Méthodologie de la programmation orientée objet

S5-POO2

L3 informatique – Université de Rouen

Année 2023-2024

`http://dpt-info-sciences.univ-rouen.fr/~andarphi/lbb/index.php?category/POO2`

Philippe Andary

Table des matières

Ce document est en constante évolution : il est remanié ou complété d'une année sur l'autre. N'hésitez pas à me faire part de vos remarques...

Chapitre 1 : Rappels

1) Généralités

1.1) Variables, valeurs et types de données

Dans les langages impératifs, une variable est un nom ou une étiquette, définie dans le code source, représentant un emplacement mémoire qui contiendra une certaine valeur lors de l'exécution. On peut ainsi manipuler une valeur dans le code source, encore inconnue à ce stade, en la désignant par un nom. À l'exécution, l'emplacement mémoire associé à la variable contiendra des valeurs différentes au cours du temps, en fonction des actions prévues dans le programme.

Dans les langages à typage statique, les variables doivent être déclarées avec leur type avant utilisation. Ce type indique au compilateur (ou à l'interpréteur, selon le cas) comment reconstruire la donnée à partir de la séquence d'octets présente en mémoire. Il détermine aussi le domaine de variation de la variable, c'est-à-dire l'ensemble des valeurs qu'elle est susceptible de prendre, ainsi que l'ensemble des opérations disponibles portant sur ces valeurs.

1.2) Compatibilité entre types

Dans un langage de programmation, on dit qu'un type S est compatible avec un type T lorsque le langage nous autorise à affecter une valeur de type S à une variable déclarée de type T.

Concrètement, cela signifie qu'une valeur de type S peut être convertie (pour les types primitifs) ou momentanément considérée (pour les types références) comme une valeur de type T, et manipulée dans le code source à travers une variable de type T. On peut donc lui appliquer toutes les opérations portant sur T, et seulement celles-là.

1.3) Objets, classes et instances

Un objet est un élément logiciel, un modèle, qui n'existe que dans la tête du programmeur. Mais ce modèle est représenté dans la mémoire vive de la machine lors de l'exécution par une valeur qui l'identifie : sa référence. Parler d'un objet ou de sa référence, c'est donc parler de la même chose, mais à des niveaux différents : respectivement, au niveau de la conception ou au niveau de la machine).

Puisque l'on peut confondre objet et référence d'objet, nous emploierons souvent le terme d'objet là où en toute rigueur il faudrait parler de référence d'objet. Le lecteur traduira...

La référence d'un objet désigne une valeur complexe correspondant à une structure de données qui agrège différents champs. Cette valeur est stockée dans la partie dynamique de la mémoire allouée au programme : le tas. Le type de cette valeur est appelé le type de l'objet ; il détermine l'ensemble des opérations applicables sur tous les objets de ce type.

Un objet doit être créé par le programmeur au moment où ce dernier en a besoin, puis détruit lorsqu'il devient inutile afin de ne pas encombrer la mémoire. Si vous avez bien

suivi mon raisonnement, ce n'est pas l'objet qui est créé, mais sa référence ainsi que l'enregistrement qu'elle désigne dans le tas.

L'état d'un objet en un moment précis de l'exécution correspond à la séquence des valeurs prises par les champs de son enregistrement à cet instant là.

Le comportement d'un objet est l'ensemble des sollicitations (que l'on appelle messages) auxquelles il sait répondre, et qui lui permettent d'interagir avec son environnement. Pour chaque message reçu par l'objet, une des opérations du type de l'objet est activée.

Dans les langages de classes, les objets sont des valeurs typées et leur type s'appelle une classe. Au niveau du code source, une classe est définie par un texte structuré qui décrit des attributs (les structures de données qui accueilleront l'état des objets) et des méthodes (les opérations qui s'appliqueront sur les objets).

L'action qui consiste à créer un objet conformément au texte d'une classe s'appelle instanciation. L'objet ainsi créé est appelé une instance de cette classe.

2) Types Java

Java est un langage impératif à typage statique. Mais c'est aussi un langage objet, ce qui signifie qu'il permet de manipuler des objets, et c'est un langage de classes, ce qui signifie que les objets sont créés à partir de classes. On parle de la classe génératrice d'un objet pour désigner la classe dont il est une instance directe.

2.1) Types primitifs

Un type primitif est un type de données prédéfini au niveau du langage. En Java il en existe huit, aucun n'est une classe :

- le type des booléens : `boolean` ;
- le type des caractères : `char` ;
- les types entiers : `byte`, `short`, `int`, `long` ;
- et les types flottants : `float`, `double`.

Les valeurs de ces types sont codées en machine sur un certain nombre d'octets contigus en mémoire, en fonction du type auquel elles appartiennent.

On peut se représenter mentalement une variable d'un type primitif (une variable primitive) comme une boite contenant une certaine valeur de ce type. L'affectation entre deux variables primitives recopie la valeur contenue dans la variable de droite (de l'affectation) dans la variable de gauche.

