le rayon du cercle */
    private double rayon;

    /**
     * Initialise un cercle avec un centre et un rayon.
     * @param centre Le centre.
     * @param rayon Le rayon.
     */
    public Cercle2D(Point2D centre, double rayon) { /* ... */ }

    /**
     * Initialise un cercle centre a l'origine et de rayon 1.
     */
    public Cercle2D() { /* ... */ }

    public Point2D getCentre() { /* ... */ }
    public double getRayon() { /* ... */ }

    /**
     * Translate le cercle.
     * @param dx deplacement en abscisse.
     * @param dy deplacement en ordonnees.
     */
    public void translate(double dx, double dy) { /* ... */ }
}
```

Listing 3.2 – La classes `Cercle2D` en Java

3.2.2 Classe et objet

Instanciation d'une classe

— Le mécanisme d'*instanciation* permet de créer des objets à partir d'une classe

— Chaque objet est une instance d'une classe

— Lors de l'instanciation,

- de la mémoire est allouée pour l'objet,
- l'objet est initialisé afin de respecter l'invariant de la classe.

Exemple

Classe et objet

(voir figure 3.5).

— Les objets `p1`, `p2` et `p3` sont des instances de la classe `Point2D`

— Ce type de lien est représenté par le *stéréotype « instanceof »* en UML

— On représente rarement classes et objets sur le même schéma (pas le même point de vue)

— *Attention à bien différencier classe et objet*

3.2.3 Classe et type

Type

— Un *type* est un modèle abstrait réunissant à un haut degré les traits essentiels de tous les êtres ou de tous les objets de même nature

— En informatique, un *type (de donnée)* spécifie :

- l'ensemble des valeurs possibles pour cette donnée (définition en *extension*),
- l'ensemble des opérations applicables à cette donnée (définition en *intention*).

— Un type spécifie l'*interface* par laquelle une donnée peut être manipulée

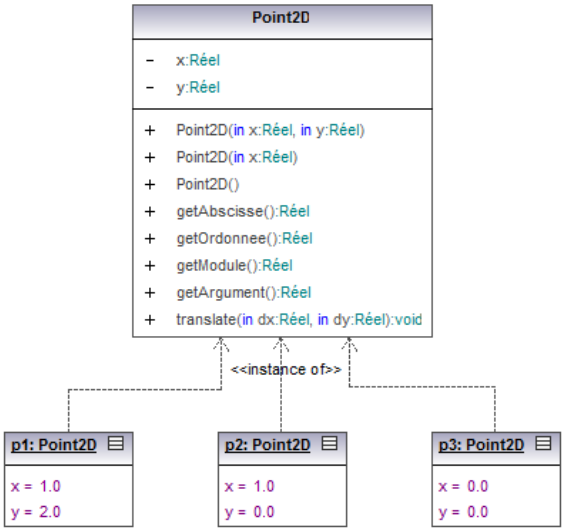


FIGURE 3.5 – Lien entre classe et objet.

Exemple
Représentation d'un type comme une interface
(voir figure 3.6).

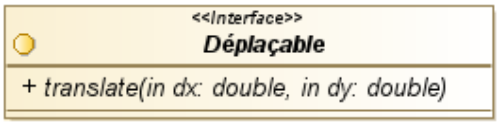


FIGURE 3.6 – Représentation d'un type comme une interface.

60

Exemple
L'interface `Deplacable` *en Java*

```
interface Deplacable {
    /**
     * Translate l'objet.
     * @param dx deplacement en abscisse.
     * @param dy deplacement en ordonnees.
     */
    void translate(double dx, double dy);
}
```

Listing 3.3 – L'interface `Deplacable` en Java

61

Classe et type
- Une classe implémente un ou plusieurs types, i.e. respecte une ou plusieurs interfaces
- Un objet peut avoir plusieurs types mais est une instance d'une seule classe
- Des objets de classes différentes peuvent avoir le même type

62

Exemple
Interface et classe
(voir figure 3.7).

