Introduction
à la Programmation Orientée Objet

S4-POO1

L2 informatique – Université de Rouen

Année 2022-2023

`http://dpt-info-sciences.univ-rouen.fr/~andarphi/lbb/index.php?category/POO1`

Philippe Andary

Table des matières

Références bibliographiques du cours

1) Sur la toile

Page de ressources de ce cours
http://dpt-info-sciences.univ-rouen.fr/~andarphi/lbb/

The Java Tutorial : Learning the Java Language. Oracle.
https://docs.oracle.com/javase/tutorial/java/TOC.html

The Java language Specification (Third Edition). James GOSLING, Bill JOY, Guy STEELE
https://docs.oracle.com/javase/specs/jls/se6/html/j3TOC.html

2) Sur le papier

Bertrand MEYER
Conception et programmation orientées objet
Eyrolles, 2017 (ISBN : 978-2212675009)

Robert MARTIN
Coder proprement
Pearson, 2009 (ISBN : 978-2744023279)

David J. BARNES, Michael KÖLLING
Conception objet en Java avec Bluej (4° éd.)
Pearson Education, 2009 (ISBN : 978-2326001176)
Objects First with Java: A Practical Introduction Using BlueJ (5° éd.)
Pearson, 2011 (ISBN : 978-0132835541)

Philippe DRIX
Le langage C ANSI, vers une pensée objet en Java (3° éd.)
Dunod, 2000 (ISBN : 978-2100051939)

Chapitre 1 : Programmation orientée objet (POO)

1) Le changement de point de vue apporté par la POO

1.1) Facteurs de qualité

- Fiabilité : exprime le niveau de sûreté du logiciel en termes de correction (respect des spécifications) et de robustesse (comportement approprié en cas de situation anormale).
- Modularité : exprime la facilité de décomposition du logiciel en éléments autonomes, recouvrant ainsi la notion de réutilisabilité (capacité d'être utilisé dans plusieurs applications différentes) et celle d'extensibilité (facilité d'adaptation aux changements de spécification).
- Maintenabilité : simplicité de modification du logiciel après livraison au client.

Le but avoué de la POO est de maximiser la qualité du logiciel dans le contexte actuel de l'industrie du logiciel, à savoir :
- gros logiciels aux nombreuses fonctionnalités ;
- grand nombre de programmeurs, eux-mêmes répartis en équipes nombreuses ;
- dispersion géographique des équipes.

1.2) Question centrale de la POO

En programmation classique (impérative et structurée), la question fondamentale que doit se poser le développeur est « Que fait le système ? » ; il faut déterminer les actions à réaliser pour simuler l'évolution du système.
Par comparaison, dans la programmation orientée objet, la bonne question est « De quoi est fait le système ? » ; il faut cette fois déterminer quels sont les objets constituant le système et comment ils communiquent entre eux lorsque le système évolue.
L'exemple présenté en cours est celui de la cabine d'ascenseur, permettant de comparer la décomposition fonctionnelle descendante classique avec la décomposition en objets.

Dans le premier paradigme, le système est vu comme ayant une mémoire centrale à laquelle peuvent accéder, en lecture / écriture, de nombreuses routines (procédures ou fonctions). Le mécanisme de base est l'affectation. Un programme est la description d'algorithmes qui agissent sur la mémoire centrale, un processus est un programme en cours d'exécution.
Dans le paradigme objet, le logiciel est constitué de plusieurs petits systèmes indépendants mais qui collaborent entre eux, bénéficiant ainsi du principe d'émergence : « le tout est plus que la somme de ses parties ». Le mécanisme de base est ici l'envoi de message entre objets. Un programme est la description d'objets (leur structure et la nature des messages qu'ils échangent), un processus est un ensemble d'objets qui coopèrent.

2) Notion d'objet

Un objet représente, au sein du logiciel, une chose du monde réel ou virtuel. C'est un élément logiciel qui modélise cette chose et permet donc de la manipuler, ou d'en simuler le comportement. C’est une machine individuelle qui n’existe vraiment qu’au moment de l'exécution. Elle regroupe des données et des traitements sur ces données au sein d'une

même structure, et communique avec d'autres objets par envoi de messages.

La mémoire disponible pour une application est traditionnellement répartie en plusieurs zones distinctes dont les trois suivantes :
- une zone de code qui contient le code de l'application ;
- une pile, zone dans laquelle sont allouées les variables locales et stockée les adresses de retour d'appel de fonctions ;
- un tas, zone d'allocation dynamique à l'usage du programmeur.

C'est dans cette dernière zone que sont créés les objets.

Du point de vue interne, un objet est un assemblage d'éléments « caractéristiques » :
- des champs contigus en mémoire, placés dans le tas ;
- des sous-programmes dont le code est situé dans la zone de code, et qui permettent de manipuler ces champs et d'effectuer des traitements.

Au niveau du code source, l'objet doit être utilisé comme une boîte noire, à travers sa référence, par l'intermédiaire d'une variable. Certaines de ses caractéristiques sont accessibles aux clients, d'autres non.

La notion d'objet est définie par l'équation caractéristique des objets :

Objet = Identité + État + Comportement

2.1) État d'un objet

Pour pouvoir fonctionner en tant que machine logicielle participant à l'exécution d'un programme, un objet a accès à une portion de mémoire qui lui est personnellement réservée, dans le tas, sous la forme d'un enregistrement.

L'état interne de l'objet à un instant donné de l'exécution est l'état de sa mémoire à cet instant là. Il est représenté par la séquence des valeurs des champs de l'enregistrement. La correspondance entre les champs et les éléments de la séquence est arbitrairement fixée une fois pour toutes.

2.2) Messages et comportement d'un objet

Les objets collaborent entre eux dans le but de réaliser les fonctions du logiciel auquel ils appartiennent. Cette collaboration se traduit par un échange permanent de messages entre les différents objets. Le message est ainsi l'unité élémentaire de communication, il correspond à une demande de service qui émane d'un objet émetteur, et qui est à destination d'un objet récepteur. Il est composé d'un identificateur et de la séquence des paramètres effectifs requis pour le traitement du message.

Chaque objet a accès à un ensemble de sous-programmes qu'il peut exécuter pour interagir avec sa mémoire privée et y stocker les résultats des calculs qu'il effectue. L'objet accède à ses sous-programmes *via* un catalogue qui met en relation un identificateur de sous-programme et le bloc de code qui lui est associé.

Pour satisfaire la demande d'un émetteur, l'objet récepteur détermine, grâce à son catalogue, le code à exécuter en fonction de l'identificateur véhiculé par le message. Ainsi deux objets différents pourront choisir chacun d'exécuter du code différent en réponse au

même message.

Le comportement d'un objet est l'ensemble des messages qu'il est capable de traiter, c'est-à-dire ceux qui sont basés sur les identificateurs apparaissant dans son catalogue.

2.3) Identité d'un objet

L'identité est une caractéristique qui permet de distinguer les objets entre eux, de façon non ambiguë et indépendamment de leur état. Elle fonctionne à la manière du numéro national d'identification qui permet de différencier les ressortissants d'un même pays, quand bien même ce seraient des jumeaux physiquement indistinguables.

En programmation objet, le concept d'identité coïncide avec celui de référence : une valeur à l'exécution qui représente l'objet de manière unique.

3) Créer un objet

3.1) Classes

Pour pouvoir créer un objet, il faut d'abord définir son type : décrire comment est organisée sa mémoire et quels sont les messages qu'il saura traiter. Cette description générique de tous les objets ayant même état et même comportement est fournie dans un texte structuré. Celui-ci définit un type d'objets, qui agira comme un moule permettant de créer des objets conformes à la description donnée. Un tel type sera appelé une classe.

La description d'une classe se trouve dans un fichier source contenant :
- les déclarations d'attributs qui définissent les noms permettant, au sein du programme, d'accéder aux champs des objets de cette classe ;
- les définitions de constructeurs qui contiennent le code d'initialisation des champs des objets de cette classe ;
- les définitions de méthodes qui codent les sous-programmes communs à tous les objets de cette classe.

3.2) Instanciation

Instancier une classe, c'est créer un objet conforme à la description qu'en donne cette classe. On se sert de la classe comme d'un moule à objets. L'objet produit est appelé une instance de la classe. Cette action de création est appelée instanciation.

La règle de l'objet dit que dans un langage de classes (comme Java), tout objet est instance directe d'une et une seule classe.
Le type d'un objet est ainsi la classe génératrice dont il est une instance directe, et une classe est l'ensemble de toutes ses instances.

Java est un langage de classes, mais il existe d'autres langages qui n'en sont pas. Par exemple JavaScript, bien qu'ayant un nom proche de Java, n'est pas un langage de classes mais un langage de prototypes[1].

1 Cela signifie que les objets ne sont pas créés à partir d'une classe mais plutôt par clonage d'objets préexistants, appelés prototypes.

4) Manipuler un objet

Manipuler un objet, c'est l'inciter à interagir avec son environnement en lui envoyant un message : ordre à exécuter ou demande d'information. Mais au moment où les objets évoluent (lorsque le processus est en cours d'exécution) le développeur n'est pas là pour leur adresser la parole. Il faut donc tout planifier à l'avance au niveau du programme.

4.1) Références

Dans la vie courante, si vous souhaitez parler à l'une des personnes d'un groupe, il faut pouvoir l'identifier pour lui signifier que c'est à elle que vous vous adressez. De la même manière, pour pouvoir vous adresser à un objet, il faut d'abord pouvoir l'identifier. On a vu que cela se faisait au moyen de la référence de l'objet ; il est temps de décrire plus précisément ce qu'est une référence.

Une référence est une valeur à l'exécution qui est vide ou attachée. Lorsqu'elle est attachée, la référence désigne de manière unique un certain objet. Cette valeur est fournie par le système à la création de l'objet, elle n'est évidemment pas connue avant, quand le développeur écrit son programme.

Une référence désigne un objet, de la même façon qu'une adresse mémoire désigne un emplacement dans la mémoire. Mais à la différence d'une adresse, une référence est typée : le type d'une référence est la classe génératrice de l'objet auquel elle est attachée.

Dans les schémas que je ferai, j'utiliserai une représentation métaphorique des objets sous une forme qui évoque un canard en plastique avec lequel les petits enfants jouent dans leur bain : le bec est la référence attachée à l'objet, la tête donne le type de l'objet et le corps sa structure. C'est par son bec que l'on accède à un canard et que l'on peut lui attacher une variable. La référence vide est représentée sous la forme d'un bec relié à... une absence de canard.

4.2) Variables

Pour manipuler un objet depuis le texte d'un programme, on utilise une variable[2]. Lors de l'exécution, cette variable contiendra la référence d'un objet, établissant ainsi un lien à travers le temps entre le développeur qui écrit le programme et l'objet qui participe à son exécution (un peu comme un compositeur — le développeur — écrit une partition qui sera jouée plus tard par des musiciens — les objets).

Mentalement, on peut imaginer une variable comme une boîte étiquetée contenant une valeur à l'exécution. Le programme décrit la manipulation de la variable, et de la valeur qu'elle contient, par l'emploi du nom écrit sur cette étiquette. On confond généralement la boite avec son nom, de sorte que l'on parle de « la variable `x` » alors que `x` n'est en fait que le nom associé à la boîte.

Une variable référence est une variable dont les valeurs possibles sont uniquement des références d'objets. L'attachement est l'action qui établit un lien entre une variable et un objet (en stockant la référence de l'objet dans la variable). Une fois l'attachement effectué, la variable est dite attachée. Cette variable est la cible de l'attachement, l'objet auquel elle

2 Il s'agit d'un abus de langage, en fait on utilise un nom de variable.

va être attachée en est la source. L'affectation d'un objet à une variable, ainsi que le passage d'un objet comme paramètre effectif d'un appel de sous-programme, sont les deux actions dans un programme qui provoqueront un attachement à l'exécution.

Une variable peut aussi être détachée : elle contient alors la référence vide (valeur spéciale attachée à aucun objet), ce qui représente l'absence d'objet. Un attachement ultérieur d'une variable à un autre objet, ainsi que le détachement de cette variable, peuvent entraîner la « mort » de l'objet auquel la variable était initialement attachée ! En effet, un objet auquel aucune variable n'est attachée n'est plus accessible. Inutile, il sera recyclé par le glaneur de cellules (un processus qui tourne en permanence pour trouver les zones de mémoire inaccessibles afin de les libérer).

Dans un langage statiquement typé (comme Java) toute variable doit être typée au moment de sa déclaration. Cela contraint l'ensemble des valeurs que la variable pourra contenir par la suite, valeurs qui devront nécessairement être compatibles avec le type déclaré pour la variable. Pour chaque attachement, le compilateur pourra alors vérifier que le type de l'expression source (dont l'évaluation à l'exécution donnera la référence d'un objet) est compatible avec celui de la variable cible. Les classes sont des types disponibles à la compilation, utilisables dans ce contexte.

Pour terminer cette section, on pourrait penser qu'une variable référence est un pointeur... et l'on n'aurait pas tout à fait tort, mais pas tout à fait raison non plus. Contrairement aux pointeurs du C, il n'y a pas « d'arithmétique sur les références » ni d'opérateur d'indirection.

4.3) Accès à un champ

À chaque champ d'un objet correspond un attribut dans sa classe génératrice. Dans le code, pour dénoter l'accès au champ de l'objet, il faut utiliser l'attribut qui lui correspond, préfixé par le nom de la variable référençant l'objet.

Par exemple, en Java, si `x` est une variable attachée à un objet possédant un champ défini par l'attribut `n`, on accédera au champ de cet objet de la manière suivante : `x.n`. L'opérateur d'accès est noté par un point en Java, mais cela peut varier selon le langage utilisé.

4.4) Envoi de message

Un envoi de message correspond à un ordre, donné à un objet, pour qu'il exécute un traitement. Par exemple, si un objet `y` envoie le message « remplis-toi avec l'entier 123 » à un objet `x`, de type `Box`, `y` est l'émetteur du message, `x` en est le récepteur, et on s'attend à ce que, suite à la demande de `y`, `x` modifie son champ `content`[3] avec la valeur `123`.

Dans notre exemple, l'action résultant du traitement du message par l'objet `x` modifie la structure interne de cet objet. Comme on le verra par la suite, le traitement d'un message par un objet n'aboutit pas systématiquement à la modification de son état : il peut aussi simplement effectuer un calcul, ou permettre la consultation de son état sans modifier

3 Remarquez l'abus de langage : il ne s'agit pas exactement du champ `content`, mais précisément du champ associé à l'attribut `content`...

l'objet ni son environnement.

Un envoi de message se traduit au sein du programme par un appel de sous-programme. Dans notre exemple où `y` envoie un message à `x`, cela se traduit par le fait que le code exécuté par `y` contient l'instruction d'appel du sous-programme `fill` de l'objet `x`. Cette instruction n'est valide que si le type de `x` (`Box`) définit bien un sous-programme identifié par `fill` et admettant un argument de type entier.

4.5) Appel de méthode

Le code que sait exécuter un objet est représenté au sein du programme par des sous-programmes particuliers, que l'on appelle des méthodes et qui sont définis dans la classe génératrice de l'objet. Un envoi de message se traduit donc, à proprement parler, par un appel de méthode sur l'objet récepteur du message.

Pour coder cet appel, on préfixe le nom de la méthode par un opérateur d'accès et par le nom de la variable référençant l'objet. On trouvera donc l'instruction d'appel `x.fill(123);` dans le code exécuté par l'objet `y`. On dit que la variable `x` est la cible de cet appel.

Plus généralement, la cible d'un appel n'est pas toujours une variable. Cela peut être une expression, dont le type est un type référence, et dont la valeur est une référence d'objet.
Le nom de la méthode appelée, ainsi que le nombre et la position des arguments, doivent être compatibles avec ceux d'une méthode définie dans le type de l'expression cible.
Si l'une quelconque des contraintes ci-dessus n'est pas respectée, il se produit une erreur dès la compilation.
L'expression cible d'un appel doit représenter un objet au moment de l'exécution, sous peine de lever, en Java, une `NullPointerException`. Typiquement, c'est ce qui vous arrivera lorsque la cible d'un appel sera une variable détachée.

4.6) Objet courant

Un appel de méthode dans le programme conduira l'objet cible de cet appel à exécuter des instructions (celles contenues dans le corps de la méthode) lors de l'exécution du programme.

Dans le contexte de l'exécution d'une méthode, on appelle objet courant la variable spéciale qui fait référence à l'objet qui est en train d'exécuter cette méthode (en réponse au message qu'il a reçu).
Dans le langage Java, l'objet courant est une variable synthétique[4], appelée **`this`**.

Pour poursuivre sur notre exemple, le corps de la méthode `fill`, exécutée par `x`, contiendrait l'instruction « **`this`**`.content = val;` » (où `val` est le paramètre formel de la méthode `fill`, associé à la valeur `123` au moment de l'appel). Sur cet exemple précis, la variable **`this`** contiendrait la même référence que celle stockée dans `x`. La référence contenue dans **`this`** varie d'un appel à l'autre en fonction de l'objet cible de l'appel.

4 Une variable synthétique est une variable synthétisée (ou ajoutée) par le compilateur.

4.7) Différences entre méthode et fonction

À ce stade, on peut déjà justifier l'emploi du terme de méthode plutôt que de celui de fonction par le fait que les premières sont appelées dans le contexte d'un objet particulier, alors que les secondes sont appelées dans un contexte global, indépendant de tout objet. On peut encore exprimer cela en disant que l'objet cible d'un appel de méthode représente un paramètre privilégié pour la méthode. Par contre, aucun paramètre n'est privilégié lors d'un appel de fonction : dans ce cas tous les paramètres ont la même importance par rapport à l'exécution du code.

Plus précisément, un appel de méthode bénéficie du mécanisme de liaison dynamique, alors qu'un appel de fonction non.
La notion de liaison dynamique met en œuvre le principe selon lequel « lors d'un appel de méthode, c'est l'objet cible qui choisit le code qu'il doit exécuter. »

Pour un appel de fonction classique, l'adresse de branchement vers le code à exécuter est calculée par le compilateur, puis cette adresse est insérée dans le code compilé, sous la forme d'un branchement inconditionnel vers une adresse mémoire dans la zone de code. En revanche, pour un appel de méthode, l'adresse de branchement vers le code à exécuter dépend du catalogue des sous-programmes du type générateur de l'objet cible. Par exemple, un appel à la méthode `fill` ne doit pas exécuter le même code selon que la cible est une baignoire ou une personne[5].
Le compilateur ne pouvant pas calculer à l'avance l'adresse de branchement vers le bloc d'instructions du corps de la méthode, les mécanismes mis en place ne sont pas les mêmes dans un cas que dans l'autre, d'où la différence de terminologie employée.

5 On joue ici sur la polysémie du verbe *fill* en anglais, qui peut être utilisé dans le sens de « remplir un contenant de quelque chose » ou de « remplir les fonctions pour un poste particulier ».

Chapitre 2 : Syntaxe Java

La syntaxe du langage Java est issue de celle du langage C. On retrouvera donc un large fond commun, mais certaines particularités (comme par exemple l'absence de pointeurs ou encore la présence d'un vrai type booléen) différencient nettement les deux langages.

1) Commentaires

Ils peuvent être de trois sortes :

- le commentaire de fin de ligne :
```
du code // un commentaire
```
- le commentaire simple sur plusieurs lignes :
```
/*
  un commentaire qui se déroule
  sur plusieurs lignes
*/
```
- le commentaire de documentation (sur plusieurs lignes aussi) :
```
/**
 * Par convention on place une étoile
 * au début de chaque ligne
 */
```

2) Identificateurs

Ils sont construits à l'aide des lettres (lettres accentuées comprises), des chiffres et des symboles `$` et `_`. La casse des lettres est prise en compte (minuscule différent de majuscule), leur longueur est illimitée et ils ne peuvent pas commencer par un chiffre.

Attention toutefois : le symbole `$` est utilisé en interne par le compilateur pour nommer certains types. Pour éviter toute confusion, il n'est donc pas recommandé de l'utiliser dans ses propres identificateurs.

3) Types primitifs

Il y a deux catégories de types en Java : les types références et les types primitifs. Les premiers sont les types permettant de travailler sur les objets, les seconds sont les types permettant de travailler avec des valeurs qui ne sont pas des objets : les booléens, les caractères, les entiers et les flottants.

