Cours de Programmation Orientée Objet

Licence Informatique

Denis Béchet, Marc Champesme, Guillaume Vauvert
Marc.Champesme@lipn.univ-paris13.fr
Département d'Informatique
Institut Galilée
Université Paris 13

27 septembre 2019

Table des matières

Chapitre 1

Introduction

1.1 But de ce cours

Ce cours traite des idées principales de la programmation orientée objet dans le contexte du langage de programmation JAVA. Précisons cependant bien les choses :

— le but principal de ce cours est de vous faire assimiler les concepts et notions de base de la programmation orientée objet ;

— le but de ce cours n'est pas de vous enseigner tous les détails du langage JAVA.

Il s'agit de vous apprendre à appliquer l'approche orientée objet dans la conception et l'implémentation de logiciels. Ces connaissances devraient par la suite pouvoir être appliquées quelque soit le langage de programmation que vous serez amené à utiliser, y compris des langages non orientés objet, où ces concepts peuvent être aussi appliqués, quoique avec moins de facilité.

Il ne faut, en effet, pas confondre "connaitre la syntaxe d'un langage de programmation" avec "savoir l'utiliser au mieux de ces possibilités". Une connaissance parfaite de la syntaxe du langage JAVA, vous permettra d'écrire des programmes en JAVA, mais pas nécessairement d'écrire des programmes orientés objet.

1.2 Références bibliographique

1.2.1 Livres

Ce cours est basé en partie sur le livre de B. Meyer "Conception et programmation orientées objet", qui est en même temps le concepteur du langage Orienté Objet "Eiffel".

Les références essentielles sont :

— Bertrand Meyer, "Conception et programmation orientées objet", Editions Eyrolles, paru en 2008.

— Joshua Bloch, "Effective Java : Third Edition", Addison Wesley, paru en 2017.

— David J. Barnes et Michael Kölling, "Conception objet en Java avec BlueJ. Une approche interactive", Pearson Education, 4ème édition, 2013.

1.2.2 Sur le web

Tous les logiciels utilisés dans le cadre de ce cours sont disponibles gratuitement (JML et BlueJ sont des logiciels "libres" sous licence GPL) et peuvent être installés sur toutes les plateformes (Linux, MacOS X, Windows).

— Le site "officiel" de ce cours :
`https://moodle.univ-paris13.fr/course/view.php?id=66`

— Le site "officiel" de JAVA pour télécharger le SDK (Software Development Kit) JAVA :
`http://java.oracle.com`
Version 5.0 (appelée aussi version 1.5.0) ou ultérieure requise pour la généricité ;

— Le site de BlueJ : `http://www.bluej.org`

— Le site de JML : `http://www.openjml.org`

Chapitre 2

Une approche informelle de la programmation orientée objet

2.1 Pourquoi un nouveau paradigme de programmation ?

Dans ce cours, nous allons étudier ce que l'on appelle un nouveau *paradigme* de programmation. Vous avez vu jusqu'à présent le *paradigme impératif* (avec le langage C), puis le *paradigme* fonctionnel (avec le langage CAML) nous allons compléter maintenant le tableau avec le *paradigme objet* (et si vous continuez vos études en informatique, vous verrez peut-être un quatrième paradigme : la *programmation logique*).

Ce terme *paradigme*, est employé volontairement pour marquer le fait qu'il ne s'agit pas simplement d'étudier un nouveau langage de programmation, mais plutôt toute une nouvelle famille de langages de programmation. Ce qui différencie ces différentes familles, c'est le fait que chaque famille possède sa propre manière de voir la programmation. Le but de ce cours n'est donc pas de vous faire découvrir un nouveau langage, mais plutôt, quelle est cette nouvelle manière de voir le monde de la programmation.

La question que vous vous posez surement maintenant est : "Mais pourquoi donc *encore* un nouveau paradigme de programmation ?"

Pour répondre à cette question, il faut faire un peu d'histoire : les premiers langages de programmation utilisés étaient les assembleurs (je passe sur le langage machine que les pionniers de l'informatique ont utilisé dans les tout débuts de l'histoire des ordinateurs), des langages très proches de la machine (utilisation directe des instructions du processeur et de la mémoire), tellement proches que lorsque l'on changeait de machine il fallait apprendre un nouveau langage. La première évolution importante a consister à concevoir, à partir des langages d'assemblage, des langages indépendants des machines. De là est née une famille de langages de programmation, à laquelle on a attaché le qualificatif d'impératif (le fameux *paradigme impératif*).

Quelques années plus tard, une équipe de chercheurs qui cherchaient un langage de programmation permettant d'effectuer des simulations du monde réel a créé le langage SIMULA en 1962 (aujourd'hui disparu). En reprenant certaines des idées qui avaient été utilisées pour la conception de ce langage, une autre équipe donna naissance en 1980 à un langage appelé SMALLTALK, que l'on considère aujourd'hui comme le premier langage orienté objet. SMALLTALK fut utilisé (il possède encore une communauté de supporters) longtemps surtout pour écrire des prototypes de systèmes informatiques car il avait l'avantage de permettre un développement très rapide mais qu'il n'était pas suffisamment efficace pour permettre de réaliser des systèmes utilisables en production.

Le paradigme objet était né, mais il a végété encore quelques années jusque dans les années 1990. En effet, avec l'essor important de l'informatique, les producteurs de logiciels se sont trouvés confronté à un grave problème : les systèmes informatiques (i.e. les logiciels) devenaient de plus en plus sophistiqués, mais on commençait à avoir de sérieux problèmes pour les fabriquer :

— délais de fabrication jamais tenus ;
— logiciels de mauvaise qualité (nombreux bugs) ;
— très grande difficulté à réutiliser le travail de programmation effectué pour les logiciels antérieurs ;
— logiciels très difficiles à faire évoluer (ex : bug de l'an 2000).

Un thème de recherche en informatique a alors pris une grande importance : le génie logiciel, un axe de recherche qui se préoccupe essentiellement des problèmes de méthodes pour le développement de logiciels (comment procéder pour produire de *bons logiciels*). A l'intérieur de cette communauté de recherche de nombreux chercheurs se sont alors rendu compte que l'approche orientée objet pouvait aider à résoudre les problèmes précédents.

C'est à peu près à cette époque que les principaux langages orientés objet encore utilisés actuellement ont été conçus (C++, Objective-C, JAVA, Pascal Objet, Eiffel, CLOS...).

Ce qu'il est important de retenir dans ce rapide historique, ce sont les raisons pour lesquelles les langages orientés objet (dont JAVA) ont été conçu (cf. les points précédents). En effet, les choix de conception de ces langages s'ex-

pliquent la plupart du temps par la volonté de résoudre ces problèmes et ils permettent donc souvent de mieux comprendre pourquoi ces langages fonctionnent comme ils le font et surtout comment ces langages doivent être utilisés. C'est pourquoi il est très important que vous reteniez les points suivants pour vous guidez lorsque vous aurez à faire des choix de programmation : lorsque vous aurez un choix à faire, essayez de choisir la solution qui permet de satisfaire au mieux les objectifs suivants :

— possibilité de développer de grosses applications complexes en répartissant le travail de programmation entre un nombre important de programmeurs ;
— faire du logiciel correct *par construction* ;
— permettre la réutilisation de composants logiciels ;
— possibilité de faire évoluer le logiciel, de l'adapter aux besoins de l'utilisateur et cela sans avoir besoin de tout refaire à partir de zéro à chaque modification.

2.2 Une manière de voir le monde

Pour illustrer les idées principales de la programmation orientée objet, nous allons prendre une situation du monde réel et examiner comment nous pourrions nous inspirer de cette situation pour la conception de logiciels.

Supposons que Lionel, étudiant à Paris 13, veuille envoyer des fleurs à son amie Bernadette qui habite en Corrèze. Du fait de la distance, Lionel ne peut pas aller lui porter les fleurs directement. Néanmoins, il y a une solution simple à son problème : Lionel peut aller chez le fleuriste de Villetaneuse, lui dire quelles fleurs il veut envoyer, lui donner l'adresse de Bernadette et quelques pièces jaunes. Ensuite, il n'a plus rien à faire, il est assuré que les fleurs seront livrées chez Bernadette.

2.2.1 Agents et communautés

Examinons maintenant un peu plus attentivement ce qui s'est passé, pour résoudre le problème de Lionel :

— il a d'abord recherché un *agent* approprié (le fleuriste de Villetaneuse)
— et il lui a transmis un *message* contenant sa demande.

Il a choisi ce fleuriste parce que Lionel savait qu'il était capable de satisfaire sa demande. En effet, il sait que le fleuriste de Villetaneuse connaît une *méthode* pour traiter ce type de demande, c'est-à-dire, qu'il connaît la suite des actions à entreprendre pour la satisfaire.

Cependant, pour choisir la personne à qui soumettre sa requête, Lionel n'a pas eu besoin de savoir comment le fleuriste de Villetaneuse allait s'y prendre pour faire parvenir les fleurs à Bernadette, et la plupart du temps nous ne souhaitons pas connaître ce genre de détails. Ces informations nous sont *masquées* et c'est beaucoup mieux ainsi. En effet, de cette manière, c'est le fleuriste de Villetaneuse qui prend l'entière responsabilité de la satisfaction de la requête de Lionel et Lionel n'a pas à se soucier des éventuels problèmes que le fleuriste pourrait rencontrer. D'un autre côté, en lui cachant cette information, le fleuriste est assuré que Lionel n'interviendra pas dans son travail, et qu'il pourra s'y prendre de la manière qui lui convient le mieux, du moment que Bernadette reçoit les fleurs dans un délai raisonnable.

Si, maintenant je mène mon enquête pour essayer de découvrir tous les détails de ce qui s'est réellement passé, je vais d'abord découvrir que le fleuriste de Villetaneuse à lui même envoyé un *message* à un fleuriste de la ville où habite Bernadette. Un message très probablement différent de celui que Lionel avait transmis au fleuriste de Villetaneuse. En allant plus loin, nous pourrions découvrir que cet autre fleuriste a lui même fait appel à plusieurs autres agents pour traiter la commande de Lionel :

— un grossiste en fleurs ;
— une personne pour arranger les fleurs en un bouquet présentable et emballer le bouquet pour qu'il soit transportable sans subir de dommages ;
— un livreur pour acheminer le bouquet de la boutique au domicile de Bernadette.

Le grossiste, à son tour, a peut-être du contacter un horticulteur, qui lui même possède des employés à qui il a transmit un message qui n'avait plus grand chose à voire avec le message de Lionel à son fleuriste.

Cette description de la résolution d'un problème, correspond en fait très bien à la façon *orientée objet* de résoudre les problèmes : la résolution de ce problème d'envoi de fleur, a nécessité l'intervention d'un grand nombre d'agents (le fleuriste de Villetaneuse, le deuxième fleuriste en Corrèze, le grossiste, la personne qui fait le bouquet, le livreur...). Sans leur aide, le problème de Lionel n'aurait pas pu être résolu simplement. Transposées en "orienté objet", nous pourrions exprimer ces remarques de la manière suivante :

> Un programme orienté objet est structuré en une *communauté d'agents* qui interagissent. Ces agents sont appelés *objets*. Chaque objet fournit un ou plusieurs services aux autres membres de la communauté.

2.2.2 Messages et méthodes

La réaction en chaîne qui a finalement aboutit à résoudre ce problème de fleurs a été initié par la demande de Lionel faite au fleuriste de Villetaneuse. Cette requête a elle même déclenché d'autres requêtes de ce fleuriste en direction d'autres agents, lesquels ont à leur tour déclenché d'autres requêtes, tout cela jusqu'à ce que les fleurs soient finalement livrées à Bernadette. Cette observation conduit à mettre en lumière un autre principe fondamental de la programmation orientée objet :

> En programmation orientée objet, toute action est initiée par la transmission d'un *message* à un agent (un *objet*) qui prend la responsabilité de l'action à effectuer. Ce message, en plus de la nature de l'action à effectuer (e.g. "livrer des fleurs"), est accompagné d'informations supplémentaires (des *arguments* : quelles fleurs, combien de fleurs, adresse de livraison ?) nécessaires pour effectuer cette action. Le *receveur* (i.e. le destinataire) est l'objet à qui le message est envoyé. Si le receveur accepte le message, il accepte en même temps de prendre la responsabilité que l'action demandée soit menée à bien selon les informations fournies. En réaction à un message reçu, le receveur va exécuter une *méthode* pour satisfaire le requête exprimée dans le message.

Voyons maintenant de quelle manière l'approche "orientée objet" est différente de l'approche "programmation impérative" que vous connaissez (à travers l'utilisation du langage C). En effet, il peut sembler à première vue que les deux approches se ressemblent beaucoup : par exemple, qu'y a-t-il comme différence entre un envoi de message et un appel de fonction, ou entre l'exécution d'une méthode et l'exécution d'un fonction ?

La première différence importante est que pour tout message il y a un *receveur* désigné ; le receveur est un objet à qui le message est envoyé. A l'opposé, lors d'un appel de fonction, il n'y a aucun receveur désigné.

La seconde différence porte sur l'*interprétation* du message, c'est-à-dire la méthode — le code — utilisé pour répondre au message. En programmation objet, cette interprétation dépend du receveur et pourra être différente selon l'objet à qui le message est envoyé.

Exemple 2.2.1 Si, pour envoyer des fleurs à Bernadette, au lieu d'aller chez le fleuriste de Villetaneuse, Lionel s'adresse à son ami Jacques, il est probable que Jacques saura satisfaire sa requête, cependant, la méthode qu'il utilisera pour cela (il se contentera très probablement de transmettre la requête de Lionel à un fleuriste) sera probablement très différente de celle utilisée par le fleuriste de Villetaneuse. Si, par contre, Lionel demande à son chien d'envoyer des fleurs à Bernadette, il est très probable que le chien n'ait pas de méthode pour résoudre ce problème et Lionel ne peut pas s'attendre à ce que la réponse que son chien donnera à sa requête corresponde à son attente.

2.2.3 Masquage d'information

Un autre principe important que nous avons pu noter dans notre exemple, est celui du *masquage d'information*. L'agent qui envoie un message n'a pas besoin de savoir *comment* l'action demandée sera réalisée. Ce principe qui peut sembler évident dans la vie courante, est malheureusement loin d'être aussi évident dans l'esprit de beaucoup de programmeurs. En effet, beaucoup de programmeurs font leur travail avec l'idée qu'ils doivent tout faire eux-même et ne pas utiliser le travail des autres. A l'opposé, en programmation objet une partie importante du travail consiste à développer des composants logiciels réutilisables, or, le premier pas à franchir lorsque l'on décide de réutiliser du logiciel est justement d'accepter de faire confiance à du logiciel écrit par d'autres.

2.2.4 Responsabilités

La notion de responsabilité est un autre concept fondamental de la programmation orientée objet. La demande que Lionel formule au fleuriste de Villetaneuse mentionne uniquement le résultat souhaité : que Bernadette reçoive les fleurs qu'il a choisit. Pour aboutir à ce résultat, le fleuriste est totalement libre de la séquence d'actions concrètes à appliquer pourvu que le résultat soit celui que Lionel attend, et pour assurer cette liberté le fleuriste sait que Lionel ne viendra pas interférer dans les actions qu'il entreprendra.

Dans la vision traditionnelle de la programmation (i.e. programmation impérative), on considère qu'un programme *agit sur* des structures de données. Par opposition, un programme orientée objet *demande à* des structures de données (i.e. des objets) de lui rendre un service. Cette distinction peut s'exprimer d'une autre manière par la règle suivante :

> Ne vous-demandez pas ce que vous pouvez faire *à* vos structures de données
> Demandez-vous plutôt ce que vos structures de données peuvent faire *pour* vous.

2.2.5 Classes et instances

Dans l'exemple du fleuriste, nous avons parler d'un fleuriste particulier : "le fleuriste de Villetaneuse". Cependant, si Lionel s'est adressé à ce fleuriste en particulier, c'est parce qu'il sait qu'en qualité de fleuriste il est capable de traiter un certain nombre de demandes, comme tous les fleuristes, et que, par conséquent, Lionel est assuré qu'il est capable

de traiter sa demande de livraison de fleurs. Un manière d'exprimer cela serait de dire, que le fleuriste de Villetaneuse fait partie de la *catégorie* des fleuristes. En programmation orientée objet, cette notion de catégorie est très importante et correspond à la notion fondamentale de *classe*.

Tout objet est *instance* d'une *classe*. La méthode exécutée par un objet en réponse à un message reçu est déterminée par la classe du receveur du message. Tous les objets (ou instances) d'une classe donnée exécutent la même méthode pour des messages similaires (i.e. qui différent uniquement par leurs arguments).

Chapitre 3

Les classes

Dans ce chapitre, nous allons voir comment mettre en œuvre les notions fondamentales présentées au chapitre précédent à l'aide d'un langage de programmation orientée objet particulier : le langage JAVA. Plus précisément, ce chapitre est destiné à fournir les éléments de syntaxe JAVA nécessaires pour commencer à programmer. Le chapitre suivant apportera des éléments complémentaires, avant de revenir dans les chapitres d'après sur les notions fondamentales et la manière de les appliquer dans des programmes orientés objet.

3.1 Programmation impérative vs. POO

Programmation Impérative	POO
Deux notions distinctes : — structures de données et types — opérations/fonctions	Une seule notion : classe = type = structure de données + opérations/méthodes
On définit des variables à partir des types	On définit des instances à partir des classes
Les traitements s'effectuent par appel de fonctions sur des variables.	Les traitements s'effectuent par envoi de messages à des instances de classes. Un envoi de message à une instance déclenche l'exécution d'une méthode, si la méthode a été définie dans la classe de cette instance.
Analyse descendante : 1. découpage selon les traitements à faire ; 2. en déduire les modules ; 3. conception des structures de données nécessaires pour chaque module.	Analyse ascendante : 1. identifier des agents ; 2. déterminer les services que doivent rendre chaque agent ; 3. en déduire les modules (1 agent = 1 classe).

3.2 Structure d'un programme

Un programme objet est un ensemble de classes. Chaque classe décrit un agent et les services qu'il est capable de rendre à la communauté d'agents. Chaque classe est définie dans un fichier distinct (une classe = un fichier). Le point d'entrée d'un programme (i.e. les instructions à exécuter au lancement du programme) est spécifié en désignant une classe (appelée "root class" ou "base class" en anglais) parmi l'ensemble des classes. Dans ce cours, nous emploieront le terme "classe racine" pour désigner cette classe. En JAVA, la classe racine doit posséder une méthode de nom `main`, qui jouera le même rôle que la fonction `main` d'un programme C (c'est cette méthode qui est appelée lors du lancement du programme).

La méthode `main` d'un programme JAVA doit toujours être déclarée de la manière suivante :

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

3.3 Premier programme JAVA

Voici un exemple trivial de programme JAVA, le classique “Hello world” :

```
public class HelloWorld {

        public static void main(String[ ] args) {
                System.out.println("Hello World");
        }
}
```

Dans ce cas, le programme se limite à une unique classe, dont la définition (i.e. le code ci-dessus) devra être sauvegardée dans un fichier nommé “`HelloWorld.java`” afin de respecter la convention JAVA, selon laquelle toute classe doit être sauvegardée dans un fichier dont le nom est constitué de l’identificateur de la classe (“`HelloWorld`” dans le cas présent) auquel on ajoute le suffixe “`.java`”.

Examinons maintenant un peu plus précisément ce programme.

3.3.1 Déclaration d’une classe

La première ligne correspond à la déclaration de la classe : le premier mot (i.e. le mot clé `public`) indique la visibilité de la classe, la valeur `public` indique que cette classe peut être utilisée par n’importe quelle autre classe. Bien qu’il soit possible d’utiliser une autre valeur, il est très rarement utile de le faire. Dans ce cours nous déclarerons toujours les classes avec une visibilité `public`.

Le second mot – `class` – indique que nous définissons une classe et le troisième – `HelloWorld` – est l’identificateur de cette classe. Ensuite, la définition de la classe est donnée entre accolades { `...` }

3.3.2 Déclaration d’une méthode

Dans cet exemple, la définition de la classe se limite à la définition de la méthode `main`. Comme vous pouvez le voir la syntaxe utilisée est très proche de la syntaxe du C. Comme les fonctions en C, la définition d’une méthode en JAVA s’effectue en donnant le type de sa valeur de retour ou en utilisant le mot clé `void`, lorsqu’il n’y a pas de valeur de retour comme c’est le cas ici pour la méthode `main`. On trouve ensuite l’identificateur de la fonction – `main` – puis la liste des arguments entre parenthèses – `(String[ ] args)` –, qui indique que la fonction possède un unique argument nommé `args` dont le type est tableau de chaînes de caractères – puis la liste des instructions à exécuter entre accolades { `...` }

Le mot clé `public` est, comme pour la classe, un indicateur de visibilité (i.e. pour faire du masquage d’information). Nous verrons plus tard quelles autres valeurs il est possible d’utiliser et comment choisir, cependant la méthode `main` doit toujours être déclarée `public` et, de manière générale, pour les méthodes, c’est la valeur la plus fréquemment utilisée (mais contrairement aux classes, il n’est pas rare d’utiliser d’autres valeurs).