```
class Cercle2D implements Deplacable {
```

Listing 3.4 – `Cercle2D` implémente `Deplacable` en Java

63

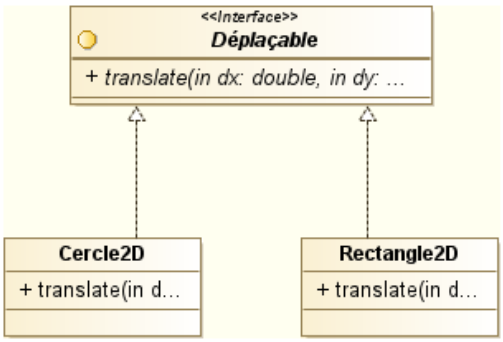


FIGURE 3.7 – Relation entre classe et interface.

Exercice
Modélisation d'objets et de classes

On veut modéliser des robots se déplaçant sur un terrain. Ce terrain est découpé en cases carrées repérées par deux coordonnées. Chaque case peut être vide ou contenir un mur ou un robot. Les robots sont très rudimentaires et ne disposent que d'une boussole. Ils ne connaissent donc que leur orientation (Nord, Est, Sud, Ouest). Un robot doit pouvoir avancer d'une case et tourner d'un quart de tour à droite. Un robot ne peut se déplacer d'une case à une autre que si la case de destination est vide.

1. Représenter avec la notation vue précédemment la classe `Robot`
2. Faire de même avec la classe `Terrain`
3. Représenter sur un diagramme de communication les échanges de messages pour le déplacement d'un robot
4. On suppose maintenant qu'un robot peut en détecter un autre qui passe devant lui. Par exemple, quand un robot se déplace, il peut passer dans le champ de vision d'un autre. Ce dernier devra alors être averti. Modifier le diagramme de communication précédent pour y intégrer la détection des déplacements

65

3.3 Héritage

3.3.1 Héritage

Héritage

— L'*héritage* permet de définir l'implémentation d'une classe à partir de l'implémentation d'une autre
— Ce mécanisme permet, lors de la définition d'une nouvelle classe, de ne préciser que ce qui change par rapport à une classe existante
— Une *hiérarchie de classes* permet de gérer la complexité, en ordonnant les classes au sein d'arborescences d'abstraction croissante
— Si Y hérite de X, on dit que Y est une classe *fille* (*sous-classe*, *classe dérivée*) et que X est une classe *mère* (*super-classe*, *classe de base*)

70

Exemple
Rectangle et rectangle plein

(voir figure 5.1).

71

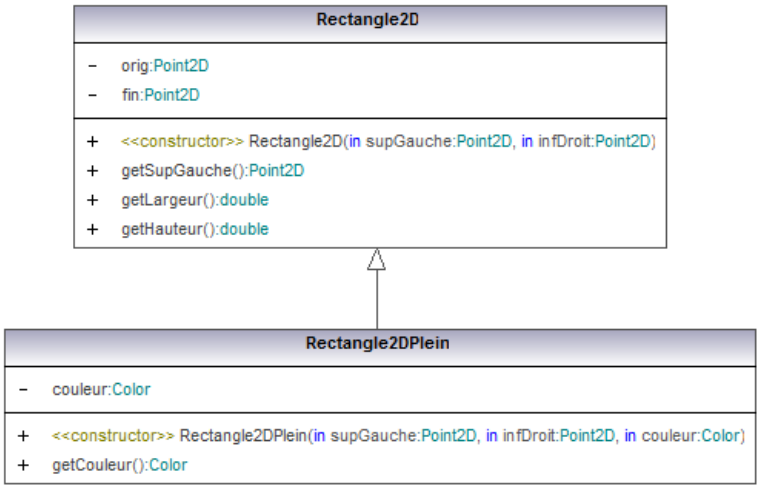


FIGURE 3.8 – Héritage entre rectangle plein et rectangle.

3.3.2 Polymorphisme

Polymorphisme

- Le *polymorphisme* est l'aptitude qu'ont des objets à réagir différemment à un même message
- L'intérêt est de pouvoir gérer une collection d'objets de façon homogène tout en conservant le comportement propre à chaque objet
- Une méthode commune à une hiérarchie de classe peut prendre plusieurs formes dans différentes classes
- Une sous-classe peut *redéfinir* une méthode de sa super-classe pour spécialiser son comportement
- Le choix de la méthode à appeler est retardé jusqu'à l'exécution du programme (*liaison dynamique ou retardée*)

72

Exemple

Une description pour le rectangle et le rectangle plein
(voir figure 5.2).

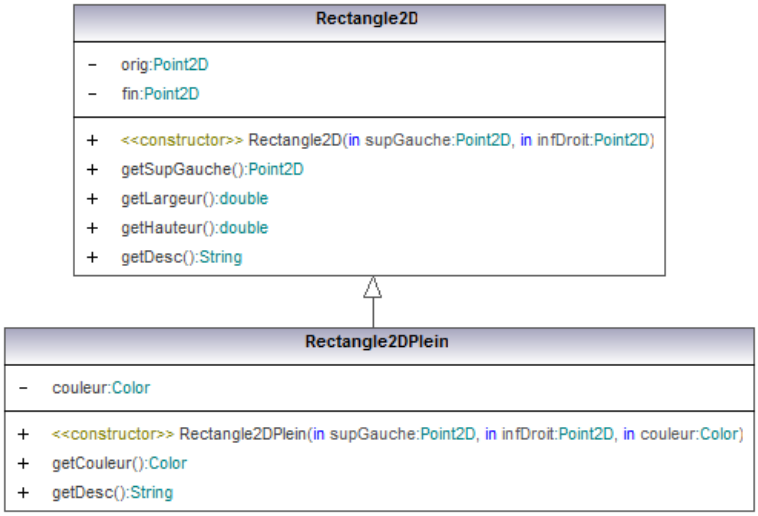


FIGURE 3.9 – Polymorphisme entre rectangle plein et rectangle.

73

3.3.3 Classe abstraite

Classe abstraite

— Une *classe abstraite* représente un concept abstrait qui ne peux pas être instancié
— En général, son comportement ne peut pas être intégralement implémenté à cause de son niveau de généralisation
— Elle sera donc seulement utilisée comme classe de base dans une hiérarchie d'héritage

74

Exemple

La hiérarchie d'héritage des figures
(voir figure 5.3).

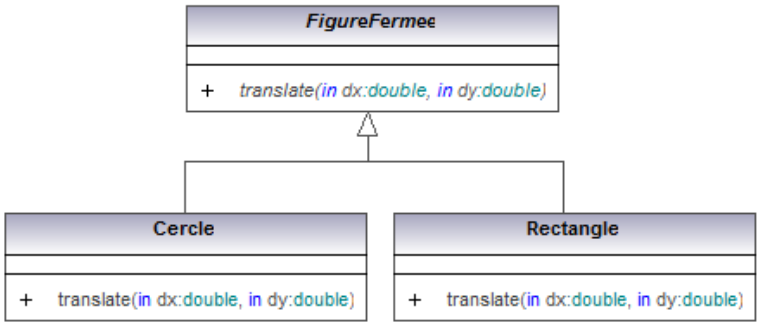


FIGURE 3.10 – Hiérarchie d'héritage des figures.

— Deux façons de représenter une classe abstraite : `abstract` entre `{}` ou nom de la classe en italique
— Méthodes abstraites en italique

75

3.3.4 Héritage multiple et à répétition

Héritage multiple et à répétition

— Un *héritage multiple* se produit lorsqu'une classe possède plusieurs super-classes
— Un *héritage à répétition* se produit lorsqu'une classe hérite plusieurs fois d'une même super-classe
— Ces types d'héritage peuvent provoquer des conflits aux niveaux des attributs et méthodes
  — deux classes de base peuvent posséder la même méthode,
  — un attribut peut être hérité selon plusieurs chemins dans le graphe d'héritage.
— L'héritage multiple de classe n'est pas supporté par Java

76

Exemple

Héritage multiple et à répétition
(voir figure 3.11).

77

3.3.5 Héritage et sous-typage

Sous-type

Sous-type

Un type T_1 est un *sous-type* d'un type T_2 si l'interface de T_1 contient l'interface de T_2

— Un sous-type possède une interface plus riche, i.e. au moins toutes les opérations du super-type
— De manière équivalente, l'extension du super-type contient l'extension du sous-type, i.e. tout objet du sous-type est aussi instance du super-type

78

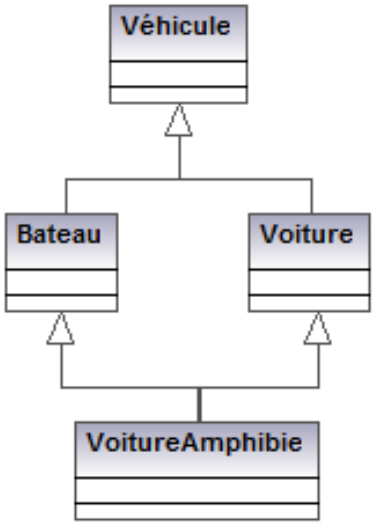


FIGURE 3.11 – Héritage multiple et à répétition.

Principe de substitution de Liskov

Principe de substitution de Liskov
Si pour chaque objet o_1 de type S, il existe un objet o_2 de type T tel que, pour tout programme P défini en terme de T, le comportement de P demeure inchangé lorsque o_1 est remplacé par o_2, alors S est un sous-type de T.

— Un objet du sous-type peut remplacer un objet du super-type sans que le comportement du programme ne soit modifié

79

Héritage et sous-typage

— L'héritage (ou *héritage d'implémentation*) est un mécanisme technique de réutilisation
— Le sous-typage (ou *héritage d'interface*) décrit comment un objet peut être utilisé à la place d'un autre
— Si Y est une sous-type de X, cela signifie que « Y est une sorte de X » (relation *IS-A*)
— Dans un langage de programmation, les deux visions peuvent être représentées de la même façon : le mécanisme d'héritage permet d'implémenter l'un ou l'autre
— En Java, les interfaces et l'héritage entre interface implémentent plus spécifiquement le concept de sous-typage

80

Exemple

Héritage d'implémentation et d'interface
(voir figure 3.12).

81

3.4 Module

Module

— Un *module* (ou *package)* est l'unité de base de décomposition d'un système
— Il permet d'organiser logiquement des modèles
— Un module s'appuie sur la notion d'*encapsulation*
  — publie une interface, i.e. ce qui est accessible de l'extérieur
  — utilise le principe de *masquage de l'information*, i.e. ce qui ne fait pas parti de l'interface est dissimulé

82

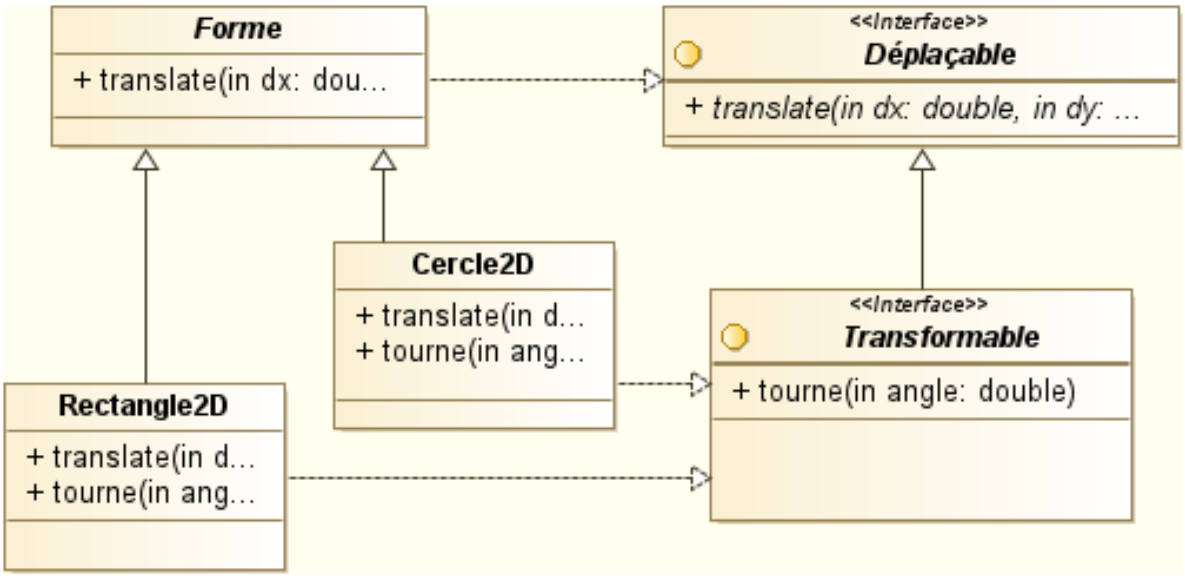


FIGURE 3.12 – Héritage d'implémentation et d'interface.

Utilité d'un module

- Sert de brique de base pour la construction d'une architecture
- Représente le bon niveau de granularité pour la réutilisation
- Est un *espace de noms* qui permet de gérer les conflits

83

Qualité d'un module

- La conception d'un module devrait conduire à un *couplage faible* et une *forte cohésion*

  couplage désigne l'importance des liaisons entre les éléments ⇒ *doit être réduit*

  cohésion mesure le recouvrement entre un élément de conception et la tâche logique à accomplir ⇒ *doit être élevé*, i.e. chaque élément est responsable d'une tâche précise

84

Exemple

Un diagramme de packages UML
(voir figure 3.13).

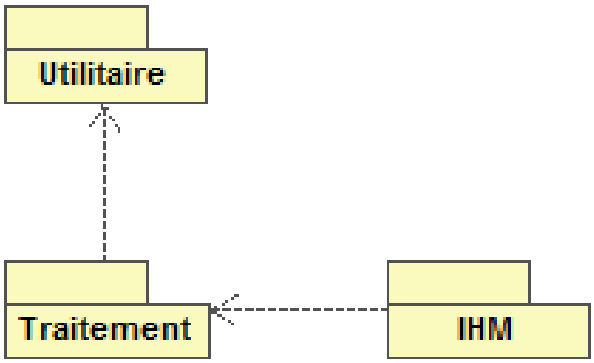


FIGURE 3.13 – Diagramme de packages UML.

- Liens d'utilisation entre module
- D'autres types de liens sont également possible comme l'héritage

85

Module en Java
- Le mot clé `package` placé en début de fichier permet l'ajout d'éléments dans un module
- Le mot clé `import` permet l'accès aux éléments d'un module

3.5 Exercices

CONTRAINTES
- Utilisation de l'environnement de développement BLUEJ.
- Les seules sources d'information externes autorisées sont : (i) la référence de l'API Java, (ii) le tutoriel Java, (iii) la documentation de BLUEJ.
- Exécution interactive uniquement avec BLUEJ (pas de méthode `main` ni d'affichage).

Exercice 3.1 (Découverte de BlueJ)
Dans cette exercice, vous apprendrez les manipulations de base de l'environnement BLUEJ.

1. Ouvrez le projet d'exemple `people2` (*Projects/Open Projet...*). Les projets d'exemples se trouvent dans le répertoire d'installation de BlueJ sous Windows ou dans `/usr/share/doc/BlueJ` sous Linux.
   (a) Quelles classes y-a-t-il dans ce projet ?
   (b) Quelles relations existe-t-il entre ces classes ?
2. Compilez le projet (Bouton *Compile*)
   (a) Quels logiciels faut-il pour compiler un projet en Java ? pour l'exécuter ?
3. Création d'objet (*menu contextuel d'une classe*)
   (a) Quels constructeurs sont disponibles pour chaque classe ?
   (b) Pourquoi la classe `Person` n'en a-t-elle aucun ?
   (c) Créez un employé (*Staff*) *p1* ayant pour nom *Smith*, pour année de naissance *1980* et pour numéro de bureau *D100*.
4. Invocation de méthode (*menu contextuel d'un objet*)
   (a) Quelles méthodes propose *p1* ?
   (b) Exécutez les méthodes `getRoom()` et `getName`. Dans quelles classes sont-elles définies ?
   (c) Exécutez la méthode `setAddress` .
5. Évaluation à la volée de code Java (*View/Show Code Pad*) Les manipulations ci-dessous sont à réaliser dans le Code Pad.
   (a) Créez un étudiant en précisant son nom, son année de naissance et son identifiant.
   (b) Copiez-le (*glisser-déposer*) dans la fenêtre des objets.
   (c) Exécutez la méthode `getName` dans le *Code Pad*.
6. Inspection d'un objet (*menu contextuel d'un objet*)
   (a) Inspectez l'objet *p1* ?
   (b) De quels attributs son état est-il formé ?
   (c) Pouvez-vous inspecter tous ses attributs ? Pourquoi ?
7. Éditez le code Java de la classe `Staff` (*double-click sur la classe*)
   (a) Quels éléments du langage trouve-t-on dans cette classe ?
   (b) Á quoi correspondent les couleurs de fond ?
   (c) Visualisez la documentation de la classe (*liste déroulante en haut à droite*).

(d) Quelle relation y-a-t-il entre la documentation et le code source ? Modifiez un commentaire dans le code et visualisez à nouveau la documentation ?

(e) quel parallèle peut-on établir avec les fichiers `.h` et `.c` du langage C ?

Exercice 3.2 (Manipulation d'objets)
Dans cette exercice, vous effectuerez les manipulations tout d'abord de façon interactive, puis en tapant les instructions dans le *Code Pad*.

1. Ouvrez le projet d'exemple BlueJ *shapes* et compiler-le.
2. Réalisez les manipulations suivantes
   (a) Créez un cercle et rendez-le visible.
   (b) Déplacez-le horizontalement de 100 unités.
   (c) Changez sa couleur en rouge.
   (d) Déplacez-le verticalement de 50 unités.
3. Représentez la scène suivante :
   — une maison (carré bleu et triangle rouge) ,
   — un soleil jaune au-dessus et à droite de la maison.

Exercice 3.3 (Modification d'une classe)
Cet exercice utilise le projet `Entreprise` (cf. figure 3.14) à ouvrir sous BlueJ (*Project/Open Non Bluej...*).
(voir figure 3.14).

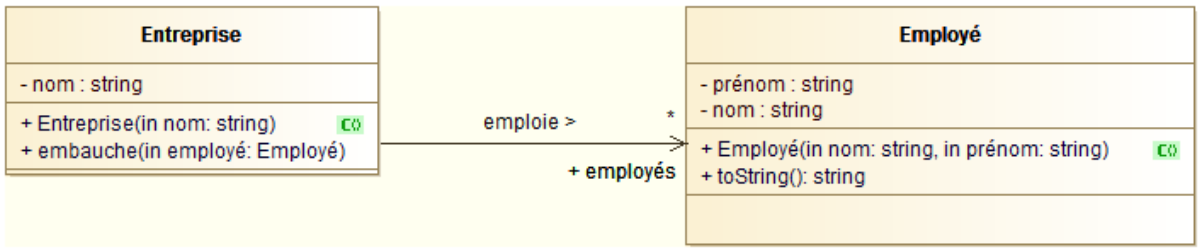


FIGURE 3.14 – Diagramme de classe du projet `Entreprise`.

1. Modifiez la classe `Employe` comme suit :
   (a) ajoutez l'attribut privé age,
   (b) modifiez les méthodes pour prendre en compte ce changement,
   (c) recompilez et testez ce changement.
2. Mettez à jour les commentaires Javadoc et vérifier la documentation de la classe.
3. Ajoutez la méthode `toString` à la classe `Entreprise` : cette méthode retourne une chaîne de caractères contenant le nom de l'entreprise et la liste de ses employés (prénom, nom et âge),
4. recompiler les deux classes et tester les nouvelles méthodes.

Exercice 3.4 (Utilisation d'un débogueur)
Cet exercice utilise le même projet que l'exercice précédent.
Pour chaque test, vous créerez une entreprise et deux employés que l'entreprise embauchera puis vous invoquerez `toString` sur l'entreprise.

1. Faites afficher le débogueur intégré à BlueJ (*View/Show Debugger*). Quels éléments sont visibles dans le débogueur ? Quelles actions peut-on effectuer à partir du débogueur ?

2. Lancer un premier test et vérifier le comportement du programme.
3. Lancer un second test en saisissant la valeur `null` lors de l'embauche du deuxième employé
   — Que se passe-t-il ?
   — Quelle instruction pose problème ?
4. Placer un point d'arrêt (*breakpoint*) au début de la méthode `toString` de `Entreprise` (*click dans la marge de l'éditeur*).
5. Reproduisez le second test :
   (a) l'exécution doit s'arrêter au niveau du *breakpoint*,
   (b) faites exécuter la méthode pas à pas en consultant la valeur des variables,
   (c) quelle différence existe-t-il entre les commandes *Step* et *Step Into* du débogueur ?
   (d) pour quel employé le problème se pose-t-il ?
   (e) quelle est donc finalement la cause de l'erreur ?
6. Modifiez la classe `Entreprise` pour éviter le problème.
7. Vérifiez en reproduisant à nouveau le second test.

Chapitre 4

Objet et classe

Sommaire

4.1 Compléments sur les objets

4.1.1 Caractéristiques d'un objet

Etat d'un objet

- L'état d'un objet est l'ensemble de ses caractéristiques *internes*, *cachées* aux autres objets
- Ces caractéristiques peuvent elles-mêmes être des objets
- Les propriétés représentant l'état d'un objet particulier sont appelées *variables d'instance* ou *attributs* ou *données membres*
- Chaque objet possède un état courant décrit par la valeur de ses attributs et donc sa propre copie des variables d'instance
- Les attributs conservent leurs valeurs durant toute la durée de vie d'un objet

87

Comportement d'un objet

- Le comportement d'un objet regroupe les caractéristiques *externes* mises à la disposition des autres objets
- Chaque opération est décrite par sa *signature* (son nom, les objets qu'elle prend comme paramètre et le type de retour)

- L'ensemble de ces opérations forme l'interface de l'objet
- Les opérations propres à un objet sont appelées *méthodes d'instance* ou *fonctions membres* (fonctions associées à l'objet)
- *Ces opérations sont invoquées par rapport à un objet particulier*
  - une méthode sait toujours quel objet l'a invoquée
  - donc, une méthode a accès aux attributs de l'objet qui l'a invoquée
  - fonction avec un paramètre caché vers l'objet lui-même
- *Les opérations sont les seules moyens de manipuler l'état interne d'un objet (*encapsulation*)*

88

Identité, égalité, affectation et copie d'objets

- Un objet possède une *identité* (identifiant interne) qui caractérise son existence de façon indépendante de son état (*égalité ≠ identité*)
  ⇒ deux objets peuvent être égaux (même état interne) sans être identiques (même objet)
- L'*affectation* d'un objet à une variable crée un *lien* entre la variable et l'objet
- Créer une *copie* d'un objet nécessite de créer récursivement une copie de ses attributs (*copie profonde* ou *deep copy*)
- Un autre sémantique possible est de ne copier que l'objet racine et de partager ses attributs (*copie superficielle* ou *shallow copy*)

89

4.1.2 Les objets en Java

Création d'objets

- Un objet est créé (instancié) à partir d'une classe en utilisant le mot-clé `new` (`new Point2D(1.0, 2.0)`)
- L'utilisation de `new` provoque la réservation de mémoire pour l'objet et l'invocation du constructeur qui initialise l'objet
  - Le constructeur est une méthode spéciale invoquée automatiquement lors de la création de l'objet
  - Son rôle est d'initialiser l'objet pour que ce dernier respecte l'invariant de la classe
- `new` retourne une référence sur l'objet créé
- Cette référence doit être liée (affectée) à un identificateur (variable) pour permettre l'accès à l'objet

90

Association d'une variable à un objet

- La syntaxe pour la déclaration d'une variable est `type nom` (`Point2D p1`)
- *La déclaration ne crée pas d'objet* mais uniquement une référence
- La déclaration crée un identificateur qui sera lié à l'objet en utilisant une affectation
- La variable est invalide tant qu'elle n'est pas liée à un objet (*null reference*)
  - contient la valeur `null`
  - il est conseillé de toujours initialiser une variable lors de sa déclaration (si possible)
- Il est possible de lier la variable lors de sa déclaration (`Point2D p1 = new Point2D(1.0, 2.0);`)

91

Exemple

Association d'une variable à un objet

(voir figure 4.1).

92

Exemple

Instanciation de cercle et de points 1/2

```
Point2D p1 = new Point2D(1.0, 2.0);
Point2D p2 = new Point2D(1.0);
Point2D p3 = new Point2D();
Point2D unAutreP3 = p3;
assert p3 == unAutreP3; // 2 points identiques

Cercle2D c1 = new Cercle2D(p1, 3.0);
```

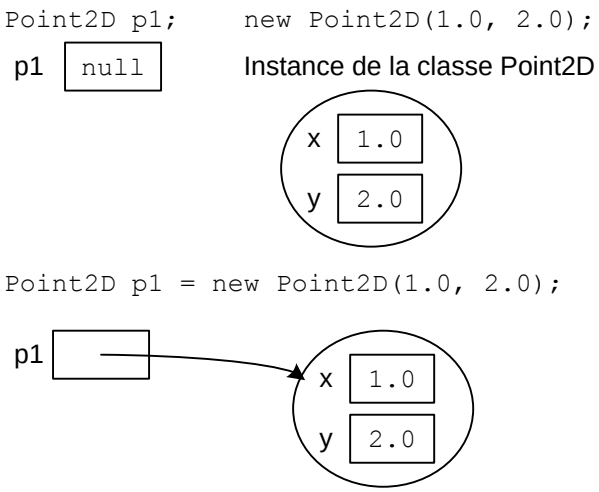


FIGURE 4.1 – Déclaration, création et affectation.