Type	Valeurs	Taille (en bits)
`boolean`	**`false`**, **`true`**	non spécifié
`char`	de `'\u0000'` à `'\uffff'`	16
`byte`	de `-128` à `127`	8
`short`	de `-32768` à `32767`	16
`int`	de `-2147483648` à `2147483647`	32

long	de -9223372036854775808 à 9223372036854775807	64
float	de 1.4E-45 à 3.4028235E38	32
double	de 4.9E-324 à 1.7976931348623157E308	64

3.1) Type boolean

Constitué de deux constantes littérales : **true** et **false**.
Les seules expressions logiques en Java sont celles typées avec **boolean**. Ainsi, le beug sournois bien connu des programmeurs C : `if (x = y) { ... } else { ... }` qui affecte à `x` la valeur de `y` juste avant de tester la valeur de `x` (devenue égale à celle de `y`) ne peut pas survenir en Java. En effet, le compilateur indiquera une erreur dans ce cas, et la seule solution compilable est `if (x == y) { ... } else { ... }` où `x == y` est bien une expression booléenne (sans effet de bord).

3.2) Type char

Ce type est en réalité un type entier (comme en C) mais il n'est pas fait pour être utilisé en tant que tel. La représentation standard des caractères est celle du jeu de caractères Unicode qui permet de représenter 65535 valeurs différentes, sur deux octets.

Les littéraux peuvent s'exprimer de trois façons différentes :
- `'A'` (classique)
- `'\101'` (octal)
- `'\u0041'` (Unicode)

Il existe plusieurs caractères spéciaux :
- `'\b'` (espace arrière)
- `'\t'` (tabulation)
- `'\n'` (nouvelle ligne)
- `'\f'` (page suivante)
- `'\r'` (retour chariot)
- `'\"'` (guillemet)
- `'\''` (apostrophe)
- `'\\'` (barre oblique inverse)

3.3) Types entiers

Ils sont au nombre de quatre : **byte**, **short**, **int** et **long**, pour des entiers signés sur 8, 16, 32 et 64 bits respectivement.

Les littéraux peuvent être exprimés dans trois bases numériques différentes : en octal (la représentation commence par 0 : `0101`), en décimal (représentation classique : `65`) ou en hexadécimal (la représentation commence par 0x : `0x41`).

Les littéraux ne représentent que des valeurs des types **int** ou **long**.
La représentation par défaut des entiers correspond au type **int**. Si l'on veut utiliser des littéraux de type **long**, ces derniers doivent se terminer par un `L` (il est fortement conseillé

d'utiliser une lettre majuscule pour distinguer la lettre `l` minuscule du chiffre `1`).

Remarque : Java ne calcule que sur les **`int`** ou les **`long`**, pas sur les **`byte`** ni les **`short`**. Par conséquent, si `b` est une variable déclarée de type **`byte`**, l'expression `b = b + b` n'est pas compilable car le résultat de l'addition est un **`int`**. La solution ici, si l'on est sûr qu'il n'y aura pas de débordement, est de coder `b = (byte) (b + b)`.

3.4) Types flottants (norme IEEE 745)

Ils sont au nombre de deux : **`float`** et **`double`**, pour des flottants sur 32 et 64 bits respectivement.

En simple précision, on dispose à peu près de 7 chiffres significatifs après la virgule et la représentation des littéraux se fait avec un `F` placé immédiatement après le dernier chiffre (exemple : `3.14F` ou `3.14f`).
En double précision, on dispose à peu près de 17 chiffres significatifs et la représentation des littéraux peut se terminer par un `D` (exemple : `3.14D` ou `3.14d` ou `3.14`). Par défaut, c'est-à-dire sans lettre à la fin du nombre, il s'agira d'un flottant double précision.

4) Conversions numériques

On parle de conversion d'un type p vers un type q lorsque le compilateur accepte de traiter une expression de type p comme si elle était de type q.
Par exemple lorsqu'on veut stocker la constante littérale 5 dans une variable de type **`float`**, une conversion entre le type **`int`** et le type **`float`** est effectuée.
Dans ces circonstances, le compilateur vérifie toujours que les types sont compatibles et produit une erreur si ce n'est pas le cas.
La conversion proprement dite se fait lors de l'exécution, la valeur convertie étant stockée dans une autre représentation que celle qui aurait été utilisée s'il n'y avait pas eu de conversion.

Les conversions peuvent entraîner des pertes de précision comme par exemple lorsque la valeur `123456795` (de type **`int`**) qui, convertie en flottant simple précision, donnera la valeur `1.23456792E8f`.
Mais elles peuvent aussi conduire à des pertes de magnitude comme lors de la conversion **`byte`** `b = (byte) 1234567` (`b` contiendra la valeur `-121`).

4.1) Conversions numériques élargissantes

Les conversions numériques élargissantes sont répertoriées dans le tableau suivant :

- de **`byte`** vers **`short`**, **`int`**, **`long`**, **`float`**, **`double`**
- de **`short`** vers **`int`**, **`long`**, **`float`**, **`double`**
- de **`char`** vers **`int`**, **`long`**, **`float`**, **`double`**
- de **`int`** vers **`long`**, **`float`**, **`double`**
- de **`long`** vers **`float`**, **`double`**
- de **`float`** vers **`double`**

Elles sont toutes implicites et n'entraînent aucune perte de magnitude.
Si p est un type entier et q un type flottant une conversion numérique élargissante de p vers q peut entraîner une perte de précision, mais ce ne sera jamais le cas si p et q sont

tous deux entiers ou tous deux flottants. Ces conversions ne génèrent jamais d'erreur.

4.2) Conversions numériques rétrécissantes

Les conversions numériques rétrécissantes sont répertoriées dans le tableau suivant :

- de **`short`** vers **`byte`**, **`char`**
- de **`char`** vers **`byte`**, **`short`**
- de **`int`** vers **`byte`**, **`short`**, **`char`**
- de **`long`** vers **`byte`**, **`short`**, **`char`**, **`int`**
- de **`float`** vers **`byte`**, **`short`**, **`char`**, **`int`**, **`long`**
- de **`double`** vers **`byte`**, **`short`**, **`char`**, **`int`**, **`long`**, **`float`**

Elles doivent toutes être formulées explicitement par le programmeur au moyen d'une opération de transtypage (*cast*) et peuvent entraîner des pertes de précision, voire de magnitude. Mais elles ne génèrent jamais d'erreur.

Remarque : une conversion de **`byte`** vers **`char`** est possible, elle se produit alors en deux temps : conversion élargissante de **`byte`** vers **`int`**, puis rétrécissante de **`int`** vers **`char`**.

5) Chaînes de caractères

Le type des chaînes de caractères est un type référence nommé `String`. Ses littéraux sont des séquences de caractères encadrées par des guillemets ; ils ne peuvent pas contenir de retour à la ligne mais ils peuvent contenir un ou plusieurs caractères `'\n'`.

6) Expressions

Une expression Java est une formule correctement parenthésée formée sur des littéraux, des noms de variables, de types ou de méthodes, et des symboles d'opérateurs.
Chaque expression a un type qui peut être calculé (à la compilation) à partir des éléments qui la composent.

6.1) Affectation

Ce n'est pas une instruction, c'est une expression dont l'évaluation produit un effet de bord[6]. La valeur produite est celle affectée à la mémoire lors de l'effet de bord. Ce n'est pas la seule expression qui produise un effet de bord lors de son évaluation (par exemple `i++`).

Une expression d'affectation (de la forme `v = e`) est composée d'un membre gauche (identificateur de variable, `v`), d'un opérateur d'affectation (`=`) et d'un membre droit (une expression, `e`, d'un type compatible avec celui du membre gauche).

Une expression d'affectation peut être multiple. Dans ce cas, l'opérateur d'affectation est associatif à droite : `i = j = k = 2` s'évalue en initialisant `k`, puis `j`, puis `i`, toutes ces variables prenant la valeur 2, qui est la valeur de cette expression.

Il existe plusieurs opérateurs composant l'affectation avec un autre opérateur : `*=`, `+=`, `-=`, `/=`, ... Par exemple, `v += e` équivaut à `v = v + e`. Il en va de même pour les autres.

6 un effet de bord est une modification de la mémoire provoquée par l'évaluation d'une expression.

6.2) Opérateurs

On se contentera de donner un tableau synthétique des principaux opérateurs :

Priorité	Opérateur	Arité	Type opérandes	Description
1	`++`, `--`, `+`, `-` `!`, `~`	1	arithmétique booléen	incrémentat°, décrémentat°, + et - unaires complément logique, complément bit à bit
2	`*`, `/`, `%`	2	arithmétiques	multiplicat°, divis°, reste modulo
3	`+`, `-` `+`	2	arithmétiques String	addition, soustraction concaténation
5	`<`, `<=`, `>`, `>=`	2	arithmétiques	comparaisons
6	`!=`, `==`	2	TT	différence, égalité
7	`&`	2	booléens ou arithmétiques	ET logique
8	`^`	2	booléens ou arithmétiques	XOR
9	`|`	2	booléens ou arithmétiques	OU logique
10	`&&`	2	booléens	ET conditionnel
11	`||`	2	booléens	OU conditionnel
12	`? :`	3	booléen, TT, TT	opérateur conditionnel
13	`=`, `*=`, `/=`, `%=`, `+=`, `-=`	2	TT	affectations

7) Instructions

Il y a douze sortes d'instructions.

7.1) Bloc d'instructions

Un bloc d'instructions est composé de zéro, une ou plusieurs instructions qui se suivent, regroupées entre deux accolades. Son exécution consiste à exécuter les instructions dans l'ordre où elles sont données.

7.2) Instruction vide

Une instruction vide est matérialisée par un simple point-virgule. Son exécution ne fait rien.

7.3) Instructions simples

Les instructions simples sont de deux sortes : les instructions simples à base d'expression (une expression suivie d'un point-virgule) et les instructions simples à base de déclaration

(une déclaration suivie d'un point-virgule).

L'exécution d'une instruction simple évalue l'expression ou effectue la déclaration selon le cas. La valeur retournée par le calcul de l'expression est ignorée.

7.4) Instructions conditionnelles

Les instructions conditionnelles sont de deux sortes : l'instruction **`if-else`** et l'instruction **`switch`**.

`if` `(<expr_bool>) I` **`[else`** `J`**`]`**
Son exécution évalue `<expr_bool>` puis exécute `I` si le résultat de l'évaluation est **`true`** ou `J` sinon. La partie **`else`** est optionnelle.

`switch` `(<expr>) {`**`{case`** `<expr_i>:` **`{`**`I`**`}*}* [default:`** **`{`**`I`**`}*]`**`}`
Son exécution :

1. évalue `<expr>` (qui doit être de type **`char`**, **`byte`**, **`short`** ou **`int`**) puis toutes les `<expr_i>` (qui doivent être compatibles avec `<expr>` et de valeurs deux à deux distinctes)
2. compare la valeur de `<expr>` avec la valeur de chaque `<expr_i>`, de la première à la dernière :
   - s'il existe `<expr_i>` telle que `<expr> == <expr_i>` alors toutes les instructions entre le **`:`** du case et la dernière `}` du switch sont exécutées ;
   - sinon, s'il existe une clause **`default`** alors toutes les instructions entre le **`:`** et la dernière `}` du **`switch`** sont exécutées ;
   - sinon aucune instruction n'est exécutée.

7.5) Instructions itératives

Il en existe trois : **`while`**, **`do-while`** et **`for`**.

`while` `(<expr_bool>) I`
Son exécution évalue `<expr_bool>` puis :

- si le résultat de cette évaluation est **`true`**, exécute `I` et recommence
- sinon l'exécution se termine.

`do` `I` **`while`** `(<expr_bool>);`
Son exécution exécute `I` puis évalue `<expr_bool>` et :

- si le résultat de cette évaluation est **`true`**, recommence
- sinon l'exécution se termine.

`for` `(`**`[`**`<expr_init>`**`];`** **`[`**`<expr_bool>`**`];`** **`[`**`<expr_maj>`**`])`** `I`
Son exécution évalue `<expr_init>`, puis `<expr_bool>` et :

- si le résultat de l'évaluation de `<expr_bool>` est **`true`**, exécute `I` puis évalue `<expr_maj>` et recommence au niveau de l'évaluation de `<expr_bool>`
- sinon l'exécution se termine.

7.6) Instructions de rupture

`break;`
Son exécution termine immédiatement l'instruction (**`switch`**, **`while`**, **`do-while`** ou **`for`**) à l'intérieur de laquelle elle se trouve.

`continue;`
Son exécution termine immédiatement l'itération courante de la boucle dans laquelle elle se trouve et entame l'itération suivante.

`return` **`[`**`<expr>`**`];`**
Son exécution termine immédiatement l'exécution de la méthode dans laquelle elle se trouve et retourne à l'appelant.

8) Variables locales

Une variable locale est une variable définie, à l'intérieur d'un bloc d'instructions, par une instruction simple de déclaration, avec ou sans initialisation.
Sa forme générale est :

`[final]` `<type>` **`{`**`[]`**`}*`** `<id>` **`[`** `= <expr>` **`]{`**`, <id>` **`[`** `= <expr>` **`]}*`**

La portée d'une variable locale est limitée à son bloc de définition.
Règles de déclaration des variables locales :

- Toute variable locale doit être déclarée et initialisée avant d'être utilisée.
- Une déclaration peut survenir n'importe où dans un bloc.
- Une variable locale n'est visible qu'à l'intérieur du bloc dans lequel elle est déclarée, à partir de sa déclaration.
- Une déclaration ne peut pas en masquer une autre.
- Une constante (variable déclarée avec **`final`**) ne peut prendre valeur qu'une seule fois.

Chapitre 3 : POO avec Java et BlueJ

1) Exemple de classe

Voici en exemple le texte d'une classe :

```
public class Box {

    // ATTRIBUTS

    private boolean closed;
    private String name;
    private int content;

    // CONSTRUCTEURS

    public Box(String n, int v) {
        this.name = n;
        this.content = v;
        this.closed = true;
    }

    public Box(int v) {
        this("Standard", v);
    }

    // REQUETES

    public String name() {
        return this.name;
    }

    public int value() {
        return this.content;
    }

    public boolean isClosed() {
        return closed;
    }

    // COMMANDES

    public void fill(int v) {
        if (this.isClosed()) {
            throw new AssertionError();
        }
        this.content = v;
    }
```

```
    public void open() {
        this.closed = false;
    }

    public void close() {
        this.closed = true;
    }
}
```

Comme vous pouvez le constater, ce texte respecte un certain nombre de conventions qu'il vous faudra respecter vous aussi lorsque vous coderez. Parmi ces conventions, il est utile d'en présenter immédiatement quelques-unes :

- chaque mot composant le nom d'un type référence s'écrit en commençant par une lettre majuscule, suivie de lettres minuscules ;
- le nom d'une méthode ou d'un attribut commence obligatoirement par une lettre minuscule, pour le reste on applique la même règle que ci-dessus ;
- chaque mot composant le nom d'une constante globale s'écrit tout en lettres majuscules, les mots étant séparés par un caractère `_` ;
- le nom d'un fichier source doit être identique (respect de la casse) au nom du type qu'il contient ; ce nom est ensuite suffixé avec `.java`.

De plus, toute variable doit être déclarée avant d'être utilisée, et cette déclaration doit mentionner le type de la variable.

Voici une description formelle de la déclaration d'une classe :

```
[public] [abstract|final] class <IdClasse> [<ClauseHéritage>] [<ClauseImplémentation>] {
    // Attributs
    [<Accessibilité>][static][final]<Type><IdAttr>[=<Expr>]{,<IdChamp>[=<Expr>]}*
    // Constructeurs
    [<Accessibilité>]<IdClasse>([<Args>])<ClauseErr><Bloc>
    // Méthodes
    [<Accessib.>][static|abstract]{<Type>|void}<IdMéth>([<Args>])<ClauseErr><Bloc>
}
```

où l'on a posé :

- `<ClauseHéritage>` ::= **`extends`** `<IdClasse>`
- `<ClauseImplémentation>` ::= **`implements`** `<IdInterface>`{`, <IdInterface>`}*
- `<ClauseErr>` ::= **`throws`** `<IdException>`{`, <IdException>`}*
- `<Accessibilité>` ::= **`public|protected|private`**
- `<Args>` ::= `<Type> <IdArg>`{`, <Type> <IdArg>`}*

2) Attributs et méthodes

2.1) Attributs

Un attribut est une variable (typée, en Java) permettant d'identifier et de manipuler, depuis le programme, un champ spécifique des instances de la classe. En Java, on utilise aussi le terme de « variable d'instance ».

Une déclaration d'attribut est de la forme : `[<Accessibilité>] [final] <Type><id>`
où `<Accessibilité>` **`::= private|protected|public`**
Nous reviendrons plus tard sur l'accessibilité, pour l'instant il suffira de savoir que

`private` réduit l'accessibilité au texte de la classe qui déclare l'attribut alors que **`public`** permet l'accès depuis le texte de n'importe quelle classe.
Lorsqu'un attribut est préfixé de **`final`** dans sa déclaration, sa valeur reste constante durant toute sa durée de vie (= la durée de vie de l'objet auquel il appartient).

L'état interne d'une instance de classe est la séquence des valeurs de chaque attribut.

2.2) Méthodes

Une méthode représente une séquence particulière d'instructions qu'exécutera un objet (en réponse à la réception d'un message). Une déclaration de méthode est de la forme :

```
[<Accessibilité>]<Type><IdMéth>([<Args>])[<ClauseErr>]<Corps>
```

dit autrement, c'est un entête (accessibilité + prototype[7] + clause d'erreur) suivi du corps de la méthode.

```
<Accessibilité> ::= private|protected|public
<Args> ::= arg_1, ..., arg_n (n ≥ 0)
<ClauseErr> ::= throws <TypeException>{, <TypeException>}*
<Corps> ::= { I_1 ... I_n } (n ≥ 0)
```

L'entête indique comment la méthode doit être utilisée, le corps est un bloc d'instructions qui décrit la séquence d'actions qu'exécutera l'objet cible de l'appel.

En Java, le passage d'arguments se fait par valeur uniquement. C'est pourquoi tout paramètre formel est assimilable à une variable locale.
Lorsqu'il n'y a pas d'argument, la paire de parenthèses est vide ; lorsqu'il y en a, chaque argument doit être déclaré, précédé de son type et séparé du suivant par une virgule.

Il faut bien distinguer variable locale et variable d'instance.

- Une variable d'instance est déclarée dans le texte de la classe, hors de toute méthode ; une variable locale est déclarée dans un bloc de code.
- La portée d'une variable d'instance s'étend à tout le code de la classe qui la déclare ; celle d'une variable locale est restreinte au bloc dans lequel elle est déclarée.
- Une variable d'instance représente un emplacement mémoire situé dans le tas ; une variable locale représente un emplacement mémoire situé sur la pile d'exécution.
- Une variable d'instance est automatiquement initialisée (initialisation standard, voir plus bas) ; une variable locale doit toujours être initialisée par le programmeur.
- Une variable d'instance a une durée de vie identique à celle de l'objet qui la contient ; une variable locale a une durée de vie limitée au bloc qui la déclare.

Le type de la valeur retournée par une méthode est indiqué devant le nom de la méthode lors de sa définition. Lorsque la méthode ne retourne pas de valeur, le type de retour indique **`void`**.

Si plusieurs méthodes définies dans un même type portent le même nom, mais avec une

7 *signature* = nom + type des paramètres ; *prototype* = type de retour + signature

liste d'arguments différente au fil des définitions, on dit que ce nom est surchargé. Toutefois, toutes les méthodes concernées par une situation de surcharge devraient conserver la même sémantique.

3) Constructeurs et instanciation

3.1) Constructeur

Un constructeur est une procédure définie dans une classe, de même nom que cette classe ; cette procédure contient le code d'initialisation des instances de la classe au moment de leur création.
Une déclaration de constructeur est de la forme :

```
[<Accessibilité>]<IdClasse>([<Args>])[<ClauseErr>]<Corps>
```

Un constructeur n'est pas une méthode, il s'utilise uniquement avec l'opérateur **`new`**, il ne contient jamais d'instruction **`return`** `<Expr>;` et son nom commence par une majuscule (le nom du constructeur et celui de la classe sont réellement identiques).

Il peut y avoir plusieurs constructeurs, dans ce cas ils ont tous le même nom et se distinguent par leur séquence d'arguments (il y a donc surcharge).
Un constructeur peut en appeler un autre de la même classe à l'aide de la forme **`this`**`(...)` utilisée obligatoirement comme première instruction dans le corps du constructeur appelant.

Lorsqu'une classe ne contient pas de constructeur explicitement donné par le programmeur, le compilateur en ajoute automatiquement un sans argument, de corps vide[8] et de même accessibilité que la classe.