Pour terminer avec l’entête de la méthode, le mot clé `static` indique que cette méthode est une *méthode de classe*. En effet, en JAVA comme dans beaucoup d’autres LOO il existe deux sortes de méthodes : les méthodes de classe et les méthodes d’instance. Nous reviendrons plus tard sur cette distinction. Cependant, l’usage de méthodes de classe doit être réservé à des cas très particulier et la plupart des méthodes que nous définirons seront des méthodes d’instance, ce que nous indiquerons en omettant le mot clé `static`.

3.3.3 Appel de méthode

Enfin, pour terminer avec ce premier programme, examinons le corps de la fonction `main`. Celui-ci ne comporte ici qu’une instruction qui est un appel de méthode (i.e. un envoi de message). Comme vous pouvez le voir, la syntaxe utilisée ressemble beaucoup à la syntaxe utilisée pour les appels de fonctions en C : un identificateur suivi de la liste des paramètres réels (ou effectifs) entre parenthèses. La seule différence importante est que l’identificateur de la méthode est composé de plusieurs mots séparés par des points. Dans cet exemple, ces mots sont au nombre de trois, cependant, nous verrons plus souvent des appels de méthodes composés de deux mots : dans ce cas, le premier mot indique l’objet à qui est envoyé le message et le second indique quelle méthode de cet objet nous souhaitons exécuter.

3.4 Un deuxième exemple : la médiathèque

Pour aller plus avant dans notre description du langage JAVA, nous allons prendre un exemple plus complexe qui nous permettra de présenter les éléments nécessaire pour écrire nos premières classes. Nous allons pour cela prendre l’exemple d’une application de gestion d’une médiathèque. Pour une telle application on peut identifier les agents suivant :

Adhérent l'agent adhérent est caractérisé par :
- un nom, un prénom et une adresse, qui permettent de faire le lien avec l'être humain qui emprunte des œuvres à la médiathèque ;
- le nombre d'emprunts actuellement en cours ;
- la possibilité pour l'adhérent d'effectuer de nouveaux emprunts ;
- un nombre maximal d'emprunts autorisés.

Emprunt qui nous permet de garder trace des informations concernant l'emprunt d'un livre par un adhérent (date de l'emprunt, livre emprunté, date limite pour rendre le livre...) ;

Livre caractèristiques dont l'application a besoin concernant les livres présents dans la médiathèque ;

Auteur caractèristiques dont l'application a besoin concernant les auteurs dont au moins un livre est présent dans la médiathèque.

Nous allons dans un permier temps, nous intéresser plus particulièrement à l'agent Adhérent, pour lequel nous allons définir une classe `Adherent`.

Pour cela, nous devons dans un premier temps déterminer précisément ce qu'est un adhérent pour l'application, c'est-à-dire quelles sont, pour l'application, les caractéristiques d'un adhérent et quelles propriétés ces caractéristiques doivent satisfaire pour qu'un objet adhérent soit utilisable par l'application.

Les caractéristiques d'un adhérent ont été données plus haut, voyons donc maintenant, quelles sont les propriétés que doivent satisfaire celles-ci afin que l'adhérent puisse rendre les services pour lesquels il a été pensé :
- l'objet adhérent étant destiné à représenter un être humain, tout objet adhérent doit posséder suffisamment d'informations pour que l'utilisateur du logiciel puisse établir un lien entre l'objet adhérent du logiciel et l'être humain correspondant : en conséquence, tout objet adhérent doit avoir ces champs nom, prénom et adresse renseignés. Nous pourrions ajouter, que ces informations doivent être pertinentes, mais la vérification de cette propriété dépasse ce que l'on peut espérer d'un logiciel : le logiciel peut s'assurer qu'une adresse est plausible (en vérifiant, par exemple, que la ville mentionnée est cohérente avec le code postal), pas qu'il s'agit d'une adresse existante ;
- le nombre d'emprunts doit être supérieur ou égal à zéro et inférieur ou égal au nombre maximal d'emprunts, deux emprunts ne peuvent faire référence au même exemplaire d'un même livre ;
- le nombre maximal d'emprunts autorisé doit être positif ou nul.

3.4.1 Les services de l'agent adhérent

Nous distinguerons différentes catégories de services, pour chacun de ces services, nous donnons les conventions de nommage :
- les services de consultation lorsqu'il s'agit d'une simple consultation d'une caractéristique de l'objet. Nous placerons dans cette catégorie la consultation des caractéristiques d'un adhérent :
  - le nom ;
  - le prénom ;
  - l'adresse ;
  - le nombre d'emprunts en cours ;
  - le nombre maximal d'emprunts simultanés autorisés ;
  - les emprunts.

  Par convention, en JAVA, les identificateurs des méthodes correspondant à ces services sont construit en juxtaposant le nom de la caractéristique au préfixe `get`. Cela donnerait ici les méthodes : `getNom, getPrenom, getAdresse, getNbEmprunts, getNbMaxEmprunts, getEmprunts;`
- les services permettant de remplacer la valeur d'une caractéristique par une nouvelle valeur :
  - le nom ;
  - le prénom ;
  - l'adresse ;
  - le nombre maximal d'emprunts simultanés autorisés.

  Par convention, en JAVA, les identificateurs des méthodes correspondant à ces services sont construit en juxtaposant le nom de la caractéristique au préfixe `set`. Cela donnerait ici les méthodes : `setNom, setPrenom, setAdresse, setNbMaxEmprunts;`
- les services permettant de tester si une propriété de l'objet est vraie. Dans le cas de l'adhérent, un service permettant de déterminer si un adhérent a le droit d'effectuer un emprunt, entrerait dans cette catégorie.
  Par convention, en JAVA, les identificateurs des méthodes correspondant à ces services sont construit en juxtaposant le nom de la caractéristique au préfixe `is`. Cela donnerait ici la méthode : `isEmpruntPossible;`
- les autres services, par exemple, ceux permettant de modifier des caractéristiques de l'objet :
  - ajout d'un emprunt ;
  - retrait d'un emprunt.

Par convention, en JAVA, les identificateurs des méthodes correspondant à ces services sont construit sans préfixe particulier. Cependant, ces services correspondant le plus souvent à des actions à effectuer sur des caractéristiques de l'objet, on choisit comme identificateur un verbe (à l'infinitif) décrivant l'action à réaliser. Cela donnerait ici les méthodes : `ajouterEmprunt, retirerEmprunt;`

3.5 Types primitifs et types objets

L'étape suivante pour définir notre classe `Adherent` consisterait à définir la liste des paramètres et la valeur de retour de chaque méthode. Pour cela, nous avons besoin de savoir quels types nous avons à notre disposition.

Les types utilisables en JAVA appartiennent à deux grandes catégories :

— les types primitifs issus de C et qui s'utilisent de façon assez proche du C, la liste est restreinte et limitative (ne peut pas être étendue) ;

— les classes (classe = type) : là encore, un certain nombre de types prédéfinis mais à la différence des types primitifs, la liste n'est pas limitative, puisque chaque nouvelle classe définit un nouveau type ;

Ajoutons à cela les tableaux et les types énuméré (les types enum), qui constituent une catégorie particulière de classes dont l'utilisation se démarque assez nettement de celle des classes "ordinaires".

Un grand absent (par rapport au C) : les pointeurs. En JAVA, il n'est pas possible de créer des types pointeurs et de manipuler directement des adresses. Cependant, ce n'est pas un problème car :

— l'absence des pointeurs élimine du même coup tous les risques d'erreurs liées à leur usage. Il s'agit donc bien d'un avantage car les risques d'erreurs liés à l'usage des pointeurs sont nombreux et souvent très difficiles à dépister et corriger ;

— en C, les pointeurs sont nécessaires pour deux raisons essentielles : le passage de paramètres par adresse et l'allocation dynamique de la mémoire. L'absence des pointeurs en JAVA ne l'empêche cependant pas de fournir ces deux fonctionnalité. La technique utilisée est différente et repose sur le fait que tout objet (i.e. instance d'une classe) est, en fait, un "pointeur masqué" (en JAVA, on ne parle pas de pointeurs ou d'adresse, mais de référence) et, deuxième principe, tout objet est alloué dynamiquement et désalloué (i.e. libéré) automatiquement sans intervention du programmeur à l'aide d'un "ramasse miettes".

3.5.1 Types primitifs

Java offre un nombre restreint de types primitifs mais dont l'intervalle des valeurs possibles est fixé indépendamment de la plate-forme, contrairement à C où selon le type de processeur de la machine, un entier de type `int` est codé sur 2 ou 4 octets, voir plus pour les machines 64 bits :

Nom	Type
`byte`	valeur entière codée sur 8 bits (-128 à 127)
`short`	valeur entière codée sur 16 bits (-32768 à 32767)
`int`	valeur entière codée sur 32 bits (-2^{31} à $2^{31}-1$)
`long`	valeur entière codée sur 64 bits (-2^{63} à $2^{63}-1$)
`float`	valeur réelle codée sur 32 bits (norme IEEE 754-1985)
`double`	valeur réelle codée sur 64 bits (même norme)
`char`	caractère (international) codé sur 16 bits (code Unicode)
`boolean`	valeur booléenne vrai/faux

Exemples de constantes littérales :

Type	Littéral
`int`	`0`, `25666`, `28` ou `0x1c` ou `0X1C` ou `034`
`long`	`0l` ou `0L`, `25666L`, `28L` ou `0x1cL` ou `0X1CL` ou `034L`
`float`	`1e1f` ou `1E1F`, `0.055f`, `.3f`, `0f`, `6.022137e+23f`
`double`	`1e1` ou `1e1d` ou `1E1D`, `0.055`, `.3`, `0d`, `6.022137e+23`
`char`	`'a'`,`'%'`,`'\n'`,`'\u03fa'`,`'\uFFFF'`
`boolean`	`true` et `false`

Contrairement à C ou C++, Java possède un type booléen retourné par tous les opérateurs de comparaison et les opérateurs logiques. Le langage C utilise les entiers comme valeur booléenne et interprète toute valeur non nulle comme vraie et 0 comme faux. Java est plus "propre" en ce domaine.

3.6 Définir des méthodes

Nous en savons maintenant suffisamment pour pouvoir définir l'entête des méthodes :

```
String getNom()
String getPrenom()
String getAdresse()
int getNbEmprunts()
int getNbMaxEmprunts()
void setNom(String nouveauNom)
void setPrenom(String nouveauPrenom)
void setAdresse(String nouvelleAdresse)
void setNbMaxEmprunts(int max)
boolean isEmpruntPossible()
void ajouterEmprunt()
void retirerEmprunt()
```

3.6.1 Argument implicite

On voit ici le résultat de l'application d'une notion du chapitre précédent : à la différence des langages de programmation impératifs, l'exécution d'une méthode est déclenchée par l'envoi d'un message *à un objet particulier*. La conséquence visible ici est que, dans la définition des méthodes, il n'est pas nécessaire de spécifier l'objet particulier sur lequel ces méthodes seront appliquées. En d'autre terme, toute méthode possède un argument implicite qui est l'objet receveur du message.

3.6.2 Passage de paramètres

En JAVA comme en C, seul existe le passage par valeur. Cependant, les objets étant des références, les paramètres qui sont des objets peuvent être modifiés. Par contre, les paramètres typés à l'aide des types de base ne peuvent jamais être modifiés par un appel de méthode.

3.7 Définir des attributs (ou champs)

Pour pouvoir remplir son rôle, un objet à besoin de pouvoir mémoriser des informations : les attributs [1] sont faits pour ça. Mais ils doivent restés au second plan → le principe général doit-être l'utilisation de la visibilité `private` pour les attributs.

```
private String nom;
private String prenom;
private String adresse;
private int nbMaxEmprunts;
private int nbEmprunts;
```

3.8 Implémenter des méthodes

```
public class Adherent {
        private String nom;
        private String prenom;
        private String adresse;
        private int nbMaxEmprunts;
        private int nbEmprunts;

        public String getNom() {
                return nom;
        }

        public String getPrenom() {
                return prenom;
        }

        public String getAdresse() {
                return adresse;
```

1. Par la suite, on utilisera indifféremment les termes “attribut” ou “champ”.

```
        }

        public int getNbEmprunts() {
                return nbEmprunts;
        }

        public int getNbMaxEmprunts() {
                return nbMaxEmprunts;
        }

        public void setNom(String nouveauNom) {
                nom = nouveauNom;
        }

        public void setPrenom(String nouveauPrenom) {
                prenom = nouveauPrenom;
        }

        public void setAdresse(String nouvelleAdresse) {
                adresse = nouvelleAdresse;
        }

        public void setNbMaxEmprunts(int max) {
                nbMaxEmprunts = max;
        }

        public boolean isEmpruntPossible() {
                return (getNbEmprunts() < getNbMaxEmprunts());
        }

        public void ajouterEmprunt() {
                nbEmprunts = nbEmprunts + 1;
        }

        public void retirerEmprunt() {
                nbEmprunts = nbEmprunts - 1;
        }
}
```

3.9 Créer et utiliser des objets

Nous allons maintenant voir comment la classe `Adherent` que nous venons de définir peut être utilisée dans un programme. Dans ce but, nous définissons une nouvelle classe `MediathequeApp`, dont cette toute première version se contente d'effectuer quelques opérations simples sur des objets de la classe `Adherent`.

3.9.1 Définition d'un constructeur

Avant de pouvoir définir cette nouvelle classe, il nous reste néanmoins à compléter la classe `Adherent` afin de préciser comment des instances de cette classe pourront être créées. En JAVA, comme dans la plupart des langages de programmation orientée objet, on distingue deux phases dans la création d'instances : l'allocation mémoire et l'initialisation. En JAVA, c'est l'opérateur `new` qui effectue l'allocation mémoire et ce sont des méthodes particulières appelées "constructeurs" qui se chargent d'initialiser les champs du nouvel objet. Le rôle du constructeur est de faire en sorte que, dès la création de l'objet, les propriétés essentielles de l'objet soient vérifiées (cf. plus haut pour la liste de ces propriétés). En JAVA, toute classe possède un constructeur par défaut, mais pour la classe `Adherent` nous devons définir un constructeur.

```
public class Adherent {
        ...

        public Adherent(String nom, String prenom, String adresse) {
```

```
                this.nom = nom;
                this.prenom = prenom;
                this.adresse = adresse;
                nbMaxEmprunts = 5;
        }

        ...
}
```

La classe `Adherent` est maintenant complète et prête à être utilisée :

```
public class MediathequeApp {

   public static void main(String[ ] args) {
      Adherent unAdherent, autreAdherent;

      // Message d'accueil
      System.out.println("Bienvenue dans l'application Mediatheque");

      // Creation des objets
      unAdherent = new Adherent("Dupont", "Jacques", "12 rue des Glycines, Villetaneuse");
      autreAdherent = new Adherent("Martin", "Bernard", "5 rue Victor Hugo, St-Denis");

      // Utilisation de quelques méthodes
      System.out.println("Coordonnnees du premier adherent:");
      System.out.println("Nom: " + unAdherent.getNom());
      System.out.println("Prenom: " + unAdherent.getPrenom());
      System.out.println("Adresse: " + unAdherent.getAdresse());

      // Message d'adieu !
      System.out.println("L'application Mediatheque vous souhaite une bonne journee...");
   }
}
```

3.9.2 Constructeur implicite

En JAVA, une classe pour laquelle un constructeur n'a pas été explicitement défini, possède un constructeur que nous qualifierons d'implicite et qui possède les caractéristiques suivantes :

- Une classe sans constructeur possède un constructeur *implicite* sans arguments. Ce constructeur initialise les attributs de la nouvelle instance à leurs valeurs par défaut (cf. section 4.3.2).
- Une classe possédant au moins un constructeur explicitement défini, ne possède pas de constructeur implicite : dans ce cas, une instruction du type `a = new MaClasse();` n'est légale que si un constructeur sans arguments a été explicitement défini. Dans ce cas, les initialisations par défaut sont faites de toute manière par un appel (implicite lui aussi et effectué avant exécution de la première instruction du constructeur explicitement défini) au constructeur sans arguments de la classe `Object`.

Lorsque nous aborderons l'héritage, nous compléterons ce point sur le fonctionnement des constructeurs (cf. section 6.1.3).

3.10 Compilation et exécution de programmes JAVA

L'exécution d'un programme JAVA nécessite de passer par une étape de compilation. Cependant, à la différence du langage C et de la plupart des langages de programmation compilés, le résultat de la compilation d'un programme JAVA n'est pas un un programme exécutable directement.

Pour la plupart des langages de programmation le compilateur traduit le code source du programme dans le langage machine spécifique au processeur sur lequel on souhaite exécuter le programme, l'exécution d'un même programme sur des plates-formes différentes nécessite donc de recompiler ce programme pour chacune de ces plates-formes.

Par opposition, le résultat de la compilation d'un programme JAVA est identique quelque soit la plate-forme sur laquelle on souhaite l'exécuter. Cette caractéristique est très importante car elle apporte une vraie solution au problème important de la portabilité des logiciels. C'est une des raisons de la popularité du langage JAVA.

Le fait que la compilation d'un programme JAVA donne exactement le même résultat quelque soit la plate-forme est rendu possible en appliquant la technique suivante :

1. le compilateur JAVA génère un code indépendant de la plate-forme que l'on appelle *bytecode*. Il s'agit d'un langage de type assembleur, à la différence que, dans ce cas, ce langage n'a pas été conçu en fonction d'une architecture de processeur, mais en fonction du langage JAVA ;
2. le *bytecode* produit par le compilateur JAVA étant indépendant de la plate-forme, il ne peut être directement exécuté. L'exécution de ce code s'effectue à l'aide d'un programme qui simule un ordinateur dont le processeur serait capable de l'exécuter directement. Le fonctionnement de cet ordinateur étant simulé, on parle alors de ce programme en employant le terme de *machine virtuelle*.
3. on développe une machine virtuelle pouvant exécuter du *bytecode* pour chaque plate-forme. De cette façon, le même code compilé d'un programme JAVA pourra être exécuté sur toute plate-forme pour laquelle existe une machine virtuelle.

Chapitre 4

Types, opérateurs et constructions de base en JAVA

Dans cette section, nous nous intéresserons aux types de base des variables ou des expressions et à la syntaxe du corps des méthodes.

4.1 Opérateurs

Opérateurs	Fonction
`obj.champ  obj.methode(...)`	accès à un champ (= attribut), appel d'une méthode
`obj instanceof Adherent`	obj est-il instance de la classe Adherent ou d'une de ses sous-classes ?
`t[i]`	accès à un élément d'un tableau
`(type) v`	cast
`new Adherent(...)`	allocation et initialisation d'un objet
`=`	affectation
Opérateurs arithmétiques	
`-x`	négation
`* / %`	multiplication, division et reste
`+ -`	addition, soustraction
`*= /= %= ...`	affectations (avec opération)
`++x --x`	pré-incrémentation, pré-décrémentation
`x++ x--`	post-incrémentation, post-décrémentation
Opérateurs relationnels	
`< > <= >=`	comparaisons
`== !=`	égalité, différent
Opérateurs logiques	
`!bool`	négation logique
`&&`	ET logique
`||`	OU logique
Opérateurs de la classe `String`	
`+`	concaténation de chaînes

Attention : Les conditions doivent être des expressions à valeur de type `boolean`. A la différence du C, les expressions à valeur entiere (ou les références) ne peuvent être utilisées comme condition.

Exemple 4.1.1 Le cas des expressions booléennes :

```
int i;
Adherent unAdherent;
```

```
i = 5;
if (i - 5) { // Légal en C, interdit en JAVA
        ...
}
if (i != 5) { // Syntaxe correcte en JAVA
        ...
}
...
if (unAdherent) { // Légal en C, interdit en JAVA
        ...
}
if (unAdherent != null) { // Syntaxe correcte en JAVA
        ...
}
```

4.2 Affectation

Une variable peut être modifiée avec l'une des instructions d'affectation ou bien avec un des opérateurs d'incrémentation ou de décrémentation.

NomVariable = Expression
NomVariable +=|-=|*=|... Expression
NomVariable++|--
++|--NomVariable

Exemple 4.2.1 affectations :

```
i = 1;
j += i;
nom = "Durand";
i = j = 0;
```

4.3 Variables

Il y a différentes sortes de variables :

Attributs (ou variables) de classe : un attribut partagé par toutes les instances de cette classe. Ces attributs existent dès le début du programme et disparaissent à la fin de son exécution. On déclare cette sorte d'attribut à l'aide du mot clé `static`. Le même mot clé peut-être utilisé pour définir des *méthodes de classe*.

Attributs (ou variables) d'instance : un attribut possédant une valeur spécifique pour chaque instance. L'attribut est créé avec l'instance et disparaît avec ce dernier.

Variable/Paramètre : un paramètre d'une méthode. La variable apparaît lors de l'appel à la méthode. Elle est initialisée avec la valeur de l'argument correspondant. Elle disparaît au retour de la méthode.

Variable locale : une variable temporaire valable uniquement à l'intérieur d'un bloc d'instructions.

Paramètres implicites : `this` qui désigne l'objet courant et `super` qui désigne la partie héritée de cet objet.

La syntaxe à utilisée pour une déclaration de variable est :

Type NomVariable,...;

4.3.1 Attributs à valeur non modifiable ou `final`

L'utilisation du mot clé `final` dans une déclaration d'attribut, spécifie que la valeur de cet attribut ne pourra pas être modifiée au cours de l'exécution du programme. Les attributs qualifiés de `final` peuvent donc être utilisés comme des constantes.