```
Cercle2D c2 = new Cercle2D(new Point2D(2.0, 4.0), 5.0);
Cercle2D c3 = new Cercle2D();
```

Listing 4.1 – Instanciation de cercles et de points

— Pour `p1, p2, p3` : déclaration de la variable, création de l'objet et initialisation de la variable en « une seule étape »
— Pour `unAutreP3` : déclaration de variable, pas de création d'objet, initialisation à partir de la référence sur `P3`

93

Exemple

Instanciation de cercle et de points 2/2

(voir figure 4.2).

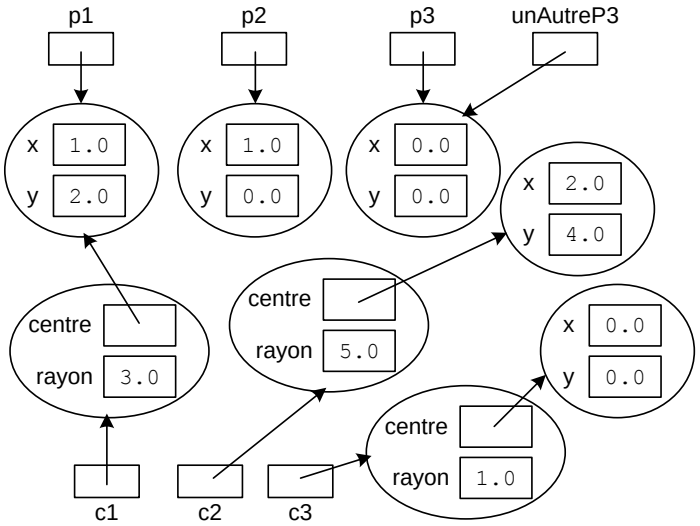


FIGURE 4.2 – Après la création des cercles et des points.

94

Manipulation

Accès aux attributs

— Juste par le nom de l'attribut quand on se trouve dans la portée de l'attribut (exemple : `x`)
— En qualifiant/préfixant avec le nom d'une référence sur l'objet à l'extérieur (exemple : `p1.x`)

— *La manipulation directe d'attributs en dehors de la classe est interdite* (violation de l'encapsulation)

Invocation d'une méthode

— Même syntaxe que pour les attributs mais avec la liste des paramètres (exemple : `p1.getAbscisse()`)

95

Destruction

— Quand un objet n'est plus utilisé, il doit être retiré de la mémoire
— La destruction des objets en Java est automatique
  — L'environnement d'exécution de Java supprime les objets lorsqu'il détermine qu'ils ne sont plus utilisés
  — Un objet est éligible pour la destruction quand plus aucune référence n'est liée à lui
— Ce processus de suppression s'appelle *ramasse-miette* (*garbage collector*)
— Avant de détruire l'objet, le ramasse-miette invoque la méthode `protected void finalize()` de l'objet
  — utilisée pour restituer les ressources allouées par l'objet
  — `finalize` est membre de la classe `Object`
  — `super.finalize()` doit être d'appeler à la fin de la méthode

96

4.1.3 Les chaînes de caractères en Java

Les chaînes de caractères en Java

— Java fournit trois classes pour les chaînes de caractères : `String`, `StringBuffer` et `StringBuilder`
— `String` est dédiée aux chaînes de caractères immuables, i.e. dont la valeur ne change pas
— `StringBuilder` est dédiée aux chaînes de caractères pouvant être modifiées (contexte *mono-thread*)
— `StringBuffer` est dédiée aux chaînes de caractères pouvant être modifiées (contexte *multi-threads*)

97

Création d'une chaîne (`String`)

— Une instance de `String` représente une chaîne au format UTF-16
— Une chaîne est souvent créée à partir d'un littéral de type chaîne (une suite de caractères entre guillemets)
  — quand Java rencontre un littéral de type chaîne, il crée un objet de type `String` dont la valeur est le littéral
— Une chaîne peut aussi être créée en utilisant l'un des constructeurs de `String`

98

Quelques accesseurs de `String`

`length()` taille de la chaîne,

`charAt(int)` caractère à l'indice spécifié,

`substring(int, int)` extraction d'une sous-chaîne,

`indexOf(...)`, `lastIndexOf(...)` recherche dans la chaîne

99

Autres usages de `String`

— *Un littéral chaîne peut être utilisé à tout endroit où un objet `String` peut l'être*
  ⇒ on peut invoquer des méthodes de `String` sur un littéral chaîne
— L'opérateur `+` permet de concaténer des objets de type `String`
  — c'est le seul opérateur surchargé pour un objet en Java
— Une chaîne peut être utilisée avec l'instruction `switch` (*Java SE 7*)

100

Les classes `StringBuilder` et `StringBuffer`

- Les instances disposent à peu prés des mêmes accesseurs que `String`
- Quelques mutateurs : `append(...)` (ajout de caractères), `delete(...)` (suppression de caractères), `insert(...)` (insertion de caractères)
- `StringBuilder` est optimisée pour un environnement *mono-thread*
- `StringBuffer` est à utiliser dans un contexte *multi-threads*

101

Exemple

Manipulation de chaînes de caractères

```
String source = "abcde";
int len = source.length();
StringBuilder dest = new StringBuilder(len);

for (int i = (len - 1); i >= 0; --i) {
  dest.append(source.charAt(i));
}

assert dest.toString().equals("edcba") : dest;
```

Listing 4.2 – Inverser une chaîne

- `toString` : conversion d'un objet en `String`
- `equals` test d'égalité, `==` test d'identité (la distinction n'a de sens que pour les objets)

102

4.2 Compléments sur les classes

4.2.1 Caractéristiques d'une classe

Catégories de méthodes

Accesseur permet de consulter l'état d'un objet

Mutateur modifie l'état d'un objet
- doit préserver l'invariant

Constructeur initialise un objet afin de le placer dans un état cohérent
- établit l'invariant de la classe
- appelé automatiquement lors de la création d'un objet

Destructeur libère les ressources allouées par l'objet
- appelé automatiquement lors de la destruction de l'objet

103

Surcharge de méthode

- La *surcharge* (ou *polymorphisme ad hoc*) d'une méthode consiste à définir plusieurs méthodes de même nom mais ayant des signatures différentes
- Une opération n'est donc plus simplement identifiée par son nom mais également par le nombre, l'ordre et le type de ses arguments
- Le choix de la méthode est résolu lors de la compilation

Exemple d'application

- La surcharge est souvent utilisée pour définir plusieurs constructeurs de façon à initialiser les objets de différentes manières
- Une alternative plus souple consiste à appliquer un modèle de conception Fabrique (Factory)

104

Accès aux membres

— Une classe peut contrôler l'accès à ses membres (données ou fonctions)

privé accès limité à la classe

protégé accès limité à la classe et à ses sous-classes

module accès limité au module (*package*) contenant la classe

public accès non limité

105

Invariant de classe

Invariant de classe

L'*invariant d'une classe* impose une contrainte sur l'état des instances de la classe. Chaque objet doit respecter les conditions imposées par l'invariant.

— L'invariant est établi lors de la création d'une instance
— Il doit être vérifié avant l'exécution d'une opération publique
— Chaque opération publique doit rétablir l'invariant avant de se terminer
— L'invariant peut être violé temporairement lors de l'exécution d'une opération
— Avec un langage de programmation, l'invariant peut être exprimé avec des assertions ou avec une construction spécifique

106

4.2.2 Les classes en Java

Définition de classe

— La *définition* d'une classe comporte deux parties : la *déclaration* et le *corps* de la classe

```
/**
 * Un bref commentaire.
 * Un commentaire plus détaillé...
 * @version oct 2008
 * @author Prénom NOM
 */
class NomClasse { // Déclaration de la classe
  // Corps de la classe
}
```

— La déclaration précise au compilateur un certain nombre d'informations sur la classe (son nom, ...)
— Le corps de la classe contient les attributs et les méthodes (les *membres*) de la classe

107

Attribut

— La déclaration d'un attribut spécifie son nom et son type

```
/** Description de l'attribut. */
Type nom;
/** Description de l'attribut. */
final Type nom;
```

— *L'initialisation des attributs se fait dans un constructeur*
— `final` est optionnel et permet de déclarer un attribut qui ne pourra être affecté qu'une unique fois

108

Pseudo-attribut `this`

— Chaque classe possède un attribut privé particulier nommé `this` qui référence l'objet courant
— Cette attribut est maintenu par le système et ne peut pas être modifié par le programmeur
— `this` n'est accessible que dans le corps de la classe
— Utilité
  — passer l'objet courant en paramètre d'une méthode

— lever certaines ambiguïtés à propos des membres
— invoquer un autre constructeur dans un constructeur

Exemple

Les attributs d'un cercle

```
/**
 * Un cercle en deux dimensions.
 *
 * @version  jan. 2015
 * @author   Stephane Lopes
 *
 */
class Cercle2D {
    /** Le centre du cercle. */
    private Point2D centre;

    /** Le rayon du cercle */
    private final double rayon;
```

Listing 4.3 – Les attributs d'un cercle

Méthode

— La *définition* d'une méthode comporte deux parties : la *déclaration* et le *corps* (l'*implémentation*) de la méthode

```
/**
 * Brève description de la méthode.
 * Une description plus longue...
 * @param param1 description du paramètre
 * @param ...
 * @return description de la valeur de retour
 */
TypeRetour nomMethode(listeDeParametres) { // Déclaration
  // Corps de la méthode
}
```

— `TypeRetour` est le type de la valeur retournée ou `void` si aucune valeur n'est retournée
— Dans le corps de la méthode, on utilise l'opérateur `return` pour renvoyer une valeur
— Un constructeur a le même nom que sa classe et ne possède pas de type de retour
  — Bien que l'on puisse initialiser les attributs directement (affection lors de leur déclaration ou bloc d'initialisation), il est préférable de toujours le faire dans le constructeur
    — plus de souplesse
    — initialisations à un seul endroit

Paramètre de méthode

— `listeDeParamètres` est une liste de déclarations de variables séparées par des virgules
  — un paramètre peut être vu comme une variable locale à la méthode
  — `final` peut préfixer la déclarations si le paramètre ne doit pas être modifié
— Le passage de paramètres se fait *par valeur*
  — la valeur d'un paramètre d'un type primitif ne peut pas être modifiée
  — la valeur d'une référence ne peut pas être modifiée mais l'objet référencé peut l'être (comme avec un pointeur en C)

Exemple

Les constructeurs de la classe `Cercle2D`

```
    /**
     * Initialise un cercle avec un centre et un rayon.
     * @param centre Le centre.
     * @param rayon Le rayon.
     */
    public Cercle2D(final Point2D centre, final double rayon) {
        this.centre = centre;
        this.rayon = rayon;
    }

    /**
     * Initialise un cercle centre a l'origine et de rayon 1
     */
    public Cercle2D() {
```

```
        this(new Point2D(), 1.0);
    }
```

Listing 4.4 – Les constructeurs du cercle

— Surcharge au niveau des constructeurs
— Utilisation de `this` pour lever l'ambiguïté ou pour appeler un autre constructeur

113

Exemple

Les accesseurs de la classe ***`Cercle2D`***

```
    /**
     * Renvoie le centre du cercle.
     * @return le centre du cercle.
     */
    public Point2D getCentre() {
        return centre;
    }

    /**
     * Renvoie le rayon du cercle.
     * @return le rayon du cercle.
     */
    public double getRayon() {
        return rayon;
    }
```

Listing 4.5 – Les accesseurs du cercle

114

Exemple

Le mutateur de la classe ***`Cercle2D`***

```
    /**
     * Translate le cercle.
     * @param dx deplacement en abscisse.
     * @param dy deplacement en ordonnees.
     */
    public void translate(final double dx, final double dy) {
        centre.translate(dx, dy);
    }
```

Listing 4.6 – Le mutateur du cercle

115

Contrôle d'accès aux membres en Java

— Le contrôle de l'accès aux membres permet de spécifier l'interface d'un classe
— Le niveau d'accès est précisé en ajoutant un mot-clé devant la déclaration du membre (attribut ou méthode)
— Il peut prendre l'une des valeurs `private, public, protected` ou être absent

Niveau	Classe	Paquetage	Sous-classe	En dehors
`private`	X			
aucun	X	X		
`protected`	X	X	X	
`public`	X	X	X	X

— La restriction d'accès s'applique au niveau de la classe et non pas de l'objet
— *Les attributs sont déclarés* ***`private`*** pour respecter l'encapsulation

116

La classe `Cercle2D` dans son ensemble

```java
/**
 * Un cercle en deux dimensions.
 *
 * @version  jan. 2015
 * @author   Stephane Lopes
 *
 */
class Cercle2D {
    /** Le centre du cercle. */
    private Point2D centre;

    /** Le rayon du cercle */
    private final double rayon;