3.2) Instanciation

La création d'un objet se fait par évaluation d'une expression de création d'objet

```
new <IdClasse>([<Args>])
```

Effet de cette évaluation :

1. création d'une instance de `<IdClasse>` :
   - allocation d'un espace-mémoire dans le tas, suffisamment vaste pour contenir tous les champs relatifs à l'objet
   - création de la référence de l'objet
2. initialisation implicite standard de chaque champ de l'objet :
   - numérique : `0`
   - caractère : `'\u0000'`
   - booléen : **`false`**
   - référence : **`null`**
3. initialisation explicite des champs selon `<args>` et le code du (ou des)

8 Ce n'est pas la vérité. On verra plus loin (héritage) ce que fait vraiment le compilateur, mais pour l'instant nous feindrons de croire que le corps de ce constructeur est vide d'instruction...

constructeur(s) exécuté(s)

4. retour de la référence attachée à l'objet

Les deux premiers points sont l'effet de l'exécution de l'opérateur **`new`**, le troisième représente l'exécution du constructeur et le dernier est la valeur retournée par l'évaluation de l'expression de création.

4) Le mot clé this

On l'a vu, le mot clé **`this`** a deux utilisations bien différentes en Java :

- la variable synthétique **`this`** désigne à chaque instant l'instance courante dans le corps d'une méthode ou d'un constructeur ;
- la forme d'appel **`this`**`(<Args>)` représente, au sein d'un constructeur, l'appel à un autre constructeur de la même classe (rappel : cet appel doit se trouver obligatoirement en première ligne du corps du constructeur appelant).

5) Contrôle d'accès

Le contrôle d'accès est un mécanisme fourni par le langage, qui permet au développeur de restreindre la possibilité d'accéder à un élément logiciel (attribut, méthode, constructeur, classe, etc) au sein du code source. L'accessibilité qui en découle est une propriété statique de l'élément, qui est calculable à partir des modificateurs d'accessibilité employés.

En java ces modificateurs sont au nombre de trois : **`public`**, **`protected`** et **`private`**. Les seuls appels autorisés sont les appels de caractéristiques accessibles. Pour l'instant, voici une description incomplète[9] et imprécise de l'effet de deux d'entre eux :

- une caractéristique publique (**`public`**) a vocation à être accessible par tout objet ;
- une caractéristique privée (**`private`**) a vocation à n'être accessible que par l'objet propriétaire.

6) Certains objets représentent des types

En Java, pendant l'exécution d'un programme, chaque classe qui lui est nécessaire est représentée par un objet (instance d'une classe spéciale `Class`). Cet objet spécial possède un état propre et un comportement propre, définis pour partie dans le type `Class`, et pour partie dans le texte de la classe elle-même. Dans le texte de la classe, la distinction entre les caractéristiques de la classe et celles de ses instances se fait par la présence ou l'absence (respectivement) du mot clé **`static`** lors de la déclaration de la caractéristique.

Un appel de caractéristique de classe doit avoir pour cible le nom de la classe concernée. Cette cible peut être implicite dans le texte de la classe, mais elle doit être explicitement formulée si l'appel est fait depuis le texte d'une autre classe (pour des caractéristiques accessibles, évidemment).

9 Nous complèterons au fil de l'eau, lorsque nous parlerons d'encapsulation (chapitre 4) et d'héritage (chapitre 5).

6.1) Attribut de classe

C'est un attribut représentant un champ propre à (l'objet qui représente) la classe, accessible à toutes ses instances bien qu'il n'appartienne à aucune d'entre elles. Il peut même être accessible au dehors de la classe selon le modificateur d'accessibilité utilisé lors de sa déclaration. Il est déclaré avec le mot clé **`static`**.
Son initialisation se fait au chargement de la classe, elle peut être définie à l'endroit de la déclaration ou dans un bloc statique d'initialisation, mais elle a toujours lieu hors des constructeurs et des méthodes.

6.2) Méthode de classe

C'est un service que l'on ne peut demander qu'à la classe, pas à ses instances. C'est une séquence d'instructions indépendantes de toute instance de la classe. Elle est déclarée avec le mot clé **`static`**.
Dans le corps d'une telle méthode, **`this`** n'est jamais définie (et la cible par défaut est la classe dans laquelle on se trouve).

7) Déclaration de classe

```
[public][abstract|final] class <IdClasse>
        [<ClauseHéritage>][<ClauseImplémentation>]{

    // ATTRIBUTS

    [<Accessibilité>][final][static]<Type><IdChamp>[=<Expr>]
            {, <IdChamp>[=<Expr>]}*

    // CONSTRUCTEURS

    [<Accessibilité>]<IdClasse>([<Args>])<ClauseErr><Corps>

    // [REQUETES|COMMANDES|OUTILS]

    [<Accessibilité>][static|abstract]<Type><IdMéth>([<Args>])
            <ClauseErr><Corps>
}
```

8) Méthodes et classes abstraites

8.1) Définitions

Une méthode abstraite est une méthode qui n'a pas de corps. Sa définition est obligatoirement précédée du mot-clé **`abstract`** :

```
public abstract double doubleValue();
```

Une classe abstraite est une classe non instanciable. Sa définition est obligatoirement précédée du mot-clé **`abstract`** :

```
public abstract class Number { ... }
```

Par opposition :
- une méthode concrète est une méthode dotée d'un corps
- une classe concrète est une classe instanciable

Toute classe contenant au moins une méthode abstraite doit être déclarée abstraite. Une classe abstraite peut ne contenir que des méthodes concrètes. L'intérêt des classes abstraites ne sera évident qu'à partir du chapitre 5 sur l'héritage !

8.2) Interfaces Java

Une interface Java est un type référence non instanciable défini avec le mot-clé **`interface`** :

```
public interface CharSequence { ... }
```

Une interface Java est constituée de déclarations de méthodes (obligatoirement) publiques et abstraites. Par exemple, dans `CharSequence` :

```
public interface CharSequence {
    [public] [abstract] int length();
    ...
}
```

mais aussi de définitions d'attributs (obligatoirement) constants, publics et statiques :

```
interface Bidon {
    [public] [static] [final] int MIN_SIZE = 5;
    ...
}
```

Attention : il n'y a pas de constructeur dans une interface !

Une classe Java peut implémenter une ou plusieurs interfaces Java, elle est alors automatiquement dotée de toutes les méthodes abstraites et de toutes les constantes des interfaces qu'elle implémente :

```
class String implements CharSequence, Comparable {
    // String récupère automatiquement toutes les méthodes
    //  abstraites et les constantes déclarées dans CharSequence,
    //  et Comparable
    ...
}
```

Pour être concrète, une telle classe doit donner corps à toutes les méthodes abstraites qu'elle récupère de ses interfaces.
Dans BlueJ, une relation d'implémentation sera représentée par une flèche à ligne pointillée et à tête triangulaire blanche, orientée de la classe vers l'interface qu'elle implémente.

8.3) Compatibilité pour l'affectation

On dit qu'un type S est compatible pour l'affectation avec un type T lorsqu'une affectation du genre « `T v = e;` » est légale, pour toute expression `e` de type S. Attention, l'objet référencé par `v` n'est alors manipulable que par le biais des méthodes du type T !

Les situations de compatibilité pour l'affectation sont nombreuses en Java. Par exemple, tout type Java (classe ou interface) est compatible avec la classe `Object`.

Une interface Java est un type référence tel que toute classe qui l'implémente est compatible avec elle pour l'affectation. Par exemple, la classe `String` est compatible pour l'affectation avec l'interface `CharSequence` :

```
CharSequence s = new String("arf");
```

car la classe `String` implémente l'interface `CharSequence`. La chaîne référencée par `s` ne sera toutefois manipulable dans le code que par le biais des méthodes de l'interface `CharSequence`.

La compatibilité a aussi à voir avec l'héritage, nous en reparlerons plus tard...

9) Mutabilité, non mutabilité

Un objet mutable est un objet dont l'état peut être modifié, un objet non mutable est le contraire d'un objet mutable : il ne peut pas changer d'état.

Le caractère mutable des objets est déterminé au niveau de leur type générateur : c'est la manière dont la classe est conçue qui fait que ses instances seront mutables ou non. On dira qu'une classe est mutable ou non mutable selon que ses instances seront (toutes) mutables ou (toutes) non mutables.

Par exemple la classe `Box` vue précédemment est mutable : elle contient des méthodes qui permettent de modifier l'état de ses objets (méthodes `fill`, `open` et `close`).
En revanche, la classe `java.lang.String` de l'API Java est non mutable puisqu'il n'existe aucune méthode permettant de modifier l'état de ses instances...

Il est souvent intéressant de travailler avec des objets mutables, notamment lorsqu'on veut stocker des informations qui évoluent avec le temps. Par exemple : des figures géométriques que l'on souhaite pouvoir déplacer dans le plan.
Lorsque les informations liées à l'objet sont uniquement dépendantes de l'objet, il faut alors bien prendre garde à ce que ces derniers ne dévoilent pas leur structure interne, sans quoi cette structure serait modifiable indépendamment de l'objet, et nous verrons bientôt que ce serait une grave erreur de conception.
Si l'on considère des cercles dotés d'un point matérialisant leur centre, et d'une méthode de translation ne pouvant fonctionner que dans certaines circonstances uniquement (dans le quart de plan positif par exemple, pas ailleurs). Ce serait une erreur que de donner accès au centre d'un tel cercle car rien n'empêcherait plus alors les clients d'en translater le centre sans aucune contrainte.

En revanche, si une partie de la structure interne de l'objet peut évoluer indépendamment de l'objet lui-même, il sera probablement utile de partager cette donnée de l'objet avec son environnement, pour permettre justement à la donnée d'évoluer indépendamment de l'objet. Ainsi, on pourrait très bien envisager qu'un objet représentant une personne, dotée d'une adresse, puisse communiquer cette adresse à un organisme d'urbanisation qui, lui, pourrait la modifier : ici, la modification de l'adresse d'une personne ne relève pas de la responsabilité de la personne.

10) Identité et équivalence

10.1) Test d'identité

Deux objets sont identiques s'ils ont même référence, c'est-à-dire s'ils ne constituent qu'un seul et même objet, s'ils occupent le même emplacement mémoire[10].

L'identité “référencielle” entre deux objets se teste au moyen de l'opérateur `==` (à ne pas confondre avec l'opérateur d'affectation) appliqué entre deux variables, ou plus généralement entre deux expressions.

10.2) Test d'équivalence

Deux objets sont équivalents s'ils ont même type et même état courant.

L'équivalence entre deux objets se teste au moyen de la méthode

```
boolean equals(Object)
```

Lorsqu'une classe ne définit pas explicitement cette méthode, le compilateur en rajoute toujours une dont la sémantique équivaut au test d'identité référencielle :

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

On peut modifier ce comportement en définissant soi-même une méthode `equals` (rigoureusement de même entête que ci-dessus) dans la classe.

11) Méthode toString

Une autre méthode est implicitement présente dans toute classe :

```
public String toString()
```

Elle permet d'obtenir une représentation textuelle de tout objet sous la forme :

```
<nom_de_la_classe_de_l'objet> + @ + <adresse_mémoire_de_l'objet>
```

Ici encore, on peut modifier ce comportement en définissant soi-même une méthode `toString` (rigoureusement de même entête que ci-dessus) dans la classe.

12) Erreurs et exceptions en Java

12.1) Définitions

Une panne logicielle est une situation anormale provoquant la fin abrupte d’une instruction. L’exécution de l’instruction concernée par la panne ne conduit donc pas à sa fin normale.

Une panne peut survenir dans l'une des situations suivantes :

- une situation anormale est détectée par le matériel ou le système d’exploitation ;

10 Cette phrase est à replacer dans le contexte où l’on assimile un objet avec la variable qui le référence...

- le chargement d'une classe ou l'édition d'un lien ne peuvent aboutir ;
- les limites d'une ressource sont dépassées ;
- l'évaluation d'une expression viole la sémantique du langage (par exemple : appel de méthode sur une cible vide, débordement d'indice pour un tableau, ...) ;
- une instruction **`throw`** est exécutée ;
- une erreur interne de la JVM survient.

Une panne se traduit en Java par la création d'un objet d'un type particulier, pouvant être « lancé » (*to throw* en anglais). Ces objets lançables peuvent être de deux sortes : erreur ou exception.

Le déclenchement d'un tel objet lançable provoque l'interruption immédiate du flot d'exécution du programme par la JVM, puis il initie la propagation de cet objet. la JVM transmet alors l'objet au code appelant, offrant à ce dernier la possibilité de le capturer pour gérer la panne.
Le langage Java met à notre disposition des mots clés permettant de définir deux zones particulières dans le code :

- **`try`** définit un bloc de code délimitant la zone de détection d'une panne ;
- **`catch`** définit un bloc de code délimitant une zone de gestion possible pour la panne.

À un bloc **`try`**, on peut associer plusieurs blocs **`catch`**, chacun permettant de repérer un type particulier d'objet lançable. Lorsqu'une panne survient dans un bloc **`try`**, le flot d'exécution est immédiatement interrompu à l'endroit où la panne est survenue. Le contrôle du flot d'exécution est alors donné à l'un des blocs **`catch`** associés au bloc **`try`**, comme traitement de remplacement du code fautif et du code restant (qui n'ont pas pu être exécutés) dans le bloc **`try`**. Si l'un des blocs **`catch`** convient, l'objet lançable est traité par ce bloc au lieu d'être propagé ; dans le cas contraire, il se propage au code appelant où il pourra être traité par un procédé similaire.
De plus, il est possible de relancer un objet lançable depuis l'intérieur d'un bloc **`catch`**.
Enfin, le mot clé **`finally`** permet de définir une troisième zone particulière, sous la forme d'un bloc de code qui sera systématiquement exécuté à la fin d'un **`try-catch`**, qu'il y ait eu récupération ou non, qu'il y ait eu rupture d'exécution ou non :

```
try {
    ... zone de détection
} catch (Exception1 e) {
    ... zone de gestion pour un problème de type Exception1
} catch (Exception2 e) {
    ... zone de gestion pour un problème de type Exception2
} catch ...
    ...
} finally {
    ... zone de code obligatoirement exécutée
}
```

12.2) Différents types d'objets lançables

En Java, il existe différents types d'objets lançables. Par exemple : `RuntimeException`, `NullPointerException`, `ArrayIndexOutOfBoundsException`, `AssertionError`, `StackOverflowError`, pour ne citer que les plus célèbres.

On constate donc qu'il existe deux grandes catégories : les erreurs et les exceptions. La première représente les pannes non récupérables, c'est-à-dire les situations fatales où le problème ne peut pas être traité par le logiciel et où il faut arrêter l'application.
La seconde regroupe les pannes pour lesquelles le problème peut être géré au sein du logiciel.
Ainsi, une `OutOfMemoryError` signale un débordement de mémoire et il n'y a rien d'autre à faire dans cette situation que d'arrêter le programme et de régler le problème soit en corrigeant le programme, soit en lui allouant plus de mémoire lors de la prochaine exécution.
Au contraire, la survenue d'une `IOException` signale une défaillance lors d'une opération d'entrée/sortie, par exemple lors de la lecture d'un fichier sur le disque. Cette situation peut se gérer sans pour autant arrêter l'exécution du logiciel, par exemple en demandant à l'utilisateur de refaire une tentative, ou de lire dans un autre fichier...
On partitionne aussi les exceptions en deux catégories : selon qu'elles sont contrôlées ou non par le compilateur.
Les exceptions contrôlées par le compilateur (comme `IOException`) sont des exceptions dont la survenue éventuelle doit obligatoirement être déclarée (mot clé **`throws`**) dans l'entête des méthodes susceptibles de les lever, afin que le compilateur puisse vérifier que le développeur sait qu'il utilise du code sensible :

```
public void m() throws IOException {
    ... du code pouvant lever une IOException
}
```

La méthode `m` contient du code qui pourrait lever une `IOException` et ne souhaite pas gérer ce problème elle-même (pas de **`try-catch`**), elle doit donc indiquer dans son entête une clause **`throws`** sinon le compilateur émettra une erreur signalant le risque au développeur. Notez bien que ce n'est pas parce qu'une méthode signale qu'elle pourrait lever une exception qu'elle en lèvera obligatoirement une...

Les exceptions non contrôlées par le compilateur peuvent, elles, ne pas être déclarées dans l'entête des méthodes susceptibles de les lever. Elles sont toutes, sans exception (hu ! hu !), des sortes de `RuntimeException`. Ce sont les exceptions de la vie de tous les jours, celles qu'il serait fastidieux de déclarer systématiquement car elles sont potentiellement fréquentes. Par exemple, une `NullPointerException` peut survenir à chaque appel de caractéristique (quand la cible vaut **`null`**). Si cette exception était contrôlée, il faudrait la signaler dans l'entête de toutes les méthodes qui manipulent des variables !

12.3) Exceptions utilisateur

Le développeur peut, s'il le souhaite, créer ses propres types d'exceptions et les utiliser ensuite dans son code.
Pour cela il faut définir une nouvelle classe ainsi :

```
public class MonException extends UnTypeDExceptionControlee {
    public MonException() { this(""); }
    public MonException(String msg) { super(msg); }
}
```

De telles exceptions sont alors utilisables dans le code, par exemple de la manière suivante :

```
...
if (panne détectée) {
    throw new MonException("message décrivant la panne");
}
...
```

Notez bien que l'on est ici dans un corps de méthode, pas dans son entête, et qu'il faut utiliser **throw** et non pas **throws** (attention au s !).

13) Tableaux

En Java, un tableau est un objet. Par conséquent il est alloué dans le tas et on y accède à l'aide de sa référence. Sa structure interne, en tant qu'objet, est constituée de ses éléments et de sa taille (un champ constant défini au moment de la création du tableau, et publiquement accessible).

Déclarer une variable d'un type tableau crée une variable référence dont les valeurs possibles seront les références d'objets de ce type tableau :

```
<Type>{[]}+ <Identificateur> [= <ExprCréationTableau>]
```

Une expression de création de tableau est de l'une des deux formes suivantes :

- **new** <Type>{[<Expr>]}+{[]}*
- **new** <Type>{[]}+ <ValeurInitiale>

où <expr> est une expression de type **int** et où une valeur initiale est une séquence de valeurs du type des éléments du tableau, donnée entre accolades.

Bien qu'un tableau soit créé dynamiquement (alloué dans le tas à la demande du programmeur), ce n'est pas une structure dynamique car sa taille ne varie pas au cours du temps : **public final int** length. Cette taille, inconnue lors de la déclaration du tableau, n'est fixée qu'au moment de l'instanciation.
Le domaine de variation des indices d'un tableau t est toujours [0 .. t.length-1].
En cas de débordement, la JVM lèvera une ArrayIndexOutOfBoundsException.

13.1) Création de tableau simple de type primitif

On peut créer un tableau en 3 étapes :

1. déclaration : **int**[] x;
2. construction (+ initialisation standard) : x = **new int**[2];
3. initialisation : x[0] = 1; x[1] = 2;

On peut aussi regrouper toutes ces étapes en une seule :

```
int[] x = new int[] {1, 2};
```

Notez que, dans ce cas, le compilateur détermine automatiquement la taille du tableau.

13.2) Création de tableau simple d'objets

Création en 3 étapes :

1. déclaration : `Box[] b;`
2. construction (+ initialisation standard) : `b = new Box[2];`
3. initialisation : b[0] = `new Box("X", 0); b[1] = new Box("Y", 0);`

Ou en une seule instruction, le compilateur déterminant automatiquement la taille :

```
Box[] b = new Box[] { new Box("X", 0), new Box("Y", 0) };
```

13.3) Tableaux à plusieurs dimensions

Un tableau à une seule dimension est un tableau simple tel que défini précédemment. Un tableau à n dimensions (n ≥ 2) est un tableau simple de tableaux à n-1 dimensions.

On peut initialiser un tableau à plusieurs dimensions d'un seul coup, à l'aide d'un initialiseur de tableau :

```
int[][] tab = new int[][] {{1, 2, 3}, {4, 5}}
```

On peut aussi repousser l'initialisation à plus tard, il faut alors préciser une ou plusieurs tailles, à partir de la gauche :

```
int[][] tab = new int[2][];
... initialisation du contenu ultérieurement
```

14) Types enveloppes

Une classe enveloppe est une classe qui permet de représenter les valeurs d'un type primitif à l'aide d'objets dont l'état ne varie pas.
À tout type primitif correspond sa classe enveloppe :

- **byte** : Byte
- **short** : Short
- **int** : Integer
- **long** : Long
- **float** : Float
- **double** : Double
- **char** : Character
- **boolean** : Boolean

Typiquement, on trouve dans ces classes des constructeurs qui permettent de passer d'une valeur primitive à une valeur du type enveloppe correspondant et des méthodes de conversion qui permettent de retrouver la valeur primitive. Par exemple :

```
Integer i = new Integer(123);
int n = i.intValue();
```

On y trouve aussi des méthodes de conversion entre les valeurs primitives et leur représentation textuelle. Par exemple :

```
int n = 123;
String s = Integer.toString(n); // ou aussi String.valueOf(n)
Integer i = Integer.valueOf(s); // ou aussi new Integer("123")
```

15) Classes de chaînes de caractères

Une chaîne de caractères est une séquence de zéro ou plusieurs caractères.