Concernant les attributs `final`, deux remarques importantes doivent être faites :

— pour qu'un attribut `final` soit utile il doit être initialisé à la déclaration ou par les constructeurs (cf. section suivante) ;

— si le type de l'attribut `final` est un type objet, seule la référence à cet objet est constante, l'objet lui-même pourra être modifié.

Notons que le mot clé `final` peut être utilisé en JAVA dans d'autres circonstances que la déclaration d'attributs. En particulier, une classe ou une méthode peut être qualifiée de `final`. Pour une classe cela signifie que cette classe ne pourra pas avoir de sous-classe. Pour une méthode, cela signifie qu'elle ne pourra pas être redéfinie dans une sous-classe.

4.3.2 Initialisation

Les champs ou attributs sont toujours initialisés avec des valeurs par défaut. Ces valeurs par défaut sont :
— 0 pour les champs numériques ;
— `false` pour les champs `boolean` ;
— le caractère `'\0'` pour les champs de type `char` ;
— la référence nulle `null` lorsque le champ est un objet.

> Seuls les attributs sont initialisés avec des valeurs par défaut : toute variable locale doit être initialisée avant utilisation. En cas d'oubli, le compilateur signale une erreur.

Lorsqu'il n'y a pas d'initialisation par défaut ou lorsqu'elle ne suffit pas, il est possible d'effectuer des initialisations à la déclaration, comme en C. Il y a cependant une différence importante avec le C : en JAVA, les expressions d'initialisation sont évaluées lors de l'exécution du programme et non à la compilation comme en C. Il n'y a donc aucune limitation sur le type d'expression d'initialisation utilisable.

Type NomVariable = Expression,... ;

Exemple 4.3.1 Initialisations :

```
char cara;
char cara2 = 'a', cara3 = '\n';
String ch = "Berlioz";
String[] tabChaines = { "ligne 1", "ligne 2", "ligne 3" };
String ch2;                 // Référence 'nulle' (pour un attribut)
System.out.println(ch2)     // Engendre une erreur (car ch2 vaut null)
```

4.4 Structure de contrôle `if`

L'instruction conditionnelle :

if (Condition) Instruction
if (Condition) Instruction else Instruction

Exemple 4.4.1 Utilisation du `if`

```
if (x == 0.0) {
        System.out.print("x est nul");
} else {
        System.out.print("x n'est pas nul");
        return 1/x;
}
```

4.5 Structure de contrôle `switch`

Comme en C.

Exemple 4.5.1 Utilisation du `switch`

```
public void classement(int n) {
        switch(n) {
                case 1 : System.out.println("un");
                                break;
                case 2 :
                case 4 : System.out.println("deux ou quatre");
```

```
                                    break;
                            case 3 : System.out.println("trois");
                            case 5 : System.out.println("trois ou cinq !");
                                    break;
                            default : System.out.println("autre");
                    }
            }
```

4.6 Structures de contrôles `for`, `while` et `do-while`

Comme en C (on peut inclure les déclaration des variables de boucle) :

`for(` Init; Test; Incrément`)` Instruction
`while (` Condition `)` Instruction
`do {` Instructions `} while (` Condition `);`

Exemple 4.6.1 `for`

```
for(int n = 1; n <= 10; n += 1) {
    System.out.println(n);    // Affiche 1  2  3  ...  10
}
int i, j;
for(i = 1, j = 10; i < 10; i++, j--) {
    System.out.println(i + ":" + j);    // Affiche 1:10  2:9  ...
}
```

Exemple 4.6.2 `while`

```
n = 1;
while (n <= 10) {
    System.out.println(n);    // Affiche 1  2  3  ...  10
    n++;
}
```

Exemple 4.6.3 `do ... while`

```
n = 10;
do {
    System.out.println(n);    // Affiche 10  9  8  ... 1
    n--;
} while (n > 0);
```

4.7 Types `enum`

Avec la version 5 de JAVA, une classe `Enum` a été introduite pour permettre la création de types jouant un rôle similaire aux types `enum` du langage C.

Exemple 4.7.1 Définition d'un type `enum` en C

```
typedef enum {CARREAU, COEUR, PIQUE, TREFLE} Famille;
```

Exemple 4.7.2 Définition équivalente en JAVA

```
public enum Famille {CARREAU, COEUR, PIQUE, TREFLE};
```

En JAVA, la définition d'un `enum` correspond à la définition d'une nouvelle classe héritant de la classe `Enum` et dont les uniques instances sont les constantes apparaissant dans la déclaration : dans l'exemple précédent, une classe `Famille` a été créée, cette classe est une sous-classe de la classe `Enum` et cette déclaration indique également que cette classe possède exactement 4 instances (aucune autre instance de cette classe ne peut être créée). De plus, les classes `enum` sont toujours non modifiables (les caractéristiques de leurs instances ne peuvent pas être modifiées. Pour accéder aux 4 instances de cette classe, on utilisera la syntaxe `Famille.CARREAU, Famille.COEUR, Famille.PIQUE, Famille.TREFLE`, comme dans l'exemple suivant :

Exemple 4.7.3 Utilisation d'un `enum` en JAVA :

```
Famille f = Famille.COEUR;
```

4.7.1 Méthodes d'instances des `enum`

Chaque classe `enum`, hérite des méthodes définies dans sa super-classe `Enum` :

int ordinal() renvoie un entier associé à cette instance, la valeur de cet entier correspond à l'ordre dans lequel les instances de la classe ont été déclarées : la valeur de `Famille.CARREAU.ordinal()` est 0, celle de `Famille.COEUR.ordinal()` est 1, ...

String name() renvoie une chaîne de caractères composée des mêmes caractères que l'identifiant de cette instance : la valeur de `Famille.CARREAU.name()` est `"CARREAU"`, celle de `Famille.COEUR.name()` est `"COEUR"`, ... Par défaut, la méthode `String toString()` renvoie la même chaîne de caractères que la méthode `name()`

int compareTo(Famille f) permet de comparer deux instances selon la relation d'ordre induite par l'ordre de déclaration des instances de l'enum (identique à la relation d'ordre entre les valeurs renvoyées par la méthode `ordinal()`):`Famille.CARREAU < Famille.COEUR < Famille.PIQUE < Famille.TREFLE`. Renvoie un entier strictement négatif si l'instance est inférieur à l'instance passée en paramètre, 0 en cas d'égalité et un entier strictement positif si l'instance est supérieure à l'instance passée en paramètre. Par conséquent, l'expression `Famille.COEUR.compareTo(Famille.TREFLE)` renvoie un entier strictement négatif.

4.7.2 Méthodes de classes des `enum`

Chaque classe `enum` possède par ailleurs, deux méthode de classes (i.e. déclarées `static`) :

Famille valueOf(String name) renvoie l'instance de `Famille` possédant le nom spécifiée :
`Famille.valueOf("COEUR") == Famille.COEUR`
Notons cependant que l'appel de cette méthode avec un argument ne correspondant pas à une constante du type énuméré considéré déclenchera une exception et l'interruption immédiate de l'exécution du programme.

Famille[] values() renvoie un tableau contenant toutes les instances de la classe `Famille`

4.8 Tableau

En Java, un tableau est un objet d'une classe qui est construite à partir du type des éléments qui sont placés dans le tableau. Il n'y a aucune contrainte sur le type des éléments, on peut utiliser aussi bien les types de bases (int, boolean, char,...), que les types classe, y compris les types tableau eux-mêmes. Le fait que les éléments puissent être de type tableau permet de définir des tableaux à plusieurs dimensions, dans ce cas il s'agit de tableaux de références vers d'autres tableaux (i.e. des tableaux de tableaux). Notons cependant que tous les éléments d'un tableau sont du même type.

Les tableaux doivent être créés avec l'opérateur `new` qui fixe le nombre d'éléments du tableaux. La taille d'un tableau est donc toujours fixée à l'exécution. Par contre, une fois le tableau créé, sa taille ne pourra pas être modifiée.

Tout objet tableau possède un champ `length` qui permet de consulter le nombre de cases du tableau. Ce champ n'est accessible qu'en lecture (c'est un champ `final`), toute affectation du champ `length` est interdite.

Exemple 4.8.1 Utilisation d'un tableau de String (cf. figure 4.1).

```
String[3] tabChaînes;     // interdit: il faut passer par new pour créer le tableau
String[] tabChaînes;      // tabChaînes a la valeur null (si c'est un attribut)
tabChaînes.length         // Engendre une erreur (car la référence est nulle)
tabChaînes = new String[3];  // tabChaînes pointe sur un tableau de 3 cases
                              // tabChaînes[0] = ... = tabChaînes[3] = null
tabChaînes[0] = "chaîne 1";   // Initialisation de la 1ère case
tabChaînes.length             // retourne 3
```

Exemple 4.8.2 Utilisation d'un tableau de tableaux (cf. figure 4.2).

```
double matrice[][];           // Matrice, un tableau de tableaux vaut null
matrice = new double[4][];   // Matrice comporte 4 cases chacune valant null
matrice[1] = new double[2];   // La deuxième ligne contient 2 doubles à 0
matrice[2] = new double[3];   // La troisième ligne contient 3 doubles à 0
matrice[2][2] = 1.5;          // Initialisation de la 3ième case de cette ligne
```

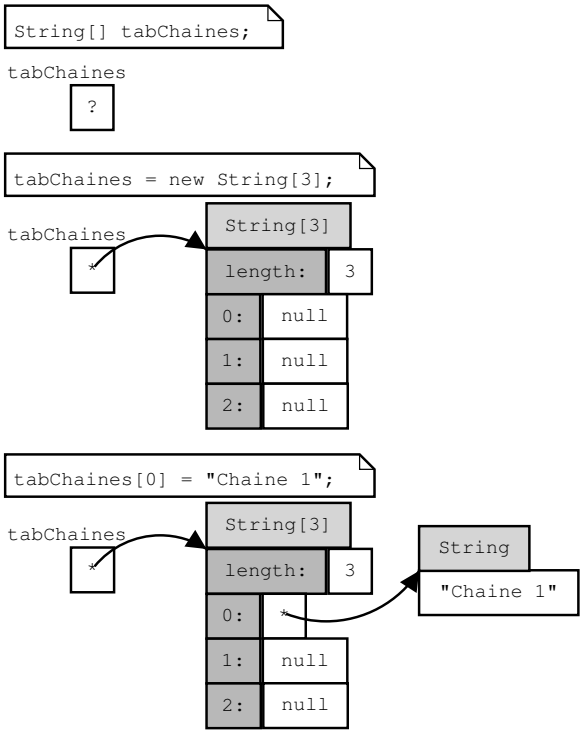

FIGURE 4.1 – Schéma illustrant l'utilisation d'un tableau de String (cf. exemple 4.8.1).

```
matrice.length                // vaut 4
matrice[1].length              // vaut 2
matrice[0].length              // Engendre une erreur
```

4.9 Classe `String`

Les chaînes de caractères ne sont pas des tableaux de caractères comme en C mais des instances de la classe `String`. Ce sont des objets constants, non-modifiables. Une chaîne peut être définie comme un littéral avec la même notation qu'en C :

```
"Philippe Aïch"        "mercredi"      "ligne 1\nLigne 2"
```

4.9.1 Affectation et initialisation

Comme pour toute classe, l'utilisation d'instances de la classe `String` passe par l'utilisation de l'opérateur `new` et d'un constructeur, cependant, le langage JAVA définie des raccourcis de notation permettant de simplifier l'utilisation de cette classe. Ainsi, l'affectation suivante :

```
nom = "Durand";
```

est un raccourci, et est donc équivalent, à l'affectation suivante :

```
nom = new String("Durand");
```

il faut, par conséquent, faire bien attention au fait, que deux variables de type `String` contenant des chaînes de caractères identiques (i.e. composées des mêmes caractères), peuvent être différentes, car contenant des références à des instances distinctes, comme dans l'exemple suivant :

```
String nom1, nom2, nom3;

nom1 = new String("Durand");
nom2 = new String("Durand"); // nom1 == nom2 est false
nom3 = nom1; // nom1 == nom3 est true
```

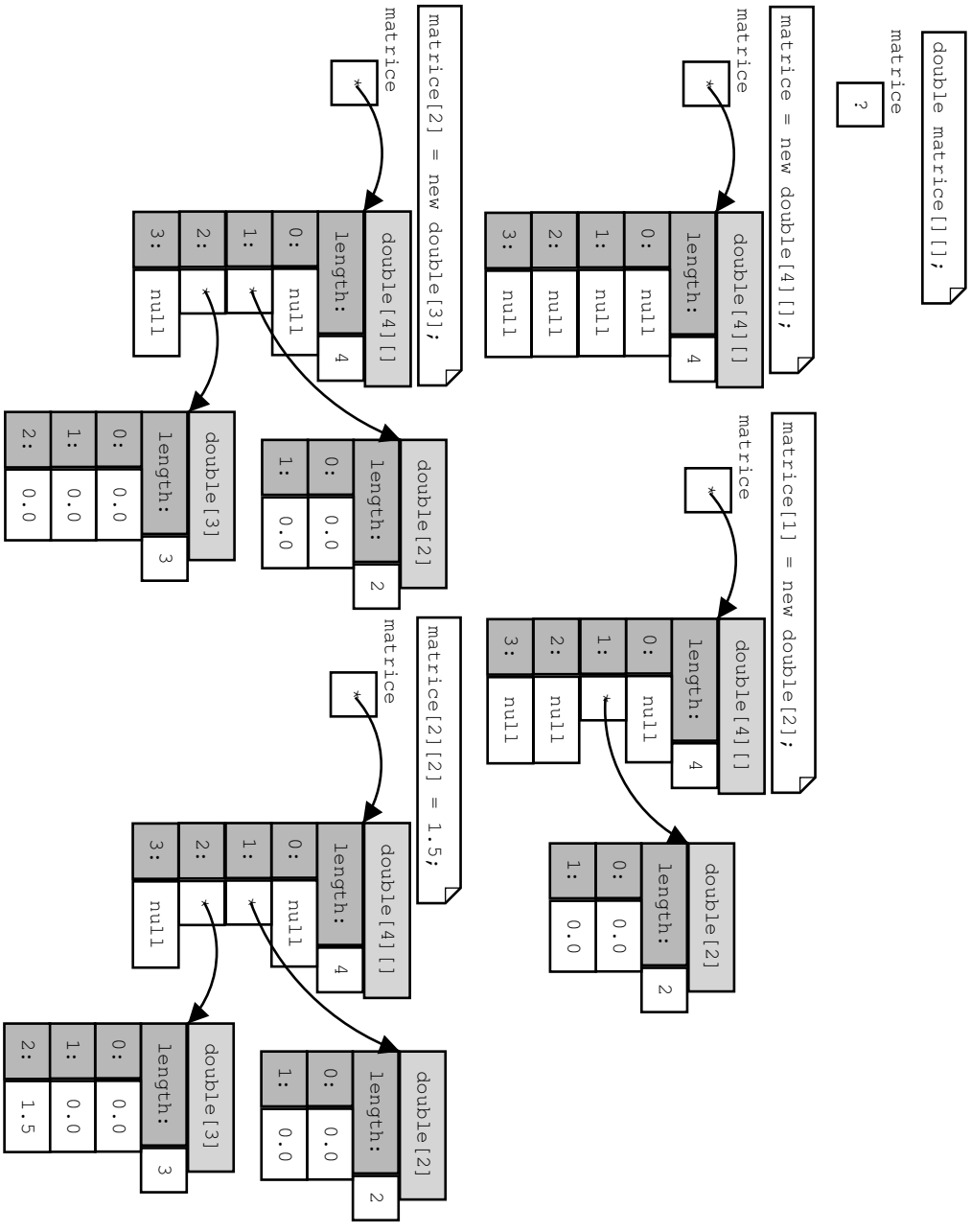

FIGURE 4.2 – Schéma illustrant l'utilisation d'un tableau de tableaux (cf. exemple 4.8.2).

Dans l'exemple ci-dessus, les variables `nom1` et `nom2` contiennent des références à 2 instances distinctes de la classe `String` contenant toutes les deux des chaînes de caractères composées des mêmes caractères. L'expression `nom1 == nom2` est donc `false`. Par contre, la variable `nom3` contient la référence à la même instance que la variable `nom1` et l'expression `nom1 == nom3` est donc bien `true`.

4.9.2 Méthodes et opérateurs

L'opérateur + est l'opérateur de concaténation de deux chaînes de caractères. Si l'un des deux arguments est un nombre, il est converti en une chaîne de caractère. Par exemple :

```
"Jean"+2       --> "Jean2"
"Jean " + "II" --> "Jean II"
```

Méthode	Opération
`int length()`	retourne la longueur de la chaîne
`char charAt(int position)`	retourne le caractère à la position `position` (≥ 0)
`boolean equals(Object o)`	compare deux chaînes
`boolean equalsIgnoreCase(String chaine)`	compare deux chaînes sans distinguer majuscules et minuscules
`int indexOf(char cara)`	retourne la position du 1er caractère `cara` ou -1 si `cara` n'appartient pas à la chaîne
`int indexOf(String ch)`	retourne la position de la sous-chaîne `ch` ou -1 si `ch` n'apparaît pas dans la chaîne
`String substring(int debut, int fin)`	retourne une sous-chaîne
`String toLowerCase()`	retourne une copie en minuscule
`String toUpperCase()`	retourne une copie en majuscule

Parmi, les méthodes présentées ci-dessus nous retiendront particulièrement la méthode `equals` qui permet de tester si deux chaînes de caractères sont identiques (au sens de "contiennent les mêmes caractères"), ce que ne permet pas l'opérateur == comme nous l'avons vu précédemment :

```
String nom1, nom2, nom3;

nom1 = new String("Durand");
nom2 = new String("Durand");        // nom1 == nom2 est false
                                    // mais nom1.equals(nom2) est true
nom3 = nom1;                        // nom1 == nom3 est true
                                    // et nom1.equals(nom2) est true
```

4.10 Comparaison d'objets

Comme nous venons de le voir avec la classe `String`, nous avons souvent besoin d'une notion d'égalité entre objets qui soit différente de celle que permet de tester l'opérateur == : en fait, ce dont nous avons besoin, c'est de pouvoir tester si deux instances d'une même classe sont interchangeables, c'est-à-dire, de pouvoir tester si utiliser une instance à la place de l'autre dans un programme aboutirait au même résultat.

Ce besoin se faisant sentir très fréquemment et quelque soit la classe, en JAVA, cette notion d'égalité est définie pour l'ensemble des classes dans la classe `Object`. En effet, la classe `Object` est une classe particulière dans laquelle sont définies quelques méthodes utilisables sur tout objet et dont toute classe est une sous-classe (quand nous aborderons l'héritage, nous verrons que, en JAVA, toute classe hérite de cette classe `Object`).

Parmi les quelques méthodes définies dans la classe `Object` figure la méthode `equals`. Par conséquent, toute classe peut utiliser cette méthode pour comparer ses instances entre elles. Cependant, le comportement exacte de cette méthode étant très dépendant de la définition de chaque classe, la méthode `equals` de la classe `Object` se comporte exactement comme l'opérateur ==. Par conséquent, toute classe dont on souhaite pouvoir comparer les instances en prenant en compte la valeur de ses attributs (c'est-à-dire en adoptant un comportement différent de l'opérateur ==) doit définir sa propre méthode `equals`.

A titre d'exemple, le code ci-après présente une définition possible de la méthode `equals` pour une classe `Point` représentant des points dans un espace de dimension 2 :

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

```
    // le reste du code est omis

    /**
     *  Test d'égalité comparant les valeurs des coordonnées des deux points.
     *
     * @param p Le point à comparer avec l'instance courante
     * @return true si les deux points ont les mêmes coordonnées false
     *      sinon
     */
    /*@ also
      @ ensures !(obj instanceof Point) ==> !\result;
      @ ensures (obj instanceof Point)  ==>
      @    \result ==  (getX() == ((Point) obj).getX())
      @                && (getY() == ((Point) obj).getY());
      @ pure
      @*/
    public boolean equals(Object obj) {
        if (!(obj instanceof Point)) {
            return false;
        }
        Point p = (Point) obj;
        return getX() == p.getX() && getY() == p.getY();
    }
}
```

Chapitre 5

Programmation par contrat

5.1 Historique

La notion de contrat est assez récente, elle a été introduite par Bertrand Meyer en 1992. Dans EIFFEL, le LOO conçu par Bertrand Meyer, la notion d'assertion est totalement intégrée au langage.

Pour JAVA, les créateurs du langage expliquent que c'est uniquement faute de temps que les assertions ne furent pas intégrées au langage dès son origine, les toutes dernières versions du langage tentent de réparer partiellement cet oubli. En TP, nous utiliserons l'extension JML (Java Modeling Language, cf. `http://www.jmlspecs.org/`) qui fournit un support complet des assertions en JAVA.

UML, un langage utilisé pour la conception de logiciels intègre maintenant le langage OCL qui permet de spécifier des assertions dans le même objectif.