    /**
     * Initialise un cercle avec un centre et un rayon.
     * @param centre Le centre.
     * @param rayon Le rayon.
     */
    public Cercle2D(final Point2D centre, final double rayon) {
        this.centre = centre;
        this.rayon = rayon;
    }

    /**
     * Initialise un cercle centre a l'origine et de rayon 1
     */
    public Cercle2D() {
        this(new Point2D(), 1.0);
    }

    /**
     * Renvoie le centre du cercle.
     * @return le centre du cercle.
     */
    public Point2D getCentre() {
        return centre;
    }

    /**
     * Renvoie le rayon du cercle.
     * @return le rayon du cercle.
     */
    public double getRayon() {
        return rayon;
    }

    /**
     * Translate le cercle.
     * @param dx deplacement en abscisse.
     * @param dy deplacement en ordonnees.
     */
    public void translate(final double dx, final double dy) {
        centre.translate(dx, dy);
    }
}
```

Listing 4.7 – La classe `Cercle2D` dans son ensemble

Exercice

Manipulation d'objets et définition de classes

Pour cet exercice, on reprendra les spécifications obtenues lors de l'exercice précédent.

1. Soit le terrain suivant

R N		M
M		R O

(M = mur, R = robot, N = nord, O = ouest). Écrire les instructions Java permettant de créer ce terrain puis de déplacer le robot (0, 0) en (1, 1).
2. Écrire en Java la classe `Robot` qui modélise les robots.

117

4.3 Métaclasse

4.3.1 Métaclasse et membres de classe

Métaclasse

— La relation qui existe entre objet et classe est une relation d'instanciation
— De même, on peut considérer une classe comme une instance de classe de plus « haut niveau », dite *métaclasse*
— Cette notion permet d'introduire des *méthodes de classe* et des *attributs de classe*, i.e. des membres associés à la classe et non pas à une instance particulière

— Une *méthode de classe* peut être invoquée sans instance particulière
  — la création d'un objet est un exemple d'une telle méthode de classe
— Un *attribut de classe* est associé à la classe elle-même et non pas à une instance particulière
  — le nombre d'instances d'une classe est un exemple d'un tel attribut

121

Exemple
Membres de classe
(voir figure 4.3).

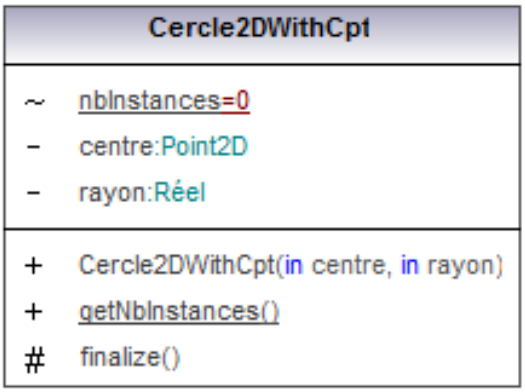


FIGURE 4.3 – Membres de classes.

— Les membres de classe sont soulignés

122

4.3.2 Membres de classe en Java

Membres de classe en Java
— Un membre de classe se déclare avec le mot-clé `static`
— *L'accès à un membre de classe se fait par l'intermédiaire de la classe*
— Pour un attribut de classe, le système alloue un espace mémoire pour un attribut par classe (et non pas un attribut par instance)
— Un attribut de classe est souvent utilisé pour définir une constante (`static final`)

```
public static final double E = 2.718281828459045d;
public static final double PI = 3.141592653589793d;
```

— L'initialisation d'un attribut de classe peut se faire directement ou en utilisant un *bloc d'initialisation statique*
— Un *bloc d'initialisation statique* est un bloc de code Java classique commençant par le mot-clé `static` et placé dans le corps de la classe
  — Permet d'éviter les limites des initialisations en une ligne
    1. l'initialisation doit pouvoir s'exprimer en une affection
    2. l'initialisation ne peut pas invoquer de méthodes susceptibles de lancer une exception vérifiée
    3. on ne peut pas faire de la reprise sur erreur en cas d'exception d'exécution
— *Une méthode de classe ne peut pas accéder aux attributs d'instance*
  — Pas de `this` dans une méthode de classe

123

Exemple
Compter les instances de cercle 1/2

```
class Cercle2DWithCpt {
    /** Le nombre d'instances de Cercle2DWithCpt */
    static int nbInstances = 0;

    /** Le centre du cercle. */
    private Point2D centre;

    /** Le rayon du cercle */
    private double rayon;

    /**
     * Initialise un cercle avec un centre et un rayon.
     * @param centre Le centre.
     * @param rayon Le rayon.
     */
    public Cercle2DWithCpt(Point2D centre, double rayon) {
        this.centre = centre;
        this.rayon = rayon;
        ++nbInstances;
    }
```

Listing 4.8 – La classe `Cercle2DWithCpt` (part. 1)

— On incrémente le compteur dans le constructeur, 124

Exemple
Compter les instances de cercle 2/2

```
    /**
     * Retourne le nombre d'instances de la classe.
     * @return le nombre d'instances.
     */
    public static int getNbInstances() {
        return nbInstances;
    }

    /**
     * Decrementation du nombre d'instances quand l'objet est detruit.
     */
    @Override
    protected void finalize() throws Throwable {
        --nbInstances;
        super.finalize();
    }
}
```

Listing 4.9 – La classe `Cercle2DWithCpt` (part. 2)

— On décrémente dans le destructeur. 125

Exemple
Invoquer une méthode de classe

```
assert Cercle2DWithCpt.getNbInstances() == 0 :
    Cercle2DWithCpt.getNbInstances();

// Creation des cercles
Cercle2DWithCpt c1 = new Cercle2DWithCpt(new Point2D(2.0, 4.0),
        5.0);
Cercle2DWithCpt c2 = new Cercle2DWithCpt(new Point2D(1.0, 2.0),
        4.0);
Cercle2DWithCpt c3 = new Cercle2DWithCpt(new Point2D(3.0, 4.0),
        2.0);

assert Cercle2DWithCpt.getNbInstances() == 3 :
    Cercle2DWithCpt.getNbInstances();

// Simple liaison
Cercle2DWithCpt unAutreC3 = c3;

assert Cercle2DWithCpt.getNbInstances() == 3 :
    Cercle2DWithCpt.getNbInstances();

// Suppression d'un instance
c1 = null; System.gc(); Thread.sleep(1000);

assert Cercle2DWithCpt.getNbInstances() == 2 :
    Cercle2DWithCpt.getNbInstances();
```

Listing 4.10 – Invoquer une méthode de classe

126

4.3.3 Le programme principal en Java

Rôle du programme principal

— Un système OO en cours d'exécution est une collection d'objets qui interagissent
— Le lancement du programme doit donc permettre d'instancier ces objets

— En général, le programme principal se limite à créer un *objet application* qui se charge d'instancier les autres objets

Le programme principal en Java

La méthode `main`

— Le point d'entrée d'une application Java est une méthode de classe nommée `main`
— Lors de l'exécution, l'interpréteur Java est invoqué avec le nom d'une classe qui doit implémenter une méthode `main`
— La déclaration de la méthode `main` est : `public static void main(String[] args)`
— Le paramètre de `main` est un tableau de chaînes de caractères contenant les *arguments de ligne de commande* passés lors de l'appel du programme
— On limite en général le code se trouvant dans le `main` au strict minimum : création d'un objet application et invocation d'une méthode

Exemple

Le programme principal en Java (avec une énumération)

```
/**
 * Represente l'application.
 *
 * @version  jan. 2015
 * @author   Stephane Lopes
 *
 */
enum ApplicationVide {
    ENVIRONNEMENT;

    /*
     * Methode principale du programme.
     * @param args les arguments de ligne de commande
     */
    public void run(String[] args) {
      // ...
    }

    /*
     * Point d'entree du programme.
     * @param args les arguments de ligne de commande
     */
    public static void main(String[] args) {
      ENVIRONNEMENT.run(args);
    }
}
```

Listing 4.11 – Le programme principal en Java

4.3.4 Énumération en Java

Complément sur le type énuméré

— Un type énuméré en Java est en fait une classe dont les instances sont connues lors de la compilation
— Les constantes d'un type énuméré peuvent être utilisées partout où un objet peut l'être
— Un type énuméré peut donc contenir des méthodes et des attributs
— De plus, le compilateur ajoute automatiquement certaines méthodes
  — `values()` retourne un tableau contenant les constantes dans l'ordre de leur déclaration
  — un type énuméré hérite implicitement de la classe `Enum`

4.4 Généricité

4.4.1 Généricité

Généricité

— La *généricité* permet de paramétrer une classe par un ou plusieurs paramètres formels (généralement des types)

— La généricité permet de définir une famille de classes, chaque classe étant instanciée lors du passage des paramètres effectifs
— Une classe paramétrée (ou générique) est donc une métaclasse particulière
  — Métaclasse étant donné que son instance est une classe
— Il peut être souhaitable de limiter les paramètres possibles : la généricité est alors *contrainte*
— Utiliser un type paramétré améliore la vérification de type lors de la compilation
— Cette notion est orthogonale au paradigme OO : on parle de *programmation générique* 131

Exemple
Un cercle générique
(voir figure 4.4).

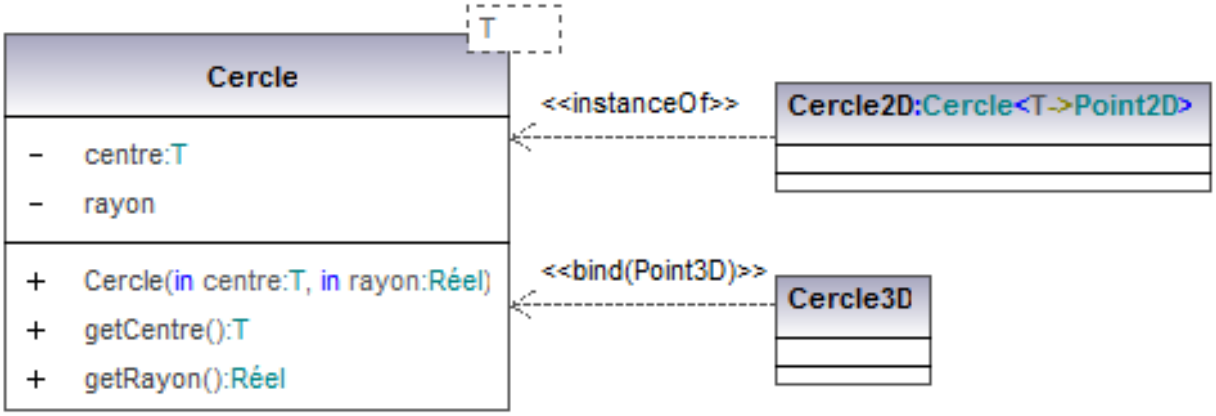


FIGURE 4.4 – Un cercle générique.

— Rectangle pointillée pour la classe paramétrée
— Deux façons de représenter les classes instanciées 132

4.4.2 Généricité en Java

Généricité en Java
— Les *paramètres formels de type* sont placés entre « < » et « > »
— Un paramètre effectif est obligatoirement une classe (pas un type primitif)
— Le mécanisme implémentant la généricité en Java se nomme *Type erasure*
— Ce mécanisme supprime toute trace de la généricité dans le bytecode ⇒ il n'existe pas d'information concernant la généricité à l'exécution 133

Classe générique en Java
— Les paramètres formels de type sont placés entre « < » et « > » juste après le nom de la classe

```
class Cercle<T> {
  //...
}
```

— Le paramètre de type peut ensuite être utilisé comme tout autre type dans la définition de la classe
— La déclaration d'une variable de ce type nécessite de passer le type effectif à la classe paramétrée

```
Cercle<Point2D> c = //...
```

— la création d'une instance nécessite également le passage du type effectif

```
Cercle<Point2D> c = new Cercle<Point2D>(/* ... */);
```

134

Exemple

Définition de la classe générique `Cercle`

```
class Cercle<T> {
    /** Le centre du cercle. */
    private T centre;

    /** Le rayon du cercle */
    private double rayon;

    /**
     * Initialise un cercle avec un centre et un rayon.
     * @param centre Le centre.
     * @param rayon Le rayon.
     */
    public Cercle(T centre, double rayon) {
        this.centre = centre;
        this.rayon = rayon;
    }

    public T getCentre() {
        return centre;
    }

    public double getRayon() {
        return rayon;
    }
}
```

Listing 4.12 – Une classe `Cercle` générique

135

Exemple

Utilisation de la classe générique `Cercle`

```
        // En 2 dimensions
  Point2D p1 = new Point2D(1.0, 2.0);
  Cercle<Point2D> c1 = new Cercle<Point2D>(p1, 3.0);

        // Invocation d'une methode de Point2D
  double xCentre = c1.getCentre().getAbscisse();

        // En 3 dimensions
  Point3D p2 = new Point3D(3.0, 4.0, 5.0);
  Cercle<Point3D> c2 = new Cercle<Point3D>(p2, 3.0);

        // Invocation d'une methode de Point3D
  double hCentre = c2.getCentre().getHauteur();
```

Listing 4.13 – Utilisation de la classe générique `Cercle`

136

Méthode générique en Java

— Il est possible de déclarer des méthodes génériques

— Le paramètre de type formel est alors précisé avant la déclaration

```
public static <T> T max(T o1, T o2) // ...
```

— La portée de ce paramètre est alors restreinte à la méthode

— L'invocation de la méthode peut préciser le type effectif ou se baser sur l'*inférence de type*

```
// Type effectif explicite
Integer i = uneClasse.<Integer>max(i1, i2);

// Type effectif déterminé par inférence de type
Integer i = uneClasse.max(i1, i2);
```

137

Généricité contrainte en Java

— Il est possible d'imposer que le paramètre de type formel soit un sous-type d'un autre type avec le mot-clé `extends`

```
public static <T extends Number> T max(T o1, T o2) // ...
```

— Le mot-clé `super` permet d'imposer que le paramètre de type formel soit un super-type d'un autre type (exemple : `<T super Number>`)
— Il peut être nécessaire d'utiliser le caractère *joker* `?` si le type effectif n'est pas connu

```
// Une boite qui peut contenir des nombres
Boite<? extends Number> b = //...