15.1) Chaînes de caractères non mutables

Le type des chaînes de caractères non mutables se nomme `String` en Java, on ne peut pas modifier l'état de ses instances. On peut construire les instances de `String` de deux manières :

- `new String("Salut")`
- `"Salut" // littéral chaîne de caractères`

Un même littéral chaîne de caractères représente toujours le même objet, ce qui n'est pas le cas des objets créés à l'aide du constructeur.

Les méthodes les plus utilisées sont :

- longueur d'une chaîne : `int length()`
- ième caractère d'une chaîne : `char charAt(int i)`

Lorsqu'on veut tester l'équivalence de deux chaînes de caractères, il faut utiliser la méthode `equals` :

```
String x = new String("bonjour");
String y = new String("bonjour");
System.out.println(x == y); // false
System.out.println(x.equals(y)); // true
```

Remarque : si `x` est de type `String` alors `x == x.toString()`.

On peut concaténer des chaînes avec l'opérateur `+` :

`"Salut " + "à tous"` donne la nouvelle chaîne `"Salut à tous"`

Attention : il y a bien trois chaînes distinctes en jeu !
On peut concaténer une chaîne avec n'importe quoi, le résultat sera toujours une chaîne :

`1 + 1 + "b" + (1 + 1 + 1)` donne la nouvelle chaîne `"2b3"`

car `chaîne + objet` équivaut à `chaîne + objet.toString()`, et `chaîne + primitif` équivaut à `chaîne + String.valueOf(primitif)`.

15.2) Chaînes de caractères mutables

Le type des chaînes de caractères mutables se nomme `StringBuffer` en Java, il possède des méthodes permettant de modifier l'état de ses instances.

Les méthodes communes aux deux classes de chaînes précédentes sont :

```
char charAt(int)
boolean equals(Object)
int indexOf(String)
int lastIndexOf(String)
int length()
String substring(int)
String substring(int, int)
String toString()
```

Les méthodes particulières aux `String` sont :

```
String concat(String)
int compareTo(String)
String replace(char, char)
```

Les méthodes particulières aux `StringBuffer` sont :

```
StringBuffer append(TT)
StringBuffer delete(int, int)
StringBuffer deleteCharAt(int)
StringBuffer insert(int, TT)
StringBuffer replace(int, int, String)
StringBuffer reverse()
void setCharAt(int, char)
void setLength(int)
```

où `T ::= boolean|char|char[]|int|long|float|double|Object`
et `TT ::= T|String`

15.3) La classe java.util.StringTokenizer

C'est une classe dont les instances permettent de découper une chaîne de caractères en mots. Dans la chaîne, les mots sont des séquences de caractères délimitées par des caractères particuliers appelés séparateurs.

Exemple simple d'utilisation :

```
StringTokenizer hachoir = new StringTokenizer("Bonjour le
monde !", "n ");
StringBuffer resultat = new StringBuffer();
while (hachoir.hasMoreTokens()) {
     String mot = hachoir.nextToken();
     mot = mot.toUpperCase();
     resultat.append(mot + "\n");
}
```

Exemple pour montrer la facilité d'utilisation des `StringTokenizer` : trier les mots d'une phrase

```
String data = "Belle Marquise, vos beaux yeux me font mourir
d'amour !";
String result = "";
StringTokenizer st = new StringTokenizer(data, " ,!", false);
String[] words = new String[st.countTokens()];
int i = 0;
while (st.hasMoreTokens()) {
    words[i] = st.nextToken();
    i += 1;
}
Arrays.sort(words);
while (i > 0) {
```

```
    i -= 1;
    result = words[i] + " " + result;
}
System.out.println(result);
```

16) Bibliothèques System et Math

Une bibliothèque de fonctions est une classe non instanciable dont tous les membres sont statiques. Elle n'a pas vocation à fabriquer des instances, c'est plutôt un réservoir de fonctionnalités.

16.1) Bibliothèque java.lang.System

- `final PrintStream err` : canal de sortie d'erreur
- `final InputStream in` : canal d'entrée standard
- `final PrintStream out` : canal de sortie standard

- `void exit(int)` : provoque la terminaison du programme avec valeur de retour
- `String getProperty(String)` : retourne une propriété système

16.2) Bibliothèque java.lang.Math

Ci-dessous, on note `T` ::= `double`|`float`|`int`|`long`

- `final double E`
- `final double PI`

- `T abs(T)`
- `T max(T, T)`
- `T min(T, T)`
- `double acos(double)`
- `double asin(double)`
- `double atan(double)`
- `double ceil(double)`
- `double cos(double)`
- `double exp(double)`
- `double floor(double)`
- `double log(double)`
- `double pow(double)`
- `double random()`
- `double rint(double)`
- `long round(double)`
- `int round(float)`
- `double sin(double)`
- `double sqrt(double)`
- `double tan(double)`
- `double toDegrees(double)`
- `double toRadians(double)`

17) Utilisation de BlueJ en TP

Page d'accueil : `http://www.bluej.org`
Téléchargements :

- des instructions d'installation `https://www.bluej.org/download/install.html`
- du logiciel `https://www.bluej.org/download/download.html`
- du tutoriel `https://www.bluej.org/tutorial/tutorial-v4.pdf`
- du (vieux) manuel de référence `http://www.bluej.org/doc/bluej-ref-manual.pdf`

18) Annexe : standards de codage

18.1) Fichier source

- nom rigoureusement identique à celui de l'élément public qu'il contient
- suffixé par ".java"

Structure d'un fichier source :

1. déclaration de paquetage : `package ...;`
2. déclarations d'importation : `import ...;`
3. documentation de l'élément (contenant la spécification de l'invariant) : `/** ...@inv... */`
4. définition de l'élément : `... class ... { ... }`

Structure du texte d'une classe : pour chacune des sections d'abord les éléments statiques puis les éléments d'instance, d'abord les éléments publics, puis protégés, puis sans modificateur d'accessibilité, et enfin privés.

1. section `// ATTRIBUTS STATIQUES` : déclarations commentées d'attributs de classe
2. section `// ATTRIBUTS` : déclarations commentées d'attributs d'instance
3. section `// CONSTRUCTEURS` : définitions spécifiées de constructeurs
4. section `// REQUETES` : définitions spécifiées de requêtes
5. section `// COMMANDES` : définitions spécifiées de commandes
6. section `// OUTILS` : définitions spécifiées de méthodes auxiliaires

18.2) Indentation et espacement

Une indentation fait quatre (4) espaces, ni plus, ni moins. Ce sont des espaces, pas des tabulations.

La longueur maximum d'une ligne est de 80 caractères, au-delà il faut la couper. Une expression ou une instruction trop longues s'étendront sur plusieurs lignes, coupées de la manière suivante :

1. après une virgule
2. avant un opérateur
3. pas à l'intérieur de parenthèses imbriquées (si possible)
4. en indentant à l'identique les expressions de même niveau
5. deux indentations (8 espaces) pour bien marquer la différence avec une indentation

normale causée par l'entrée dans un nouveau bloc

Les espaces suivent les règles suivantes :

1. une espace entre un mot-clé et la parenthèse ouvrante qui le suit : `while (condition)`
2. aucune espace entre un nom de méthode et la parenthèse qui le suit : `o.m(args)`
3. une espace après chaque virgule dans une liste : `o.m(arg1, arg2)`
4. tout opérateur binaire (sauf `.`) doit être entouré d'espaces : `x + y` mais `o.a`
5. une espace après un opérateur de transtypage : `(int) 5.2`

18.3) Commentaires

Le commentaire de documentation (`/** ... */`) sert à spécifier un élément d'une manière indépendante de son implémentation, quel que soit le niveau d'accessibilité de cet élément.
Il doit être placé juste avant l'élément, doit avoir même indentation que l'élément, peut contenir une description d'invariant (`@inv`) lorsque l'élément est une classe ou une interface et peut contenir une spécification de contrat (`@pre`, `@post`) lorsque l'élément est une méthode ou un constructeur.

Le commentaire simple (`/* ... */`) sert à décrire une portion de code.
Il doit constituer une valeur supplémentaire en soi et permettre une meilleure compréhension du code, sinon il est inutile.

Le commentaire de fin de ligne (`// ...`) peut être utilisé comme un commentaire simple à la fin d'une (courte) ligne de code.
Il peut aussi permettre de masquer certaines lignes de code au compilateur.

Remarques : N'utilisez jamais de commentaires en forme de grosse boîte délimitée par des `*` ou d'autres caractères...

18.4) Déclarations

Une seule déclaration doit être faite par ligne de code (par exemple, il ne faut pas déclarer plusieurs variables du même type sur la même ligne).

Il faut déclarer les tableaux au niveau du type de la variable, pas de son nom (par exemple faire `int[] t`, ne pas faire `int t[]`).

Dans la déclaration d'un type, d'un constructeur ou d'une méthode :

- l'accolade ouvrante doit être placée à la fin de la ligne d'entête (pas au début de la ligne suivante) ;
- l'accolade fermante doit être placée sur une ligne vide indentée au même niveau que la ligne d'entête.

Les variables locales sont déclarées au début du bloc qui les utilise (sauf pour les boucles `for` : dans l'entête). Elles sont initialisées le plus tôt possible (idéalement en même temps que leur déclaration).

Les variables d'instance sont initialisées dans les constructeurs uniquement, pas lors de leur déclaration.

Ne jamais masquer une variable d'instance par une variable locale ; éviter de les masquer par un paramètre formel.

18.5) Instructions

Une ligne de code doit correspondre à une seule instruction. Seules les instructions composées dérogent à cette règle et elles seront construites ainsi :

- une accolade ouvrante se place en fin de ligne et non sur une ligne vierge ;
- une instruction immédiatement interne gagne un niveau d'indentation ;
- il faut entourer les instructions internes d'une paire d'accolades, même s'il n'y a qu'une seule instruction interne ;
- s'il n'y a pas d'instruction interne il faut placer un bloc vide contenant un commentaire indiquant ce fait ;
- une accolade fermante se place sur une nouvelle ligne, indentée comme la première ligne de l'instruction.

```
if (condition) {
    instructions
} else if (condition) {
    instructions
} else {
    instructions
}

while (condition) {
    instructions
}

do {
    instructions
} while (condition);

for (init ; cond ; màj) {
    instructions
}

switch (expression) {
case cas1:
    instructions
    break;
case cas2:
    instructions
    // ici on continue sur le cas suivant
case cas3:
    ...
default:
    instructions
```

```
    break;
}
```

18.6) Nommage

Les paquetages s'écrivent tout en minuscules avec des noms courts. Deux noms de sous-paquetages sont séparés par un point. Par exemple `java.lang`.

Chaque mot constitutif du nom d'une classe ou d'une interface s'écrit en commençant par une lettre majuscule suivie de lettres minuscules. Par exemple : `UneClasse`.

Le nom d'un constructeur est identique à celui de la classe qui le contient (imposé par le langage).

Le nom d'une méthode ou d'une variable se construit ainsi : le premier mot s'écrit tout en minuscules, les suivants commencent par une majuscule, suivie de lettres minuscules. Par exemple : `uneMethode`.

Les constantes de classe s'écrivent tout en lettres majuscules, les mots constitutifs étant séparés par le caractère souligné. Par exemple : `UNE_CONSTANTE`.

Les identificateurs ne doivent pas commencer par les caractères `$` ni `_`, ils doivent être formés de noms significatifs, sur plusieurs lettres. On évitera les lettres accentuées, on s'efforcera de trouver des identificateurs en anglais.
Lors de l'utilisation de variables à très courte portée (exemple typique variable de contrôle d'une boucle : `int i`) on utilisera des variables "Kleenex" (vite utilisées, puis jetées) :

- une seule lettre ;
- jamais la lettre `l` (car elle ressemble trop au chiffre `1`).

18.7) Bonnes habitudes

L'accès aux caractéristiques statiques doit se faire par la classe et non par les instances (même si le langage ne l'impose pas).

Toute valeur littérale ayant une signification particulière doit être utilisée sous la forme d'une constante nommée (par exemple définir une constante `pi` plutôt que d'utiliser la valeur `3.14`).

Ne faire qu'une seule affectation par instruction (même si le langage autorise de coder l'instruction `x = y = z;`).

Ne pas faire d'affectation dans une expression.

Chapitre 4 : Du logiciel de qualité

1) Un exemple de type de données abstrait

On veut modéliser des accumulateurs d'électricité, caractérisés par leur capacité (la quantité maximale d'électricité qu'ils peuvent emmagasiner), leur charge (la quantité d'électricité qu'ils contiennent à un instant donné) et leur état de fonctionnement (« en service » ou « hors service » selon qu'ils sont fonctionnels ou pas).
De plus, on veut pouvoir modéliser le fait de fabriquer des accumulateurs non chargés mais dotés d'une certaine capacité, que l'on pourra ensuite charger ou décharger à volonté.
Un accumulateur devient hors service lorsqu'on tente de le charger au-delà de sa capacité. Un accumulateur hors service perd sa charge, et on ne peut plus alors ni le charger ni le décharger.

Pour représenter l'évolution des accumulateurs au cours du temps, on cherchera en fait à modéliser l'ensemble (A) de toutes les configurations d'accumulateur d'électricité possibles ainsi que des opérations sur A permettant de décrire ou d'obtenir les éléments de A.

A est un ensemble de configurations qui est en bijection avec l'ensemble (infini, dénombrable) $\{ (i, j) \in \mathbb{N}^2 \mid i \leq j \}$, il est donc bien défini.
Par abus de langage je dirai aussi bien accumulateur (un objet du monde réel) que configuration d'accumulateur (son modèle) ou accu (abréviation pour l'un ou l'autre terme).

Un générateur de A est une opération à valeur dans A, permettant d'obtenir des accus. On définit les trois générateurs suivants (pour tout entier q non nul et tout accu a, en service, de charge c, et de capacité k) :

- `créer(k)` : opération qui donne un accu en service, de charge vide et de capacité k ;
- `charger(a, q)` : opération qui donne un accu de capacité k, de charge c + q et en service si c + q ≤ k, ou de charge 0 et hors service sinon ;
- `décharger(a)` : opération qui donne un accu en service, de charge c - 1 et de capacité k.

Une requête de A est une opération qui n'est pas à valeur dans A, permettant de décrire les accus. Si a est dans A, on définit les trois requêtes suivantes :

- `charge(a)` : opération à valeur dans ℕ pour décrire la charge de a ;
- `capacité(a)` : opération à valeur dans ℕ pour décrire la capacité (charge maximale) de a ;
- `hs(a)` : opération à valeur dans B pour décrire l'état de service de a.

Définition de la syntaxe des six opérations précédentes :

`créer`	: ℕ	---> A
`charger`	: A × ℕ*	-/-> A
`décharger`	: A	-/-> A
`charge`	: A	---> ℕ
`capacité`	: A	---> ℕ

`hs` : A ---> B
où l'on note B l'ensemble des booléens, ℕ l'ensemble des entiers naturels et ℕ* = ℕ - {0}.
Les opérations partielles sont représentées avec des flèches barrées, les opérations totales avec des flèches non barrées.
Une opération partielle est une opération qui n'est pas définie sur la totalité de son ensemble de départ. Ici, `charger` et `décharger` sont partielles car elles ne sont définies que lorsque leur argument `a` (un élément de A) est tel que `¬hs(a)`.

Les requêtes permettent de définir l'action des générateurs : elles permettent de distinguer, par les valeurs qu'elles retournent, les différentes configurations d'accumulateurs obtenues à l'aide des générateurs.
Par exemple, les trois requêtes précédentes permettent de décrire l'opération de création d'un accu :

- `charge(créer(k)) = 0`
- `capacité(créer(k)) = k`
- `hs(créer(k)) = F`

créer un accumulateur de capacité k, c'est donc obtenir l'élément de A qui est en service, dont la charge est nulle et dont la capacité vaut k.

Donner la spécification des accus, c'est définir formellement la syntaxe des opérations disponibles (rubrique Opérations ci-dessous) ainsi que leur sémantique (rubriques Préconditions et Axiomes ci-dessous).
Voici un exemple de spécification possible pour le type de données abstrait ACCU :

```
TDA ACCU :
     A ensemble des accus
     ℕ ensemble des naturels positifs
     B ensemble des booléens

Opérations :
     capacité  : A       ---> ℕ
     charge    : A       ---> ℕ
     hs        : A       ---> B
     créer     : ℕ       ---> A
     charger   : A × ℕ*  -/-> A
     décharger : A       -/-> A

Préconditions : pour tout a∈A, n∈ℕ
     pre-charger(a, n)   ::= ¬hs(a)
     pre-décharger(a)    ::= ¬hs(a)

Axiomes : pour tout a∈A, k∈ℕ, i∈ℕ*
A1  : charge(créer(k)) = 0
A2  : capacité(créer(k)) = k
A3  : ¬hs(créer(k))
A4  : charge(décharger(a)) = max(charge(a) - 1, 0)
A5  : capacité(décharger(a)) = capacité(a)
A6  : ¬hs(décharger(a))
A7  : charge(a) + i ≤ capacité(a)
```

```
          ⇒ charge(charger(a, i)) = charge(a) + i
A8  : charge(a) + i > capacité(a) ⇒ charge(charger(a, i)) = 0
A9  : capacité(charger(a, i)) = capacité(a)
A10 : hs(charger(a, i)) = (charge(a) + i > capacité(a))
```

Les préconditions expriment quelles sont les valeurs des arguments qui sont utilisables avec une opération donnée. Par exemple, la deuxième précondition indique que pour avoir le droit d'écrire `décharger(a)`, il est nécessaire que `a` soit en service. Si ce n'est pas le cas on n'a pas le droit d'écrire l'expression `décharger(a)` ; dit autrement : `décharger(a)` n'est plus une expression correcte quand `a` est hors-service.

Les axiomes doivent se lire en supposant que les préconditions sont (implicitement) vérifiées. Par exemple l'axiome `A6` suppose que `a` est en service (précondition de `décharger`) de sorte que l'on a bien le droit d'écrire l'expression `décharger(a)` dans cet axiome.

2) Présentation des TDA

Un type de données abstrait (TDA) est un ensemble d'éléments, muni d'opérations agissant sur ces éléments.
Quelques remarques :

1. l'ensemble sous-jacent à un TDA est bien un ensemble au sens mathématique (éléments distincts, non modifiables) ;
2. cet ensemble peut être infini ;
3. les opérations sur cet ensemble sont des fonctions (au sens mathématique) qui s'appliquent sur une partie de A (ou, plus généralement, sur une partie d'un produit cartésien d'ensembles contenant A comme facteur), et/ou qui sont à valeur dans A.

Le TDA ACCU représente toutes les configurations d'accumulateurs électriques possibles et définit les opérations sur ces configurations. Il ne décrit pas la manière dont sont implantés les accumulateurs électriques.
La théorie des TDA constitue un cadre formel permettant la construction raisonnée des programmes qui manipulent des accumulateurs électriques.

Un TDA définit sans ambiguïté le comportement de ses opérations ainsi que les valeurs sur lesquelles elles opèrent.
En revanche, il ne décrit absolument pas la manière dont ces opérations se réaliseront (leur code) ni la manière dont ces valeurs seront représentées en mémoire.

2.1) Taxonomie des opérations d'un TDA

Les opérations d'un TDA peuvent être :

1. des fonctions totales : fonctions dont le domaine de définition correspond à la totalité de l'ensemble de départ (ex. : `créer`, `hs`, `capacité`, `charge`)
2. ou des fonctions partielles : fonctions dont le domaine de définition correspond à une partie propre de l'ensemble de départ (ex. : `charger`, `décharger`)

Elles peuvent aussi être :

1. des générateurs : fonctions à valeur dans l'ensemble sous-jacent (ex. : `créer`, `charger`, `décharger`)

2. ou des requêtes : fonctions à valeur dans un autre ensemble que l'ensemble sous-jacent (ex. : `hs`, `capacité`, `charge`).

On peut classer les opérations d'un TDA en trois catégories disjointes, selon la position du nom de l'ensemble sous-jacent (A) par rapport à la flèche de l'opération :

1. requêtes : opérations définies sur A et à valeur dans un autre ensemble que A ; A n'apparaît qu'**à gauche** de la flèche (`charge : A ---> ℕ`) ;
2. créateurs : générateurs (donc à valeur dans A) constants ou définis sur un autre ensemble que A ; A n'apparaît qu'**à droite** de la flèche (`créer : ℕ ---> A`) ;
3. commandes : générateurs (donc à valeur dans A) définis sur un ensemble produit dont l'un des facteurs est A ; A apparaît **à gauche** et **à droite** de la flèche (`charger : A × ℕ* -/-> A`).