5.2 Pourquoi ?

La programmation par contrat poursuit essentiellement trois objectifs :

— améliorer la correction (i.e. absence de bug) des logiciels construits. L'objectif est que le logiciel soit correct par conception afin de limiter au maximum la phase de correction de bugs qui prend place habituellement une fois que le logiciel est terminé ;

— faciliter l'utilisation et la réutilisation des classes : la programmation par contrat permet d'améliorer la qualité de la documentation et de la rendre plus précise, correcte et fiable ;

— simplifier la maintenance : la programmation par contrat simplifie la correction des bugs en permettant le dépistage de la source du bug de manière plus précise. Elle permet de plus, de clarifier quelles sont les modifications possibles de l'implémentation qui ne risquent pas d'entraîner de modification dans le code utilisateur.

5.3 Comment ?

La programmation par contrat tente d'atteindre les objectifs précédents en décrivant de manière beaucoup plus rigoureuse — voir formelle — les services que peuvent rendre les instances d'une classe.

Cette description appartient à l'interface de la classe. En ce sens, elle appartient à la documentation de la classe. Cependant, cette description permet, en plus, d'évaluer/contrôler si l'implémentation est conforme à l'interface.

Cette description est faite à l'aide d'assertions qui sont des expressions booléennes JAVA. Ces assertions spécifient qu'à l'endroit où elles sont placées certaines propriétés devraient être vraies. Elles peuvent être exécutées / évaluées au cours de l'exécution du programme, elles permettent alors de détecter des erreurs logicielles lors de la phase de test/mise au point.

En utilisation "normale" — dite phase de production — les assertions ne sont pas exécutées. Cependant, le fonctionnement d'un programme et par conséquent les résultats qu'il produit ne dépendent pas du fait que les assertions soient ou non exécutées.

5.4 Qu'appelle t'on contrat ?

La notion de contrat fait référence à la notion de contrat du monde réel qui lie un fournisseur de service à son client.

En programmation orientée objet :

— le fournisseur est celui qui fournit l'implémentation d'une classe ;

— le client est celui qui utilise cette classe sans en connaître l'implémentation ;
— le service rendu est donc l'utilisation d'une classe pour profiter des services qui y sont définis ;
— le contenu du contrat — à quoi s'engage chacune des parties — est exprimé dans l'interface de la classe à l'aide des assertions (et de commentaires).

5.4.1 Qu'est-ce qu'une assertion ?

— Comme cela a été dit précédemment, une assertion est une expression booléenne JAVA exécutable.
— Le fait d'exécuter ou de ne pas exécuter une assertion ne doit modifier ni le comportement du programme, ni les résultats du programmes (i.e. pas d'effet de bord).
— Il existe trois catégories d'assertion. A chaque catégorie, un rôle précis est assigné :

l'invariant de classe est associé à l'ensemble de la classe et décrit des propriétés qui doivent être vraies de toute instance de la classe à tout moment de l'existence de cette instance. L'expression qui lui est associée doit toujours être vraie *avant* et *après* exécution de chaque méthode de la classe et *après* exécution de chaque constructeur.

la pré-condition est relative à une méthode. L'expression qui lui est associée doit toujours être vraie *avant* exécution de la méthode correspondante. En général cette expression porte sur les arguments de la méthode ou sur l'état de l'objet ;

la post-condition est relative à une méthode. L'expression qui lui est associée doit toujours être vraie *après* exécution de la méthode correspondante. Cette expression sert à spécifier l'effet attendu de l'exécution de la méthode ;

5.4.2 Quelle est la nature du contrat ?

Comme dans la vie réelle, un contrat définit un ensemble de droits et de devoirs pour chacune des parties liées par le contrat (i.e. le(s) client(s) et le fournisseur) telles que :
— si une des parties remplit ses obligations, l'autre partie s'engage à assurer les droits du premier ;
— si une des parties ne remplit pas sa part du contrat (devoirs non assurés), il perd tous ses droits : l'autre partie n'est plus liée par le contrat et peut faire n'importe quoi.

	Obligations/Responsabilités	Droits/Garanties
Client	Assurer que la pré-condition est satisfaite	La post-condition et l'invariant sont satisfaits
Fournisseur	Assurer que la post-condition et l'invariant sont satisfaits. Offrir au client les moyens de vérifier que la pré-condition est satisfaite.	Inutile de tester la pré-condition. L'implémentation de la méthode peut supposer que la pré-condition est satisfaite.

5.5 Un exemple

Nous allons prendre comme exemple la classe `Carte` dont chaque instance doit permettre de représenter / tenir le rôle d'une carte d'un jeu de cartes utilisé dans un jeu de réussite.

5.5.1 Définir les caractéristiques et le rôle de la classe

La première chose à faire avant même de choisir les champs ou méthodes de la classe est de définir les caractéristiques et le rôle de la classe. Cela nécessite de répondre aux questions suivantes :
— quelles sont les propriétés essentielles d'une carte à jouer dans le contexte de l'application ? C'est-à-dire les propriétés qui font qu'une instance de la classe pourra tenir le rôle d'une carte ;
— comment seront utilisées les instances de la classe, à quoi servent elles ? Il peut aussi être important de préciser à quoi les instances ne pourront pas être utilisées ;
— quelles propriétés devront être accessibles à partir des autres classes ? quelles propriétés devront pouvoir être modifiées ? comment pourront elles être modifiées ? quelles propriétés devront rester invariantes ?

Pour la classe `Carte`, les propriétés essentielles sont :

une famille toute carte possède une famille qui peut être cœur, carreau, pique ou trèfle ;

une couleur toute carte possède une couleur qui dépend de sa famille : rouge pour cœur ou carreau et noir pour pique ou trèfle ;

un rang qui peut être As, 2, 3, 4, ..., 10, Valet, Dame ou Roi ;

une face sur laquelle sont indiquées son rang, sa famille et sa couleur. Cette carte peut être visible ou non ;

un ordre entre les cartes dans le jeu de réussite, cet ordre sera utilisé pour déterminer si une carte peut être posée sur une autre carte.

emplacement des cartes il n'est pas nécessaire qu'une carte connaisse son emplacement, chaque carte fera partie d'un objet conteneur (pile ou colonne) qui lui même devra connaître l'emplacement de la carte ou bien savoir le calculer en cas de besoin.

Utilisation des instances de la classe Pour le jeu du Solitaire, nous avons besoin de pouvoir consulter la famille, la couleur et le rang des cartes pour pouvoir les afficher. Il doit être possible de comparer deux cartes pour savoir si l'une peut être placée au dessus de l'autre, cette comparaison doit pouvoir se faire de 2 manières différentes :

— dans une colonne, la carte à placer doit être de couleur opposée et de rang immédiatement inférieur ;

— dans une pile, les deux cartes doivent être de même famille et celle du dessus de rang immédiatement supérieur.

Enfin, on doit pouvoir savoir si une carte est face visible ou retournée et le cas échéant il faut pouvoir la retourner.

On choisira ici de ne pas prendre en charge l'affichage dans cette classe (une autre classe devra être conçue spécialement dans ce but).

Accessibilité des caractéristiques Toutes les caractéristiques mentionnées précédemment doivent pouvoir être consultées. Par contre, seul l'état "face visible ou non" doit pouvoir être modifié durant la vie de l'objet, en effet, durant une partie de cartes, il n'est pas concevable qu'une carte change de famille, couleur ou rang.

5.5.2 Version préliminaire de l'invariant

A partir des informations précédentes, on peut définir une première ébauche en français de l'invariant puisque nous n'avons pas encore décidé quels seront les champs et méthodes des instances de la classe :

— la couleur d'une carte est toujours soit rouge, soit noire et rien d'autre ;

— la famille d'un carte est une des quatre valeurs : cœur, carreau, pique ou trèfle ;

— une carte est de couleur rouge *si et seulement si* sa famille est cœur ou carreau ;

— une carte est de couleur noire *si et seulement si* sa famille est pique ou trèfle ;

— le rang d'une carte est une des valeurs : As, 2, 3, ..., 10, Valet, Dame ou Roi ;

— une carte est soit "face visible", soit "face cachée".

5.5.3 Définition d'un constructeur

Le rôle d'un constructeur est de mettre en place l'invariant, c'est-à-dire de faire en sorte qu'il soit satisfait après l'appel du constructeur. Les autres méthodes devront s'assurer qu'il reste vérifié à la fin de chaque méthode.

L'invariant de classe étant maintenant connu, nous savons ce que devra faire le constructeur : initialiser chaque instance afin que l'invariant soit vérifié. Il ne reste donc plus qu'à déterminer les arguments du constructeur. Compte tenu du rôle de cette classe dans l'application, il semble pertinent de considérer que les instances seront créées en fixant leur famille et leur rang. La couleur sera alors déduite de la famille et la carte sera initialisée dans l'état "face cachée" (au début d'une partie de réussite, toutes les cartes sont cachées sauf quelques unes : tête de colonne ou première carte de la pioche).

5.5.4 Choix des méthodes

En fonction de ce qui précède, nous pouvons maintenant choisir les méthodes.

Pour donner l'entête des méthodes (en JAVA) nous devons néanmoins choisir un codage des différentes propriétés en JAVA : nous devons en effet choisir les types à utiliser pour les différentes propriétés des cartes. Nous choisissons ici de représenter la couleur, la famille et le rang à l'aide de 3 types `enum` et la visibilité des cartes à l'aide du type `boolean`.

Méthodes d'accès aux propriétés

Couleur `Couleur getCouleur()` renvoie la couleur de la carte ;

Famille `Famille getFamille()` renvoie la famille de la carte ;

Rang `Rang getRang()` renvoie le rang de la carte ;

Face visible ? `boolean estFaceVisible()` pour savoir si la face de la carte est visible ou non ;

Modification des propriétés La seule propriété modifiable est la visibilité de la carte :

Face visible `void retourner()` pour changer la visibilité de la carte.

Comparaison entre cartes

Piles `boolean estEmpilableSur(Carte autreCarte)` Détermine si l'instance courante peut-être empilée (placée au sommet d'une pile) sur la carte `autreCarte`;

Colonnes `boolean estSuccesseurColonneDe(Carte autreCarte)` Détermine si l'instance courante peut-être placée sur la carte `autreCarte`, dans une colonne;

5.5.5 Spécification de l'effet des méthodes par des post-conditions

Le rôle essentiel des post-conditions est de définir l'effet des méthodes.

Constructeur Le rôle principal d'un constructeur étant d'établir l'invariant de classe, l'effet du constructeur est en grande partie exprimé par l'invariant. Cependant la post-condition peut servir à exprimer un état particulier de l'instance au moment de sa création : par exemple, pour la classe `Carte` il semble pertinent de spécifier que toutes les cartes doivent être créées *face non visible*. Nous avons donc la post-condition : `!estFaceVisible()` (sous entendu `estFaceVisible() == false`).

D'autre part, lorsque le constructeur possède des arguments, on exprime comment vont être initialisées les instances en fonction des valeurs des arguments. Pour la classe `Carte`, avec un constructeur d'en-tête `public Carte(Famille famille, Rang rang)` on aurait :

```
(getFamille() == famille) && (getRang() == rang)
```

Méthodes d'accès aux propriétés Pour ces méthodes, l'effet est de retourner la valeur d'une propriété particulière. Cet effet ne peut pas être caractérisé à l'aide d'une assertion. Ce qui peut par contre être spécifié, ce sont des propriétés sur les valeurs retournées. Dans le cas de la classe `Carte`, les propriétés des valeurs retournées sont celles spécifiées par l'invariant, il n'y a donc pas lieu de définir des post-conditions.

Modifications des propriétés L'effet de la méthode `retourner()`, est d'inverser la visibilité de la carte : si elle était visible, elle devient non visible et si elle était non visible, elle devient visible. La post-condition doit donc spécifier que la valeur renvoyée est la négation – au sens négation logique – de la valeur antérieure de la propriété.

En programmation par contrat, la valeur renvoyée par une méthode peut être référencée en utilisant le mot clé `\result`. Pour faire référence à la valeur de la propriété, nous utilisons la méthode définie dans ce but : `estFaceVisible()` et pour exprimer le fait que nous voulons faire référence à la valeur antérieure de la propriété nous utilisons l'expression `\old(expr)`, qui nous permet de faire référence à la valeur de l'expression `expr` *avant l'appel de la méthode*.

L'assertion s'écrit donc :

```
\result != \old(estFaceVisible());
```

ou bien

```
\result == !\old(estFaceVisible());
```

Comparaison entre cartes Dans ce cas, la post-condition doit exprimer comment l'ordre est défini. Ces deux méthodes de comparaison nécessitent, toute deux, de définir au préalable l'ordre entre les rang des cartes. Pour simplifier, nous choisissons ici d'utiliser le fait que le rang est codé par un entier et d'exploiter l'ordre sur les entiers :

`boolean estEmpilableSur(Carte autreCarte)` La comparaison à l'intérieur d'une pile s'exprime par :

```
\result == ((this.getFamille() == autreCarte.getFamille())
        && (this.getRang().ordinal() == autreCarte.getRang().ordinal() + 1));
```

`boolean estSuccesseurColonneDe(Carte autreCarte)` La comparaison à l'intérieur d'une colonne s'exprime par :

```
\result == ((this.getCouleur() != autreCarte.getCouleur())
        && (this.getRang().ordinal() == autreCarte.getRang().ordinal() - 1));
```

5.5.6 Ajout des pré-conditions

Le rôle des pré-conditions est d'exprimer, dans quels cas les méthodes sont applicables. Le plus souvent, les pré-conditions spécifient des conditions sur les arguments de la méthode. Par exemple, lorsqu'un des arguments est un objet on demande que soit valeur soit une référence valide. Pour la classe `Carte`, seules les méthodes de comparaison sont concernées, on leur associe à toutes les deux la même pré-condition :

```
autreCarte != null
```

5.5.7 Version finale de l'invariant

On définit d'abord les classes enum permettant de représenter le rang, la famille et la couleur.

Définition de la classe enum `Rang`

```
public enum Rang {
        AS, DEUX, TROIS, QUATRE, CINQ, SIX, SEPT, HUIT, NEUF, DIX,
        VALET, DAME, ROI
}
```

Définition de la classe enum `Famille`

```
public enum Famille {
         CARREAU, COEUR, PIQUE, TREFLE
}
```

Définition de la classe enum `Couleur`

```
public enum Couleur { ROUGE, NOIR }
```

Invariant

```
getFamille() != null;
getCouleur() != null;
getRang() != null;
((getFamille() == Famille.CARREAU || getFamille() == Famille.COEUR)
    == (getCouleur() == Couleur.ROUGE)
&& (getFamille() == Famille.PIQUE || getFamille() == Famille.TREFLE)
    == (getCouleur() == Couleur.NOIR))
```

5.5.8 La classe `Carte` complète avec assertions

Les assertions sont données en respectant la syntaxe JML.

```
/**
 *  Notion usuelle de carte à jouer, par exemple pour les jeux de
 *  réussite : jeu de 52 cartes, avec les quatre familles habituelles.
 *
 * @author     Marc Champesme
 * @created    3 mars 2003
 */
public class Carte {
        /*@
          @ invariant getFamille() != null;
          @ invariant getCouleur() != null;
          @ invariant getRang() != null;
          @ invariant (getFamille() == Famille.CARREAU || getFamille() == Famille.COEUR)
                              == (getCouleur() == Couleur.ROUGE);
          @ invariant (getFamille() == Famille.PIQUE || getFamille() == Famille.TREFLE)
                              == (getCouleur() == Couleur.NOIR);
          @*/
```

```
private int famille;
private int rang;
private boolean estVisible;



/**
 *  Initialise une carte à partir des éléments fournis en paramètre.
 *
 * @param  famille  Famille de la nouvelle carte.
 * @param  rang     Rang de la nouvelle carte.
 */
/*@
  @ requires famille != null;
  @ requires rang != null;
  @ ensures getFamille() == famille;
  @ ensures getRang() == rang;
  @ ensures !estFaceVisible();
  @*/
public Carte(Famille famille, Rang rang) {
        this.famille = famille;
        this.rang = rang;
}


/**
 *  Renvoie la couleur de la carte (i.e. ROUGE ou NOIR)
 *
 * @return    La couleur de la carte
 */
//@ pure
public Couleur getCouleur() {
        int couleur;

        if ((famille == Famille.CARREAU) || (famille == Famille.COEUR)) {
                couleur = Couleur.ROUGE;
        } else {
                couleur = Couleur.NOIR;
        }

        return couleur;
}


/**
 *  Renvoie la famille de la carte c'est à dire une des valeurs
 *  CARREAU, COEUR, PIQUE ou TREFLE.
 *
 * @return    La famille de la carte
 */
//@ pure
public Famille getFamille() {
        return famille;
}


/**
 *  Renvoie le rang de la carte (i.e. AS, DEUX, ... ou ROI).
 *
 * @return    Le rang de la carte
```

```
 */
//@ pure
public Rang getRang() {
        return rang;
}


/**
 *  Indique si la face de la carte est visible. La face de la carte
 *  est le côté sur lequel le rang, la famille et la couleur sont
 *  visibles.
 *
 * @return    true si la face de la carte est visible, false sinon.
 */
//@ pure
public boolean estFaceVisible() {
        return estVisible;
}


/**
 *  Inverse la visibilité de la carte : si elle était visible, elle
 *  devient non visible et si elle était non visible, elle devient
 *  visible
 */
//@ ensures estFaceVisible() == !\old(estFaceVisible());
public void retourner() {
        estVisible = !estVisible;
}


/**
 *  Comparaison de la carte courante avec la carte fournie en
 *  paramètre. L'ordre utilisé pour la comparaison est celui valable
 *  à l'intérieur d'une pile
 *
 * @param  autreCarte  La carte avec laquelle la comparaison
 *      s'effectue
 * @return             true si la carte courante est empilable sur
 *      autreCarte
 */
/*@ requires autreCarte != null;
  @ ensures \result ==
          ((this.getFamille() == autreCarte.getFamille())
        && (this.getRang().ordinal() == autreCarte.getRang().ordinal() + 1));
  @ pure
  @*/
public boolean estEmpilableSur(Carte autreCarte) {
        return ((famille == autreCarte.getFamille())
                        && (rang == autreCarte.getRang().ordinal() + 1));
}


/**
 *  Comparaison de la carte courante avec la carte fournie en
 *  paramètre. L'ordre utilisé pour la comparaison est celui valable
 *  à l'intérieur d'une colonne
 *
 * @param  autreCarte  carte avec laquelle la comparaison s'effectue
 * @return             true si la carte courante est un successeur
```

```
         *      possible de autreCarte
         */
        /*@ requires autreCarte != null;
          @ ensures \result ==
                  ((this.getCouleur() != autreCarte.getCouleur())
                 && (this.getRang().ordinal() == autreCarte.getRang().ordinal() - 1));
          @ pure
          @*/
        public boolean estSuccesseurColonneDe(Carte autreCarte) {
                return ((this.getCouleur() != autreCarte.getCouleur())
                        && (rang == autreCarte.getRang().ordinal() - 1));
        }
}// -- Fin de la classe Carte
```

5.6 Quelques règles pour l'écriture des assertions

5.6.1 Effets de bord et méthodes "pure"

Toute expression booléenne JAVA bien formée peut être utilisée pour écrire une assertion, à la condition que l'évaluation de cette expression ne change pas les propriétés ou l'état de l'objet. Cette condition peut s'exprimer autrement de la façon suivante :

Les assertions doivent être *sans effet de bord*

La raison d'être de cette condition est que le fonctionnement d'un programme doit être strictement identique que l'on évalue ou non les assertions. Les assertions ne font pas partie du programme, elles ne font qu'exprimer ce que devrait faire le programme s'il était correct (sous réserve, bien sur, qu'il n'y ait pas d'erreur dans les assertions).

Les assertions utilisant souvent des appels de méthodes, il convient donc de s'assurer que les méthodes utilisées sont sans effet de bord. A cet effet, JML introduit le mot-clé `pure` pour qualifier de telles méthodes. Ce mot clé doit être utilisé de la manière suivante :

— Avec JML, seules les méthodes spécifiée *pure* peuvent-être utilisées dans les assertions.

— Toute méthode sans effet de bord doit être qualifiée de *pure*. En effet, même si cette méthode n'est pas utilisée dans l'écriture des assertions de la classe courante, il faut laisser le plus de souplesse possible au programmeur d'autres classes qui utiliseraient la classe courante et pourraient donc avoir besoin d'utiliser les méthodes de la classe courante pour écrire les assertions de ces nouvelles classes.

5.6.2 Visibilité

Les champs et méthodes utilisés dans l'écriture d'une assertion doivent être de visibilité égale ou supérieur à la visibilité de la méthode à laquelle se rapportent cette assertion :

— pour une méthode `public`, tous les champs ou méthodes utilisés dans l'écriture des assertions doivent être `public`. Dans certains cas cela nécessite de définir des méthodes `public` pour accéder à des champs `private`.

— pour une méthode `protected`, tous les champs ou méthodes utilisés dans l'écriture des assertions doivent être `public` ou `protected`.

— pour une méthode `private`, tout champ ou méthode (de la classe) peut être utilisé dans l'écriture des assertions.

La raison d'être de cette règle est que le client doit pouvoir tester lui-même les assertions, or le client d'une méthode `public` n'a pas en général accès aux champs et méthodes de visibilité plus restreinte. En particulier, dans la majorité des cas, le client ne connaît même pas l'existence des champs de visibilité `private`.

Ceci est particulièrement important pour les pré-conditions, puisque c'est au client de s'assurer que les pré-conditions sont satisfaites : le fournisseur doit donc mettre à la disposition du client les champs et méthodes nécessaires pour cela.