// Un boite qui peut contenir n'importe quoi
Boite<?> b = //...
```


4.5 Exercices

CONTRAINTES
— Utilisation de l'environnement de développement BLUEJ.
— Chaque exercice nécessite la création d'un nouveau projet sous BLUEJ.
— Les seules sources d'information externes autorisées sont : (i) la référence de l'API Java, (ii) le tutoriel Java, (iii) la documentation de BLUEJ.
— Exécution interactive uniquement avec BLUEJ (pas de méthode `main` ni d'affichage).

Exercice 4.1 (Création d'une classe simple)
L'objet de cet exercice est de réaliser une classe `ChaineCryptee` qui permettra de chiffrer/déchiffrer une chaîne de caractères (composée de lettres majuscules et d'espace). Le chiffrement utilise une méthode par décalage. La valeur du décalage représente la clé de chiffrement. Par exemple, une clé de valeur trois transformera un `'A'` en `'D'`.

1. Créez la classe `ChaineCryptee` avec pour attribut la chaîne **en clair** et le décalage.
2. Ajoutez un constructeur pour initialiser les instances à partir d'une chaîne en clair et du décalage.
3. Ajoutez la méthode `String decrypte()` qui retourne la chaîne en clair.
4. Ajoutez la méthode `String crypte()` qui retourne la chaîne chiffrée. Vous pourrez utilisez pour cela la méthode `decaleCaractere` (voir le listing 4.14).

```
/**
 * Décale un caractère majuscule.
 * Les lettres en majuscule ('A' - 'Z') sont décalés de <code>decalage</code>
 * caractères de façon circulaire. Les autres caractères ne sont pas modifiés.
 *
 * @param c le caractère à décaler
 * @param decalage le décalage à appliquer
 * @return le caractère décalé
 */
private static char decaleCaractere(char c, int decalage) {
    return (c < 'A' || c > 'Z')? c : (char)(((c - 'A' + decalage) % 26) + 'A');
}
```

Listing 4.14 – Méthode `decaleCaractere`

5. Comment se comporte votre classe si la chaîne passée au constructeur est `null` ? Vous pouvez utiliser le débogueur pour identifier le problème (s'il y a un problème) au niveau de `crypte`.
   (a) si la méthode a échoué, modifiez la classe pour prendre en compte la chaîne `null`,
   (b) vérifiez avec le débogueur que le problème est réglé.
6. Faites afficher l'interface de la classe (la documentation `JavaDoc`). Le comportement de votre classe en cas de chaîne `null` doit être expliqué dans la documentation.
7. Changez la représentation interne de la classe : seule la chaîne cryptée est stockée (plus la chaîne en clair).
   (a) effectuez les modifications nécessaires sans changer l'interface de la classe.
8. Ajoutez la possibilité d'initialiser une instance à partir d'une chaîne cryptée et d'un décalage. Pour éviter l'ambiguïté au niveau du constructeur, vous utiliserez le modèle de conception FABRICATION. Pour cela,

(a) créez les méthodes de classe `ChaineCryptee deCryptee(String, int)` et `ChaineCryptee deEnClair(String, int)`,

(b) rendez le constructeur privé.

La création des instances se fait maintenant à l'aide des deux méthodes de classe.

A la fin de cet exercice, la classe `ChaineCryptee` doit avoir la structure suivante (voir figure 4.5).

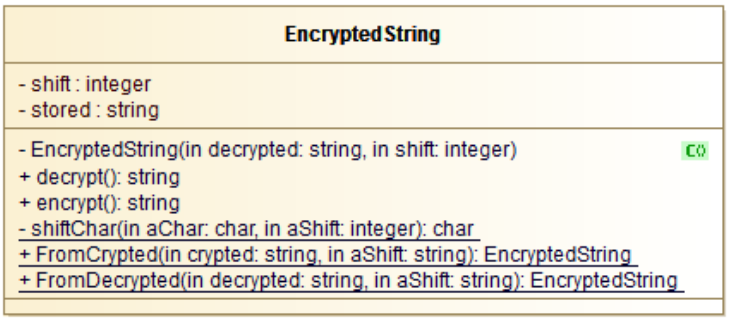


FIGURE 4.5 – La classe `ChaineCryptee`.

Exercice 4.2 (Classes et associations)
On souhaite **simuler** le comportement d'un logiciel de discussion client/serveur. Pour pouvoir discuter, un client doit au préalable se connecter au serveur. Lorsque le client envoie un message au serveur, ce dernier le transmet à tous les clients avec le nom de l'émetteur.

Un diagramme de classes possible pour cet énoncé est donné par la figure suivante (voir figure 4.6).

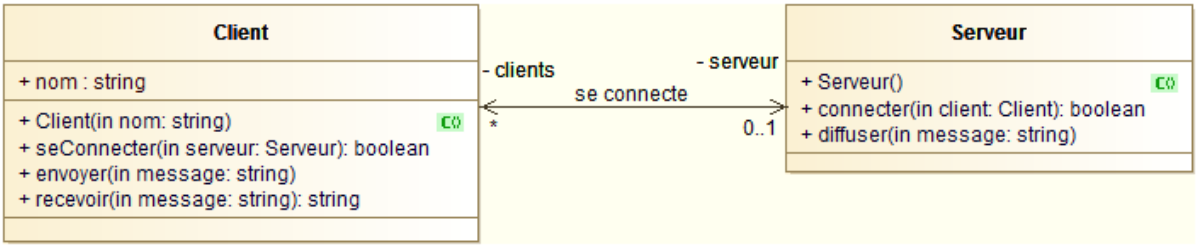


FIGURE 4.6 – Diagramme de classes pour le chat.

1. Donnez un diagramme de communication UML décrivant le scénario suivant :
   (a) le serveur s est actuellement connecté avec les clients c_1 et c_2,
   (b) le client c_3 se connecte à s,
   (c) le client envoie le message « Bonjour » aux clients connectés.
2. Implémentez les classes `Client` et `Serveur`.
3. déroulez le scénario de la question 1 dans BLUEJ.

Exercice 4.3 (Agrégation et composition)
Un document est décrit par un titre, le nom de son auteur, l'année de publication et une liste de références sur d'autres documents (par exemple, des documents traitant du même sujet). Une bibliothèque gère un ensemble de documents.

Les opérations que l'on souhaite pouvoir effectuer sont les suivantes :
— initialisation d'un document avec son titre, l'auteur et une année,
— ajout d'une référence dans un document,

— affichage (construire une chaîne de caractères le représentant) d'un document (avec ses références sous la forme *titre, auteur*),
— ajout d'un document dans la bibliothèque,
— recherche par titre d'un document dans la bibliothèque,
— recherche des documents citant un document.

1. Modélisez cette énoncé à l'aide d'un diagramme de classe.
2. Implémentez ce modèle.

Chapitre 5

Héritage en Java

Sommaire

5.1 Héritage

Héritage en Java

— On spécifie qu'une classe est une sous-classe d'une autre en utilisant `extends` dans la déclaration `class NomClasse extends NomSuperClasse //...`
— Une classe ne peut avoir qu'une seule super-classe
— Si `extends` n'est pas précisé, la classe hérite de la classe `Object`
⇒ *une classe Java a une et une seule super-classe*
— Une classe déclarée `final` ne peut plus être spécialisée (pour des raisons de conception et/ou de sécurité)

139

Héritage et membres

— Une classe C hérite de sa super-classe S les attributs et méthodes qu'elle possède
  — tous les attributs de S font parti de l'état des instances de C
  — les attributs privées de S ne sont pas accessibles dans C (mais font parti des instances de C)
  — les méthodes non privées de S sont accessibles dans C
  — les méthodes publiques de S font parti de l'interface publique de C
  — les constructeurs de S sont utilisables dans les constructeurs de C mais ne font pas parti de l'interface publique de C

140

Masquage de membres

— Une classe peut masquer un membre de sa super-classe si elle possède un membre de même nom (ou de même signature)
— Le mot clé `super` permet d'accéder aux membres masqués d'une super-classe

141

Exemple

Rectangle et rectangle plein

(voir figure 5.1).

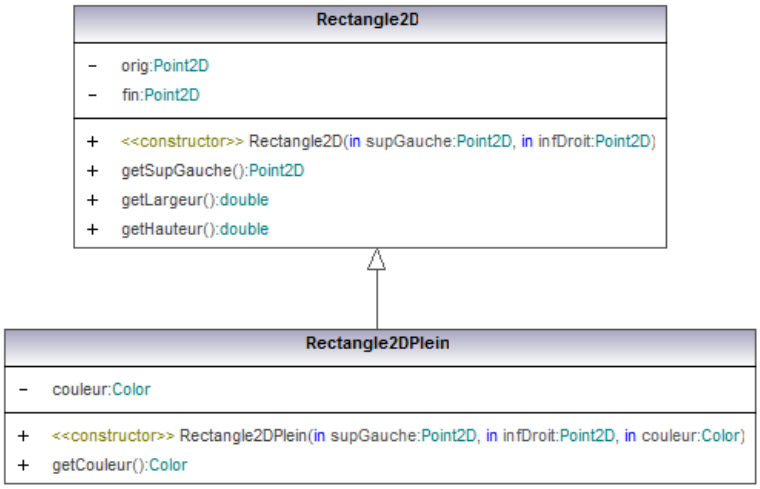


FIGURE 5.1 – Héritage entre rectangle plein et rectangle.

142

Exemple

Définition de la classe `Rectangle2DPlein` en Java

```
class Rectangle2DPlein extends Rectangle2D {
    /** La couleur de remplissage */
    private Color couleur;

    /**
     * Initialise le rectangle plein.
     * @param supGauche Le coin supérieur gauche.
     * @param infDroit Le coin inférieur droit.
     * @param couleur La couleur de remplissage.
     */
    public Rectangle2DPlein(Point2D supGauche,
            Point2D infDroit,
            Color couleur) {
        super(supGauche, infDroit);
        assert couleur != null;
        this.couleur = couleur;
    }

    /**
     * Renvoie la couleur.
     * @return la couleur.
     */
    public Color getCouleur() { return couleur; }
}
```

Listing 5.1 – la classe `Rectangle2DPlein` en Java

— `extends` exprime l'héritage
— seule les attributs supplémentaires sont déclarés dans la sous-classe
— dans le constructeur, `super` permet d'appeler le constructeur de la super-classe
— seule les méthodes supplémentaires sont définies dans la sous-classe

143

Exemple

Utilisation de la classe `Rectangle2DPlein`

```
// Déclaration et création d'un rectangle rouge
        Rectangle2DPlein rp1 = new Rectangle2DPlein(new Point2D(1.0, 2.0),
                                                    new Point2D(3.0, 0.0),
                                                    Color.RED);
        assert rp1.getCouleur() == Color.RED;
```

```
// Déclaration d'un rectangle et
// liaison avec un rectangle plein
Rectangle2D r1 = new Rectangle2DPlein(new Point2D(1.0, 2.0),
                                      new Point2D(2.0, 1.0),
                                      Color.YELLOW);
assert r1.getLargeur() == 1;
```

Listing 5.2 – Utilisation de la classe `Rectangle2DPlein`

— L'affectation d'une instance de `Rectangle2DPlein` à une références sur un `Rectangle2D` respecte le *principe de substitution de Liskov*
— Á partir de `r1`, l'accès à `getCouleur` est impossible (échoue à la compilation) car `getCouleur` ne fait pas parti de l'interface de `Rectangle2D`

144

Exercice
L'héritage

Deux nouveaux types de robot sont créés :
— les transporteurs qui peuvent ramasser un objet, le transporter et le déposer,
— les destructeurs qui peuvent détruire ce qui se trouve sur la case devant eux.

1. Modéliser les robots
2. Soit le terrain suivant

T S	M	
D E	M	
	M	

(M = mur, T = transporteur, D = destructeur, S = sud, E = est). Écrire les instructions permettant de créer ce terrain puis de déplacer l'objet se trouvant en (0, 0) en (2, 2).
3. Donner l'implémentation des deux classes `Transporteur` et `Destructeur`.

145

5.2 Polymorphisme

5.2.1 Polymorphisme en Java

Redéfinition de méthode
— Une sous-classe peut *redéfinir* (*override*) une ou plusieurs méthodes de sa super-classe
— La *redéfinition* (*overriding*) consiste à définir dans une sous-classe, une méthode ayant même signature et même type de retour qu'une méthode de la super-classe
  — la méthode de la super-classe est alors masquée
  — il est toujours possible d'appeler la méthode redéfinie en utilisant le mot-clé `super`

151

Redéfinition et polymorphisme
— Le polymorphisme est le mécanisme permettant de sélectionner la méthode redéfinie appropriée
— La redéfinition ne permet plus au compilateur de sélectionner la méthode adéquat
— C'est le type de l'objet (et non pas de la référence) qui permettra de déterminer la méthode à invoquer et ce type ne peut être connu qu'au moment de l'exécution

152

Compléments sur la redéfinition de méthode
— La déclaration de la méthode redéfinie est toujours précédée de l'annotation `@Override`
— Le contrôle d'accès peut être relaxé lors de la redéfinition
— Une méthode déclarée `final` ne peut pas être redéfinie
— Une méthode de classe ne peut pas être redéfinie

153

Exemple
Une description pour le rectangle et le rectangle plein
(voir figure 5.2).

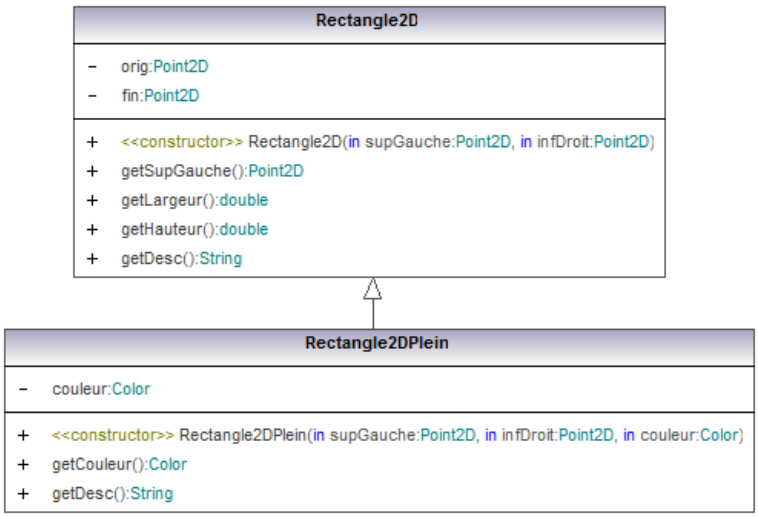


FIGURE 5.2 – Polymorphisme entre rectangle plein et rectangle.

154

Exemple
La classe `Rectangle2D` 1/2