2.2) Séparation commandes/requêtes

Dans notre vision impérative des TDA, les requêtes retournent une information (par mémorisation préalable ou par calcul) sans modifier l'état de l'objet cible (du moins son état observable) et les commandes modifient l'état sans retourner d'information. C'est ce que l'on appelle le principe de séparation commandes/requêtes.
L'application de ce principe nous autorise à observer l'effet d'une commande par appel aux requêtes autant de fois que nous le souhaitons, et aussi (et surtout) à tester que nos objets sont bien dans un état donné, ce qui est fondamental pour pouvoir faire de la programmation par contrats.

3) Présentation de la programmation par contrats

On implante un TDA dans un langage de programmation en lui faisant correspondre un type de données. À ce stade, n'ayant défini que les classes comme types de données[11], nous ne décrivons l'implantation d'un TDA qu'à travers ces dernières.

3.1) Correspondance classe - TDA

On peut faire correspondre une classe à un TDA de la manière suivante :

- en définissant dans la classe une méthode ou un constructeur public pour chaque opération du TDA ;
- en introduisant dans la classe les spécifications du TDA sous forme de commentaires de documentation ;
- en faisant en sorte que le code des méthodes et constructeurs de la classe respecte les spécifications ainsi introduites.

On dit alors que la classe réalise le TDA. Ce faisant, une instance de la classe est une réalisation possible pour un élément du TDA, et l'appel d'une méthode publique sur cette instance réalise l'application d'une opération du TDA avec cet élément en paramètre.

Mais il faut bien comprendre qu'un élément du TDA ne représente qu'une configuration possible d'un objet, pas l'objet lui-même. Ainsi un élément du TDA peut correspondre à une certaine instance de la classe à un instant t1, puis après modification de l'objet à un instant t2 ne plus lui correspondre du tout : l'objet a changé de configuration, cet objet a évolué dans le temps, passant d'une configuration à une autre.
De même, un élément du TDA peut correspondre à plusieurs objets en même temps (s'ils

11 Il s'agit en fait de la notion de classe concrète. Nous verrons au chapitre 5 qu'il existe aussi en Java des classes dites abstraites, ainsi que des interfaces...

sont dans le même état observable).

3.2) Fonction d'abstraction

On définit une fonction particulière entre une classe et le TDA qu'elle réalise. Cette fonction permet d'associer à une instance de la classe l'élément qui lui correspond naturellement dans le TDA à un instant donné (sa configuration). Cette fonction est appelée fonction d'abstraction.
Étant donnés un TDA T et une classe C qui le réalise, la fonction d'abstraction $F_{C,T}$ est une fonction de C vers T qui fait correspondre aux instances de C les éléments de T que ces instances réalisent, lorsque c'est possible.
On remarquera que cette fonction peut être partielle (elle peut ne pas être définie pour chaque instance de C), ou totale quand C est correcte. En revanche, elle est rarement injective (*a priori*, deux instances différentes peuvent avoir la même image).

3.3) Invariant de représentation

L'invariant de représentation est une conjonction d'expressions booléennes qui caractérisent les valeurs autorisées pour l'état interne des instances de la classe. De ce fait, il représente la fonction caractéristique du domaine de définition de $F_{C,T}$ et permet de définir celles, parmi les instances de la classes, qui correspondent bien à des éléments du TDA.

3.4) Correspondance méthode - opération

Lorsqu'une classe X réalise un TDA, à chaque opération du TDA il doit correspondre une méthode ou un constructeur publics dans la classe X.
Dans ce qui suit, on notera :

- A l'ensemble sous-jacent au TDA réalisé par la classe Java X ;
- E un ensemble quelconque sous-jacent à un autre TDA réalisé par une classe Java Y.

1. À un créateur du TDA (`cr : E1×...×En ---> A`) correspond un constructeur public de la classe (`public X(Y1, ..., Yn)`).
2. À une requête du TDA (`req : A×E1×...×En ---> E`) correspond une requête publique de la classe (`public Y req(Y1, ..., Yn)`).
3. À une commande du TDA (`com : A×E1×...×En ---> A`) correspond une commande publique de la classe :
    1. commande mutative (méthode qui ne retourne pas de valeur mais qui modifie l'instance courante) : `public void com(Y1, ..., Yn)`
    2. commande constructive (méthode qui retourne une nouvelle valeur de type X) : `public X com(Y1, ..., Yn)`

3.5) Programmation par contrat

Une spécification est un énoncé qui définit sans ambiguïté une partie du comportement ou de l'état des instances d'une classe.
Le contrat d'une méthode (ou d'un constructeur) est la spécification associée à une méthode (ou un constructeur), qui en définit les conditions de bonne utilisation ainsi que l'effet qu'elle (ou qu'il) produit.
L'invariant d'un type (classe ou interface) est une spécification qui décrit “ce qui est toujours vrai” par rapport à l'état (observable) de ses instances.

Le contrat d'une classe (ou d'une interface) est alors l'ensemble de toutes les spécifications qui interviennent dans ce type (contrats des méthodes et constructeurs + invariant de type).

Exemple de contrat pour la méthode `push` sur les piles bornées.
On suppose donnés un type de piles bornées et les méthodes `push`, `full`, `empty`, `size` et `top` dans ce type (pour empiler, détecter si la pile est pleine, si elle est vide, obtenir la taille de la pile et consulter le sommet). On s'intéresse au contrat de la méthode `void push(Element e)`. Ce contrat est constitué de deux parties :

- une précondition qui indique que, avant d'exécuter la méthode, la pile n'est pas pleine : `!full()`
- une postcondition qui indique que, après avoir exécuté la méthode, la pile contient un élément de plus et que le sommet de la pile est cet élément : `size() == old size() + 1` et `top() == e`

Extrait d'invariant pour la classe des piles bornées, la partie décrivant le fait qu'une pile vide est de taille nulle et vice-versa : `empty() <==> (size() == 0)`.

Le contrat d'une méthode est défini par une précondition et une postcondition. Ce sont des assertions qui expriment les contraintes qui doivent être vérifiées pour que la méthode s'exécute correctement.
La précondition est un ensemble de contraintes qui portent sur l'état du système (objet cible, paramètres, environnement) juste avant l'appel de la méthode, c'est une contrainte pour le client (utilisateur) qui doit placer le système dans un état vérifiant cette précondition avant d'appeler la méthode.
La postcondition est un ensemble de contraintes qui portent sur l'état du système juste avant le retour d'appel de la méthode, c'est une contrainte pour le fournisseur (développeur) qui doit placer le système dans un état vérifiant la postcondition avant de retourner.
Lors de l'appel de la méthode, le fournisseur fera confiance au client (par rapport aux préconditions) ; lors du retour de l'appel, c'est le client qui fera confiance au fournisseur (par rapport aux postconditions).

L'état observable d'un objet est l'ensemble des valeurs qui caractérisent l'élément du TDA qui est associé à l'objet par la fonction d'abstraction. En pratique, l'état observable d'un objet est la séquence des valeurs que retournent ses requêtes, prises dans un ordre arbitraire mais fixé une fois pour toutes.

Nous dirons d'un objet qu'il est en phase stable lorsqu'il est inactif, après sa création et entre deux appels de commandes publiques. Cette phase est caractérisée par le fait que si l'on observe répétitivement l'objet, son état observable ne change pas.
Notez bien que l'état interne d'un objet ne correspond pas nécessairement à son état observable.
L'invariant de type est une spécification qui décrit des relations sémantiques entre requêtes pour une instance quelconque de la classe lorsqu'elle est en phase stable. Dit autrement, l'invariant de type décrit les propriétés immuables de l'état observable des instances de ce type en phase stable.

Comme on le verra plus loin à la section 10 de ce chapitre, il est utile de partitionner l'ensemble des requêtes en deux : un ensemble dit de base, qui constitue un plus petit sous-ensemble des requêtes d'une classe permettant de reconstruire l'état observable de ses instances, et un ensemble dit dérivé, qui contient le reste des requêtes de la classe. Bien qu'artificielle, cette séparation simplifie la tâche de spécification d'un TDA en diminuant le nombre d'axiomes.

Par ailleurs on constate que, bien souvent, les requêtes de l'ensemble de base retournent la valeur d'un attribut alors que celles de l'ensemble dérivé retournent la valeur d'un calcul. Ce constat n'est toutefois pas universel. Par exemple, dans une classe de piles, dotée seulement des requêtes `top` et `isEmpty`, ces requêtes constituent précisément l'ensemble de base et l'ensemble dérivé est vide. Mais dans une classe de piles plus complète, dotée des requêtes `top`, `isEmpty`, `size` et `elem`, l'ensemble de base sera constitué de `size` et d'`elem`, les deux autres constituant l'ensemble dérivé.

3.6) Notation des contrats en Java

En Java, on note chaque contrat à l'intérieur du commentaire de documentation associé à la méthode qu'il spécifie :

- précondition : conjonction d'expressions booléennes basées sur l'état de l'objet courant et les arguments de la méthode avant exécution de la méthode ;
- postcondition : conjonction d'expressions booléennes basées sur l'état de l'objet courant et les arguments de la méthode après exécution de la méthode.

Exemple :

```
/**
 * Ajoute un élément sur la pile.
 * @pre
 *      !full()
 * @post
 *      size() == old size() + 1
 *      top() == e
 */
public void push(Element e) { ... }
```

Dans la postcondition d'une méthode, on utilise le pseudo-opérateur unaire `old` pour décrire a posteriori la valeur d'une expression Java calculée avant l'appel de la méthode. L'utilisation de `old` permet, après avoir exécuté une méthode, de connaître la valeur qu'avait une expression Java avant cette exécution. Attention le pseudo-opérateur `old` n'existe pas en Java, il n'est utilisé qu'au niveau des spécifications, dans la postcondition des méthodes et a la précédence maximale d'un opérateur unaire par rapport aux (vrais) opérateurs de Java.

Dans la postcondition d'une méthode, on peut aussi utiliser la pseudo-variable `result` (n'existe pas en Java) pour désigner la valeur retournée par la méthode.
Exemple :

```
/**
 * @pre
```

```
 *      p != null
 * @post
 *      forall i >= 0 :
 *          result.coeff(i) == this.coeff(i) + p.coeff(i)
 */
public Polynom add(Polynom p) { ... }
```

En Java, on note l'invariant de type à l'intérieur du commentaire de documentation réservé à la classe ou à l'interface.
Exemple :

```
/**
 * @inv
 *      side() >= 0
 *      perimeter() == 4 * side()
 *      ...
 */
class Square {
    private int side;
    ...
    public int side() { return side; }
    public int perimeter() { return side * 4; }
    public void setSide(int s) { ...side = s;... }
    ...
}
```

4) Encapsulation et modules

4.1) Effet de bord

L'effet de bord d'une action est la modification de l'état d'un objet suite à l'exécution de cette action par l'objet. Un effet de bord est observable lorsqu'on peut le décrire à l'aide des valeurs retournées par les requêtes que peut exécuter l'objet (c'est-à-dire lorsqu'il modifie l'état observable).

De même que la logique de Hoare pose comme axiome de base que l'évaluation d'une expression ne doit pas modifier l'état des variables, nous posons comme règle de spécification que seules les expressions sans effet de bord observable seront acceptées dans une spécification.
Il en découle que les requêtes ne devraient pas provoquer des effets de bord observables. Mais vous noterez que les effets de bord non observables sont acceptés. Il s'agit d'un léger assouplissement du principe de séparation commandes/requêtes.

4.2) Principe d'encapsulation, module

Le principe d'encapsulation des données s'énonce ainsi :

Seul l'objet a le droit de modifier ses propres données.

Un élément logiciel qui met en œuvre ce principe doit donc masquer sa structure interne à ses utilisateurs. Un module est, par définition, un élément logiciel qui met en œuvre

l'encapsulation des données.
Un objet dont la classe génératrice est un module peut ainsi être vu comme une boîte noire proposant des fonctionnalités connues (services), mais dont l'implémentation est inconnue de ses clients (on connait le quoi, mais pas le comment).
Java permet de travailler avec des modules (classes, paquetages), mais attention, une classe dont les attributs sont publics (en tout cas publics et variables) ne peut pas être considérée comme un module !

4.3) Règles d'accessibilité en Java

En Java, nous avons vu qu'il était possible de restreindre à certaines portions de code l'accès aux caractéristiques ou aux constructeurs. Il est temps de définir précisément les règles qui régissent cette accessibilité.

L'accessibilité d'un type est définie ainsi :
- si le type est déclaré avec le modificateur **`public`**, il est accessible en tout point du code source, où qu'il se trouve ;
- si le type est déclaré sans ce modificateur, il n'est accessible que dans le code source des types situés dans le même paquetage que lui.

L'accessibilité d'un élément déclaré dans un type *T* s'exprime alors de la manière suivante :
- si l'élément est déclaré avec le modificateur **`public`**, il est accessible uniquement depuis tout point *e* pour lequel *T* est un type accessible ;
- si l'élément est déclaré sans modificateur d'accessibilité, il est accessible uniquement depuis tout point *e* situé dans le (texte de tout type du) même paquetage que *T* ;
- si l'élément est déclaré avec le modificateur **`private`**, il est accessible uniquement depuis tout point *e* situé dans (le texte de) *T*.

Il reste encore à définir la règle d'accessibilité pour l'emploi du modificateur **`protected`**, c'est ce que nous verrons au chapitre 5.

5) Classe correcte

5.1) Règle de conception numéro 1

Toute classe doit être spécifiée par un TDA et en proposer une réalisation.

Concrètement, le texte d'une classe est séparé en deux parties :
- une spécification, sur laquelle se basera le code client pour utiliser les instances de la classe ;
- une implémentation, partie totalement ignorée par le code client.

La première partie contient les entêtes des méthodes publiques (qui correspondent aux opérations du TDA) ainsi que le contrat de la classe. La seconde partie contient toutes les structures de données internes, le corps des méthodes publiques et les méthodes auxiliaires nécessaires à la factorisation et à la modularisation du code.

5.2) Règle de conception numéro 2

Toute classe doit être conçue comme un module

Par conséquent :
- les structures de données propres aux objets doivent être inaccessibles aux clients ;
- toute modification de l'état d'un objet doit passer par l'appel d'une commande.

5.3) Correction d'une classe

Une classe correcte est une classe dont chaque instance respecte le contrat de la classe. Dit autrement, les instances d'une classe correcte évoluent selon les schémas pré-post suivants :
- $\{pre_{cons}\}$ exécution de constructeur $\{inv \wedge post_{cons}\}$
- $\{inv \wedge pre_{meth}\}$ exécution de méthode publique $\{inv \wedge post_{meth}\}$
- $\{pre_{meth}\}$ exécution de méthode privée $\{post_{meth}\}$

Tout non respect de la spécification est un beug.
Le non respect d'une précondition est un beug dans le code du client, alors que le non respect d'une postcondition ou d'un invariant est un beug dans le code du fournisseur.
Lorsqu'un beug survient :
- le système doit interrompre le processus et rapporter une erreur ;
- le responsable de l'erreur doit la corriger.

6) Implanter un TDA en Java

Comme on l'a vu, les notions de spécification et de module sont centrales dans la construction du logiciel objet fiable. Une classe ne peut être correcte que par rapport à une spécification, elle doit être spécifiée par un contrat obtenu à partir d'un TDA (règle de conception 1), et une classe spécifiée n'est correcte que si elle respecte son contrat, ce qui ne peut être assuré que si elle encapsule ses SdD internes et sa « cuisine fonctionnelle » (règle de conception 2).

Lorsqu'on implante un TDA en Java, il faut traduire les spécifications du TDA en spécifications du logiciel :
- les préconditions du TDA deviennent les préconditions des méthodes et constructeurs correspondants ;
- les axiomes exprimant les valeurs des requêtes de l'ensemble dérivé par rapport aux requêtes de l'ensemble de base deviennent l'invariant ;
- les axiomes exprimant les valeurs des requêtes de l'ensemble de base sur les générateurs deviennent les postconditions des constructeurs et des commandes correspondants.

Si nécessaire, rajouter en plus :
- dans l'invariant : des restrictions sur l'ensemble des valeurs possiblement retournées par les requêtes ;
- dans les préconditions : des restrictions sur l'ensemble des valeurs que peuvent prendre les arguments.

Concrètement, dans le code, on va retrouver systématiquement toutes les spécifications sous forme de commentaires de documentation (elles seront donc non effectives), et une partie seulement de ces spécifications sera intégrée sous forme de tests directement dans le code opérationnel (celles-ci seront donc effectives) :

- les préconditions doivent toujours se trouver dans la documentation **et** être codées ;
- les postconditions et les invariants doivent toujours se trouver dans la documentation.

Pour déboguer, on peut intégrer au code :

- le moyen de tester les postconditions ;
- le moyen de tester les invariants (de classe et de représentation) ;
- le moyen de calculer la fonction d'abstraction.

Bien entendu, pour que le code soit efficace il vaut mieux ne vérifier qu'un minimum de spécifications, c'est pourquoi ces dernières n'apparaissent que peu dans la partie opérationnelle du code.
Mais s'il est possible d'éliminer les vérifications de postconditions et d'invariants dès lors que le niveau de confiance en la qualité de notre code est suffisamment élevé, il n'en va pas de même pour les préconditions. Pourquoi ? Parce qu'une précondition indique que l'opération est une fonction partielle du TDA, son domaine de définition est donc une restriction de l'ensemble de départ. Or le compilateur accepte toutes les valeurs de l'ensemble de départ et on ne sera jamais certain que le client n'utilise que des valeurs du domaine de définition. Il est donc nécessaire d'inclure les tests de validité correspondants dans le code.

6.1) Codage des préconditions

6.1.1) Expression d'assertion

Une expression d'assertion s'écrit : **`assert`** `<condition>` **`[`**`: <description>`**`]`**
Après évaluation de `<condition>`, elle ne fait rien si le résultat vaut **`true`** ; sinon elle évalue `<description>` et passe la valeur obtenue au constructeur d'une `AssertionError` qui est aussitôt levée.
Le compilateur Java inclut toujours les assertions dans le code compilé, mais par défaut la machine Java ne les exécute pas. Si l'on veut modifier ce comportement (notamment pour le débogage) sous BlueJ, il faut modifier le fichier `bluej.properties` en y rajoutant la ligne suivante : `bluej.vm.args = -ea`

6.1.2) Cas des méthodes

Règle de codage des préconditions numéro 1 :

Une précondition doit être codée au tout début du corps de la méthode par une levée gardée d'AssertionError

Par exemple :

```
/**
 * Calcule la racine carrée de son argument.
 * @pre
 *      x >= 0
 * @post
 *      result >= 0
 *      |result * result - x| <= DELTA
 */
public double sqrt(double x) {
```

```
    if (x < 0) {
        throw new AssertionError("Argument négatif " + x);
    }
    ...
}
```

Règle de codage des préconditions numéro 2 :

Si le client est différent du fournisseur, la précondition doit être codée à l'aide d'une instruction conditionnelle

Si le client et le fournisseur sont les mêmes, la précondition doit être codée à l'aide d'une instruction d'assertion

Par exemple :

```
/**
 * @pre
 *      x >= 0
 * @post ...
 */
private double sqrt(double x) {
    assert x >= 0 : "Arg. négatif " + x;
    ...
}
```

Justification de la deuxième règle :

D'un côté, évaluer la précondition protège le client de la méthode contre lui-même. De l'autre, ne pas évaluer la précondition permet de ne pas impacter l'efficacité du logiciel. Nous devons donc choisir, entre deux intérêts antagonistes, celui qu'il faut privilégier.

Lorsque le client d'une méthode et son fournisseur sont la même personne à tous les stades de développement du logiciel, la bonne attitude sera d'activer le test de la précondition au cours du développement (pour protéger le développeur de lui-même) et de le désactiver en phase de production (quand le logiciel s'exécute sur la machine du client) lorsqu'on a atteint un niveau de confiance suffisant dans le code fourni. Pour cela on utilisera une instruction d'assertion, qui peut être activée ou désactivée selon les besoins.

En revanche, lorsque le client et le fournisseur d'une méthode peuvent être des personnes différentes, l'optimisation précédente (utiliser une assertion pour la désactiver en phase de production) ne doit pas être mise en place afin de continuer à protéger le client. Il est donc important de tester la précondition à l'aide d'une instruction conditionnelle.