5.6.3 Eléments syntaxiques spécifiques aux assertions

La syntaxe définie pour l'écriture des assertions est basée sur la syntaxe JAVA, cependant, la syntaxe des assertions est plus étendues Nous avons vu en particulier deux mots clés supplémentaires utilisables uniquement dans les assertions :

`\result` : Ce mot-clé s'utilise uniquement dans la post-condition d'une méthode renvoyant un résultat. Dans la post-condition, `\result` s'utilise comme une variable dont la valeur est la valeur renvoyée par la méthode. `\old()` : Comme`\result`, le mot-clé `\old()` s'emploie uniquement dans une post-condition. `\old()` s'utilise comme une expression quelconque dont le type est celui de l'expression passée en paramètre. La valeur d'une expression de la forme `\old(expr)` est la valeur de l'expression `expr` *avant* exécution de la méthode.

Chapitre 6

Héritage

La raison d'être essentielle de l'héritage est la réutilisation, or, nous avons vu précédemment que la réutilisation était un des objectifs majeurs de la programmation orientée objet. L'héritage est par conséquent une notion centrale en programmation orientée objet.

6.1 Un exemple classique

Nous allons tout d'abord voir sur un exemple à quoi peut servir l'héritage et comment il s'utilise. Voici une présentation de l'exemple sur lequel nous allons travailler :

> Nous allons supposer que nous voulons réaliser une application qui :
> — affiche des polygones ;
> — peut leur appliquer certaines transformations comme, par exemple, la symétrie par rapport à l'axe des ordonnées ;
> — et est capable de calculer certaines de leurs caractéristiques comme le périmètre ou la surface.

Nous avons déjà abordé en TD la réalisation d'une classe `Polygone`. Nous avons aussi vu en TD les classes `Triangle` et `Rectangle`. La géométrie nous dit que tout triangle *est un* polygone et que tout rectangle *est un* polygone (cf. figure 6.1). Un premier aperçu de ce qu'est l'héritage peut-être donné en disant que l'héritage permet de prendre en compte explicitement cette relation *est un* dans la programmation de ces trois classes : le rôle de l'héritage sera d'introduire entre les classes une relation correspondant au mieux à cette relation *est un*. En programmation orientée objet, cette relation est la relation *est sous-classe de* synonyme de *hérite de*. Pour mettre en œuvre cette relation nous définirons les classes `Triangle` et `Rectangle` comme des sous-classes de la classe `Polygone`. On dira alors que les classes `Triangle` et `Rectangle` héritent de la classe `Polygone` et que la classe `Polygone` est la super-classe des classes `Triangle` et `Rectangle`.

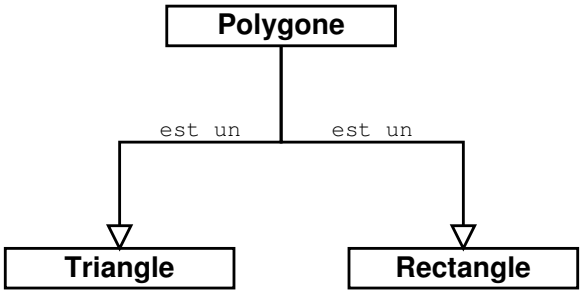

FIGURE 6.1 – La relation *est un*.

6.1.1 Pourquoi pas une seule classe ?

Avant de définir plus précisément la notion d'héritage revenons à la classe `Polygone` : puisque tout triangle et tout rectangle sont des polygones, pourquoi ne pas se contenter d'utiliser une seule classe au lieu d'en définir plusieurs ?

Nous pourrions en effet tout à fait imaginer d'utiliser la classe `Polygone` pour représenter et traiter des rectangles ou des triangles. Nous pourrions définir dans la classe `Polygone` des méthodes s'appliquant à tout polygone et, par

conséquent, s'appliquant aussi aux rectangles et aux triangles. Dans ce cas, quelle nécessité avant nous de définir des classes `Triangle` ou `Rectangle` ?

La première réponse que nous pouvons apporter est que, la prise en compte des propriétés spécifiques des rectangles et des triangles permettrait de réaliser certaines des opérations de manière beaucoup plus efficace. Par exemple, pour les rectangles, les côtés sont égaux deux à deux : pour calculer le périmètre d'une rectangle, il suffit de calculer la longueur et la largeur puis d'additionner le double de la longueur avec le double de largeur. Par opposition, le calcul du périmètre pour un polygone quelconque, nécessite de calculer la longueur de chaque côté et de faire la somme de toutes les longueurs obtenues.

Nous pourrions bien sûr, intégrer cette méthode de calcul spécifique aux rectangles dans la classe `Polygone` : pour calculer le périmètre, il faudrait alors tester si le polygone est un rectangle, puis, le cas échéant, exécuter le code optimisé pour les rectangles au lieu du code prévu pour le cas général. Le problème d'une telle approche est que le code de la classe `Polygone` devient très complexe : pour chaque méthode nous allons devoir tester si le polygone est un rectangle, un triangle, un carré, un losange... avant de pouvoir utiliser le *bon code performant*. De plus, l'exécution des tests peut en lui-même entraîner une pénalité en terme de performance, nous risquons ainsi de perdre tout le bénéfice escompté.

Un deuxième argument importante est que la création de plusieurs classes distinctes est seule à même de nous permettre de définir des caractéristiques de rectangle ou triangle qui n'ont pas de sens pour un polygone en général. Par exemple, les notion de largeur ou de longueur sont importantes pour les rectangles, mais n'ont pas de sens pour les polygones en général (et pour les triangles en particulier).

Enfin, cette approche nécessite de modifier cette classe unique (y compris son interface et son contrat) à chaque fois que nous souhaitons prendre en compte une nouvelle figure, ce qui est une pratique à éviter, car elle risque d'introduire des erreurs dans le code pré-existant utilisant cette classe.

6.1.2 Comment prendre en compte les similarités ?

Nous en arrivons donc à la conclusion que la création de plusieurs classes distinctes est nécessaire pour prendre en compte les spécificités de certaines instances de la classe `Polygone`, comme les triangles ou les rectangles. Cependant, dans cette approche les fortes relations qui existent entre les trois classes ne sont pas du tout prises en comptes : les trois classes sont totalement indépendantes et, au niveau du programme il n'existe aucun lien entre elles. Chaque classe contient donc sa propre méthode pour calculer la surface, le périmètre, retourner un objet symétrique ou afficher (cf. figure 6.2).

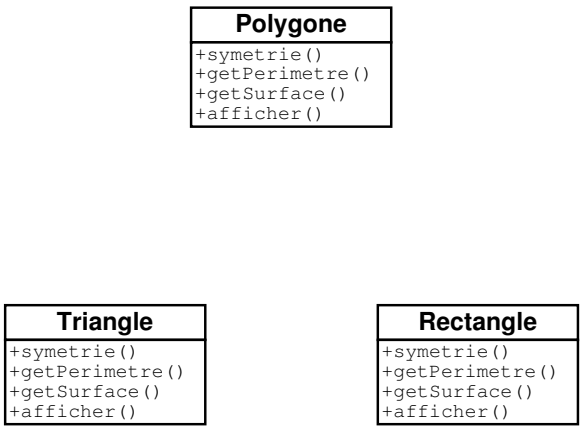

FIGURE 6.2 – Trois classes distinctes.

Or, bien qu'il soit nécessaire que certaines méthodes soit programmées de manière spécifique selon la classe, cela n'est pas nécessairement le cas pour toutes les méthodes : par exemple, la méthode de calcul du périmètre de la classe `Polygone` est probablement tout à fait adaptée à la classe `Triangle`. Cette solution en trois classes distinctes nous oblige à réécrire le code de la méthode périmètre dans `Polygone` et `Triangle` sans réelle nécessité. Dans ce cas, il serait nettement préférable d'avoir une seule méthode à écrire et de pouvoir utiliser cette même méthode qu'il s'agisse d'un triangle ou d'un polygone quelconque. En particulier, avoir une seule méthode limite le travail de maintenance (amélioration des performances et correction de bogues) à assurer.

Dans ce cas, nous aimerions pouvoir dire que la méthode à utiliser pour calculer le périmètre pour les instances de la classe *Triangle* est la même que celle de la classe *Polygone*. C'est ici que l'héritage va pouvoir nous aider : le simple fait de dire que les classes `Triangle` et `Rectangle` héritent de la classe `Polygone` donne à toutes les instances des classes `Triangle` et `Rectangle` l'accès à toutes les caractéristiques de la classe `Polygone`. En JAVA, il suffit d'écrire :

```
public class Polygone {
        public double getSurface() {
                // Code pour calculer la surface d'un polygone quelconque
        }

        public double getPerimetre() {
                // Code pour calculer le périmètre d'un polygone quelconque
        }

        public Polygone symetrie() {
                // Code pour générer le symétrique d'un polygone quelconque
        }

        public void afficher() {
                // Code pour afficher un polygone quelconque
        }

}

public class Triangle extends Polygone {
        // L'utilisation du mot clé extends indique que la classe Triangle hérite
        // de la classe Polygone:
        // les instances de la classe Triangle ont accès à (héritent de) toutes les
        // méthodes de la classe Polygone
}

public class Rectangle extends Polygone {
        // L'utilisation du mot clé extends indique que la classe Rectangle hérite
        // de la classe Polygone:
        // Les instances de la classe Rectangle ont accès à (héritent de) toutes les
        // méthodes de la classe Polygone
}
```

Redéfinition Cependant, le simple usage du mot clé `extends` ne suffit pas : dans les déclarations précédentes `Rectangle` et `Triangle` possèdent exactement les mêmes méthodes que la classe `Polygone`. Ce que nous voulons en fait, c'est pouvoir adapter les caractéristiques de la classe `Polygone` aux spécificités des triangles et des rectangles. Par exemple, pour la classe `Rectangle`, nous voulons qu'une version plus efficace des méthodes `getSurface` et `getPerimetre` soit utilisée à la place des méthodes correspondantes de la classe `Polygone`, et nous voulons que ces méthodes se substituent à celles de `Polygone`. Pour faire cela, nous allons utiliser la redéfinition :

```
public class Rectangle extends Polygone {
        // L'utilisation du mot clé extends indique que la classe Rectangle hérite
        // de la classe Polygone:
        // Les instances de la classe Rectangle ont accès à (héritent de) toutes les
        // méthodes de la classe Polygone sauf getSurface et getPerimetre qui sont
        // redéfinies:

        @Override
        public double getSurface() {
                // Code spécifique à la classe Rectangle pour calculer la surface
        }

        @Override
        public double getPerimetre() {
                // Code spécifique à la classe Rectangle pour calculer le périmètre
        }

        // Les méthodes symetrie() et afficher() de Polygone restent utilisables
        // sur les instances de la classe Rectangle
}
```

Ajout de nouvelles caractéristiques L'héritage ne nous empêche aucunement d'ajouter de nouvelles caractéristiques à la sous-classe : toujours pour la classe `Rectangle`, nous pouvons ajouter des caractéristiques qui n'ont de sens que pour les rectangles, comme, par exemple, des méthodes `getLargeur()` et `getLongueur()`.

```
public class Rectangle extends Polygone {
        // L'utilisation du mot clé extends indique que la classe Rectangle hérite
        // de la classe Polygone:
        // Les instances de la classe Rectangle ont accès à toutes les
        // méthodes de la classe Polygone sauf getSurface et getPerimetre qui sont
        // redéfinies:

        @Override
        public double getSurface() {
                // Code spécifique à la classe Rectangle pour calculer la surface
        }

        @Override
        public double getPerimetre() {
                // Code spécifique à la classe Rectangle pour calculer le périmètre
        }

        public double getLargeur() { // Méthode spécifique à la classe Rectangle
                // Code calculant la largeur du rectangle
        }

        public double getLongueur() { // Méthode spécifique à la classe Rectangle
                // Code calculant la longueur du rectangle
        }
        // Les méthodes symetrie() et afficher() de Polygone restent utilisables
        // sur les instances de la classe Rectangle
}
```

Redéfinition partielle : utilisation du mot clé `super` Dans certains cas, la redéfinition d'une méthode est nécessaire, mais l'implémentation de cette méthode dans la super-classe peut ou doit être réutilisée dans le code de la méthode redéfinie. Dans ce cas, le langage offre le moyen de faire appel dans la sous-classe à l'implémentation définie dans la super-classe. En JAVA, ce mécanisme repose sur l'utilisation du mot clé `super`, qui permet – en l'utilisant de manière analogue au mot clé `this` – de faire référence à l'instance comme s'il s'agissait d'une instance de la super-classe (sans pour autant enfreindre les règles de visibilité), comme dans l'exemple ci-dessous :

```
public class CowBoy {
    private String nom;

    public CowBoy(String nom) {
        this.nom = nom;
    }

    public void presentation() {
        System.out.print("Je suis " + nom + " un cow-boy");
    }
}

public class Brigand extends CowBoy {
    private int recompense;

    public Brigand(String nom, int recompense) {
        super(nom); // cf. section suivante
        this.recompense = recompense;
    }

    @Override
    public void presentation() {
```

```
        super.presentation(); // exécution de l'implémentation de la super-classe
        System.out.println(" malhonnête);
        System.out.println(" Ma tête est mise a pris " + recompense + " $");
    }
}
```

Retirer des méthodes est interdit L'ajout et la redéfinition sont les seules modifications qu'une sous-classe peut apporter à sa super-classe. Ce qui pourrait sembler être une limitation est en faite une conséquence directe de ce que signifie la notion d'héritage (i.e. la relation *est un*) : en effet, la signification de la relation *est un* est justement que tout ce qu'il est possible de faire avec la super-classe (`Polygone` dans l'exemple) est possible sur la sous-classe (`Rectangle` et `Triangle` dans l'exemple).

Cette contrainte va même encore plus loin : toutes les méthodes redéfinies dans la sous-classe doivent posséder la même sémantique. Par exemple, la méthode `getPerimetre` redéfinie dans la classe `Rectangle` doit effectivement calculer le périmètre du rectangle et non la longueur de la diagonale (par exemple). Cependant, autant le compilateur ou la définition du langage peuvent empêcher le programmeur de supprimer une méthode dans une sous-classe, autant la préservation de la sémantique d'une méthode ne peut être contrôlée de manière automatique. Nous verrons cependant que la programmation par contrat peut nous offrir certaines garanties : lorsqu'une classe hérite d'une autre, en plus de ses caractéristiques elle hérite de son contrat (mais nous reviendrons plus tard sur ce point).

6.1.3 Constructeurs et héritage

En ce qui concerne l'héritage, les constructeurs (cf. section 3.9.2) se comportent d'une manière très différente des méthodes :

- Il n'y a pas d'héritage des constructeurs.
- Avant d'exécuter les instructions composant le corps d'un constructeur un constructeur de la super-classe doit impérativement être exécuté, ce même principe s'appliquant de manière identique pour la super-classe, chaque appel à un constructeur implique un chaînage des appels de constructeurs jusqu'au sommet de la hiérarchie d'héritage (cf. figure 6.3). L'appel à ce constructeur de la super-classe peut-être implicite ou explicite :

  Appel implicite si le constructeur de la sous-classe ne contient pas d'appel explicite à un constructeur de la super-classe, c'est le constructeur sans arguments (implicite ou explicite) de la super-classe qui est appelé : le constructeur contient donc implicitement un appel de la forme `super()` comme première instruction. Dans ce cas, il est donc impératif que la super-classe possède un constructeur sans arguments, qu'il soit implicite ou explicite.

  Appel explicite si la super-classe ne possède pas de constructeur sans arguments (implicite ou explicite) ou si on veut choisir le constructeur de la super-classe à appeler, on doit faire un appel de la forme `super(arg1, arg2, ..., argn)` pour exécuter le constructeur à n arguments ($n > 0$) de la super-classe. Cet appel doit *impérativement* être la première instruction du constructeur.

6.2 Polymorphisme

Nous allons voir maintenant un autre avantage que nous pouvons tirer de l'héritage : le polymorphisme. Comme nous l'avons vu dans la section précédente, lorsqu'une classe hérite d'une autre, la sous-classe possède nécessairement toutes les méthodes de la super-classe et, si le programmeur fait correctement son travail les méthodes héritées possèdent la même sémantique. Tant que nous utilisons des caractéristiques (champs ou méthodes) de la super-classe, les instances de la super-classe et de ses sous-classes s'utilisent de la même manière. Ce que l'on appelle *polymorphisme* repose justement sur cette propriétés : le principe du polymorphisme est que des instances de classes différentes peuvent être utilisées comme si elles étaient de la même classe pourvu que toutes ces classes soient liées par un lien d'héritage.

Concrètement cela signifie que l'on va pouvoir utilisée une variable de type `Polygone` (i.e. la super-classe) pour stocker et utiliser indifféremment des instances des classes `Polygone`, `Rectangle` ou `Triangle`.

```
Polygone poly1, poly2, poly3;
double surfaceTotale;
poly1 = new Polygone();
poly2 = new Rectangle();
poly3 = new Triangle();

surfaceTotale  = poly1.getSurface() + poly2.getSurface()
                        + poly3.getSurface();
```

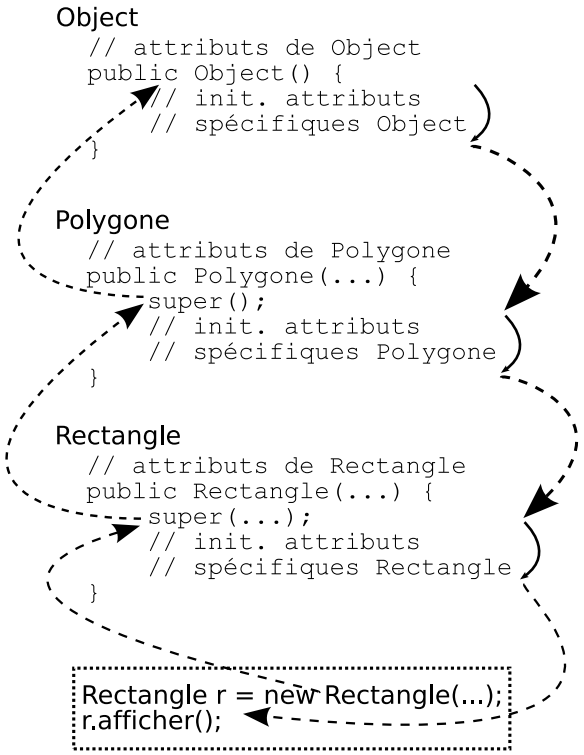

FIGURE 6.3 – Chaînage des appels de constructeurs

6.2.1 Typage statique/ liaison dynamique

Quelle version de la méthode est exécutée ? `poly2.getSurface()` exécute la version de `Rectangle` car `poly2` est une instance de la classe `Rectangle` : c'est la *liaison dynamique* car la méthode a exécutée est choisie au moment de l'exécution.

Quelles méthodes peuvent être appelées ? `poly2.getLargeur()` est illégal, seules les méthodes définies dans la classe correspondant au type de la variable peuvent être appelées : c'est le *typage statique*. Avec le polymorphisme on n'est jamais sûr du type de la valeur contenue dans la variable : on sait seulement que c'est un sous-type (au sens large) du type de la variable.

Dans d'autres langages comme Smalltalk on utilise le typage dynamique : aucune restriction sur la méthode appelable. Le typage dynamique est risqué car il ouvre la voie à un grand nombre d'erreurs possibles que le typage statique évite.

6.2.2 Utilisation du polymorphisme dans la conception et l'implémentation de logiciels

La possibilité offerte par le polymorphisme de traiter des instances de classes différentes comme des instances d'une même classe (super-classe commune aux différentes classes), a des répercutions importantes sur la conception de logiciels. En effet, à chaque fois qu'une partie d'une application n'utilise que les caractéristiques et le contrat défini dans la super-classe, cette partie de l'application peut être écrite en faisant uniquement référence à la super-classe. Le polymorphisme nous assure alors que cette partie du logiciel pourra traiter de manière adaptée toutes les instances de toutes les sous-classes de la super-classe (y compris les sous-classes qui pourraient être définies ultérieurement), et le mécanisme de liaison dynamique nous assure que ce sont les implémentations les plus adaptées qui seront effectivement exécutées.

Exemple 6.2.1 L'utilisation de l'héritage pour la définition des classes `Polygone`, `Rectangle` et `Triangle` nous permet d'utiliser le polymorphisme pour définir une classe `Dessin` dont les instances représentent des assemblages de polygones :

```
public class Dessin {
        private List<Polygone> listePoly;
        ...
        public void ajouterPolygone(Polygone p) {
```

```
                listePoly.add(p);
        }

        public void afficher() {
                for (Polynome p : listePoly) {
                        p.afficher();
                }
        }

        public void symetrie() {
                for (int i = 0; i < listePoly.size(); i++) {
                        Polynome p = listePoly.get(i);
                        listePoly.set(i, p.symetrie());
                }
        }
}
```

La classe `Dessin` ainsi définie en ne faisant référence qu'à la classe `Polygone` est capable de traiter correctement les instances de toutes les classes héritant de la classe `Polygone` :

```
        Dessin monDessin = new Dessin(...);
        monDessin.ajouterPolygone(new Polygone(...));
        monDessin.ajouterPolygone(new Rectangle(...));
        monDessin.ajouterPolygone(new Rectangle(...));
        monDessin.ajouterPolygone(new Triangle(...));
        monDessin.symetrie();
        monDessin.afficher();
```

... y compris les instances de classes définies ultérieurement, comme, par exemple, la classe `HexagoneRegulier` :

```
public class HexagoneRegulier extends Polygone {
        ...
        public void afficher() {
                // redéfinition de la méthode afficher
        }
}
```

... dont les instances pourront être utilisées pour créer un nouveau dessin sans qu'il soit nécessaire de modifier la classe `Dessin` :

```
        Dessin monDessin = new Dessin(...);
        monDessin.ajouterPolygone(new Rectangle(...));
        monDessin.ajouterPolygone(new HexagoneRegulier(...));
        monDessin.ajouterPolygone(new Triangle(...));
        monDessin.symetrie();
        monDessin.afficher();        // l'affichage de l'héxagone faisant partie du dessin
                                     // se fait en exécutant la méthode afficher()
                                     // redéfinie dans la classe HexagoneRegulier
```

6.3 Caractéristiques et classes abstraites

Les classes abstraites sont des classes dont certaines méthodes peuvent ne pas avoir d'implémentation. En conséquence, ces classes ne peuvent pas être instanciées (i.e. il est impossible de créer des instances d'une classe abstraite).