```
class Rectangle2D implements Cloneable {
    /** Coordonnées du coin supérieur gauche */
    private Point2D orig;

    /** Coordonnées du coin inférieur droit */
    private Point2D fin;

    /**
     * Initialise le rectangle.
     * @param supGauche Le coin supérieur gauche.
     * @param infDroit Le coin inférieur droit.
     */
    public Rectangle2D(Point2D supGauche, Point2D infDroit) {
        assert supGauche.getAbscisse() <= infDroit.getAbscisse() &&
               supGauche.getOrdonnee() >= infDroit.getOrdonnee();
        orig = supGauche;
        fin = infDroit;
    }

    public Point2D getSupGauche() { return orig; }

    public double getLargeur() {
        return fin.getAbscisse() - orig.getAbscisse();
    }

    public double getHauteur() {
        return orig.getOrdonnee() - fin.getOrdonnee();
    }
```

Listing 5.3 – La classe `Rectangle2D` (part. 1)

155

Exemple
La classe `Rectangle2D` 1/2

```
    /**
     * Retourne une description du rectangle.
     * @return la description.
     */
    public String getDesc() {
        StringBuilder str = new StringBuilder();
        str.append("O = (");
        str.append(orig.getAbscisse());
        str.append(", ");
        str.append(orig.getOrdonnee());
        str.append(") L = "); str.append(getLargeur());
        str.append(" H = "); str.append(getHauteur());
        return str.toString();
    }
}
```

Listing 5.4 – La classe `Rectangle2D` (part. 2)

156

Exemple
La classe `Rectangle2DPlein` 1/2

```
class Rectangle2DPlein extends Rectangle2D {
    /** La couleur de remplissage */
    private Color couleur;

    /**
     * Initialise le rectangle plein.
     * @param supGauche Le coin supérieur gauche.
     * @param infDroit Le coin inférieur droit.
     * @param couleur La couleur de remplissage.
     */
    public Rectangle2DPlein(Point2D supGauche,
          Point2D infDroit,
          Color couleur) {
        super(supGauche, infDroit);
        assert couleur != null;
        this.couleur = couleur;
    }

    /**
     * Renvoie la couleur.
     * @return la couleur.
     */
    public Color getCouleur() { return couleur; }
```

Listing 5.5 – La classe `Rectangle2DPlein` (part. 1)

157

Exemple
La classe `Rectangle2DPlein` 2/2

```
    /**
     * Retourne une description du rectangle plein.
     * @return la description.
     */
    @Override
    public String getDesc() {
        StringBuilder str = new StringBuilder(super.getDesc());
        str.append(", couleur : (");
        str.append(couleur.getRed());
        str.append(", ");
        str.append(couleur.getGreen());
        str.append(", ");
        str.append(couleur.getBlue());
        str.append(")");
        return str.toString();
    }
}
```

Listing 5.6 – La classe `Rectangle2DPlein` (part. 2)

158

Exemple
Utilisation du polymorphisme

```
        // Création d'un tableau de références sur des Rectangle2D
        final int NB_RECTANGLES = 2;
        Rectangle2D[] figures = new Rectangle2D[NB_RECTANGLES];

        // Un rectangle
        figures[0] = new Rectangle2D(new Point2D(0.0, 5.0),
             new Point2D(2.0, 2.0));

        // Un rectangle plein
        figures[1] = new Rectangle2DPlein(new Point2D(1.0, 3.0),
            new Point2D(3.0, 2.0),
            Color.BLUE);

        // getDesc() de Rectangle2D
        assert figures[0].getDesc().equals("O = (0.0, 5.0) L = 2.0 H = 3.0");

        // getDesc() de Carre2D
        assert figures[1].getDesc().equals("O = (1.0, 3.0) L = 2.0 H = 1.0, couleur : (0, 0, 255)");
```

Listing 5.7 – Utilisation du polymorphisme en Java

159

Exercices
Le polymorphisme et la redéfinition

Une amélioration importante a été apportée aux robots : ils sont maintenant programmables. Chaque type de robot possède son propre comportement. Quand ils en reçoivent l'ordre, ils exécutent l'action programmée (un ensemble d'instructions élémentaires). Plusieurs robots de types différents se trouvent sur le terrain. On veut pouvoir déclencher l'exécution du programme de l'ensemble des robots.

1. Modéliser cet énoncé
2. Implémenter les changements dans les classes `Robot`, `Transporteur` et `Destructeur`
3. Écrire un programme déclenchant l'exécution de l'action pour l'ensemble des robots

160

5.2.2 La classe `Object`

La classe `Object`

— La classe `Objet` définit et implémente le comportement dont chaque classe Java a besoin
— C'est la plus générale des classes Java
— Chaque classe Java hérite directement ou indirectement de `Object` (tout objet y compris les tableaux implémente les méthodes de `Object`)

167

Les méthodes de la classe `Object`

— Certaines méthodes de `Object` peuvent être redéfinies pour s'adapter à la sous-classe
  — `protected Object clone()` permet de dupliquer un objet
  — `boolean equals(Object obj)` permet de tester l'égalité de deux objets et `int hashCode()` de renvoyer une valeur de hashage
    — `Object.equals` teste l'identité
    — `equals` et `hashCode` doivent être redéfinies ensembles
  — `protected void finalize()` représente le destructeur d'un objet
  — `String toString()` retourne une chaîne représentant l'objet
    — `toString` est très utile pour le débogage ⇒ toujours la redéfinir
— Autres méthodes
  — `Class getClass()` retourne un objet de type `Class` représentant la classe de l'objet
    — La classe `Class` est par exemple utile pour créer des objets dont la classe n'est pas connu à la compilation
  — quelques méthodes pour les *threads*

168

Exemple

Redéfinition de la méthode `toString()` de `Rectangle2D`

```
/**
 * Retourn une chaîne représentant l'objet.
 * @return la chaîne.
 */
@Override
public String toString() {
    StringBuilder str = new StringBuilder();
    str.append("O = "); str.append(orig.toString());
    str.append(" L = "); str.append(getLargeur());
    str.append(" H = "); str.append(getHauteur());
    return str.toString();
}
```

Listing 5.8 – Méthode `toString()` de `Rectangle2D`

169

Exemple

Redéfinition de la méthode `toString()` de `Rectangle2DPlein`

```
/**
 * Retourn une chaîne représentant l'objet.
 * @return la chaîne.
 */
@Override
public String toString() {
    StringBuilder str = new StringBuilder(super.toString());
    str.append(" couleur = "); str.append(couleur.toString());
    return str.toString();
}
```

Listing 5.9 – Méthode `toString()` de `Rectangle2DPlein`

```
// Test de toString
        assert figures[0].toString().equals("O = (0.0, 5.0) [5.0, 0.0] L = 2.0 H = 3.0");
        assert figures[1].toString().equals("O = (1.0, 3.0) [3.1622776601683795, 0.3217505543966422] L = 2.0 H = 1.0
    couleur = java.awt.Color[r=0,g=0,b=255]");
```

Listing 5.10 – Appel de `toString()`

170

Copie d'objet

Le mécanisme de clonage

- Le *clonage* d'objet est destinée à créer une *copie profonde* (*deep copy*) d'un objet
  - un objet peut contenir des références vers d'autres objets (récursif)
  - l'opération de copie peut avoir différentes sémantiques
- Le clonage est réalisé en redéfinissant en `public` la méthode protégée `clone` de `Object`
- La méthode `clone` de `Object` doit être appelée pour réserver la mémoire et effectuer une *copie superficielle* (*shallow copy*)
  - `Object.clone` alloue la mémoire et effectue la copie de tous les bits de l'objet
- L'interface `Cloneable`, qui ne comporte aucune méthodes, doit aussi être implémentée
  - `Cloneable` sert simplement d'indicateur
- Les conditions suivantes doivent être vérifiées :
  - la copie et l'objet ne sont pas identiques (`x.clone() != x`),
  - la copie et l'objet sont de même classe (`x.clone().getClass() == x.getClass()`),
  - la copie et l'objet sont égaux (`x.clone().equals(x)`).
- Rappel : l'affectation d'un objet a une variable ne crée pas de copie de l'objet mais juste une nouvelle référence vers l'objet

171

Exemple

Clonage d'un rectangle

```
    /**
     * Retourne une copie "profonde" de l'objet.
     * @return la copie.
     */
    @Override
    public Object clone() throws CloneNotSupportedException {
        Rectangle2D r = (Rectangle2D)super.clone();
        r.orig = (Point2D)orig.clone();
        r.fin = (Point2D)fin.clone();
        return r;
    }
```

Listing 5.11 – Redéfinition de `clone` dans `Rectangle2D`

```
// Test du clonage d'objets
        Rectangle2D copie = (Rectangle2D)figures[0].clone();
        assert copie != figures[0]; // Pas identiques
        assert copie.getClass() == figures[0].getClass(); // Même classe
assert copie.equals(figures[0]);                          // Egaux
```

Listing 5.12 – Utilisation de `clone`

172

Egalité d'objets

La méthode ***`equals`***

- La méthode `boolean equals(Object o)` teste l'égalité de deux objets
- La méthode `equals` de la classe `Object` se content de tester l'égalité des références des objets, i.e. l'identité
- Il est donc en général nécessaire de redéfinir `equals` pour le test d'égalité
- Rappel : l'opérateur `==` teste l'identité de ses opérandes, i.e. l'égalité des références

173

Egalité d'objets

Contraintes de ***`equals`***

- `equals` implémente une relation d'équivalence pour des références d'objet non nulles

  - `x.equals(x) == true`
  - `x.equals(y) == true` si et seulement si `y.equals(x) == true`
  - si `x.equals(y) == true` et `y.equals(z) == true` alors `x.equals(z) == true`
  - `x.equals(null) == false`
- Toute classe qui redéfinit `equals` doit également redéfinir `hashCode()`
  - si deux objets sont égaux au sens de `equals` alors `hashCode` doit produire le même résultat pour les deux objets

174

Exemple
Egalité de rectangles

```
/**
 * Teste l'égalité de deux rectangles.
 * @param obj le rectangle à comparer.
 * @return true si les objets sont égaux.
 */
@Override
public boolean equals(Object obj) {
    if (obj instanceof Rectangle2D) {
        Rectangle2D r = (Rectangle2D)obj;
        return orig.equals(r.orig) && fin.equals(r.fin);
    }
    return false;
}

/*
 * Retourne une valeur de hashage pour l'objet.
 * @return la valeur de hashage.
 */
@Override
public int hashCode() {
    return orig.hashCode() ^ fin.hashCode();
}
```

Listing 5.13 – Redéfinition de `equals` et `hashCode()` dans `Rectangle2D`

175

Exemple
Contraintes de ***`equals`***

```
// Test de l'égalité
Rectangle2D r1 = new Rectangle2D(new Point2D(0.0, 5.0),
        new Point2D(2.0, 2.0));
Rectangle2D r2 = new Rectangle2D(new Point2D(0.0, 5.0),
        new Point2D(2.0, 2.0));
Rectangle2D r3 = new Rectangle2D(new Point2D(0.0, 5.0),
        new Point2D(2.0, 2.0));
assert r1.equals(r1); // Réflexivité
assert r1.equals(r2) && r2.equals(r1); // Symétrie
assert r1.equals(r2) && r2.equals(r3) &&
    !r1.equals(r3) == false; // Transitivité
assert r1.equals(null) == false;
assert r1.hashCode() == r2.hashCode();
```

Listing 5.14 – Contraintes de `equals`

176

5.3 Classe abstraite

5.3.1 Classe abstraite en Java

Classe abstraite en Java

- Une classe est spécifiée abstraite en ajoutant le mot-clé `abstract` dans sa déclaration
- L'instanciation d'une telle classe est alors refusée par le compilateur
- Une classe abstraite contient généralement des *méthodes abstraites*, i.e. qui ne possèdent pas d'implémentation
  - une classe abstraite peut cependant ne pas avoir de méthodes abstraites
- Une méthode est déclarée abstraite en utilisant le mot-clé `abstract` lors de sa déclaration
- Toute sous-classe non abstraite d'une classe abstraite doit redéfinir les méthodes abstraites de cette classe
- Une classe possédant des méthodes abstraites est obligatoirement abstraite

177

Exemple
La hiérarchie d'héritage des figures
(voir figure 5.3).

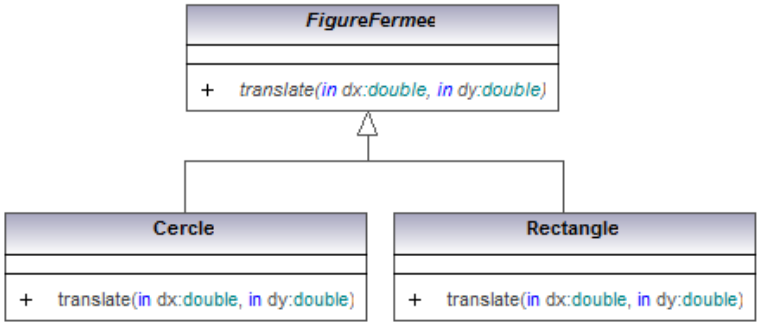


FIGURE 5.3 – Hiérarchie d'héritage des figures.

178

Exemple
La classe abstraite ***FigureFermee2D***

```
/**
 * Une figure fermée.
 *
 * @version  oct. 2008
 * @author   Stéphane Lopes
 *
 */
abstract class FigureFermee2D {
    /**
     * Translate la figure.
     * @param dx déplacement en abscisse.
     * @param dy déplacement en ordonnées.
     */
    public abstract void translate(double dx, double dy);
}
```

Listing 5.15 – La classe abstraite `FigureFermee2D`

179

Exemple
Redéfinition de la translation