Enfin, il faut remarquer qu'une méthode privée garantit que le client et le fournisseur sont une même personne car elle n'est appelée que dans le code de la classe qui la définit. De plus, une méthode sans modificateur d'accès offre une garantie équivalente au niveau du paquetage, on peut donc considérer là encore que client et fournisseur ne font qu'un. Par contre, une méthode publique ou protégée[12] peut toujours être appelée directement par un client différent du fournisseur.

12 Nous reviendrons sur ce modificateur d'accès au chapitre 5.

6.1.3) Cas des constructeurs

Le codage des préconditions doit parfois être adapté pour les constructeurs de par leur syntaxe, différente de celle des méthodes. Nous verrons que la première instruction d'un constructeur c1 ne peut *qu'être* un appel à un autre constructeur c2 (chapitre 5 sur l'héritage). De ce fait il n'est pas possible d'effectuer des calculs dans c1 avant d'appeler c2. Les calculs nécessaires pour les paramètres de c2 doivent être effectués dans des méthodes privées et statiques (une méthode pour chaque paramètre de c2 qui le nécessite).

Quand les deux règles précédentes de codage des préconditions ne peuvent pas s'appliquer pour un constructeur, on peut toujours appliquer la règle de codage des préconditions suivante :

Dans un constructeur c1 doté d'une précondition non triviale et faisant un appel explicite à un constructeur c2 doté d'au moins un argument, il faut créer une méthode privée et statique pour le premier argument de c2, qui teste la validité de la précondition de c1 avant de retourner la valeur adéquate pour le premier argument de c2.

6.2) Codage des postconditions

Une postcondition peut être codée à la fin du corps d'une méthode à l'aide d'assertions

```
/**
 * Calcule la racine carrée de son argument.
 * @pre
 *      x >= 0
 * @post
 *      result >= 0
 *      |result * result - x| <= DELTA
 */
public double sqrt(double x) {
    // vérification de la précondition
    ...
    // calcul de la racine carrée de x dans r
    double r;
    ...
    assert r >= 0
        : "Résultat négatif " + r;
    assert Math.abs(r * r - x) <= DELTA
        : "Résultat incorrect " + (r * r) + " != " + x;
    return r;
}
```

Attention, vérifier une postcondition peut être (très, très) coûteux. Le code vérificateur ne doit pas être exécuté en phase de production (quand la classe est réputée correcte) et c'est pourquoi on utilise une instruction d'assertion. En général, on ne teste pas les postconditions directement dans le logiciel...

6.3) Codage des invariants

On peut fournir une méthode boolean inv() qui teste la validité de l'invariant de type.

```
// Invariant des estomacs
public boolean inv() {
    return
        size() > 0
        && foodQuantity() >= 0
        && isEmpty() == (foodQuantity() == 0)
        && isExploded() == (foodQuantity() > size());
}
```

En général, on ne teste pas les invariants directement dans le logiciel...

6.4) Représentation textuelle des éléments du TDA

On peut utiliser la méthode `String toString()` pour décrire, sous forme de chaîne de caractères, l'image d'une instance par la fonction d'abstraction.

```
// Dans la classe Pile
public String toString() {
    String result = "Pile[";
    if (size > 0) {
        result += data[0];
        for (int i = 1; i < size; i++) {
            result += ";" + data[i];
        }
    }
    result += "]";
    return result;
}
```

6.5) Équivalence de deux instances

Deux instances d'une même classe sont considérées équivalentes lorsqu'elles ont la même image par la fonction d'abstraction (donc lorsqu'elles correspondent au même état observable)

```
// dans la classe Pile
public boolean equals(Object o) {
    if (o == null || o.getClass() != this.getClass()) {
        return false;
    }
    Pile s = (Pile) o;
    boolean equiv = (this.size == s.size);
    int i = 0;
    while (equiv && i < size) {
        equiv = (this.data[i] == s.data[i]);
        i += 1;
    }
    return equiv;
}
```

6.6) Codage de l'invariant de représentation

On peut aussi fournir une méthode **`boolean`** `repInv()` qui teste la validité de l'invariant de représentation (ce qui permet de détecter les monstres !).

```
// Invariant de représentation des piles
public boolean repInv() {
    return
        size >= 0
        && data != null
        && size <= data.length;
}
```

7) Exemples d'implantations de TDA en Java

7.1) Différents niveaux d'implantation

Nous avons vu à la section 3 de ce chapitre comment implanter un TDA à l'aide d'une classe (concrète). En réalité, lorsqu'on veut implanter un TDA en Java, nous pouvons choisir entre trois niveaux de réalisation :

- **`interface`** : description totalement abstraite d'un TDA en Java (spécification sans aucune réalisation)
- **`abstract class`** : description abstraite d'un TDA en Java (spécification + réalisation partielle)
- **`class`** : description concrète d'un TDA en Java (spécification + réalisation totale)

7.2) Implantation par interface + classe

Dans ce premier exemple, on implante le TDA ESTOMAC dans une interface `IStomach`, puis on implémente l'interface à l'aide d'une classe `Stomach`.

```
TDA ESTOMAC(ℕ, B)

Opérations :
  – créer     : ℕ*      ---> S
  – quantité  : S       ---> ℕ
  – taille    : S       ---> ℕ*
  – vide      : S       ---> B
  – explosé   : S       ---> B
  – remplir   : S × ℕ*  -/-> S
  – vivre     : S       -/-> S

Préconditions : pour tout s∈S, q∈ℕ
P1 : pre-remplir(s, q) : ¬explosé(s)
P2 : pre-vivre(s) : ¬explosé(s) ∧ ¬vide(s)

Axiomes : pour tout s∈S, n,q∈ℕ*
A1 : vide(s) = (quantité(s) = 0)
A2 : explosé(s) = (quantité(s) > taille(s))
A3 : quantité(créer(n)) = n
```

```
A4 : taille(créer(n)) = n
A5 : quantité(remplir(s, q)) = quantité(s) + q
A6 : taille(remplir(s, q)) = taille(s)
A7 : quantité(vivre(s)) = quantité(s) - 1
A8 : taille(vivre(s)) = taille(s)
```

```
/**
 * Spécifie le TDA des estomacs.
 *
 * @inv
 *      size() > 0
 *      foodQuantity() >= 0
 *      isEmpty() == (foodQuantity() == 0)
 *      isExploded() == (foodQuantity() > size())
 *
 * @cons
 *      $DESC$ Un estomac plein, de taille t.
 *      $ARGS$ int t
 *      $PRE$  t > 0
 *      $POST$
 *          foodQuantity() == t
 *          size() == t
 */
public interface IStomach {

    // REQUETES

    /**
     * Indique la quantité de nourriture actuellement dans cet
     *  estomac.
     */
    int foodQuantity();

    /**
     * Indique la taille de cet estomac, ce qui correspond à la
     *  quantité maximale de nourriture qu'il peut absorber.
     */
    int size();

    /**
     * Indique si cet estomac est vide.
     */
    boolean isEmpty();

    /**
     * Indique si cet estomac a été trop rempli.
     */
    boolean isExploded();
```

```
    // COMMANDES

    /**
     * Remplit cet estomac de quantity nourriture.
     * @pre
     *     quantity > 0
     *     !isExploded()
     * @post
     *     foodQuantity()
     *             == old foodQuantity()
     *                + quantity
     */
    void fill(int quantity);

    /**
     * Fait vivre cet estomac un instant.
     * @pre
     *     !isExploded() && !isEmpty()
     * @post
     *     foodQuantity()
     *                 == old foodQuantity() - 1
     */
    void spendLife();
}

public class Stomach implements IStomach {

    // ATTRIBUTS

    // Taille de cet estomac
    private final int size;

    // Quantité courante de nourriture dans cet estomac
    private int foodQty;

    // CONSTRUCTEURS

    public Stomach(int maxSize) {
        if (maxSize <= 0) {
            throw new AssertionError();
        }
        size = maxSize;
        foodQty = maxSize;
    }

    // REQUETES

    public int foodQuantity() {
        return foodQty;
```

```
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return (foodQty == 0);
    }

    public boolean isExploded() {
        return (foodQty > size);
    }

    // COMMANDES

    public void fill(int quantity) {
        if (quantity < 0) {
            throw new AssertionError();
        }
        if (isExploded()) {
            throw new AssertionError();
        }
        foodQty += quantity;
    }

    public void spendLife() {
        if (isExploded() || isEmpty()) {
            throw new AssertionError();
        }
        foodQty -= 1;
    }

    // OUTILS

    public String toString() {
        return "Stomach[foodQty:" + foodQty + ";size:"
                + size + "]";
    }

    public boolean inv() {
        return size() > 0 && foodQuantity() >= 0
                && isEmpty() == foodQuantity() == 0
                && isExploded() == foodQuantity() > size();
    }

    public boolean repInv() {
        return size > 0 && foodQty >= 0;
    }
```

```
}
```

7.3) Implantation par classe uniquement

Dans cet exemple, on implante le TDA PILE directement dans la classe qui le réalise.
À titre d'exemple pédagogique, on a inséré dans le code des méthodes les vérifications des postconditions et de l'invariant de type. Dans la réalité, cela est rarement le cas.

```
TDA PILE(B, E)

Opérations :
   – créer     :          ---> P
   – vide      : P        ---> B
   – sommet    : P        -/-> E
   – empiler   : P × E    ---> P
   – dépiler   : P        -/-> P

Préconditions : pour tout p∈P
P1 : pre-dépiler(p) : ¬vide(p)
P2 : pre-sommet(p) : ¬vide(p)

Axiomes : pour tout p∈P, x∈E
A1 : vide(créer())
A2 : sommet(empiler(p, x)) = x
A3 : ¬vide(empiler(p, x))
A4 : dépiler(empiler(p, x)) = p
```

```java
/**
 * Modélise le type des piles.
 */
public class Stack {

    // ATTRIBUTS

    // Les éléments de la pile
    private final ArrayList data;

    // La taille de la pile
    private int size;

    // CONSTRUCTEURS

    /**
     * Constructeur de pile vide.
     * @post
     *     isEmpty()
     */
    public Stack() {
        data = new ArrayList();
        size = 0;
```

```
        assert isEmpty();
    }

    /**
     * Constructeur duplicateur à usage interne.
     * @pre
     *      s != null
     * @post
     *      this.equals(s)
     */
    private Stack(Stack s) {
        assert s != null;

        size = s.size;
        data = new ArrayList(size);
        for (int i = 0; i < size; i++) {
            data.add(s.data(i));
        }

        assert this.equals(s);
    }

    // REQUETES

    /**
     * Indique si cette pile est vide.
     */
    public boolean isEmpty() {
        return size == 0;
    }

    /**
     * Le sommet de la pile.
     * @pre !isEmpty()
     */
    public int top() {
        if (isEmpty()) {
            throw new AssertionError();
        }

        return ((Integer) data.get(size - 1)).intValue();
    }

    // COMMANDES

    /**
     * Empile un élément.
     * @post
```

```
     *      !isEmpty()
     *      top() == n
     *      le nouvel état de la pile correspond à l'ancien état
     *          augmenté du nouveau sommet
     */
    public void push(int n) {
        Stack oldThis = null;
        assert (oldThis = new Stack(this)) != null;

        data.set(size, new Integer(n));
        size += 1;

        assert !isEmpty();
        assert top() == n;
        assert this.size == oldThis.size + 1;
        assert sameButTheTop(oldThis);
    }

    /**
     * Dépile un élément.
     * @pre
     *      !isEmpty()
     * @post
     *      le nouvel état de la pile correspond à l'ancien état
     *          privé de son sommet
     */
    public void pop() {
        if (isEmpty()) {
            throw new AssertionError();
        }
        Stack oldThis = null;
        assert (oldThis = new Stack(this)) != null;

        size -= 1;

        assert this.size == oldThis.size + 1;
        assert sameButTheTop(oldThis);
    }

    // OUTILS

    public String toString() {
        String result = "Stack[";
        if (size > 0) {
            result += data.get(0)
            for (int i = 1; i < size; i++) {
                result += "," + data.get(i);
            }
        }
```

```
        result += "]";
        return result;
    }

    public boolean repInv() {
        boolean result = (data != null && size >= 0
                && size <= data.size());
        if (result) {
            int i = 0;
            while (result && (i < size)) {
                result = (data.get(i) instanceof Integer);
                i += 1;
            }
        }
        return result;
    }

    public boolean inv() {
        return true; // il n'y a pas d'invariant
    }

    /**
     * Compare this et s, retourne true ssi la différence de
     *  taille entre elles est de 1, et que la plus grande des
     *  deux est équivalente à la plus petite plus un élément
     *  (au sommet).
     * @pre s != null
     */
    private boolean sameButTheTop(Stack s) {
        assert s != null;

        int minSize = Math.min(s.size, size);
        boolean result = (Math.max(s.size, size) - minSize == 1);
        int i = 0;
        while (result && i < minSize) {
            result = data.get(i).equals(s.data.get(i));
            i += 1;
        }
        return (i == minSize);
    }
}
```

8) Définition formelle des TDA

Une définition de TDA est un texte constitué de cinq sections :

- définition du nom de ce TDA ;
- déclaration des autres TDA utilisés par ce TDA ;
- définition des opérations de ce TDA ;
- définition des préconditions de ce TDA ;

- définition des axiomes de ce TDA.

La section « Opérations » donne la syntaxe des opérations, c'est-à-dire pour chaque opération son nom, son ensemble de départ et son ensemble d'arrivée.
Cette section définit les expressions bien formées du TDA (EBF) : les expressions correctement parenthésées, que l'on peut construire en utilisant les opérations du TDA appliquées à des arguments du bon type.
Exemples d'EBF :

- `créer(capacité(a))`
- `décharger(charger(a, capacité(a)+1))`

Exemples d'expressions mal formées :

- ~~`charge(hs(a))`~~
- ~~`décharger(a, 5)`~~

La section « Préconditions » donne le domaine de définition des opérations qui sont des fonctions partielles, c'est-à-dire pour chaque fonction partielle des formules qui décrivent la partie de l'ensemble de départ sur laquelle l'opération est bien définie.
Cette section définit les expressions correctes du TDA (EC) : les EBF telles que les arguments des fonctions partielles qui apparaissent dans ces EBF appartiennent au domaine de définition de ces fonctions. Par parenthèses, une EBF ne faisant pas intervenir de fonction partielle est une EC.
Exemples d'EC :

- `charger(a, capacité(a)+1)` si `¬hs(a)`
- `décharger(créer(n))`

Exemple d'EBF qui ne sont pas des EC :

- ~~`décharger(charger(a, capacité(a)+1))`~~

La section « Axiomes » donne la sémantique des opérations, c'est-à-dire l'ensemble des règles de réécriture des EC du TDA, qui permettent de déterminer une unique valeur (indépendante des éléments du TDA) pour toute expression correcte se terminant par une requête (ECTR).
Intuitivement, cette section permet de décrire, à l'aide des requêtes, les éléments du TDA que l'on obtient en appliquant les générateurs.
Pour des raisons de clarté dans l'énoncé des axiomes, on n'indique jamais le type des arguments utilisés (dans A4 de ACCU, le type de `a` est implicitement A) et on ne rappelle jamais les préconditions des opérations devant être vérifiées pour que les expressions soient correctes (dans A4 de ACCU, on suppose que `¬hs(a)`).

La section des axiomes est la plus délicate à construire car, tout en étant suffisamment complète, il faut qu'elle soit lisible et compréhensible.
Il est temps de remarquer que plusieurs expressions distinctes peuvent avoir la même valeur et qu'on peut donc les simplifier. Par exemple, pour les accus :

- `charger(charger(créer(5), 3), 5)` équivaut à `charger(créer(5), 8)`
- `décharger(créer(2))` équivaut à `créer(2)`
- `décharger(charger(créer(5), 3))` équivaut à `charger(créer(5), 2)`
- `décharger(décharger(charger(créer(3), 2)))` équivaut à `créer(3)`
- `charger(décharger(charger(créer(3), 2)), 10)` équivaut à `charger(créer(3), 4)`

Par conséquent, pour des raisons de simplicité évidente, on aura intérêt à trouver des axiomes les plus simples possibles.

On constate que l'on peut obtenir tous les éléments de A uniquement à l'aide des générateurs `créer` et `charger` : ils constituent ce que nous appellerons une base de générateurs.

```
A = {créer(0), charger(créer(0),1), charger(créer(0),2), ...,
créer(1), charger(créer(1),1), charger(créer(1),2), ...}
```

Une base de générateurs étant fixée, on peut alors définir un ensemble de formes dites canoniques (pour cette base) en choisissant une partie de l'ensemble des EC qui soit en bijection avec l'ensemble des éléments du TDA.
Par exemple l'ensemble des EC suivantes permet d'accéder, par évaluation, à tous les éléments de A, et ce de manière unique : cet ensemble d'EC est en bijection avec A.

```
(∀n ∈ ℕ) créer(n)
(∀n ∈ ℕ, ∀i : 1⩽i⩽n+1) charger(créer(n),i)
```

Mais notez bien que plusieurs bases sont possibles. Par exemple `créer`, `remplir`, `décharger` et `surcharger` constituent une deuxième base de générateurs :

```
A = {créer(0), surcharger(créer(0)), créer(1), remplir(créer(1)),
surcharger(créer(1)), créer(2), remplir(créer(2)),
décharger(remplir(créer(2))), surcharger(créer(2)), ...}
```

Notez aussi que, pour une même base de générateurs, plusieurs ensembles de formes canoniques sont possibles. Par exemple avec la première base (`créer`, `charger`), l'ensemble d'EC suivantes est un ensemble possible de formes canoniques :

```
(∀n ∈ ℕ) créer(n)
(∀n ∈ ℕ*, ∀i : 1⩽i<n) charger(créer(n),i)
(∀n ∈ ℕ) charger(créer(n),n+2)
```

On appelle ensemble de base de requêtes un ensemble minimal de requêtes qui permettent de distinguer chaque élément du TDA.
Pour le TDA ACCU, toutes les requêtes font parties de l'ensemble de base car :

- sans l'opération `hs`, on ne peut pas distinguer `charger(créer(n), n+1)` et `créer(n)`
- sans l'opération `charge`, on ne peut pas distinguer `charger(créer(n), i)` et `charger(créer(n), j)` pour `i ≠ j` (avec `i` et `j` inférieurs à `n`)
- sans l'opération `capacité`, on ne peut pas distinguer `créer(n)` et `créer(m)` pour `n ≠ m`

Un ensemble de base étant fixé parmi les requêtes, l'autre partie des requêtes constitue l'ensemble dérivé. Dans ACCU cet ensemble est vide, mais on aurait pu définir la requête `vide : A ---> B` par exemple (un accu est vide lorsque sa charge est nulle). Dans ce cas, `vide` aurait fait partie de l'ensemble dérivé car l'ensemble de requêtes qui contiendrait `vide` et `charge` ne serait pas minimal puisque `vide(a) = (charge(a) = 0)`.

Enfin, quand on a construit un système d'axiomes, on peut (au cas par cas) prouver qu'il constitue une spécification correcte. Par exemple, vous trouverez en section 10 l'idée de la manière dont on pourrait prouver que le système d'axiomes ci-dessous fait du TDA ACCU une spécification correcte :

```
A1  : charge(créer(k)) = 0
A2  : capacité(créer(k)) = k
A3  : ¬hs(créer(k))
A4  : charge(décharger(a)) = max(charge(a) - 1, 0)
A5  : capacité(décharger(a)) = capacité(a)
A6  : ¬hs(décharger(a))
A7  : ¬hs(charger(a, i)) ⇒ charge(charger(a, i)) = charge(a) + i
A8  : hs(charger(a, i)) ⇒ charge(charger(a, i)) = 0
A9  : capacité(charger(a, i)) = capacité(a)
A10 : hs(charger(a, i)) = (charge(a) + i > capacité(a))
A11 : charge(a) <= capacité(a)
```

9) Techniques de construction des systèmes d'axiomes

9.1) Approche descriptive

Cette méthode consiste à donner des axiomes qui expriment la valeur de chaque requête de l'ensemble de base sur chaque générateur. Le cas échéant, on rajoutera des axiomes pour préciser le domaine de variation des valeurs prises par les requêtes, et des axiomes pour exprimer les valeurs prises par les requêtes de l'ensemble dérivé en fonction de celles de l'ensemble de base.