Les *interfaces JAVA* (à ne pas confondre avec la notion d'interface d'une classe que nous avons souvent utilisée précédemment) correspond à une notion spécifique au langage JAVA. Les *interfaces JAVA* peuvent être considérées comme des cas particuliers de classes abstraites ne possédant pas d'attributs *et* dont *aucune* méthode ne possède d'implémentation.

Nous allons voir dans les sections suivantes le rôle et l'utilisation de ces nouveaux concepts.

6.3.1 Rôle des classes abstraites

L'objectif essentiel recherché dans la définition et la mise en œuvre du concept de classe abstraite au sens large (i.e. aussi bien les classes abstraites que les interfaces JAVA) est de permettre une utilisation plus large du polymorphisme.

En effet, sans les classes retardées, la mise en œuvre du polymorphisme se trouve limitée par le fait que les super-classes utilisées doivent posséder une implémentation complète et fonctionnelle de chacune de leurs méthodes. Nous allons voir sur un exemple en quoi cela constitue une limitation.

Exemple 6.3.1 Toujours dans le cadre du développement d'un logiciel de dessin il parait naturel que les dessins puissent contenir, outre des polygones, d'autres formes géométriques simples comme, par exemple, des ellipses ou des cercles. Pour ces nouvelles formes géométriques, l'implémentation des méthodes définies dans la classe `Polygone` (i.e. `getPerimetre()`, `getSurface()`, `symetrie()`, `afficher()`) ne pose à priori aucun problème et il semblerait donc envisageable que la classe `Dessin` évoquée dans l'exemple 6.2.1 puisse être adaptée afin de pouvoir traiter aussi bien des polygones que des cercles ou des ellipses. Il suffirait pour cela de définir une super-classe commune aux classes `Polygone` et `Ellipse` [1], que nous pourrions appeler, par exemple, `FigureGeometrique`. Cependant, une telle définition s'avère impossible du fait de la difficulté à définir une implémentation complète de la classe `FigureGeometrique`.

Si nous examinons plus attentivement l'exemple ci-dessus, nous pouvons nous rendre compte que :
— nous avons besoin de définir une super-classe `FigureGeometrique` définissant un contrat commun à un ensemble de classes (`Polygone`, `Rectangle`, `Ellipse`, `Cercle`, ...) pour lesquelles nous sommes en mesure de définir une implémentation satisfaisant totalement ce contrat ;
— la création d'instances de la super-classe `FigureGeometrique` n'aurait aucun intérêt : les seules figures géométriques que nous aurons besoin de manipuler effectivement seront soit des polygones, soit des rectangles, soit des ellipses, ... mais jamais des "figures géométriques" quelconques ;

Exemple 6.3.2 La hiérarchie des classes conteneurs de la librairie standard JAVA (cf. package `java.util`) illustre l'intérêt des classes retardées dans un cadre différent : celui où l'on souhaite proposer plusieurs implémentations alternatives d'un même type, toutes ces implémentations satisfaisant le même contrat. Si on considère par exemple une structure de données de liste, il existe deux catégories d'implémentation possédant chacune ses avantages propres : les implémentations par liste chaînée (cf. classe `java.util.LinkedList`) et les implémentations par tableau (cf. classe `java.util.ArrayList`). Il est donc souhaitable que ces deux implémentations soient mises à disposition du programmeur, cependant, dans d'autres circonstances, le programmeur doit pouvoir traiter indifféremment des instances de ces différentes classes. C'est le cas, en particulier, lorsqu'une méthode se contente d'utiliser les résultats fournis par d'autres classes et qu'il n'a donc aucune maîtrise sur les choix d'implémentation fait par d'autres. Dans ce cas, il y a un intérêt réel à utiliser une classe spécifiant le contrat commun à ces différentes classes, sans prendre partie pour une implémentation particulière : dans ce cas et dans le contexte des classes de la librairie standard JAVA implémentant une structure de données de liste le programmeur utilisera alors la classe `java.util.List`, une "interface JAVA" spécifiant le contrat commun implémenté par les classes `java.util.LinkedList` et `java.util.ArrayList`.

6.3.2 Héritage multiple et interfaces JAVA

L'héritage multiple est le mécanisme selon lequel une classe peut être sous-classe de *plusieurs* super-classes (ces super-classes n'ayant elles-mêmes aucun lien d'héritage entre elles).

A titre d'exemple, si nous définissons une classe `Losange` [2] héritant de `Polygone`, il semblerait pertinent de définir la classe `Carre` comme sous-classe de `Rectangle` et de `Losange`.

Certains langages orientés objet comme C++ ou Eiffel permettent de telles définitions. Cependant, l'utilisation de l'héritage multiple peut donner lieu à des conflits d'héritage entre plusieurs implémentations concurrentes d'une même méthode : en supposant que les classes `Rectangle` et `Losange` possèdent chacune leur propre implémentation d'une méthode (par exemple, `getPerimetre()`), quelle implémentation sera utilisée sur une instance de la classe `Carre` ?

La résolution des conflits d'héritage étant un problème assez complexe, les concepteurs de certains langages ont fait le choix de ne pas permettre l'héritage multiple, ou bien de n'en mettre en œuvre qu'une forme très limitée. En effet, les conflits ne pouvant survenir que lorsque plusieurs super-classes possèdent des implémentations concurrentes, un moyen radical d'éviter tout conflit est d'interdire que plus d'une des super-classes ne possède une implémentation. C'est le choix qui a été fait pour le langage JAVA en n'autorisant l'héritage multiple que dans des cas où tout risque de conflit d'héritage est exclu.

Dans ce but, le langage JAVA définie les *interfaces JAVA* qui ne sont en fait que des cas particuliers de classes abstraites. Leur particularité est d'interdire complètement toute implémentation : comme pour toute classe, une *interface JAVA* spécifie un contrat, mais à la différence des classes abstraites elles ne peuvent contenir que des entêtes de méthodes (i.e. aucun attribut et aucune implémentation de méthode). Cependant cette dénomination particulière

1. la classe `Cercle` étant supposée définie comme sous-classe de la classe `Ellipse`
2. Un losange est un parallélogramme à quatre côtés égaux.

autorise à les utiliser dans des conditions spécifiques : alors qu'une classe ne peut hériter (`extends`) que d'au plus une classe (concrète ou abstraite) elle peut hériter (`implements`) d'un nombre illimité d'interfaces.

6.3.3 Syntaxe JAVA

Pour les classes abstraites, la syntaxe à utiliser pour la définition est :

```
public abstract class UneClasseAbstraite {
        // Définition possible d'attributs

        /**
         * La définition d'un constructeur est possible,
         * elle est indispensable lorsque des attributs
         * sont définis dans cette classe et nécessitent une
         * initialisation autre qu'une initialisation par défaut.
         *
         * Les constructeurs n'étant pas hérités, la notion
         * de constructeur "abstract" n'a pas de sens.
         */
        public UneClasseAbstraite(...) {
                // code initialisant les attributs
                // définis dans cette classe
        }

        public int methodeImplementee(...) {
                // certaines méthodes peuvent posséder
                // une implémentation
        }

        /**
         * Les méthodes non implémentées doivent impérativement
         * être déclarées "abstract".
         */
        public abstract String methodeAbstraite(...);
}
```

Pour la définition d'une interface la syntaxe est :

```
// Les mots clé "class" et "abstract" n'apparaîssent pas
// dans l'entête de la définition d'une interface.
public interface UneInterface {
        // Définition d'attributs interdite

        // Constructeurs interdits (inutile puisque
        // la définition d'attributs est interdite)

        /**
         * Inutile de préciser "abstract" puisque,
         * par définition d'une interface JAVA, TOUTES
         * les méthodes sont abstraites.
         */
        public Object[] uneMethode(...);

        public void autreMethode(...);
}
```

Voyons maintenant comment utiliser une classe abstraite ou une interface dans la définition d'une autre classe :

```
public class UneClasseConcrete extends UneClasseAbstraite
                                       implements UneInterface, AutreInterface {
        public String methodeAbstraite(...) {
                // Toutes les méthodes abstraites définies dans la
```

```
                // super-classe abstraite doivent obligatoirement
                // être implémentées.
        }

        public Object[] uneMethode(...) {
                // De la même manière que pour les méthodes abstraites
                // définies dans la super-classe abstraite, toutes les
                // méthodes définies dans les interfaces (ici UneInterface
                // et AutreInterface doivent être implémentées.
        }

        public void autreMethode(...) {
                // Même remarque que pour la méthode précédente.
        }
}
```

... mais une classe abstraite ou une interface peut aussi avoir des sous-classes abstraites :

```
public abstract class AutreClasseAbstraite extends UneClasseAbstraite
                                                implements UneInterface {
        // Héritage des méthodes non abstraites de la classe UneClasseAbstraite
        // avec leur implémentation. Si besoin, ces méthodes
        // peuvent être redéfinies.

        // Les méthodes abstraites de la classe UneClasseAbstraite
        // ainsi que les méthodes définies dans UneInterface,
        // sont des méthodes abstraites de cette classe

        // De nouvelles méthodes abstraites ou non peuvent être
        // ajoutées.

}
```

... de même qu'une classe concrète peut avoir des sous-classes abstraites :

```
public abstract class EncoreUneClasseAbstraite extends UneClasseConcrete
                                                implements UneInterface {
        // Héritage des méthodes de la classe UneClasseConcrete
        // avec leur implémentation. Si besoin, ces méthodes
        // peuvent être redéfinies.

        // Les méthodes définies dans UneInterface, non
        // implémentées dans UneClasseConcrete ni dans cette classe
        // sont des méthodes abstraites de cette classe

        // De nouvelles méthodes abstraites ou non peuvent être
        // ajoutées.
}
```

Par contre, une interface ne peut hériter ni d'une classe concrète, ni d'une classe abstraite, elle ne peut hériter que d'une (ou plusieurs) interface :

```
public interface EncoreUneInterface implements UneInterface {
        // Les méthodes définies dans UneInterface peuvent être
        // redéfinies: dans le cas d'une interface, redéfinition
        // signifie uniquement un raffinement du contrat (il est
        // hors de question de donner une implémentation à une méthode).

        // De nouvelles méthodes (abstraites) peuvent être ajoutées.
}
```

6.3.4 Assertions et classes abstraites ou interfaces

Le fait qu'une classe ou une méthode ne possède pas d'implémentation – comme c'est le cas avec les classes abstraites ou les *interfaces JAVA* – n'empêche pas de lui attribuer des assertions.

Toute classe ou méthode sans implémentation doit posséder des assertions (i.e. invariant, préconditions, postconditions) au même titre qu'une classe ou méthode possédant une implémentation.

6.4 Héritage du contrat

Problème : Quand une classe `B` hérite d'une classe `A`, quelles assertions sont applicables aux instances de la classe `B` ?

Réponse générale : la classe `B` hérite aussi du contrat de `A`, c'est-à-dire des assertions de `A`.

Conséquences :
- lorsque l'on implémente la classe `B` on doit tenir compte du contrat de `A`, toute instance de `B` doit *aussi* remplir le contrat définit pour la classe `A`.
- on ne peut pas écrire n'importe quelle assertion pour `B` : les assertions de `B` doivent être cohérentes et non contradictoires avec celles de `A`.

Avantage : l'héritage des assertions est un bon cadre pour nous permettre de bien comprendre et utiliser l'héritage. Si nous rencontrons un conflit ou une incohérence dans l'héritage des assertions, c'est que nous avons mal utilisé l'héritage (ou les assertions).

6.4.1 Héritage de l'invariant

Règle :
- si une classe `B` hérite d'une classe `A`, alors l'invariant de classe de `A` s'applique à la classe `B`.
- toute instance de `B` respecte l'invariant de `A` *et* l'invariant de `B`.

Conséquence : l'invariant de `B` ne doit pas être contradictoire avec l'invariant de `A`.

Exemple 6.4.1 Héritage d'invariants :

```
public class Polygone {
        /*@
          @ ...
          @ invariant getNbCotes() > 2;
          @ invariant getNbSommets() == getNbCotes();
          @*/
        ...
}
```

```
public class Rectangle extends Polygone {
         /*@ also
           @ ...
           @ invariant getNbCotes() == 4;
           @ invariant getLongueur() > 0;
           @ invariant getLargueur() > 0;
           @*/
        ...
}
```

```
public class Carre extends Rectangle {
         /*@ also
           @ ...
           @ invariant getLongueur() == getLargueur();
           @*/
        ...
}
```

Les propriétés vraies de toute instance de la classe `Carre` sont alors :
- héritage de la classe `Polygone` :
  - `getNbCotes() > 2`
  - `getNbSommets() == getNbCotes()`
- héritage de la classe `Rectangle` :
  - `getNbCotes() == 4`
  - `getLongueur() > 0`
  - `getLargueur() > 0`
- invariant de la classe `Carre` :
  - `getLongueur() == getLargueur()`

En simplifiant les assertions redondantes (i.e. `getNbCotes() > 2` et `getNbCotes() == 4` se simplifie en `getNbCotes() == 4`), nous obtenons :
- `getNbCotes() == 4`
- `getNbSommets() == 4`
- `getLongueur() > 0`
- `getLargueur() > 0`
- `getLongueur() == getLargueur()`

6.4.2 Préconditions et postconditions

Note : un problème peut se poser uniquement dans le cas de redéfinition de méthode. En effet, c'est uniquement dans ce cas qu'il peut y avoir conflit entre des préconditions ou des post-conditions.

Principe général : les conditions à respecter découlent directement des principes de fonctionnement du polymorphisme et de la liaison dynamique : toute méthode définie (avec visibilité `public`) dans une classe, est applicable / utilisable dans chacune de ses sous-classes, sous les conditions spécifiées (y compris par les assertions) dans la super-classe.

Règle : Soit `B` une sous-classe de la classe `A` et `m` une méthode de `A` redéfinie dans `B`, telle que `m` possède une pré-condition P_A et une post-condition Q_A dans `A` et une pré-condition P_B et une post-condition Q_B dans `B` (cf. exemple 6.4.2). Alors, les assertions applicables lors de l'exécution de la redéfinition de la méthode `m` (i.e. pour les instances de B et de ses sous-classes) sont :

Pré-condition : $P_A || P_B$

Post-condition : $P_A ==> Q_A$

Post-condition : $P_B ==> Q_B$

Exemple 6.4.2 Héritage des pré-conditions et des post-conditions.

```
public class A {
        ...
        /*@
          @ requires PA;
          @ ensures QA;
          @*/
        public Truc m(...) {
        }
        ...
}

public class B extends A {
        ...
        /*@ also
          @ requires PB;
          @ ensures QB;
          @*/
        public Truc m(...) {
        }
```

```
        ...
}
```

Toute méthode de `A` doit pouvoir être appliquée à toute instance de `B` sous les conditions (i.e. les préconditions) spécifiées dans `A`. Les préconditions spécifiées dans `B` peuvent éventuellement être moins contraignantes que celles de `A`.

A chaque fois qu'une méthode de `A` est appliquée à une instance de `B`, les postconditions spécifiées dans `A` doivent être satisfaites après exécutions de la méthode. Les postconditions de `B` peuvent éventuellement satisfaire des conditions supplémentaires, mais qui n'invalident pas les postconditions spécifiées dans `A`.

Exemple 6.4.3 Nous considérons une méthode d'en-tête :

```
double calculer(double precision)
```

Le rôle de cette méthode est d'effectuer un calcul numérique, en prenant en paramètre la précision souhaitée par l'utilisateur. Cette méthode est définie dans une classe `A` et redéfinie dans `B`, une sous-classe de `A`. La précision qu'il est possible d'atteindre étant limitée par l'implémentation de la méthode, nous utilisons les préconditions et les postconditions pour spécifier quelle est la précisions maximale que permet d'obtenir une implémentation donnée. Nous supposerons ici que l'implémentation de `B`, réalisée postérieurement à celle de `A` permet de travailler avec une plus grande précision.

Dans la classe `A` où l'implémentation de `calculer` est basique nous spécifions :

précondition $precision \geq 10^{-6}$

postcondition $precisionResultat \leq precision$

Dans la classe `B` où l'implémentation est meilleure, nous spécifions :

précondition $precision \geq 10^{-20}$

postcondition $precisionResultat \leq precision/2$

On peut maintenant vérifier que les assertions spécifiées dans la classe B satisfont les règles données précédemment :

règle des préconditions la précondition de B n'est pas plus restrictive que celle de A : si la précondition de A est satisfaite (i.e. $precision \geq 10^{-6}$), alors la précondition donnée dans B est nécessairement satisfaite (i.e. $precision \geq 10^{-6} \Rightarrow precision \geq 10^{-20}$)

règle des postconditions la postcondition de B ne remet pas en cause la postcondition de A : lorsque la méthode implémentée dans B est exécutée, le résultat satisfait la condition $precisionResultat \leq precision/2$, ce qui n'est pas contradictoire avec la postcondition spécifiée dans A (i.e. $precisionResultat \leq precision$).

6.5 Implémenter clone en prenant en compte l'héritage

La prise en compte de l'héritage a plusieurs conséquences directes sur la façon dont la méthode `clone()` doit être implémentée :

1. Lorsque la méthode clone est implémentée dans une classe, nous devons porter une attention particulière au fait que des sous-classes de cette classe pourront être ultérieurement définies. L'implémentation de `clone()` doit donc être conçue pour pouvoir, selon les circonstances, être héritée sans redéfinition ou bien pour pouvoir être facilement redéfinie.
2. Toute définition d'une sous-classe par héritage s'accompagne obligatoirement de l'héritage du contrat de sa super-classe (et éventuellement du contrat de toutes les interfaces héritées).
3. La connaissance du mécanisme de l'héritage permet d'expliquer de manière plus claire, de quelle manière utiliser l'implémentation de `clone()` fournie par la classe `Object`.

6.5.1 Quelques principes de base

Avant de donner des explications détaillées, voyons tout d'abord les points essentiels à retenir :

1. Toute implémentation de `clone` est une redéfinition directe ou indirecte de la méthode `clone` de la classe `Object`, et doit donc impérativement respecter le contrat de clone défini dans cette classe, c'est à dire :
   — `x.equals(x.clone())`
   — `x != x.clone()`
   — `x.getClass() == x.clone().getClass()` : `x` et son clone sont des instances de la même classe
2. Toute classe implémentant `clone()` doit être définie en mentionnant `implements Cloneable` dans son entête (sauf si cette classe hérite d'une classe implémentant déjà `clone()` et mentionnant `implements Cloneable` dans son en-tête).

3. L'implémentation de la méthode `clone()` doit toujours contenir un appel `super.clone()`, s'il s'agit d'un appel direct à la méthode `clone` de la classe `Object`, cet appel doit être placé dans un bloc `try/catch`.
4. Implémenter `clone()` n'a de sens que si la classe concernée possède ou hérite une méthode `equals` redéfinie distincte de l'égalité de référence (sinon le contrat est impossible à satisfaire).

6.5.2 Définition de `clone` dans la classe `Object`

L'essentiel de ce qui est dit dans cette note est expliqué de manière assez précise (bien que en anglais) dans la documentation de la méthode `clone` de la classe `Object`.

Comme nous l'avons vu en cours, le contrat de la méthode `equals`, tel qu'il est définit dans la classe `Object`, rend impossible une implémentation correcte (i.e. satisfaisant le contrat) de la méthode `clone` dans la classe `Object`. En effet, si `x.equals(y)` est équivalent à `x == y`, il est impossible d'avoir en même temps, `x.equals(x.clone())` et `x != x.clone()`.

Cependant, les concepteurs de la librairie Java ont considéré qu'il était important de donner une implémentation de la méthode `clone` pour simplifier le travail des programmeurs souhaitant implémenter la méthode `clone` dans une nouvelle classe. Devant cette contradiction, un dispositif spécifique a été conçu, afin que la méthode `clone` ne puisse pas être utilisée sans le décider explicitement, mais que l'implémentation de cette méthode puisse être utilisée si le concepteur d'une classe juge nécessaire que les instances de cette classe puissent être clonées. Ce dispositif peut être décrit de la manière suivante :

— Dans la classe `Object`, la méthode `clone` est déclarée avec la visibilité `protected`. Par conséquent, sauf précaution particulière, la méthode `clone` ne peut être appelée sur une instance, que dans le code source de la classe de cette instance. En dehors de ce cas la compilation provoquera une erreur.