```
    /**
     * Translate le rectangle.
     * @param dx déplacement en abscisse.
     * @param dy déplacement en ordonnées.
     */
    @Override
    public void translate(double dx, double dy) {
        orig.translate(dx, dy);
        fin.translate(dx, dy);
    }
```

Listing 5.16 – Transaltion dans la classe `Rectangle2D`

```
    /**
     * Translate le cercle.
     * @param dx déplacement en abscisse.
     * @param dy déplacement en ordonnées.
     */
    @Override
    public void translate(double dx, double dy) {
        centre.translate(dx, dy);
    }
```

Listing 5.17 – Transaltion dans la classe `Cercle2D`

180

Exemple
Utilisation de la classe abstraite ***FigureFermee2D***

```
// Création du tableau de références
      final int NB_FIGURES = 4;
      FigureFermee2D[] figures = new FigureFermee2D[NB_FIGURES];

// Création des formes
      figures[0] = new Rectangle2D(new Point2D(0.0, 5.0),
           new Point2D(2.0, 2.0));
      figures[1] = new Cercle2D(new Point2D(1.0, 2.0), 3.0);
      figures[2] = new Rectangle2D(new Point2D(5.0, 5.0),
           new Point2D(7.0, 3.0));
      figures[3] = new Cercle2D(new Point2D(4.0, 5.0), 2.0);

// Réalise une translation de la figure
      for (int i = 0; i < figures.length; ++i) {
          figures[i].translate(1.0, 2.0);
      }
```

Listing 5.18 – Utilisation de la classe abstraite `FigureFermee2D`

181

Exercices

Les classes abstraites

On souhaite maintenant ajouter sur le terrain un type de case sensible à la charge (pont par exemple). Chaque élément mobile (robot ou objet transportable) devra donc posséder un poids. Le poids d'une instance de robot sera toujours de 1 unité. Une instance de destructeur aura un poids de 2 unités de plus que le robot. Une instance de transporteur aura un poids de 1 unités de plus que le robot auquel il faudra ajouter le poids de l'objet transporté. Le poids par défaut des objets transportables est de 1 unité mais chaque objet pourra avoir un poids différent précisé lors de sa création. On souhaite pouvoir connaître le poids de tout élément se trouvant sur le terrain.

1. Modéliser la prise en compte du poids au niveau des éléments mobiles
2. Implémenter les classes correspondantes

182

5.3.2 La classe `Number`

La classe `Number` et les classes Adaptateur

- La classe `Number` est une classe abstraite de la librairie Java
- Elle définit le comportement commun aux classes pour la gestion des nombres (les conversions)
  - exemples de méthodes de `Number` : `floatValue`, `intValue`, ...
- Elle possède plusieurs sous-classes
  - les classes Adaptateur : `Byte`, `Double`, `Float`, `Integer`, `Long`, `Short`
  - `BigDecimal`, `BigInteger`
- Ces classes offrent une vue « objet » des types primitifs
- Il existe d'autres classes Adaptateur : `Boolean`, `Character`, `Void`
  - Ces classes n'héritent pas de `Number`
- Les sous-classe de `Number` sont des exemples du pattern de conception Adaptateur

Remarque

La plupart des fonctions arithmétiques sont des méthodes de classe de la classe `Math`.

192

autoboxing/autounboxing

- Ce mécanisme permet d'éviter la conversion manuelle entre type primitif et classe Adaptateur
- C'est simplement une facilité d'écriture (*sucre syntaxique*)

Avant JDK 1.5

```
Integer i = Integer.valueOf(12); // préférable à new Integer(12)
int n = i.intValue();
```

`Integer.valueOf(int i)` peut placer en cache les valeurs fréquemment utilisées.

A Partir du JDK 1.5

```
Integer i = 12;
int n = i;
```

193

5.4 Interface

5.4.1 Définition

Interface

Du point de vue des types

- Une *interface* regroupe uniquement des signatures d'opérations et des déclarations de constantes mais aucune implémentation
- Une interface permet donc de définir un type
- Les interfaces peuvent être organisées en hiérarchies d'héritage
- Un lien d'héritage entre interface est une relation de sous-typage

194

Interface

Du point de vue des services

- Une *interface* fournit une vue totale ou partielle d'un ensemble de services offerts par une classe (un composant)
- Une interface est analogue à un protocole de comportement (un contrat sur un comportement)
- Une classe peut *implémenter* une ou plusieurs interfaces
- Une interface est formellement équivalente à une classe abstraite ne possédant que des méthodes abstraites
  - une interface ne peut pas implémenter de méthodes, une classe abstraite le peut
  - une interface ne fait pas partie de la hiérarchie de classes donc même des classes qui ne sont pas en relation peuvent implémenter la même interface

195

Utilisation d'une interface

- Permet à des objets d'interagir même s'ils ne sont pas en relation
- Permet de capturer des similarités entre classes non reliées sans définir artificiellement une relation
- Permet de déclarer des méthodes qu'une ou plusieurs classes doivent implémenter
- Permet de révéler l'interface de programmation d'un objet sans révéler sa classe

196

Exemple

Interface et implémentation d'interface

(voir figure 5.4).

- Trois classes (LecteurDVD, Magnétoscope, Télécommande), deux interfaces (Lecture, Enregistrement)
- Sous forme de classe avec le mot clé « `interface` » (on visualise la liste des opérations, évidemment pas d'attributs)
- LecteurDVD et Magnétoscope implémentent l'interface Lecture
- Magnétoscope implémente Enregistrement
- Télécommande utilise les deux interfaces

197

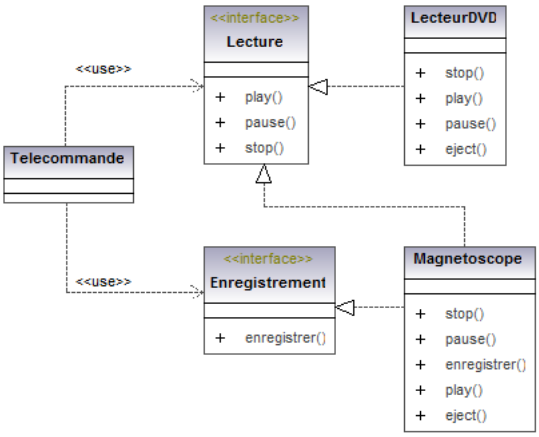


FIGURE 5.4 – Vue détaillée des interfaces.

Exemple
Interface et implémentation d'interface
(voir figure 5.5).

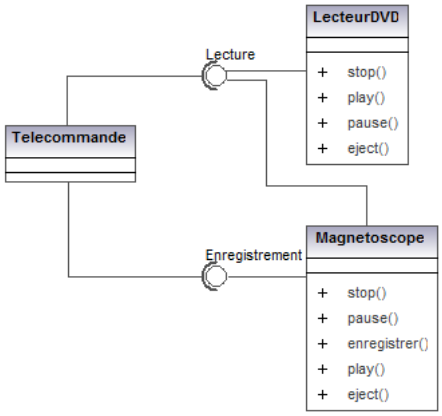


FIGURE 5.5 – Vue simplifiée des interfaces.

— Sous forme d'un trait terminé par un cercle (on visualise juste le nom de l'interface) 198

5.4.2 Interface en Java

Définition d'une interface

— La définition d'une interface comporte une déclaration et un corps

```
interface UneInterface extends UneSecondeInterface, UneAutreInterface {
  final String uneChaine = "abcde";
  final double unDouble = 123.456;
  void uneMethode(int unEntier, String uneChaine);
}
```

— Une interface peut avoir plusieurs super-interfaces
— Toutes les méthodes de l'interface sont implicitement `public` et `abstract`
— Toutes les constantes de l'interface sont implicitement `public`, `static`, et `final` 199

Utilisation d'une interface

— Pour déclarer une classe qui implémente une ou plusieurs interfaces, on ajoute `implements ListeInterfaces` dans sa déclaration (après la clause `extends` si elle existe)
— La classe doit alors implémenter toutes les méthodes de l'interface ou être déclarée abstraite
— Une interface est utilisée comme un type (par exemple comme paramètre d'une méthode) 200

Exemple
*Définir l'*ordre naturel *des objets : l'interface* `Comparable`

```
// Comparable est une interface de la librairie Java

/**
 * This interface imposes a total ordering on the objects
 * of each class that implements it. ...
 */
interface Comparable<T> {
  /**
   * Compares this object with the specified object for order. ...
   */
  public int compareTo(T o);
}

// ...
class Rectangle2D implements Comparable<Rectangle2D> {
  // ...
  public int compareTo(Rectangle2D o) {
    // Code pour la comparaison
  }
  // ...
}
```


5.5 Exercices

CONTRAINTES
- Utilisation de l'environnement de développement BLUEJ.
- Chaque exercice nécessite la création d'un nouveau projet sous BLUEJ.
- Les seules sources d'information externes autorisées sont : (i) la référence de l'API Java, (ii) le tutoriel Java, (iii) la documentation de BLUEJ.

Exercice 5.1 (Héritage et polymorphisme)
On souhaite réaliser une application pour gérer une collection de CD et de DVD. Un CD possède un titre, le nom de l'artiste ou du groupe, et le nombre de titres. Un DVD possède un titre, un réalisateur et une année de sortie.

L'application devra permettre de :
- créer des documents (CD ou DVD),
- consulter les informations d'un document,
- ajouter un document dans la collection,
- lister les documents de la collection,
- rechercher les documents contenant un mot clé dans le titre ou dans le groupe/réalisateur.

1. Donner un diagramme de classe UML modélisant ce problème.
2. Proposer une implémentation de cette modélisation.

Exercice 5.2 (Création d'une classe respectant les conventions du langage Java)
L'objectif de cet exercice est de réaliser une classe **immuable** `Fraction` qui représente un nombre rationnel. Un exemple d'interface pour une telle classe est donné par la classe `Fraction` de la bibliothèque Apache Commons Math.

Une fraction comporte un numérateur et un dénominateur (nombres entiers).

CONTRAINTES Vous respecterez les conventions Java quand cela est approprié (égalité, conversion, comparaison, ...).

Implémentez la classe en fournissant l'interface suivante :

1. initialisation avec (i) un numérateur et un dénominateur, (ii) juste avec le numérateur (dénominateur égal à 1) ou (iii) sans argument (numérateur égal 0 et dénominateur égal à 1),
2. les constantes ZERO (0, 1) et UN (1, 1),
3. consultation du numérateur et du dénominateur,

4. consultation de la valeur sous la forme d'un nombre en virgule flottante (`double`),
5. addition de deux fractions,
6. test d'égalité entre fractions (deux fractions sont égales si elles représentent la même fraction réduite),
7. conversion en chaîne de caractères,
8. comparaison de fractions selon l'ordre naturel.

Exercice 5.3 (Héritage, polymorphisme et classe abstraite)
On veut simuler le fonctionnement d'un système de fichiers. Un fichier est représenté par son nom et sa taille. Un répertoire est défini par son nom et peut contenir des fichiers et/ou des répertoires. La base de l'arborescence du système de fichier est le répertoire racine.

On veut pouvoir calculer la taille totale d'un répertoire.

1. Représenter sur un diagramme de classes UML une hiérarchie de classe modélisant ce problème. Le modèle de conception Composite peut être utilisé pour cela.
2. Proposer une implémentation de ce modèle.
3. Créer une arborescence de test et vérifier le calcul de la taille.
4. Modifier le programme pour qu'un répertoire ne puisse pas être ajouté à lui-même.
5. Modifier le programme pour qu'un répertoire ne puisse pas être ajouté comme descendant de lui-même.

Exercice 5.4 (Utilisation de la ligne de commande)
Le but de cet exercice est de vous familiariser avec les outils Java en ligne de commande.

1. Créer le répertoire `FileSystem` contenant les sous-répertoires `src` et `classes`.
2. Placer les fichiers `.java` de l'exercice précédent dans le sous-répertoire `src`.
3. Compiler les sources en incluant les informations de débogage et de façon à placer les `.class` dans le répertoire `classes` (cf. documentation de `javac`).
4. Ajouter un programme principal contenant le code de test de l'exercice précédent et recompiler.
5. Exécuter le programme (cf. documentation de `java`).
6. Créer une archive Java pour le programme (cf. documentation de `jar`) et exécuter à nouveau le programme à partir de l'archive.

Chapitre 6

Module en Java

Sommaire

6.1 Relations entre classes

6.1.1 Introduction

Quatre types de relations entre classes

Association relation structurelle

Spécialisation/généralisation relation d'héritage ou de sous-typage (voir le chapitre précédent)

Réalisation relation entre une classe et une interface (voir le chapitre précédent)

Dépendance liaison limitée dans le temps entre objets (non structurelle)

202

6.1.2 Association

Association

— Une *association* représente une connexion sémantique bidirectionnelle entre une (*association réflexive*) ou plusieurs classes

— L'*arité* d'une association représente le nombre de participants à l'association (association *binaire*, *ternaire*, *n-aire*)

203

Exemple 1/4

Une association binaire

(voir figure 6.1).

204

Exemple 2/4

Représentation de l'association binaire en Java

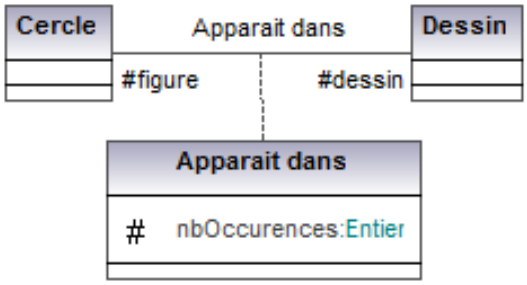


FIGURE 6.1 – Association binaire entre `Cercle` et `Dessin`.