Par application de ce principe, on peut construire le système d'axiomes suivant pour le TDA ACCU augmenté de la requête dérivée `vide` :

```
A1  : charge(créer(k)) = 0
A2  : capacité(créer(k)) = k
A3  : ¬hs(créer(k))
A4  : charge(décharger(a)) = max(charge(a) - 1, 0)
A5  : capacité(décharger(a)) = capacité(a)
A6  : ¬hs(décharger(a))
A7  : ¬hs(charger(a, i)) ⇒ charge(charger(a, i)) = charge(a) + i
A8  : hs(charger(a, i)) ⇒ charge(charger(a, i)) = 0
A9  : capacité(charger(a, i)) = capacité(a)
A10 : hs(charger(a, i)) = (charge(a) + i > capacité(a))
A11 : charge(a) <= capacité(a)
A12 : vide(a) = (charge(a) = 0)
```

9.2) Approche constructive

Dans certains cas, il est difficile d'utiliser la méthode précédente. Il faut alors déterminer une base de générateurs et un ensemble de formes canoniques. On peut ensuite donner des axiomes qui expriment la valeur de chaque requête de l'ensemble de base sur chaque forme canonique, puis compléter par des axiomes permettant de réduire toute expression correcte sans requête en une forme canonique équivalente, et enfin rajouter éventuellement des axiomes pour préciser le domaine de variation des valeurs prises par

les requêtes, et des axiomes pour exprimer les valeurs prises par les requêtes de l'ensemble dérivé en fonction de celles de l'ensemble de base.
Par application de ce second principe, on peut construire le système d'axiomes suivant pour le TDA ACCU, après avoir choisi la base de générateurs `créer` et `charger`, ainsi que l'ensemble de formes canoniques `créer(k)` et `charger(créer(k), i)` pour toute valeur de `k` positive ou nulle, et toute valeur de `i` et de `j` non nulle :

```
A1' : charge(créer(k)) = 0
A2' : capacité(créer(k)) = k
A3' : ¬hs(créer(k))
A4' : 1 ≤ i ≤ k ⇒ charge(charger(créer(k), i)) = i
A5' : charge(charger(créer(k), k+1)) = 0
A6' : 1 ≤ i ≤ k+1 ⇒ capacité(charger(créer(k), i)) = k
A7' : 1 ≤ i ≤ k+1 ⇒ hs(charger(créer(k), i)) = (i = k+1)
A8' : i > k+1 ⇒ charger(créer(k), i) = charger(créer(k), k+1)
A9' : charger(charger(créer(k), i), j) = charger(créer(k), i+j)
A10': décharger(créer(k)) = créer(k)
A11': décharger(charger(créer(k), 1)) = créer(k)
A12': 1 < i ⇒ décharger(charger(créer(k), i)) = charger(créer(k), i-1)
A13': charge(a) ≤ capacité(a)
A14': vide(a) = (charge(a) = 0)
```

N'oubliez pas que les axiomes sont supposés ne contenir que des expressions correctes. Ainsi dans l'axiome `A12'`, il n'est pas utile d'indiquer que `i ≤ k`, cette borne étant implicite puisque l'expression `décharger(charger(créer(k), i))` doit être correcte.

Notez que l'on peut récrire ce système un peu plus simplement de la façon suivante :

```
A1' : charge(créer(k)) = 0
A2' : capacité(créer(k)) = k
A3' : ¬hs(créer(k))
A4' : charge(charger(créer(k), i)) = min(i, k+1) mod (k+1)
A5' : capacité(charger(créer(k), i)) = k
A6' : hs(charger(créer(k), i)) = i > k
A7' : charger(charger(créer(k), i), j) = charger(créer(k), i+j)
A8': décharger(créer(k)) = créer(k)
A9': décharger(charger(créer(k), 1)) = créer(k)
A10': 1 < i ⇒ décharger(charger(créer(k), i)) = charger(créer(k), i-1)
A11': charge(a) ≤ capacité(a)
A12': vide(a) = (charge(a) = 0)
```

L'exemple, vu précédemment, du TDA PILE doté uniquement des opérations `créer`, `vide`, `sommet`, `empiler` et `dépiler` est un bon exemple de situation dans laquelle il est nécessaire d'utiliser une approche constructive pour exprimer la spécification :

Requêtes de base
    vide, sommet

Générateurs de base
    créer, empiler

```
Expressions canoniques : C
C0 = { créer() }
Cn = { empiler(p, x) | x ∈ E, p ∈ Cn-1 } pour n > 0
C = C0 ∪ C1 ∪ ... ∪ Cn ∪ ...

Valeur des requêtes de base sur les expressions canoniques
cas n = 0 : donne l'axiome A1 du TDA PILE
     vide(créer()) = V
cas n > 0 : donne les axiomes A2 et A3 du TDA PILE
     ∀q ∈ Cn : ∃p ∈ Cn-1, ∃x ∈ E tq q = empiler(p, x)
          sommet(q) = x
     ∀q ∈ Cn
          vide(q) = F
Règle de réduction des expressions qui ne sont pas canoniques
donne l'axiome A4 du TDA PILE
     ∀n > 0, ∀q ∈ Cn : ∃p ∈ Cn-1, ∃x ∈ E tq q = empiler(p, x)
          dépiler(q) = p
```

10) Preuve de la correction du TDA ACCU

Prouver la correction du TDA ACCU, c'est prouver que l'on est capable de faire correspondre à toute expression correcte (EC) une unique valeur de A, N ou B.

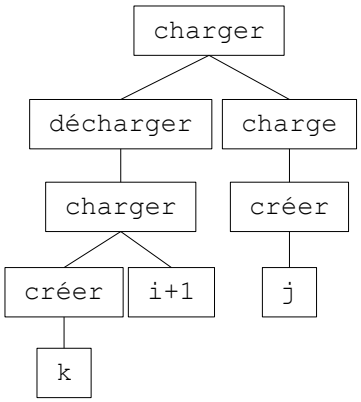
La question est donc d'être capable de déterminer, pour une part, les expressions bien formées (EBF) qui sont des EC, c'est-à-dire de déterminer quels sont les programmes corrects, et d'autre part la valeur de toute expression correcte se terminant par une requête (ECTR), c'est-à-dire la valeur que produira un programme correct à l'exécution.

Il faut donc prouver (par induction sur les expressions) que les assertions suivantes sont vraies pour tout i :

- P(i) : « Le TDA permet de dire si une EBF de niveau au plus i est une EC »
- Q(i) : « Toute ECTR de niveau i a une valeur bien déterminée »
- R(i) : « Toute ECSR de niveau i admet une forme canonique équivalente »

Commençons par remarquer que toute EBF du TDA est identifiée par un unique arbre qui représente sa structure en termes d'opérations du TDA. On définit pour cela le niveau d'une EBF comme la hauteur de l'arbre qui lui est associé.

Par exemple, l'EBF « `charger(décharger(charger(créer(k), i+1)), charge(créer(j)))` » est de niveau 4
car elle est représentée par l'arbre suivant :



Notons ensuite les ensembles d'expressions bien formées de la manière suivante :

- EBF_0 = { constantes et variables entières }
- EBF_1 = { `créer(n)` | `n` $\in$ EBF_0 }
- EBF_2 = { `charge(a)`, `capacité(a)`, `hs(a)`, `décharger(a)`, `charger(a, n)` | `a` $\in$ EBF_1 et `n` $\in$ EBF_0 }
- pour k ≥ 3 : EBF_k = { `charge(a)`, `capacité(a)`, `hs(a)`, `créer(n)`, `décharger(a)`, `charger(a, n)` | `a` $\in$ EBF_{k-1} et `n` $\in$ EBF_{k-1}, et `a` se termine par `créer`, `décharger` ou `charger`, et `n` se termine par `charge` ou `capacité` }
- $EBF_{\leq n} = \bigcup_{k \leq n} EBF_k$

Remarquons que EBF_2 = { `charge(créer(n))`, `hs(créer(n))`, `capacité(créer(n))`, `décharger(créer(n))`, `charger(créer(n), n')` }

Notons aussi les ensembles d'expressions correctes ainsi :

- $EC_0 = EBF_0$ et $val(EC_0) = \mathbb{N}$
- $EC_1 = EBF_1$ et $val(EC_1) \subset A$
- EC_k = { e | e $\in$ EBF_k et (c1(e) ou c2(e)) }
- $EC_{\leq n} = \bigcup_{k \leq n} EC_k$

où

- c1(e) : si e = `décharger(a)` avec `a` $\in$ EC_{k-1}, alors val(`hs(a)`) = F
- c2(e) : si e = `charger(a, n)` avec `a` $\in$ EC_{k-1} et `n` $\in$ EC_{k-1}, alors val(`hs(a)`) = F et val(`n`) ≠ 0

Notons encore les ensembles de formes canoniques de la manière suivante :

- FK_1 = { `créer(n)` | `n` $\in$ EC_0 }
- FK_2 = { `charger(créer(n), q)` | `n`, `q` $\in$ EC_0, val(`q`) $\in$ [1 .. val(`n`)] }
- $FK = FK_1 \cup FK_2$

Avec les notations précédentes, nous pouvons reformuler nos trois propositions :

- P(i) : $\forall$ e $\in$ $EBF_{\leq i}$, on peut savoir si e $\in$ $EC_{\leq i}$
- Q(i) : $\forall$ e $\in$ $ECTR_i$, val(e) est bien définie
- R(i) : $\forall$ e $\in$ $ECSR_i$, si e $\notin$ FK alors $\exists$ c $\in$ FK telle que val(e) = val(c)

qu'il reste à démontrer par induction sur le niveau des expressions manipulées.

Cas de base :

- P(0) et P(1) sont trivialement vraies car $EBF_{\leq 0} = EC_{\leq 0}$ et $EBF_{\leq 1} = EC_{\leq 1}$

- R(1) est trivialement vraie car $ECSR_1 = FK_1$
- P(2) est trivialement vraie d'après les préconditions
- Q(2) est vraie en utilisant les axiomes A2, A3 et A4
- R(2) est vraie car $ECSR_2 = FK_2 \cup \{ \texttt{décharger(créer(n))} \mid n \in EC_0 \}$ et val(`décharger(créer(n))`) = val(`créer(n)`)

L'hypothèse d'induction exprime pour k ≥ 2 fixé que :

- P(k) est correcte
- Q(i) est correcte pour tout i ≤ k
- R(i) est correcte pour tout i ≤ k

L'étape d'induction, bien que simple, est fastidieuse à rédiger. Elle est donc laissée en exercice pour le lecteur. Elle consiste à montrer (dans l'ordre) que sous ces hypothèses :

- P(k+1) est correcte
- Q(k+1) est correcte
- R(k+1) est correcte

Chapitre 5 : Héritage

1) Illustration : compteurs et horloges

On définit la notion de compteur modulaire comme une machine qui permet de parcourir cycliquement une séquence finie d'entiers successifs, en commençant par 0. On définit ensuite la notion d'horloge simple comme une machine qui permet d'indiquer l'heure qu'il est en donnant le nombre d'heures, de minutes et de secondes. On définit enfin la notion d'horloge digitale comme une machine qui permet d'indiquer l'heure qu'il est en donnant le nombre d'heures, de minutes et de secondes à l'aide d'afficheurs LCD (Liquid Crystal Display).

Traduite en Java, la notion de compteur modulaire devient l'interface :

```
/**
 * Type des compteurs modulaires.
 *
 * @inv 0 <= getValue() < getModulo()
 * @cons
 *      $ARGS$ int m
 *      $PRE$  m > 0
 *      $POST$ getValue() == 0 && getModulo() == m
 */
public interface IModCounter {

    // REQUETES

    /**
     * La valeur de ce compteur modulaire.
     */
    int getValue();

    /**
     * Le modulo de ce compteur modulaire.
     */
    int getModulo();

    // COMMANDES

    /**
     * Incrémente la valeur de ce compteur modulaire.
     * @post getValue() == (old getValue() + 1) % getModulo()
     */
    void increment();

    /**
     * Remet à zéro la valeur de ce compteur modulaire.
     * @post getValue() == 0
```

```
     */
    void reset();
}
```

Nous supposerons cette interface implantée dans une classe `ModCounter`.
La notion d'horloge, quand à elle, conduit à l'interface `IClock` :

```
/**
 * Type des horloges.
 *
 * @inv
 *      (getH() >= 0) && (getH() < HOURS_IN_DAY)
 *      (getM() >= 0) && (getM() < MINUTES_IN_HOUR)
 *      (getS() >= 0) && (getS() < SECONDS_IN_MINUTE)
 *      getTime() != null
 *      getTime() est une représentation textuelle fonction de
 *          getH(), getM() et getS()
 */
public interface IClock {

    // CONSTANTES STATIQUES

    int HOURS_IN_DAY = 24;
    int MINUTES_IN_HOUR = 60;
    int SECONDS_IN_MINUTE = 60;

    // REQUETES

    /**
     * Les heures que cette horloge indique.
     */
    int getH();

    /**
     * Les minutes que cette horloge indique.
     */
    int getM();

    /**
     * Les secondes que cette horloge indique.
     */
    int getS();

    /**
     * Une représentation textuelle des valeurs retournées
     *  par getH(), getM() et getS().
     */
    String getTime();

    // COMMANDES
```

```
    /**
     * Donne une impulsion à cette horloge, augmentant
     *  l'heure indiquée d'une seconde.
     * @post
     *      getS() == (old getS() + 1) % SECONDS_IN_MINUTE
     *      (getS() == 0)
     *          ==> getM() == (old getM() + 1) %  MINUTES_IN_HOUR
     *      (getS() == 0) && (getM() == 0)
     *          ==> getH() == (old getH() + 1) % HOURS_IN_DAY
     */
    void impulse();

    /**
     * Remet cette horloge à zéro.
     * @post (getH() == 0) && (getM() == 0) && (getS() == 0)
     */
    void reset();

    /**
     * Fixe l'heure indiquée par cette horloge.
     * @pre
     *      (0 <= h) && (h < HOURS_IN_DAY)
     *      (0 <= m) && (m < MINUTES_IN_HOUR)
     *      (0 <= s) && (s < SECONDS_IN_MINUTE)
     * @post (getH() == h) && (getM() == m) && (getS() == s)
     */
    void set(int h, int m, int s);
}
```

Dans l'état actuel de nos connaissances, si l'on cherche à écrire le moins de code possible, on peut réaliser les horloges simples à l'aide d'une classe `SimpleClock` qui implante l'interface `IClock` et dont les instances *utilisent* chacunes trois instances distinctes de `ModCounter` :

```
public class SimpleClock implements IClock {

    // ATTRIBUTS

    private IModCounter hour;
    private IModCounter minute;
    private IModCounter second;

    // CONSTRUCTEURS

    public SimpleClock() {
        hour = new ModCounter(HOURS_IN_DAY);
        minute = new ModCounter(MINUTES_IN_HOUR);
        second = new ModCounter(SECONDS_IN_MINUTE);
```

```
    }

    // REQUETES

    public int getH() {
        return hour.getValue();
    }

    public int getM() {
        return minute.getValue();
    }

    public int getS() {
        return second.getValue();
    }

    public String getTime() {
        return getH() + ":" + getM() + ":" + getS();
    }

    // COMMANDES

    public void impulse() {
        second.increment();
        if (second.getValue() == 0) {
            minute.increment();
            if (minute.getValue() == 0) {
                hour.increment();
            }
        }
    }

    public void reset() {
        second.reset();
        minute.reset();
        hour.reset();
    }

    public void set(int h, int m, int s) {
        if ((0 > h) || (h >= HOURS_IN_DAY)) {
            throw new AssertionError("h : " + h);
        }
        if ((0 > m) || (m >= MINUTES_IN_HOUR)) {
            throw new AssertionError("m : " + m);
        }
        if ((0 > s) || (s >= SECONDS_IN_MINUTE)) {
            throw new AssertionError("s : " + s);
        }
```

```
        reset();
        for (int i = 0; i < h; i++) {
            hour.increment();
        }
        for (int i = 0; i < m; i++) {
            minute.increment();
        }
        for (int i = 0; i < s; i++) {
            second.increment();
        }
    }
}
```

Mais il serait peut-être astucieux de penser à une instance de `SimpleClock` comme à une instance particulière de `ModCounter` : dans ce cas, une `SimpleClock` serait *une sorte de* `ModCounter` de modulo 86400 (60 s × 60 m × 24 h = 86400 s), et on aimerait disposer d'un mécanisme qui permette de déclarer que le code de la classe `SimpleClock` est le même que celui de la classe `ModCounter` avec juste ce qu'il faut de modifications pour que le comportement soit celui d'une instance de `IClock`... patience !

Pour ce qui concerne la classe des horloges digitales, `DigitalClock`, elle implante `IClock` et peut économiser pas mal de code si ses instances *utilisent* chacune une `SimpleClock` et un tableau de `LCDDisplay` :

```
public class DigitalClock implements IClock {

    // ATTRIBUTS STATIQUES

    private static final int NB_DISPLAYS = 8;

    // ATTRIBUTS

    private IClock clock;
    private LCDDisplay[] displays;

    // CONSTRUCTEURS

    public DigitalClock() {
        clock = new SimpleClock();
        displays = new LCDDisplay[NB_DISPLAYS];
        for (int i = 0; i < displays.length ; i++) {
            // initialisation des displays[i]
        }
    }

    // REQUETES

    public int getH() {
        return clock.getH();
    }
```

```
    public int getM() {
        return clock.getM();
    }

    public int getS() {
        return clock.getS();
    }

    public String getTime() {
        StringBuffer res = new StringBuffer();
        updateDisplays();
        for (int i = 0 ; i < Symbol.HEIGHT ; i++) {
            for (int j = 0 ; j < displays.length ; j++) {
                res.append(displays[j].getLine(i));
            }
            res.append("\n");
        }
        return res.toString();
    }

    // COMMANDES

    public void impulse() {
        clock.impulse();
    }

    public void reset() {
        clock.reset();
    }

    public void set(int h, int m, int s) {
        clock.set(h, m, s);
    }

    // OUTILS

    /**
     * Inscrit l'heure courante dans le tableau displays.
     */
    private void updateDisplays() { ... }
}
```

Ici encore il serait probablement astucieux de voir une instance de `DigitalClock` comme *une sorte de* `SimpleClock` mais dotée d'un afficheur LCD en supplément. Dans ce cas on aimerait encore disposer d'un mécanisme permettant de déclarer que le code de `DigitalClock` est le même que celui de `SimpleClock` avec pour seule différence le code de la méthode `getTime`...

Ce que l'on sait faire jusqu'à présent, c'est donc *réutiliser* des classes déjà construites pour en construire de nouvelles. Et ce que l'on aimerait, c'est disposer d'un mécanisme permettant de déclarer que de nouvelles classes *obtiennent* le même code que celui de classes déjà développées sans avoir à le récrire une deuxième fois, et que ce mécanisme soit suffisamment souple pour pouvoir adapter le code ainsi obtenu aux spécificités des nouvelles classes.
En Java, ce mécanisme existe et il s'appelle « héritage ». Il permet, par une simple déclaration dans l'entête d'un type, de désigner un ou plusieurs autres types déjà développés et dont on veut hériter du contenu pour le réutiliser tel quel ou le modifier selon notre volonté sans avoir à tout récrire.

2) Héritage

2.1) Héritage

Récrivons les deux classes `SimpleClock` et `DigitalClock` en utilisant le mécanisme d'héritage de Java :

```
public class SimpleClock extends ModCounter implements IClock {

    // ATTRIBUTS STATIQUES

    private static final int SECONDS_IN_MINUTE = 60;
    private static final int MINUTES_IN_HOUR = 60;
    private static final int SECONDS_IN_HOUR =
            SECONDS_IN_MINUTE * MINUTES_IN_HOUR;
    private static final int HOURS_IN_DAY = 24;
    private static final int SECONDS_IN_DAY =
            SECONDS_IN_HOUR * HOURS_IN_DAY;

    // CONSTRUCTEURS

    public SimpleClock() {
        super(SECONDS_IN_DAY);
    }

    // REQUETES

    public int getH() {
        return getValue() / SECONDS_IN_HOUR;
    }

    public int getM() {
        return (getValue() % SECONDS_IN_HOUR) / SECONDS_IN_MINUTE;
    }

    public int getS() {
        return getValue() % SECONDS_IN_MINUTE;
    }
```

```
    public String getTime() {
        return getH() + ":" + getM() + ":" + getS();
    }

    // COMMANDES

    public void impulse() {
        increment();
    }

    public void set(int h, int m, int s) {
        if ((0 > h) || (h >= HOURS_IN_DAY)) {
            throw new AssertionError("h : " + h);
        }
        if ((0 > m) || (m >= MINUTES_IN_HOUR)) {
            throw new AssertionError("m : " + m);
        }
        if ((0 > s) || (s >= SECONDS_IN_MINUTE)) {
            throw new AssertionError("s : " + s);
        }

        int value = h*SECONDS_IN_HOUR + m*SECONDS_IN_MINUTE + s;
        for (int i = 0; i < value; i++) {
            increment();
        }
    }
}

public class DigitalClock extends SimpleClock {

    // ATTRIBUTS STATIQUES

    private static final int NB_DISPLAYS = 8;

    // ATTRIBUTS

    private LCDDisplay[] displays;

    // CONSTRUCTEURS

    public DigitalClock() {
        super();
        displays = new LCDDisplay[NB_DISPLAYS];
        for (int i = 0; i < NB_DISPLAYS; i++) {
            // initialisation des display[i]
        }
    }

    // REQUETES
```

```
    public String getTime() {
        StringBuffer res = new StringBuffer();
        updateDisplays();
        for (int i = 0; i < Symbol.HEIGHT; i++) {
            for (int j = 0; j < NB_DISPLAYS; j++) {
                res.append(displays[j].getLine(i));
            }
            res.append("\n");
        }
        return res.toString();
    }

    // OUTILS

    /**
     * Inscrit l'heure courante dans le tableau displays.
     */
    private void updateDisplays() { ... }
}
```

Dans la seconde version ci-dessus, la classe `SimpleClock` contient le même code que la classe `ModCounter` (grâce à **`extends`** `ModCounter`) mais avec les modifications apportées par le code fourni dans le corps de la classe `SimpleClock`, et la classe `DigitalClock` contient le même code que la classe `SimpleClock` (grâce à **`extends`** `SimpleClock`) avec les modifications apportées par le code fourni dans le corps de la classe `DigitalClock`.