— L'implémentation de `clone` dans la classe `Object` commence par tester si l'instance à cloner *implements* l'interface `Cloneable`, si ce n'est pas le cas une exception `CloneNotSupportedException` est lancée et le clonage n'est pas effectué. Par conséquent, une classe contenant un appel à la méthode `clone` de la classe `Object` mais ne déclarant pas qu'elle implémente l'interface `Cloneable`, lancera systématiquement une exception `CloneNotSupportedException` à chaque appel de clone.

— Dans la classe `Object`, la méthode `clone` est déclarée avec une clause `throws CloneNotSupportedException`. Cela impose à toute classe faisant directement appel à cette méthode de placer l'appel dans un bloc `try/catch` ou bien de déclarer la méthode faisant appel au `clone` de `Object` avec une clause : `throws CloneNotSupportedException`.

Ce dispositif complexe assure donc que la méthode `clone` définie dans la classe `Object` ne peut pas être utilisée "par inadvertance".

Implémentation de clone dans la classe Object

L'implémentation de clone dans la classe Object est donc de la forme :

```
protected Object clone() throws CloneNotSupportedException {
        if (!(this instanceof Cloneable)) {
                throw new CloneNotSupportedException();
        }

        Object leClone = // création d'une instance de la même
                         // classe que this (pas forcément Object)

        // Copie par simple affectation (pas de clonage) de tous
        // les attributs de this vers les attributs correspondant
        // de leClone.

        return leClone;

}
```

6.5.3 Schéma d'implémentation de la méthode `clone`

Cas d'une sous-classe directe de la classe `Object`

L'implémentation sera de la forme :

```
public class ClasseClonable implements Cloneable {
        // Définition des attributs
```

```
        // Définition du ou des constructeurs

        // Définition d'autres méthodes

        // Redéfinition IMPÉRATIVE de equals, hashCode et toString

        @Override
        public Object clone() {
                Object leClone = null;
                try {
                        leClone = super.clone();
                } catch (CloneNotSupportedException e) {
                        throw new InternalError("Clone aurait du fonctionner");
                }
                // Clonage des attributs pour lesquels une simple
                // copie n'est pas satisfaisante

                return leClone;
        }
}
```

Cas d'une sous-classe d'une classe `Cloneable`

```
public class SousClasseClonable extends ClasseClonable {
        // Définition éventuelle d'attributs spécifique à cette classe

        // Définition du ou des constructeurs

        // Définition d'autres méthodes

        // Redéfinition facultative de equals, hashCode et toString

        @Override
        public Object clone() {
                Object leClone = null;

                leClone = super.clone();

                // Clonage des attributs spécifiques à cette classe
                // pour lesquels une simple copie n'est pas
                // satisfaisante. Si la copie par simple affectation
                // est satisfaisante pour tous les attributs spécifiques
                // à cette classe, alors cette redéfinition de clone
                // est inutile.

                return leClone;
        }
}
```

6.5.4 Clonage des tableaux

Comme indiqué dans la spécification du langage Java, tout tableau est instance d'une classe héritant de la classe `Object` et implémentant l'interface `Cloneable`. La spécification précise en même temps que l'implémentation de `clone` pour un tableau dont les éléments sont de type `T` est équivalente à :

```
@Override
public T[] clone() {
        try {
                return (T[]) super.clone(); // unchecked warning
        } catch (CloneNotSupportedException e) {
```

```
                throw new InternalError(e.getMessage());
        }
}
```

La spécification précise en même temps, que la méthode `clone` est la seule méthode héritée de la classe `Object` qui soit redéfinie. Par conséquent, tout appel de la méthode `equals` sur un tableau utilise l'implémentation de `equals` définie dans la classe `Object` (i.e. égalité de référence).

Les tableaux ne respectent donc pas le contrat de la classe `Object` défini pour la méthode `clone`.

Exemple :

```
String[] tab1, tab2;

tab1 = new String[2];

tab1[0] = "aa";

tab1[1] = "bb";

tab2 = tab1.clone();    // Les tableaux sont clonables

// tab1 != tab2 est true ; tab1.equals(tab2) est false
// ==> le contrat de clone définit dans la classe Object
// n'est pas respecté.

boolean pareil = (tab1.length == tab2.length);

// pareil est true ==> le clone est un tableau de même dimension
// que l'original

for (int i = 0; i < tab1.length; i++) {

        pareil = pareil && (tab1[i] == tab2[i]);
}
// pareil est true ==> le clone contient les mêmes valeurs que
// l'original (les éléments sont copiés par simple affectation, ils
// ne sont pas clonés).
```

6.6 Différentes utilisations de l'héritage

6.6.1 Dans quels cas utiliser l'héritage ?

Spécialisation La sous-classe est un cas particulier de sa super-classe ; c'est la forme d'héritage qui nous a servie d'illustration pour introduire l'héritage dans les sections précédentes. Dans cette utilisation de l'héritage c'est la relation *est un* qui relie une classe à sa super-classe. Exemple : un `Rectangle` *est un* `Polygone`.

Spécification La super-classe définie un comportement, qui est implémentée dans la sous-classe mais pas dans la super-classe. Cette forme d'héritage est, en général, combinée avec une autre forme d'héritage. Exemple : la classe `FigureFermee` définie un service pour calculer le périmètre d'une surface, mais n'implémente pas ce service ; seules les sous-classe `Polygone` ou `Ellipse` (et, éventuellement, leurs sous-classes) définissent une implémentation de ce service. Dans cet exemple l'héritage pour spécification est combiné avec un héritage pour spécialisation.

Construction La sous-classe se contente d'hériter le comportement de sa super-classe sans en être pour autant un sous-type.

Extension La sous-classe se contente d'ajouter de nouvelles fonctionnalités sans modifier les comportements hérités de la super-classe.

Combinaison La sous-classe hérite des fonctionnalités de plusieurs classes. Cela correspond aux cas d'utilisation de l'héritage multiple (par utilisation des interfaces en JAVA).

6.6.2 Dans quels cas *ne pas* utiliser l'héritage ?

Dans certains cas, l'utilisation de l'héritage doit être évité :

Limitation La sous-classe restreint l'usage de fonctionnalités définies dans la super-classe. Ce type d'héritage enfreint les règles sur l'héritage du contrat telle que définies dans la section précédente et doit donc être proscrit. Exemple : la classe `Pile` ou la classe `Ensemble` ne doivent pas être définies comme sous-classes de la classe `ArrayList`. Dans ces deux exemples, il n'est cependant pas interdit d'utiliser la classe `ArrayList` comme structure de données pour stocker les éléments, mais dans ce cas, on définit un champ de type `ArrayList` et on implémente les méthodes d'accès aux éléments en utilisant les méthodes d'accès définies dans la classe `ArrayList`. La différence essentielle dans ce cas est que l'on peut contrôler totalement comment et dans quelles conditions l'accès aux éléments pourra se faire (par exemple, pour une pile, seul l'élément au sommet de pile est consultable...).

Composition Lorsque la relation qui relie deux classes est une relation de composition, comme entre la classe `Voiture` et la classe `Roue`, l'héritage ne doit pas être utilisé : une voiture n'est pas une roue, une roue n'est pas une voiture et les comportements de ces deux types d'objets sont totalement différents.

6.7 Les classes *conteneur* de la bibliothèque JAVA

La bibliothèque standard de JAVA, contient un nombre important de classes permettant de gérer des groupes d'objets. Ces classes sont appelées des *conteneurs*. Ces classes reprennent des structures de données classiques en algorithmique, comme les listes, les tableaux ou les tables de haschage. Cette hiérarchie est composée d'interfaces, de classes abstraites et de classes concrètes (i.e. classes non abstraites) :

— les interfaces décrivent un comportement général attendu (ensemble de méthodes que chaque classe implémentant cette interface doit implémenter). Elles servent de guide général lorsque de nouvelles classes conteneur doivent être implémentées. Elles sont aussi utilisées lorsque certaines classes on besoin d'utiliser certaines sortes de conteneur (comme paramètre ou valeur de retour d'une méthode), sans pour autant vouloir imposer une implémentation particulière ;
— les classes abstraites définissent une implémentation partielle évitant d'avoir à tout réécrire lors de l'implémentation d'une nouvelle classe ;
— les classes concrètes fournissent des classes prêtes à l'emploi pour les utilisations les plus courantes.

Les classes conteneurs sont réparties en 2 hiérarchies distinctes correspondant aux deux interfaces : `Collection` et `Map`. L'interface `Iterator` décrit des objets qui ne sont pas des conteneurs, mais définissent des méthodes d'accès aux éléments des classes conteneur et apparaissent fréquemment dans la définition des classes conteneur.

6.7.1 L'interface Collection

C'est la classe racine pour les structures de données les plus usuelles : listes, ensembles, tableaux. Les principales méthodes requises par cette interface sont :

— `boolean` **`contains`**`(Object o)` Renvoie true si cette collection contient l'élément spécifié.
— `Iterator<E>` **`iterator`**`()` Renvoie un itérateur sur les éléments de cette collection.
— `Object[]` **`toArray`**`()` Renvoie un tableau contenant tous les éléments de cette collection.
— `boolean` **`isEmpty`**`()` Renvoie true si cette collection ne contient aucun élément.
— `int` **`size`**`()` Renvoie le nombre d'éléments de cette collection.

Pour une collection modifiable, des méthodes d'ajout/suppressions d'éléments doivent être implémentées, mais l'implémentation de ces méthodes est optionnelle (en violation de la règle "pas d'héritage pour restriction" énoncée à la section précédente). La classe abstraite `AbstractCollection` fournit une implémentation basique de cette interface dans laquelle seules deux méthodes sont abstraites : `iterator` et `size`.

L'interface possède deux sous-interfaces : `Set` et `List`.

L'interface Set

Cette interface modélise la notion mathématique d'*ensemble*. Elle ne spécifie aucune nouvelle méthode par rapport à l'interface `Collection`. La différence essentielle entre ces deux interfaces est dans le contrat (sous forme de commentaires) : dans les conteneurs implémentant l'interface `Set`, il ne doit pas y avoir d'éléments dupliqués. Comme pour `Collection`, une classe abstraite `AbstractSet` sous-classe de `AbstractCollection` est fournie qui se contente de fournir une nouvelle implémentation de certaines méthodes, sans fournir d'implémentation des méthodes abstraites de `AbstractCollection`.

Autres caractéristiques attendues des classes implémentant cette interface :

— les éléments ne sont pas ordonnés ;
— les tests d'appartenance sont des opérations rapides (en O(1))
— les opérations d'ajout/suppression sont des opérations rapides (en O(1))
— l'énumération des éléments peut-être une opération lente.

L'interface List

Les classes implémentant l'interface `List` sont des collections ordonnées, elles fournissent un contrôle précis de la position où chaque élément est inséré et permettent un accès aux éléments par l'intermédiaire de leur index (position dans la liste).

A la différence des ensembles (interface `Set`), les listes autorisent la présence d'éléments dupliqués.

L'accès aux éléments par leur index et la recherche d'un élément donné peuvent nécessiter un temps proportionnel à la valeur de l'index (cas d'une implémentation où l'accès à un élément par son index nécessite le parcours de tous les éléments de la liste d'index inférieur) ou au nombre d'éléments. En conséquence, lorsqu'il est nécessaire de parcourir un par un les éléments de la liste il est préférable d'utiliser un itérateur.

L'interface `List` se différencie aussi d'une `Collection` en fournissant un itérateur (`ListIterator`) permettant l'insertion / suppression d'éléments ainsi qu'une possibilité d'itération bidirectionnelle.

6.7.2 L'interface Map

Les classes implémentant l'interface `Map` appartiennent à une hiérarchie distincte des collections. Les éléments de ces groupes d'objets sont des paires d'objets (clé, valeur), pour lesquelles l'accès à la valeur se fait à en utilisant la clé à la manière des dictionnaires (clé = mot, valeur = définition).

Les implémentations courantes de ces classes utilisent la technique du hashage (hash coding), qui consiste à associer / calculer un entier à partir de la clé, cet entier – calculé à l'aide de la méthode `hashCode()` – étant alors utilisé comme index.

Les caractéristiques habituelles de ces classes sont :

— éléments non ordonnés, aucun contrôle sur l'emplacement auquel un élément est ajouté ;
— pas d'éléments dupliqués : deux éléments distincts ne peuvent avoir la même clé ;
— accès par la clé et ajout d'un élément sont des opérations efficaces (temps constant) ;
— itération peu efficace.

Les principales opérations sont :

— `boolean` **`containsKey`**`(Object key)` Renvoie true si la map contient une correspondance pour la clé spécifiée.
— `V` **`get`**`(Object key)` Renvoie la valeur correspondant à la clé spécifiée.
— `int` **`size`**`()` Renvoie le nombre de couples (clé, valeur) de cette map.
— `boolean` **`isEmpty`**`()` Renvoie true si la map ne contient aucune association (clé, valeur).
— `Set<K>` **`keySet`**`()` Renvoie un ensemble (`Set`) contenant toutes les clés de la map.
— `Collection<V>` **`values`**`()` Renvoie une collection contenant toutes les valeurs de la map.
— `V` **`put`**`(K key, V value)` Ajoute l'association (key, value) spécifiée (optionnel, pour les implémentations de classes modifiables).
— `V` **`remove`**`(Object key)` Supprime la correspondance dont la clé est spécifiée (optionnel, pour les implémentations de classes modifiables).

6.7.3 Les itérateurs

L'interface `Iterator` définie une méthode d'accès générale et indépendante des implémentations permettant d'effectuer des itérations sur les éléments d'une collection.

Tout itérateur définie les deux méthodes :

— `boolean` **`hasNext`**`()` Renvoie true si il reste des éléments non encore parcourus.
— `E` **`next`**`()` Renvoie l'élément suivant.
— à sa création un itérateur est positionné sur le premier élément. Chaque nouveau parcours nécessite la création d'une nouvelle instance d'itérateur.

Chapitre 7

Généricité

7.1 Introduction

La version 5.0 de l'environnement de développement JAVA introduit plusieurs extensions au langage de programmation JAVA. L'une d'entre elle est la généricité. La généricité est présente, sous des formes différentes, dans de nombreux autres langages de programmation, comme Eiffel ou C++, mais nous allons l'aborder ici à travers son implémentation dans le langage JAVA.

La généricité est un outil permettant de définir des types (i.e. des classes) de manière plus abstraite d'une manière différente de l'héritage. Les exemples les plus courant d'utilisation de la généricité dans la librairie standard JAVA sont les classes conteneurs que l'on trouve dans le paquetage `java.util` et qui sont, en général, des classes implémentant les interfaces `Collection` ou `Map`. Cependant, l'utilisation de la généricité ne se limite pas à ses classes.

L'exemple 7.1.1 ci-après illustre un usage typique de ces classes avant l'introduction de la généricité :

Exemple 7.1.1 Premier exemple sans généricité.

```
List myIntList = new LinkedList();                          // 1
myIntList.add(new Integer(0));                             // 2
Integer x = (Integer) myIntList.iterator().next();          // 3
```

Dans cet exemple, le "cast" de la ligne 3 est assez ennuyeux, car alors que, habituellement, le programmeur sait quelle sorte de donnée a été placée dans la liste, le "cast" n'en reste pas moins indispensable. En effet, la seule chose que le compilateur puisse garantir est qu'une instance de la classe `Object` sera retournée par l'itérateur, et, dans ce contexte, la seule possibilité pour assurer que l'affectation à une variable de type `Integer` soit sûre est de faire ce "cast".

Ce qui est encore plus gênant dans cette pratique est qu'elle introduit la possibilité d'une erreur à l'exécution (i.e. "run time error") en cas de méprise de la part du programmeur.

Pour résoudre ce problème, il suffirait que le programmeur puisse exprimer le fait que la liste est en fait restreinte à ne contenir qu'un type de données en particulier. C'est l'idée générale de la généricité. L'exemple ci-après reprend le même exemple 7.1.1, mais, ici, en utilisant la généricité :

Exemple 7.1.2 Premier exemple avec généricité.

```
List<Integer> myIntList = new LinkedList<Interger>();                  // 1'
myIntList.add(new Integer(0));                                        // 2'
Integer x = myIntList.iterator().next();                               // 3'
```

Notez tout d'abord la déclaration de la variable `myIntList` : elle spécifie qu'il ne s'agit pas d'une `List` arbitraire, mais d'une `List` d'`Integer`, ce qui s'écrit `List<Integer>`. Nous disons que `List` est une interface générique prenant un *type en paramètre*, dans ce cas `Integer`. De la même manière, nous spécifions aussi un type en paramètre lors de la création de l'objet liste.

La deuxième chose importante que nous pouvons remarquer dans cet exemple est que le "cast" a disparu (cf. ligne 3'). A première vu, on pourrait penser que nous n'avons fait que déplacer le code indésirable : en effet, au lieu d'un cast à la ligne 3, nous avons maintenant un type en paramètre à la ligne 1'. Il y a cependant une différence très importante : le compilateur peut maintenant vérifier la concordance des types *à la compilation*. Le fait que `myIntList` soit déclarée avec le type `List<Integer>` apporte une précision à propos de la variable `myIntList` qui est vrai partout et à tout moment de son utilisation et que le compilateur est capable de nous garantir. A l'opposé, le cast ne fait que nous dire quelque chose que le programmeur *pense* être vrai *à un point particulier du code*.

La principale conséquence, et plus particulièrement pour de grands programmes, est une amélioration de la lisibilité et de la robustesse du code.

7.2 Une première utilisation de la généricité

Voici, ci-après, un petit extrait des définitions des interfaces `List` et `Iterator` du paquetage `java.util` :

```
public interface List<E> {
        void add(E x);
        Iterator<E> iterator();
}

public interface Iterator<E> {
        E next();
        boolean hasNext();
}
```

Tout cela devrait vous paraître familier à l'exception de ce qui se trouve entre crochets (i.e. '<' et '>'). Il s'agit des déclarations de type comme paramètres formels des interfaces `List` et `Iterator`.

Les types utilisés comme paramètres (comme `E` dans le code ci-dessus) peuvent être utilisés dans une déclaration générique de la même manière que des types ordinaires.

Dans l'introduction, nous avons vu des invocations de la déclaration du type générique `List` comme `List<Integer>`. Dans ces invocations (habituellement appelées `types paramétrés`), toutes les occurrences du type utilisé comme paramètre formel (`E` dans ce cas) sont remplacées par le type utilisé comme paramètre réel (`Integer` dans ce cas). Vous pouvez donc imaginer que `List<Integer>` représente une version du type `List` dans laquelle `E` aurait été uniformément remplacé par `Integer` :

```
public interface IntegerList {
        void add(Integer x);
        Iterator<Integer> iterator();
}
```

Bien qu'utile, cette intuition peut en même temps être trompeuse.

Cette intuition est utile, car le type paramétré `List<Integer>` possède en effet des méthodes qui ressemblent précisément à cette expansion.

Cette intuition est cependant trompeuse car la déclaration d'un type générique n'est jamais effectivement expansée de cette manière. Il n'existe jamais plusieurs copies du code : ni sous forme de code source, ni sous forme de code compilé dans aucune sorte de mémoire. et il s'agit là d'une différence importante avec les "template" C++ qui jouent un rôle équivalent en C++.

Une déclaration de type générique est compilée une seule fois pour toute sous forme d'un unique fichier "`.class`", exactement de la même manière que n'importe quelle déclaration de type ou d'interface ordinaire.

En fait, les types utilisés comme paramètres sont analogues au paramètres ordinaires utilisés dans les méthodes ou les constructeurs. De la même manière qu'une méthode possède des paramètres formels qui décrivent sur quelle valeurs cette méthode peut effectuer des traitements, une déclaration générique possède des types utilisés comme paramètres formels. Quand une méthode est appelée, les paramètres formels sont remplacés par les paramètres réels et le corps de la méthode est exécuté. Quand une déclaration générique est utilisée, les types utilisés comme paramètres réels remplacent les types utilisés comme paramètres formels dans la déclaration du type générique.

Note sur les conventions de nomage. Pour les identificateurs de types utilisés comme paramètres formels, il est recommandé d'utiliser des noms courts (un seul caractère si possible) et autant que possible évocatifs. Il est préférable d'éviter les caractères minuscules afin de pouvoir distinguer facilement les types utilisés comme paramètres formels des identificateurs de types ordinaires. Ainsi, beaucoup de types conteneurs utilisent l'identificateur `E` – comme element – comme dans les exemples ci-dessus).

7.3 Généricité et héritage

Testons maintenant notre compréhension de la généricité. Le code suivant est il légal ?

```
List<String> ls = new ArrayList<String>();        // 1
List<Object> lo = ls;                             // 2
```

La ligne n° 1 est bien sûr légale. La difficulté se trouve à la ligne n° 2. Cela revient à répondre à la question : une `List` de `String` est elle une `List` d'`Object` ? A première vue, la plupart des gens répondrait : "Bien sûr !".