```
class ApparaitDans {
  private Cercle figure;
  private Dessin dessin;
  private int nbOccurences;

  public ApparaitDans(Cercle figure, Dessin dessin, int nbOccurences) {
    this.figure = figure;
    this.dessin = dessin
    this.nbOccurences = nbOccurences;
  }
  // ...
}
```

205

Exemple 3/4

Une association ternaire

(voir figure 6.2).

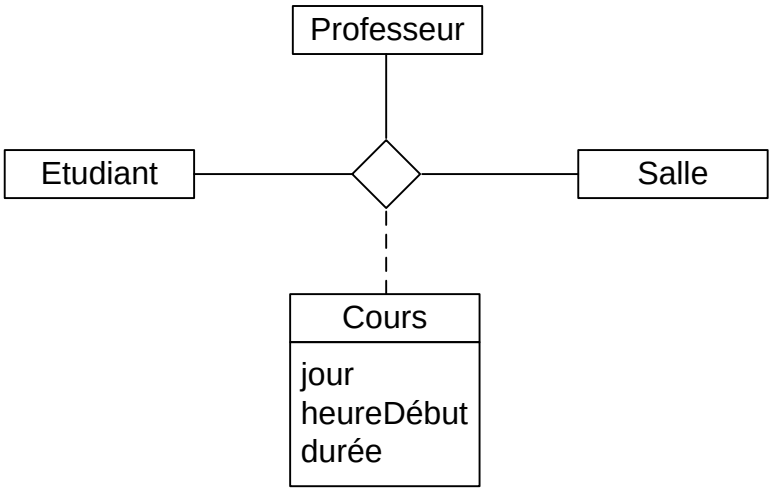


FIGURE 6.2 – Association ternaire.

206

Exemple 4/4

Représentation de l'association ternaire en Java

```
class Cours {
    private Etudiant etudiant;
    private Professeur professeur;
    private Salle salle;
    private int jour;
    private int heureDebut;
    private int duree;
    // ...
}
```

207

Agrégation

- L'*agrégation* est une association non symétrique, qui exprime un couplage fort et une relation de subordination
- Elle représente une relation de type « ensemble/élément » (ou « tout/partie ») entre des classes
- La classe représentant l'ensemble est parfois appelée *agrégat*
- À un même moment, une instance d'élément agrégé peut être liée à plusieurs agrégats (l'élément agrégé peut être partagé)
- Une instance d'élément agrégé peut exister sans agrégat (et inversement) : les cycles de vies de l'agrégat et de ses éléments agrégés peuvent être indépendants
  - si on détruit une agrégation, on ne détruit pas ses composants
  - agrégation par « référence »

208

Exemple 1/2

Agrégation

(voir figure 6.3).

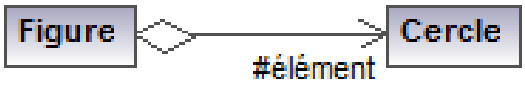


FIGURE 6.3 – Agrégation entre `Figure` et `Cercle`.

209

Exemple 2/2

Représentation de l'agrégation en Java

```
class Figure {
  /** L'ensemble de cercles composants la figure. */
  private List<Cercle> elements;
  // ...
}
```

210

Composition

- Une *composition* est une agrégation forte (agrégation par valeur)
- À un même moment, une instance de composant ne peut être liée qu'à un seul agrégat
- Les cycles de vies des éléments (les « composants ») et de l'agrégat sont liés
  - si l'agrégat est détruit (ou copié), ses composants le sont aussi

Remarque

- Le choix de modélisation entre agrégation et composition est souvent subjectif

211

Exemple 1/2

Composition

(voir figure 6.4).

FIGURE 6.4 – Composition entre un cercle et son centre.

212

Exemple 2/2

Représentation de la composition en Java

```
class Cercle {
  /** Le centre du cercle. */
  private Point centre;

  public Cercle(final Point centre, final double rayon) {
    this.centre = centre.clone();
    // ...
  }
  // ...
}
```

213

6.1.3 Relation de dépendance

Relation de dépendance

- Un élément A dépend d'un élément B si A utilise les services de B
- Un changement dans B peut donc avoir des répercussions sur A
- Par exemple, une *objet local à une méthode* ou *un paramètre de méthode* sont génèrent des dépendances

214

6.2 Modules

Définition d'un module

- Pour créer un module ou y ajouter une classe ou une interface, on place une instruction `package` au début du fichier source

```
package monpackage;
```

- Tout ce qui est défini dans le fichier source fait alors parti du module
- Sans instruction de ce type, les éléments se trouvent dans le module par défaut (non nommé)
- Les noms des module respectent en général une convention (par exemple, `uvsq.in404.monpackage`)
- La librairie Java est organisée en module (`java.lang`, `java.util`, `java.io`, ...)

215

Interface d'un module

- Seul les éléments publics sont accessibles à l'extérieur du module
- Pour rendre une classe ou une interface publique, on spécifie le mot-clé `public` dans sa déclaration

```
public class MaClasse { //...
```

216

Utilisation d'un module

- Différentes façons d'utiliser les éléments public d'un module
  - utiliser son nom qualifié (par exemple, `uvsq.in404.monpackage.MaClasse`)
  - importer l'élément (par exemple, `import uvsq.in404.monpackage.MaClasse;` placé en début de fichier source)
  - importer le module complet (par exemple, `import uvsq.in404.monpackage.*;` placé en début de fichier source)
  - importer les classes imbriquées (`import uvsq.in404.monpackage.MaClasse.*;`)
  - importer les membres de classes (`import static uvsq.in404.monpackage.MaClasse.*;`)
- Les directives `import` se placent avant toute définition de classes ou d'interfaces mais après l'instruction `package`
- Deux modules sont automatiquement importés : le module par défaut et `java.lang`

217

Module et gestion des sources en Java

- Dans un fichier source
  - plusieurs éléments (classes, interfaces, ...) peuvent être définies
  - un seul élément peut être public
  - le nom de l'élément public doit être le même que le nom du fichier
- On se limite de préférence à une classe par fichier source
  - le nom du fichier `.java` est le même que le nom de l'élément qu'il contient
- Le nom du répertoire doit refléter le nom du paquetage
  - la classe `uvsq.in404.monpackage.MaClasse` doit se trouver dans le fichier `MaClasse.java` du répertoire `uvsq/in404/monpackage`

218

Module et compilation

- Lors de la compilation, un fichier `.class` est créé pour chaque élément
- La hiérarchie de répertoires contenant les `.class` reflète les noms des modules
- Les répertoires où sont recherchées les classes lors de l'exécution sont listés dans le *class path*
- Par défaut, le répertoire courant et la librairie Java se trouve dans le *class path*
- La façon dont le *class path* est défini dépend de la plateforme
  - en général, on définit une variable d'environnement `CLASSPATH`
- Le *class path* contient des chemins vers
  - des répertoires contenant une arborescence de `.class`
  - des fichiers `.jar`
  - des fichiers `.zip`

219

6.3 Utilisation d'une bibliothèque tierce

Écosystème Java et bibliothèques

- L'écosystème Java fournit un nombre important de bibliothèques et d'outils de développement
- Dans un projet de développement logiciel, le choix des bibliothèques à utiliser est une étape importante
  - fonctionnalités, complexité, support de la communauté, licence, ...
- La plupart des programmes Java font appel à des *bibliothèques tierces* (*third party libraries*)

220

Utilisation d'un bibliothèque tierce

1. Récupérer la bibliothèque
   - manuellement (téléchargement)
   - automatiquement (outils de gestion des dépendances comme *maven* ou *gradle*)
2. Inclure la bibliothèque dans le projet
   - le `CLASSPATH` doit être modifié pour faire référence aux archives (`jar` en général) de la bibliothèque
3. Consulter l'interface de la bibliothèque
   - toute bibliothèque Java est distribuée avec sa documentation au format *javadoc*
4. Importer les modules nécessaires dans les fichiers sources
   - l'utilisation d'une classe de la bibliothèque nécessite d'importer le package Java adéquat

221

Exemple 1/4

Utilisation de la bibliothèque Apache Commons Math

- Télécharger et décompresser le fichier commons-math3-3.4.1-bin.tar.gz

```
~/commons-math3-3.4.1 $ ls
commons-math3-3.4.1.jar          docs/        NOTICE.txt
commons-math3-3.4.1-javadoc.jar  LICENSE.txt  RELEASE-NOTES.txt
```

222

Exemple 2/4

Utilisation de la bibliothèque Apache Commons Math

— Ajouter la bibliothèque au projet (IDE ou outil de *build*) (voir figure 6.5).

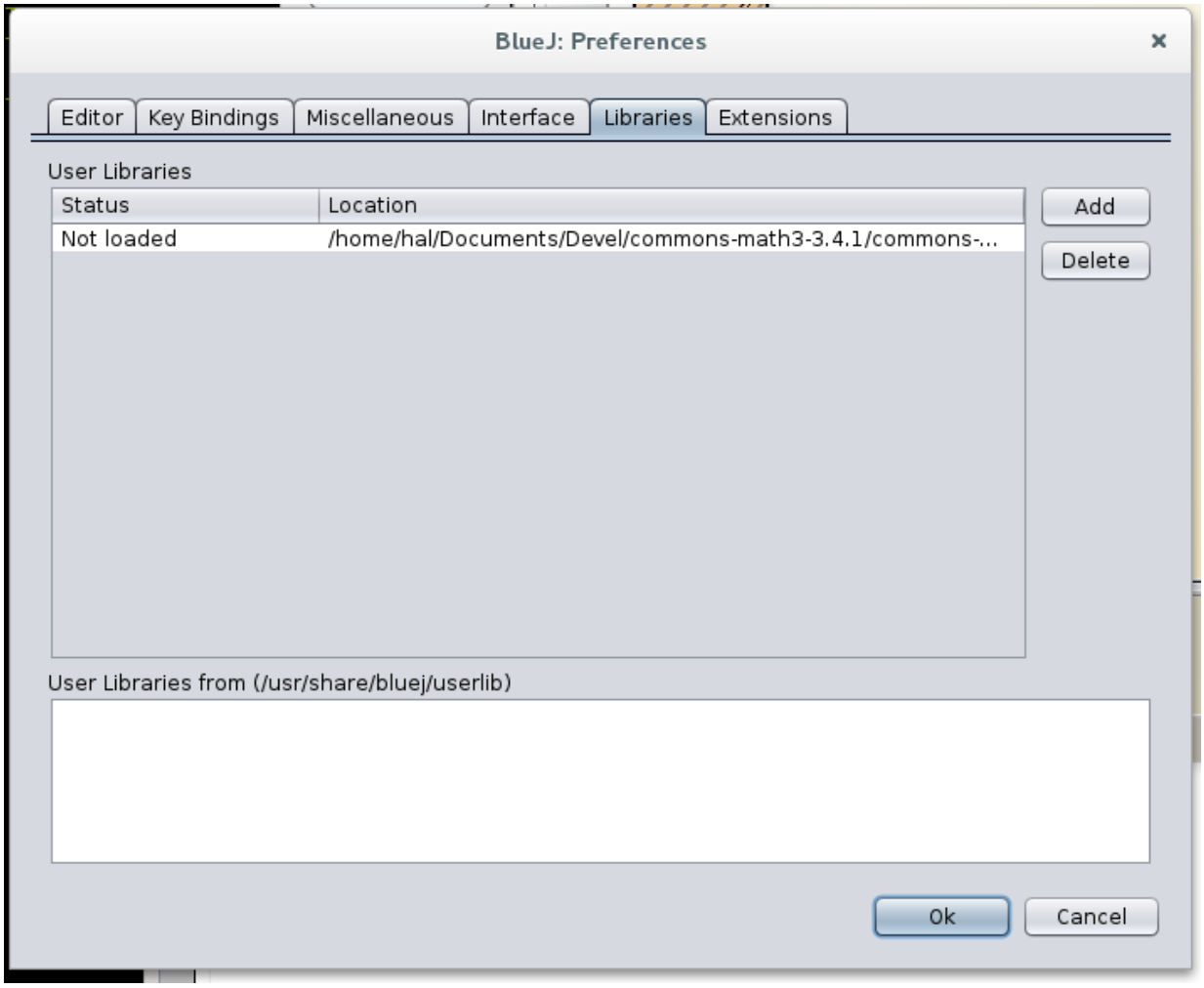


FIGURE 6.5 – Apache Commons Math dans BlueJ.

223

Exemple 3/4

Utilisation de la bibliothèque Apache Commons Math

— Importer les classes des packages nécessaires

```
import org.apache.commons.math3.fraction.Fraction;

public class Main {
    public static void main(String[] args) {
        Fraction f = new Fraction(1, 3);
        System.out.println(f);
    }
}
```

224

Exemple 4/4

Utilisation de la bibliothèque Apache Commons Math

— Compiler en précisant la bibliothèque dans le CLASSPATH (en ligne de commande)

```
$ javac -cp ../commons-math3-3.4.1/commons-math3-3.4.1.jar Main.java
```

— Éxécuter en précisant la bibliothèque dans le CLASSPATH (en ligne de commande)

```
$ java -cp ../commons-math3-3.4.1/commons-math3-3.4.1.jar:. Main
```

225

6.4 Exercices

Exercice 6.1 (Module et bibliothèque)
Dans cet exercice, vous reprendrez le code source de l'exercice 4.2 (simulation de client/serveur).

1. Placez l'ensemble du code source dans le package `in404.exo61`.
2. Ajoutez une classe contenant le programme principal qui implémentera le scénario de la question 1 de l'exercice 4.2.
3. Déplacez la classe `Client` dans le package `in404.exo61.client` et la classe `Serveur` dans le package `in404.exo61.serveur`. Apportez les modifications nécessaires au programme principal.
4. Construisez une archive `jar` contenant les classes compilées `in404.exo61.client.Client` et `in404.exo61.serveur.Serveur`. Modifiez le projet pour qu'il utilise cette bibliothèque.

Exercice 6.2 (Utilisation d'une bibliothèque tierce)
Vous utiliserez ici la bibliothèque Apache Commons Math.

1. En consultant la documentation Javadoc sur le site, identifiez la classe (et son package) permettant la manipulation de nombres complexes.
2. Ajoutez la bibliothèque à votre projet.
3. Écrivez un programme principal pour tester quelques méthodes de la classe.

Chapitre 7

Conclusion

Sommaire

7.1 Début de la conclusion

Bilan

1. blabla

226

Pour aller plus loin...

1. blabla

227