D'une manière générale, l'héritage est un mécanisme fourni par un LOO, qui permet de définir une nouveau type en récupérant en son sein les caractéristiques d'autres types. Un type héritier peut modifier les caractéristiques dont il hérite et/ou en ajouter d'autres.
On appelle ancêtre d'un type T, le type T lui-même ou l'un de ses parents (directs ou indirects) dans la relation d'héritage. On appelle descendant d'un type T, le type T lui-même ou l'un de ses enfants (directs ou indirects) dans la relation d'héritage. Pour un ancêtre propre, on parle aussi de super-classe ou de super-interface, directe ou indirecte. Pour un descendant propre, on parle de sous-classe ou de sous-interface, directe ou indirecte.

Le mécanisme d'héritage fourni par Java se décline en trois versions :

- héritage simple entre classes : une classe ne peut avoir qu'une seule super-classe directe ;
- héritage multiple entre interfaces : une interface peut avoir plusieurs super-interfaces directes ;
- implémentation multiple entre classes et interfaces : une classe peut avoir plusieurs super-interfaces directes.

De plus, toute classe qui ne dit pas explicitement qu'elle hérite d'une autre classe, hérite en fait de la classe `Object`, cette dernière est donc la super-classe de toute classe Java.

La hiérarchie d'héritage de toutes les classes est représentée par un arbre indiquant les relations d'héritage entre ces classes. Si l'on rajoute à cet arbre les relations d'héritage avec les interfaces, on obtient un graphe orienté sans cycle.

La hiérarchie d'héritage d'une classe est la branche constituée de toutes les classes que l'on parcourt en partant de celle-ci et en remontant jusqu'à la racine de l'arbre : la classe `Object`.

2.2) Retour sur les modificateurs d'accessibilité

Nous complétons ici les règles d'accessibilité données à la section `4.3` du chapitre 4.

L'accessibilité d'un élément déclaré dans le type *T* avec le modificateur **`protected`** est possible en un point *e* du code source uniquement :

- si *T* est un type accessible pour *e*, et si *e* est situé dans le texte d'un type héritier de *T* ;
- si *e* est situé dans le texte d'un type du même paquetage que *T*.

Reste à compléter les remarques faites précédemment pour déterminer la manière de coder la précondition du contrat d'une méthode[13]. Il faut remarquer qu'une méthode protégée d'une classe non publique ne sera accédée que par des classes clientes faisant partie du même paquetage. De ce fait, le codage de la précondition d'une telle méthode peut se faire au moyen d'une instruction d'assertion.

3) Polymorphisme

Le polymorphisme est la capacité d'une variable à référencer, à l'exécution, des objets d'un type différent de celui avec lequel elle a été déclarée.
Par exemple, le fait qu'une variable typée par une interface puisse être initialisée avec un objet est une manifestation du polymorphisme :

```
CharSequence s = new String("arf")
```

Plus généralement, un attachement polymorphe est un attachement pour lequel le type de la source est différent du type de la cible ; la variable cible est alors dite polymorphe.

Attention, il ne s'agit pas de dire que le type d'une variable polymorphe évolue au cours du temps ! Dire qu'une variable est polymorphe, c'est dire que l'on peut l'attacher à un objet d'un certain type, puis plus tard à un objet d'un type différent.

Le polymorphisme sans aucune contrainte n'est pas compatible avec la notion de langage statiquement typé. Les situations légales de polymorphisme sont repérées par le terme de compatibilité : on dit que le type référence S est compatible (pour l'affectation) avec le type T s'il est possible d'attacher à toute variable de type T un objet de type S. En Java, S est compatible avec T ssi :

- T est le type `Object` ;
- ou bien S est un descendant de T.

13 Section 6.1.2, remarques faisant suite à la règle numéro 2.

3.1) Type à la compilation et à l'exécution

Le type à la compilation d'une variable est le type utilisé lors de sa déclaration ; ce type est donc détectable lors de la compilation.
Le type à l'exécution d'une variable est le type de l'objet auquel elle est attachée à un instant donné ; ce type n'est connu, en toute généralité, qu'à l'exécution.

Ces définitions se généralisent aisément aux expressions de types références, en considérant que le type à la compilation d'une expression se calcule à partir du type à la compilation des variables et des constantes dont elle est constituée, ainsi que de la signature des opérateurs ou méthodes auxquels elle fait appel.
Le type à l'exécution d'une expression est le type de la référence qui constitue la valeur que produit son évaluation.

3.2) Règle de compatibilité de types

Le polymorphisme est contraint par la compatibilité de types.

Dit autrement : l'attachement d'une source à une cible polymorphe n'est valide que si le type à la compilation de la cible est `Object` ou si le type à la compilation de la source est un descendant du type à la compilation de la cible.

3.3) Règle d'appel de caractéristique

Un appel x.c est compilable ssi la caractéristique c est accessible depuis le site d'appel, et déclarée dans l'un des ancêtres du type statique de x.

Concrètement, cette règle garantit que si le compilateur accepte de compiler un appel de caractéristique c'est qu'il a été capable de découvrir une définition dans le type à la compilation de la cible et que cette définition est accessible depuis l'endroit où l'appel est effectué.

La combinaison de cette règle avec celle qui la précède nous autorise à dire que le type à l'exécution d'une expression à un instant donné est nécessairement compatible avec le type à la compilation de cette expression. Ainsi, l'objet référencé par la cible d'un appel disposera toujours de la caractéristique en question lors de l'exécution.

4) Transmission par héritage

4.1) Non transmission des constructeurs

Les constructeurs ne sont jamais hérités dans les sous-classes.

L'héritage permet de bénéficer du code écrit dans une autre classe, mais pas de *tout* le code, seulement de certains attributs et de certaines méthodes. Les constructeurs, eux, ne sont pas hérités.

Qu'on le veuille ou non, la première instruction d'un constructeur est obligatoirement un appel à un autre constructeur de la même classe (appel de la forme `this(...);`) ou bien un appel à un constructeur de la super-classe (appel de la forme `super(...);`).
En effet, si l'on donne une instruction différente des deux citées ci-dessus, le compilateur ajoute avant toute instruction du constructeur un appel au constructeur sans argument de

la super-classe (`super();`). Bien entendu, dans ce cas, il faudra s'assurer que la super-classe en question soit dotée d'un constructeur sans argument...
De ce fait, l'exécution d'un constructeur d'une classe enchaîne en réalité l'exécution de plusieurs constructeurs, au moins un constructeur de chaque classe faisant partie de la hiérarchie d'héritage de cette classe. Cet enchaînement se déroule depuis le constructeur sans argument de la classe `Object` jusqu'au constructeur de la classe que l'on utilise avec l'opérateur **`new`**.

4.2) Transmission des caractéristiques

Les caractéristiques d'un type sont potentiellement transmissibles par héritage aux sous-types seulement si :

- *elles sont publiques ou protégées ;*
- *ou elles sont sans modificateur et le sous-type concerné est alors situé dans le même paquetage que son super-type.*

Rq. : les caractéristiques d'une interface sont obligatoirement publiques, donc elles sont potentiellement transmissibles à leurs sous-types.

4.2.1) Masquage d'attribut

Lorsqu'on définit dans un type un attribut de même nom que l'un des attributs potentiellement transmissibles depuis l'un des super-types de ce type, on est dans une situation de masquage d'attribut : celui qui est défini dans le sous-type empêche l'héritage de celui qui est défini dans le super-type.

On peut quand même accéder à un attribut d'instance masqué en utilisant la notation **`super`**, ou la variable synthétique **`this`** après l'avoir correctement transtypée.
Pour un attribut statique, on peut évidemment y accéder en le préfixant du nom du super-type.

4.2.2) Masquage de méthode statique

Lorsqu'on définit dans une classe une méthode statique de même signature que l'une des méthodes statiques potentiellement transmissibles depuis l'un des super-types de cette classe, on est dans une situation de masquage : la méthode statique définie dans la sous-classe empêche l'héritage de celle définie dans le super-type.

On peut évidemment accéder à une méthode statique masquée en la préfixant du nom du super-type.

4.2.3) Redéclaration de méthode d'instance

Lorsqu'on définit dans un type une méthode d'instance de même signature que l'une des méthodes d'instances potentiellement transmissibles depuis l'un des super-types de ce type, on est dans une situation de redéclaration.
Une redéclaration est une déclaration qui empêche la transmission d'une méthode par héritage en la remplaçant par une nouvelle méthode. Ce remplacement peut se faire de deux manières différentes :

- par concrétisation d'une méthode abstraite : on fournit dans la classe héritière, en remplacement d'une méthode abstraite présente dans l'ancêtre, une nouvelle

méthode, concrète, de même signature ;
- par redéfinition d'une méthode concrète : on fournit dans la classe héritière, en remplacement d'une méthode concrète présente dans la super-classe, une nouvelle méthode concrète de même signature mais de corps différent.

Dans le code de la sous-classe, on peut accéder à la méthode de la super-classe en préfixant l'appel par la notation **`super`** (au lieu de **`this`**). Cela nous permet de changer le corps de la méthode soit en le remplaçant totalement soit en intervenant avant, après ou autour du corps de la méthode de l'ancêtre.

5) Mécanismes de liaison

5.1) Liaison dynamique

C'est un mécanisme qui, dans le contexte de l'appel d'une méthode d'instance, consiste à retarder au moment de l'exécution et en fonction du type à l'exécution de la cible de l'appel, le choix de la méthode devant être exécutée. C'est ce qui fonde la différence entre une fonction et une méthode.

La mise en place de ce mécanisme est possible du fait que chaque objet dispose d'un pointeur vers la table des méthodes que lui fournit sa classe génératrice. C'est cette table qui permet à l'objet d'accéder efficacement à chacune de ses méthodes.
Lorsqu'une classe hérite d'une autre, la nouvelle table des méthodes construite à cette occasion est une copie de l'ancienne table dans laquelle, pour chaque méthode redéclarée, l'adresse de branchement vers le code hérité pour cette méthode est remplacée par l'adresse de branchement vers le nouveau code fourni dans la classe héritière.

En fait, lors d'un appel `obj.m()`, tout se passe comme si, à l'exécution, la machine Java exécutait l'algorithme de recherche suivant :

```
Soit x une variable initialisée avec la classe de obj
Tant que x <> null Faire
    Si x contient une définition pour m() Alors
        cette méthode est retrouvée et exécutée
        terminer
    Sinon
        x <- super-classe de x
    FinSi
FinTantQue
Si x vaut null Alors erreur : NoSuchMethodError FinSi
```

En toute logique, l'erreur ne devrait jamais se produire dans l'algorithme ci-dessus puisque le compilateur a accepté de compiler l'appel (application de la règle d'appel de caractéristique qui garantit qu'une méthode existe, et de la règle de compatibilité de types qui garantit que le type à l'exécution descend du type à la compilation et donc qu'il contient une définition, au moins héritée, pour la méthode). Néanmoins il est possible de manipuler le code des classes compilées de telle sorte que la machine Java ne retrouve pas de définition pour la méthode `m()` dans la hiérarchie d'héritage du type de `obj`, c'est pourquoi l'algorithme se termine sur une possibilité d'erreur...

5.2) Liaison statique

C'est un mécanisme qui, dans le contexte d'un appel de méthode, consiste à déterminer dès la compilation et en fonction du type à la compilation de la cible de l'appel, l'adresse de branchement vers le code qui devra être exécuté.
En Java, les méthode de classes (`static`) sont résolues à la compilation par liaison statique (comme n'importe quel appel de fonction en C) ainsi que les méthodes d'instances lorsqu'elles sont déclarées avec `private` ou `final`, ou encore préfixées de `super` lors de l'appel[14].

6) Différence entre redéfinition et surcharge

Attention, les termes de redéfinition et de surcharge ne sont pas synonymes.
Une méthode est surchargée si elle est définie plusieurs fois dans une même classe, mais avec une liste différente de paramètres dans chaque cas.

Par ailleurs, le mécanisme de sélection de méthode en cas de surcharge est basé sur le type à la compilation des arguments car c'est le compilateur qui détermine la signature de la méthode à exécuter.

7) Retour sur la mutabilité

Pour qu'une classe soit non mutable il faut qu'elle ne définisse aucune méthode permettant de modifier l'état de ses instances[15].

Si l'on considère que les instances d'une sous-classe sont aussi des instances de ses super-classes, cela a pour conséquence que toutes les sous-classes d'une classe non mutable doivent aussi être non mutables. Or lors de la conception d'une classe non mutable, il est impossible de contraindre ses futures sous-classes à être non mutables elles aussi. Par conséquent, une classe non mutable devrait être déclarée avec `final`, ce qui garantit qu'elle n'aura pas de sous-classe.

Par ailleurs, les variables d'instances d'une classe non mutable ne devant pas changer de valeur au cours du temps, il est préférable de les déclarer avec `final`.

8) Quand et comment hériter ?

8.1) Sous-typage

8.1.1) Principe de substitution

La notion de sous-type dans la théorie des TDA est caractérisée par le principe de substitution de Liskov (PSL) :

> *Si à chaque élément de T il correspond un élément de S, de sorte que dans tout programme P défini en fonction de T on peut substituer aux éléments de T les éléments de S qui correspondent, sans changer la sémantique de P, alors S est un sous-type de T.*

14 On rappelle que dans le cas général (= ni `private`, ni `final`, ni appel avec `super`) une méthode d'instance bénéficie de la liaison dynamique.
15 Ce qui ne veut pas nécessairement dire qu'elle ne contient aucune commande !

Ce principe exprime que S est sous-type de T si tout programme qui fonctionne en utilisant des éléments de T peut, utiliser sans le remarquer des éléments de S à la place de ceux de T. Le terme « sans le remarquer » est à rapprocher du terme « sans changer la sémantique du programme ».

8.1.2) Hériter pour sous-typer

Si l'on considère les implantations Java (classes ou interfaces) X et Y pour S et T respectivement, la traduction en Java d'une relation de sous-typage entre S et T nous amène à considérer les points suivants :

- Déclarer que X hérite de Y est obligatoire, car (règle de compatibilité de types) c'est ce qui nous permet de substituer des instances de X lorsque des instances de Y sont attendues dans les programmes utilisant des variables de type Y.
- Le comportement défini par X peut étendre le comportement défini par Y : les instances de X savent répondre aux mêmes messages que celles de Y plus d'autres messages encore (on a rajouté des méthodes dans X).
- L'invariant de X peut restreindre celui de Y : les instances de X vérifient les mêmes propriétés en phase stable que celles de Y, plus d'autres encore (on a rajouté des propriétés invariantes dans X).
- Les contrats de X peuvent raffiner les contrats hérités de Y : les instances de X se comportent de manière plus spécialisée que celles de Y (les contrats sont plus agréables pour les clients de X que pour ceux de Y : on a affaibli les préconditions et/ou renforcé les postconditions).

Le sous-typage entre TDA se traduira donc par une relation d'héritage entre types Java.

8.1.3) Pratique du sous-typage

Concernant les invariants :

- dans la documentation de X : ajouter une rubrique `@inv` qui ne déclare que les nouvelles propriétés invariantes (les propriétés de Y sont ici sous-entendues) ;
- dans le code de X : il est possible d'ajouter une méthode `public boolean inv()` pour tester l'invariant complet de X (la partie définie dans Y et aussi celle définie dans X).

Concernant le raffinement des préconditions :

- dans la documentation de la méthode : ajouter une rubrique `@pre` faisant état de l'affaiblissement de la précondition uniquement (la précondition de la méthode telle que définie dans Y est ici sous-entendue) ;
- dans le code de la méthode : il faut ajouter au début du corps un test sur la précondition complète (c'est-à-dire la précondition définie dans Y affaiblie par l'avenant défini lors de la redéclaration de la méthode dans X).

Concernant le raffinement des postconditions :

- dans la documentation de la méthode : ajouter une rubrique `@post` faisant état du renforcement de la postcondition uniquement (la postcondition de la méthode telle que définie dans Y est ici sous-entendue) ;
- dans le code de la méthode : il est possible d'ajouter à la fin du corps une instruction d'assertion sur la postcondition complète (c'est-à-dire la postcondition définie dans Y renforcée par l'avenant défini lors de la redéclaration de méthode dans X).

8.2) Hériter pour construire

Malheureusement, par la règle de compatibilité de types, l'héritage entre deux types Java entraîne systématiquement un lien de sous-typage entre ces deux types. De sorte que, en Java, on est obligé de confondre héritage et sous-typage.
Pourtant, dans certains cas, on peut vouloir utiliser la relation d'héritage simplement pour factoriser du code commun à plusieurs classes, ou réutiliser le code d'une classe existante, et ceci que l'on soit en situation de sous-typage ou non.

Dans le cas où l'on veut factoriser une partie du code commun à deux classes X et Y qui réalisent deux TDA qui ne sont pas en relation de sous-typage, on utilisera une super-classe abstraite pour le code factorisé. Si cette classe représente une réalisation partielle de TDA, elle pourra être publique. Au contraire, si elle ne représente rien en termes de TDA, elle sera non publique puisque sans signification pour les clients.
Dans ce dernier cas, il faudra que les interfaces qu'implémentent X et Y ne soient pas en relation d'héritage. C'est normal puisque les deux TDA ne sont pas en relation de sous-typage. Surtout, il faudra typer les variables avec ces interfaces plutôt qu'avec les classes qui les implantent. Ceci afin d'empêcher toute substitution.

8.3) Délégation

On parle de délégation lorsqu'un objet se décharge sur un autre objet de la réalisation d'un service.
La délégation est une alternative possible à l'héritage lorsqu'on n'est pas en présence de sous-typage.
Elle permet aussi de simuler l'héritage lorsqu'on ne maîtrise pas la hiérarchie d'héritage des classes, voire de simuler l'héritage multiple.

8.4) Retour sur la hiérarchie des horloges

Notre hiérarchie du début du chapitre était-elle correcte ? Non pas tout à fait !

Le PSL n'est pas vérifié entre horloges et compteurs modulaires puisqu'il n'y a pas de relation d'héritage entre les interfaces `IClock` et `IModCounter`. Mais ici on a voulu hériter tout de même, pour réutiliser dans `SimpleClock` du code et des fonctionnalités présents dans `ModCounter`. On a vu que cela était légitime tant qu'il n'y a pas d'héritage entre les interfaces. On observera toutefois que le gain entre les deux versions de `SimpleClock` (celle avec délégation et celle avec héritage) n'est pas énorme. Dans ces conditions, il est préférable de choisir la délégation.

Ce qui est plus grave, c'est la relation entre horloges digitales et horloges simples : cette fois-ci le PSL ne peut pas être vérifié à cause d'une incompatibilité entre les invariants de ces deux classes. En effet, `getTime()` ne peut pas retourner à la fois une chaîne de la forme "xx:xx:xx" et de la forme rendue par un afficheur LCD.

Au final, la solution est de développer une classe abstraite `Clock` qui va factoriser tout le code des horloges sauf la méthode `getTime`. Ensuite `SimpleClock` et `DigitalClock` hériteront de `Clock` pour récupérer tout le code des horloges et chacune rajoutera sa spécificité, c'est-à-dire le code nécessaire pour calculer `getTime`.

8.5) Conclusion

Considérons la situation de deux TDA S et T implantés en Java respectivement par deux interfaces I et J, chacune implémentée par une classe, disons X pour I, et Y pour J.

Si S $\subset$ T, alors I doit *étendre J, et X devrait étendre Y.*
Si S $\not\subset$ T, alors I ne doit pas *étendre J, mais X pourrait étendre Y.*