Voyons donc maintenant les quelques lignes de code suivantes :

```
        lo.add(new Object());        // 3
        String s = ls.get(0);        // 4 tentative d'affectation d'un Object à
                                     // une variable de type String !
```

Ici (en supposant que la ligne 2 soit valide), `ls` et `lo` sont des alias (i.e. `ls` et `lo` sont des variables qui contiennent toutes les deux une référence vers le même objet). En accédant à `ls` – une liste de `String` – par l'intermédiaire de l'alias `lo`, nous pouvons lui ajouter n'importe quel objet. En conséquence, `ls` ne contient plus désormais uniquement des instances de la classe `String` et si nous essayons maintenant d'en récupérer un élément nous risquons d'avoir une grosse surprise !

Bien entendu, le compilateur Java fera ce qu'il faut pour qu'une telle chose n'arrive pas et la ligne n° 2 provoquera une erreur à la compilation.

De manière générale, si `Foo` est un sous-type de `Bar`, et que `G` est un type générique, il est **faux** de dire que `G<Foo>` est un sous-type de `G<Bar>`. C'est probablement la chose la plus difficile que vous aurez à comprendre à propos de la généricité, car cela va contre notre intuition.

Le problème avec cette intuition est qu'elle suppose que les collections ne changent pas. Notre intuition considère ces objets comme ne pouvant être modifiées.

Par exemple, si le service des permis de conduire fournie une liste de conducteurs au bureau du recensement, cela semble raisonnable. Nous pensons dans ce cas qu'une `List<Conducteur>` est une `List<Personne>`, en supposant que `Conducteur` soit un sous-type de `Personne`. En fait, ce qui est transmis au bureau du recensement est une **copie** du registre des conducteurs. S'il en était autrement, le bureau du recensement pourrait ajouter à cette liste de nouvelles personnes qui ne sont pas des conducteurs et pourrait ainsi corrompre le registre du service des permis de conduire.

7.4 Types paramètres joker

Considérons le problème consistant à écrire une méthode affichant tous les éléments d'une collection. Le code ci-après montre comment nous aurions pu l'écrire avec une ancienne version du langage :

```
void printCollection(Collection c) {
        Iterator i = c.iterator();
        for (k = 0; k < c.size(); k++) {
                System.out.println(i.next());
        }
}
```

Le code suivant montre une tentative naïve d'utilisation de la généricité (et de la nouvelle syntaxe des boucles `for`) pour l'écriture de cette même méthode :

```
void printCollection(Collection<Object> c) {
        for (Object e : c) {
                System.out.println(e);
        }
}
```

Le problème avec cette nouvelle version est qu'elle est beaucoup moins utile que la précédente. En effet, alors que l'ancienne version de la méthode pouvait être appelée avec n'importe quelle collection en paramètre, la nouvelle version ne fonctionne que pour des `Collection<Object>` (i.e. cette méthode ne peut pas être appelée avec un paramètre de type `Collection<String>` ou `Collection<Integer>`), qui, comme nous venons de le montrer *n'est pas* un super-type de toutes les collections !

Mais alors, quel est donc le super-type de toutes les collections ? C'est le type qui s'écrit `Collection<?>` (à prononcer "collection d'inconnus"), c'est-à-dire, une collection dont le type des éléments est un type quelconque. Un tel type est appelé *type joker* (ou *wildcard type* en anglais), par analogie avec la carte joker des jeux de carte qui peut se substituer à n'importe quelle carte. Nous pouvons alors écrire :

```
void printCollection(Collection<?> c) {
        for (Object e : c) {
                System.out.println(e);
        }
}
```

et maintenant, nous pouvons appeler cette méthode avec n'importe quel type de collection. Vous remarquerez par ailleurs que, dans le code de cette méthode, il reste toujours possible de consulter les éléments de `c` et de les considérer

comme étant de type `Object`. En effet, quelque soit le type réel de la collection passée en argument, le type des objets qu'elle contient est un sous-type de `Object`, les considérer comme étant de type `Object` ne présente donc aucun risque. Néanmoins, il n'en est pas de même si l'on souhaite ajouter des objets à cette collection :

```
Collection<?> c = new ArrayList<String>();
c.add("Bonjour");          // erreur de compilation
```

en effet, le type des éléments de `c` étant inconnu, nous ne pouvons pas lui ajouter des objets car le paramètre de la méthode `add()` n'est pas d'un type quelconque, il est de type `E`, un type fixé une fois pour toute lors de la création de l'instance qui recevra ce message à l'exécution.

Annexe A

Assertions avec JML

On peut inclure trois types d'assertions : les invariants, associés à la classe, les préconditions et les postconditions, toutes deux associées à des méthodes. Avec JML, les assertions s'écrivent sous forme de commentaires situés dans le code de la classe ou avant l'entête des méthodes, à des emplacements précis et selon une syntaxe précise. Le but de ce document est de décrire cette syntaxe.

A.1 Syntaxe commune à tous les types d'assertion

Quelque soit le type d'assertion (invariant, précondition ou postcondition), certaines règles syntaxiques sont identiques :

- Le commentaire contenant l'assertion est délimité par les caractères `/*@` qui indiquent le début du commentaire et par `@*/` pour indiquer la fin du commentaire. Lorsque les assertions s'étendent sur plus d'une ligne, il est recommandé, afin d'assurer une meilleure lisibilité, de débuter chaque ligne par le caractère `@`.
- Le commentaire contenant l'assertion doit être placé en dehors du commentaire au format javadoc, l'emplacement recommandé est immédiatement après le commentaire javadoc c'est-à-dire, entre le commentaire javadoc et l'entête de la méthode.
- Lorsque l'assertion occupe une seule ligne, il est possible d'utiliser le délimiteur de commentaire `\\@`. Dans ce cas, le commentaire se termine en fin de ligne et il n'y a pas de délimiteur de fin de commentaire.
- Après la séquence de caractères indiquant le début du commentaire, un mot-clé indique le type d'assertion : `invariant` pour un invariant, `requires` pour une précondition et `ensures` pour une postcondition.
- Juste après le mot-clé indiquant le type d'assertion, il est possible de placer un label indiquant le rôle de cette assertion. Ce label apparaîtra dans le message affiché si JML détecte une violation d'assertion lors de l'exécution du programme, l'intérêt de ce label est donc d'aider le programmeur à détecter la source de l'erreur diagnostiquée par cette violation d'assertion. D'autre part, ce label apparaîtra dans la documentation de la classe générée par javadoc. La syntaxe à utiliser pour donner un label à une assertion est la suivante :
  - `(\lblneg label assertion );`
  - la chaîne de caractères constituant le label doit respecter les mêmes règles que celles définies pour les identificateurs JAVA : en particulier, un label ne doit pas contenir le caractère espace.
- Chacune des expressions booléennes constituant une assertion doit être terminée par un point-virgule. Du point de vue de la signification de l'assertion, ce point-virgule est équivalent à un et logique (opérateur JAVA `&&`).
- Lorsqu'il n'est pas possible d'exprimer l'assertion sous forme d'une expression JAVA, l'assertion pourra être spécifiée uniquement par un commentaire délimité par `(*` et `*)` (il s'agit d'un commentaire multiligne).

A.2 Invariant de classe

Il doit être placé dans un commentaire situé juste après la déclaration de la classe (i.e. la ligne `public class IdentificateurDeClasse {`).

Exemple A.2.1 Exemple d'invariant

```
/**
 * Cette classe permet d'effectuer des calculs sur les nombres complexes ...
 *
 * @author Moi
 * @since 18/12/2001
 * @version 0.1
```

```
 */
public class Complexe {
        /*@ invariant getNorme()
          @                == Math.sqrt(getPartieReelle()*getPartieReelle()
          @                        + getPartieImaginaire()* getPartieImaginaire());
          @*/
        ...
        //@ pure
        public double getPartieReelle() {
                ...
        }
        ...
        //@ pure
        public double getPartieImaginaire() {
                ...
        }
        ...
        //@ pure
        public double getNorme() {
                ...
        }
        ...
}
```

A.3 Préconditions

Les préconditions se placent juste avant l'entête des méthodes.

Exemple A.3.1 Exemple de précondition

```
        /**
           * Permet de calculer ...
           * @param x paramètre utilisé pour calculer ...
           * @return le résultat du calcul ...
           */
        /*@
          @ requires (\lblneg x_existe  x != null);
          @ requires x.getNorme() != 0;
          @ pure
          @*/
        public double getResultatDuCalcul(Complexe x) {
                // Le code de la méthode
        }
```

A.4 Postconditions

Les postconditions se placent juste avant l'entête des méthodes, au même emplacement que les préconditions.

Exemple A.4.1 Méthode qui donne une valeur à une propriété :

```
        /**
         * Permet de donner une nouvelle valeur à  ...
         * @param x La nouvelle valeur de ...
         */
        /*@
          @ requires        (\lblneg x_existe  x != null);
          @ ensures        getValue() == x;
          @*/
        public void setValue(Object x) {
                // Le code de la méthode
        }
```

Exemple A.4.2 Méthode qui retourne une valeur :

```
/**
 * Renvoie ...
 * @param x Nombre complexe ...
 * @return La valeur de ...
 */
/*@
  @ requires        (\lblneg x_existe  x != null);
  @ requires        x.getNorme() != 0;
  @ ensures        \result == (x.getNorme() * getDeviation());
  @ pure
  @*/
public double getNormalDeviation(Complex x) {
        // Le code de la méthode
}
```

A.4.1 Faire référence au résultat de la méthode : la variable `result`

Pour les méthodes renvoyant un résultat, il est souvent nécessaire de faire référence à ce résultat dans les postconditions de la méthode. Dans ce but, JML met à notre disposition la variable `\result`. Cette variable possède les caractéristiques suivantes :

— La valeur de cette variable est la valeur de l'expression spécifiée dans l'instruction `return` de la méthode.
— Le type de cette variable est le type déclaré pour la valeur de retour de la méthode.
— La variable `\result` est utilisable *uniquement* pour l'écriture des postconditions, elle n'est pas définie en dehors ce contexte. Elle ne peut donc être utilisée, ni dans le code JAVA de la méthode, ni dans les autres types d'assertion.

Exemple A.4.3 Utilisation de la variable `\result`

```
/**
 * Renvoie la norme de l'instance.
 */
/*@
  @ ensures \result == Math.sqrt(getPartieReelle()*getPartieReelle()
  @                        + getPartieImaginaire()* getPartieImaginaire();
  @ pure
  @*/
public void getNorme() {
        ...
}
```

A.4.2 Faire référence à l'état antérieur de l'instance : opérateur `old(...)`

Dans l'écriture des postconditions, il est souvent nécessaire de faire référence à l'état antérieur de l'instance. Dans ce but, JML met à notre disposition l'opérateur `\old(..)`. Cet opérateur possède les caractéristiques suivantes :

— La valeur d'une expression `\old(..)` est calculée par JML immédiatement avant l'exécution du code de la méthode.
— Une expression `\old(..)` est utilisable *uniquement* pour l'écriture des postconditions, elle n'est pas définie en dehors ce contexte. Elle ne peut donc être utilisée, ni dans le code JAVA de la méthode, ni dans les autres types d'assertion.

Exemple A.4.4 Utilisation d'une expression `\old(..)` :

```
public class Pile {
        ...
        /**
         * Place l'objet passé en paramètre sur la pile.
         * @param o L'Objet à mettre sur la pile.
         */
        /*@
          @ ensures        getSize() == \old(getSize()) + 1;
```

```
          @*/
        public void empiler(Object o) {
                ...
        }
}
```

A.5 Quel code a-t-on le droit de mettre dans une assertion ?

A.5.1 Prise en compte de la visibilité des caractéristiques

Une assertion est une expression booléenne destinée à être évaluée. Il faut que toutes ses variables :

— soient déclarées et visibles au moment de leur vérification, ce qui exclu les variables locales à une méthode pour les préconditions et les invariants ;

— soient visibles par l'utilisateur de la classe (sinon, l'assertion n'a aucun sens pour lui), ce qui exclu les propriétés privées (et généralement, toutes les propriétés sont déclarées privées, ce qui oblige à utiliser des méthodes du type `getX()`, `getValue()`, ...), et les variables locales à une méthode (déclarées dans le corps).

Exemple A.5.1 Ce qu'il ne faut pas faire :

```
public class Complexe {
    private double partieReelle;
    private double partieImaginaire;
    /**
     * Calcule et renvoie le résultat de la division de l'instance courante
     * par un autre nombre complexe.
     * @param z Le nombre complexe qui divise le complexe courant.
     * @result Le résultat de la division de l'instance courante par un
     * autre nombre complexe.
        */
    /*@
      @ requires    partieReelle != 0;
      @ requires    partieImaginaire != 0;
      @         ...
      @ requires    normeZ != 0;
      @ pure
      @*/
    public Complexe diviserPar(Complexe z) throws ComplexeDivisionParZero {
        double normeZ = z.getNorme();
        if (normeZ == 0) {
            throw new ComplexeDivisionParZero();
        } else {
            return (this.multiplierPar(z.conjugue())).produitScalairePar(1/normeZ);
        }
    }
    ...
}
```

A corriger par :

```
public class Complexe {
    private double partieReelle;
    private double partieImaginaire;
    ...
    //@ pure
    public double getPartieReelle() {
        return partieReelle;
    }

    //@ pure
    public double getPartieImaginaire() {
```

```
        return partieImaginaire;
    }

    //@ pure
    public double getNorme() {
        ...
    }

    /**
     * Calcule et renvoie le résultat de la division de l'instance courante par un
     * autre nombre complexe.
     * @param z Le nombre complexe qui divise le complexe courant.
     * @result Le résultat de la division de l'instance courante par un
     * autre nombre complexe.
        */
    /*@
      @ requires    getPartieReelle() != 0;
      @ requires    getPartieImaginaire() != 0;
      @         ...
      @ requires    z.getNorme() != 0;
      @ pure
      @*/
    public Complexe diviserPar(Complexe z) throws ComplexeDivisionParZero {
        double normeZ = z.getNorme();
        // Le test sur la valeur de la norme de z, n'a pas sa place ici puisque
        // la précondition joue justement ce rôle.
        return (this.multiplierPar(z.conjugue())).produitScalairePar(1/normeZ);
    }
    ...
}
```

A.5.2 Méthodes sans effets de bord : utilisation du mot-clé `pure`

Les assertions utilisant souvent des appels de méthodes, il convient de s'assurer que les méthodes utilisées sont sans effet de bord. A cet effet, JML introduit le mot-clé `pure` pour qualifier de telles méthodes. Ce mot clé doit être utilisé de la manière suivante :

— Avec JML, seules les méthodes spécifiée *pure* peuvent-être utilisées dans les assertions.

— Toute méthode sans effet de bord doit être qualifiée de *pure*. En effet, même si cette méthode n'est pas utilisée dans l'écriture des assertions de la classe courante, il faut laisser le plus de souplesse possible au programmeur d'autres classes qui utiliseraient la classe courante et pourraient donc avoir besoin d'utiliser les méthodes de la classe courante pour écrire les assertions de ces nouvelles classes.

A.6 Autres opérateurs de JML

La grammaire de JML pour les assertions est un peu plus riche que celle de Java. En effet, en plus des expressions booléennes standards de Java, JML permet de construire des expressions booléennes utilisant des quantificateurs universels ou existentiels.

A.6.1 Quantificateurs universel et existentiel

`\forall` (quelque soit) est un quantificateur universel et `\exists` (il existe) est un quantificateur existentiel.

Exemple A.6.1 Expression quantifiée universellement.

```
(\forall int i, j ; 0 <= i && i < j && j < 10 ; tab[i] < tab[j])
```

spécifie que les éléments du tableau `tab` dont l'indice est compris entre 0 et 9 sont triés par ordre croissant.

Les variables quantifiées (i.e. `i` et `j` dans l'exemple A.6.1) prennent toute les valeurs possibles satisfaisant l'expression booléenne située en deuxième partie de l'expression quantifiée entre les points-virgule (`;`) (i.e. `0 <= i && i < j && j < 10` dans l'exemple A.6.1). Si cette expression booléenne est absente, sa valeur par défaut est l'expression `true` (i.e. toujours vraie) et l'expression porte sur toutes les valeurs possibles des variables quantifiées. Le type d'une expression quantifiée est `boolean`.

```
(\forall <type> <var1>, <var2>, ... ; <expr_valeurs_possibles> ; <Expr_var>);
(\forall int <var> ; <borne_inf> <= <var> && <var> <= <borne_sup> ; <Expr_var>);
(\exists <type> <var1>, <var2>, ... ; <expr_valeurs_possibles> ; <Expr_var>);
(\exists int <var> ; <borne_inf> <= <var> && <var> <= <borne_sup> ; <Expr_var>);
```

Exemple A.6.2 Exemple avec une assertion `exists` et une assertion `forall` :

```
public class TestJml {
    public int[] tab;

    public static void main(String[] args) {
        System.out.println("Création d'une instance:");
        TestJml obj = new TestJml();
        System.out.println("Instance créée avec succès: Bye...");
    }
    /*@
      @ ensures (* Déclenche une violation de postcondition si
      @               un des éléments du tableau est égal a 2 : *) &&
      @        (\lblneg
      @            aucun_2
      @            (\forall int i ; 0 <= i && i < tab.length ; tab[i] != 2) );
      @ ensures (* Au moins un des éléments du tableau doit être égal a 3 : *) &&
      @        (\lblneg
      @             au_moins_un_3
      @            (\exists int i ; 0 <= i && i < tab.length ;  tab[i] == 3) );
      @*/
    public TestJml() {
        tab = new int[10];
        tab[2] = 2; // Violation de l'exception aucun_2
    }
}
```

A.6.2 Quantificateurs généralisés

Les quantificateurs `\max`, `\min`, `\product` et `\sum` sont des quantificateurs généralisés qui renvoient, respectivement, le maximum, le minimum, le produit ou la somme des valeurs de l'expression donnée, quand les variables satisfont l'expression déterminant les valeurs possibles. L'expression déterminant les valeurs possibles doit être de type `boolean`. L'expression dont les valeurs sont prises en compte pour le calcul doit être d'un type de base numérique comme `int` ou `double`; le type de l'expression quantifiée est du même type que cette dernière expression. L'expression dont les valeurs sont prises en compte pour le calcul est la dernière expression, c'est-à-dire celle suivant immédiatement l'expression déterminant les valeurs possibles. A titre d'exemple, les expressions suivantes sont toutes vraies :

```
(\sum int i; 0 <= i && i < 5; i) == 0 + 1 + 2 + 3 + 4
(\product int i; 0 < i && i < 5; i) == 1 * 2 * 3 * 4
(\max int i; 0 <= i && i < 5; i) == 4
(\min int i; 0 <= i && i < 5; i-1) == -1
```

Exemple A.6.3 Exemple d'utilisation des quantificateurs généralisés :

```
public class TestJml {
        public int[] tab;
        public int[] autreTab;

        public static void main(String[] args) {
                System.out.println("Création d'une instance:");
```

```
                TestJml obj = new TestJml();
                obj.testQuantificateurGeneralise();
                System.out.println("Instance créée avec succès: Bye...");
        }

        public TestJml() {
                tab = new int[10];
                tab[2] = 3;
                tab[3] = 1;
                tab[4] = 3;
        }

        /*@
          @ requires (\lblneg  min_0
          @                (\min int i ; 0 <= i && i < tab.length ;  tab[i]) == 0 );
          @ requires (\lblneg max_3
          @                (\max int i ; 0 <= i && i < tab.length ;  tab[i]) == 3 );
          @ ensures (\lblneg somme_sup_tab_length
          @           (\sum int i ; 0 <= i && i < tab.length ;  tab[i]) > tab.length );
          @ ensures (\lblneg produit_sup_4
          @                (\product int i ; 0 <= i && i < tab.length ;  tab[i]) > 4 );
          @ ensures (\lblneg min_1
          @                (\min int i ; 0 <= i && i < tab.length ;  tab[i]) == 1 );
          @*/
        public void testQuantificateurGeneralise() {
                for (int i = 0; i < tab.length; i++) {
                        tab[i] = 1;
                }
                tab[5] = 3;
                tab[6] = 2;
        }
}
```

A.6.3 Le mot-clé `fresh`

L'opérateur `\fresh` spécifie que des objets ont été alloués depuis le début de l'exécution de la méthode. Cet opérateur ne doit donc être utilisé que dans des postconditions. Par exemple, l'assertion `\fresh(x, y)` spécifie que les variables `x` et `y` ne sont pas `null` et qu'elle contiennent des références à des objets qui n'étaient pas encore créés au début de l'exécution de la méthode. Les arguments de `\fresh` peuvent être de n'importe quel type objet et le type d'une expression `\fresh(...)` est boolean.

Exemple A.6.4 Exemple d'utilisation de `\fresh(...)` :

```
public class TestJml {
        public int[] tab;
        public int[] autreTab = new int[10];

        public static void main(String[] args) {
                System.out.println("Création d'une instance:");
                TestJml obj = new TestJml();
                obj.testFresh();
                System.out.println("Instance créée avec succès: Bye...");
        }

        /*@
          @ ensures \fresh(tab);
          @ ensures \fresh(autreTab);
          @*/
        public void testFresh() {
                tab = new int[10];
                tab[2] = 2;
```

```
                // Violation de l'assertion \fresh(autreTab)
        }
}
```

Note : L'utilisation de l'expression `\fresh(this)` dans la spécification d'un constructeur est une erreur, car c'est l'opérateur JAVA `new` qui effectue l'allocation mémoire de l'objet ; le constructeur se contente uniquement d'initialiser cet espace mémoire déjà alloué.