2.2) Types références

Si l'on fait abstraction de la manière dont elle est implémentée au sein de la *JVM*, la notion de référence fait penser[1] à celle de pointeur vue en C, avec toutefois des restrictions importantes.

Dans un programme C, avec une variable pointeur `p`, on peut manipuler au choix la valeur pointeur (avec `p`) ou bien la valeur pointée (avec `*p`). On bénéficie aussi d'une

1 Le lecteur curieux pourra lire https://stackoverflow.com/a/2629758

arithmétique sur les pointeurs (par exemple si `p` est un pointeur vers le début d'un tableau, `p++` permet de pointer vers l'élément suivant dans le tableau).

En Java, rien de tout cela : une variable qui contient une référence d'objet, et dont le type est une classe, « pointe » bien vers une valeur mais elle permet seulement d'accéder à la valeur pointée. Bien que stockée dans la variable, la référence n'est donc jamais accessible au programmeur. Par ailleurs, aucune arithmétique sur les références n'est fournie.

On aura compris que lorsqu'on déclare une variable avec un type référence `X`, cette variable ne contiendra pas directement des instances de `X`, mais plutôt des références vers des instances de `X`. Il ne faut pas perdre de vue que l'affectation entre deux variables références recopie la valeur de celle de droite (une référence) dans la variable de gauche : il n'y a donc pas duplication d'un objet, mais copie de sa référence.

En Java, tout type qui n'est pas primitif est un type référence. Les classes génératrices d'objets sont donc des types références. Il y a en tout trois sortes de types références : les classes, les interfaces et les types tableaux.

Le langage Java est fourni avec une énorme bibliothèque de types références (plus de 3700 pour Java SE 6), que le développeur peut (et doit) enrichir avec ses propres types références.

2.2.1) Classes et interfaces

Une classe peut être concrète : il s'agit alors d'un type Java définissant des structures de données et un ensemble de méthodes complètement implémentées. Ce genre de classe est donc instanciable, c'est-à-dire que l'on peut s'en servir comme générateur d'objets.

Mais une classe peut aussi être abstraite : il s'agit alors d'un type Java non instanciable, définissant des structures de données et un ensemble de méthodes partiellement ou totalement implémentées. Une classe abstraite doit être déclarée explicitement avec le mot-clé **`abstract`**.

Une interface est un type Java non instanciable, définissant uniquement un ensemble de signatures de méthodes, sans aucune implémentation[2]. Elle est implicitement déclarée avec le mot-clé **`abstract`**.

Alors qu'une classe concrète ne peut contenir que des méthodes concrètes et qu'une interface ne peut contenir que des méthodes abstraites, une classe abstraite peut contenir à la fois des méthodes abstraites et des méthodes concrètes, le nombre d'éléments dans chacune de ces catégories pouvant être égal à zéro. En revanche, une classe qui contient au moins une méthode abstraite doit obligatoirement être déclarée abstraite.

2.2.2) Types tableaux

Les types tableaux sont construits sur la base de types existants en y ajoutant une ou plusieurs paires de crochets. Le type des éléments d'un tableau peut être n'importe quel type, primitif ou référence. Les types tableaux sont des types références et les tableaux sont donc des objets.

2 Le terme « uniquement » est exagéré : en réalité, on peut aussi y ajouter des constantes statiques.

Un type tableau à une dimension hérite de toutes les méthodes de la classe `Object`. Il définit une variable (constante) d'instance qui mémorise le nombre d'éléments du tableau :

```
public final int length
```

Un tableau à n dimensions est, en Java, un tableau à une dimension de tableaux à n-1 dimensions. Ainsi les tableaux à deux dimension peuvent avoir des « formes » multiples : rectangle, triangle, ou en dents de scie.

2.3) Description textuelle d'un type référence

Le texte d'une classe décrit des attributs, des méthodes, des constructeurs et, on le verra par la suite, d'autres types Java, ainsi que des blocs d'initialisation, statiques ou non.

Par comparaison, le texte d'une interface décrit seulement des attributs mais implicitement publics, statiques et constants ; des méthodes mais implicitement publiques et abstraites ; et d'autres types Java mais implicitement publics et statiques. Le texte d'une interface ne contient jamais de bloc d'initialisation ni de constructeur.

Les types de tableaux sont des types internes au langage, ils ne sont pas décrits par des textes particuliers mais gérés automatiquement.

3) Méthodes

Une méthode est, dans un langage objet, ce qui correspond à une routine (procédure ou fonction) dans un langage impératif classique. Mais, comme on le verra ci-dessous, une méthode ne se réduit pas à une simple routine.

En Java, une méthode ne peut pas contenir la définition d'une autre méthode. En revanche, elle peut s'appeler elle-même. Si c'est le cas, elle est alors récursive.

3.1) Méthodes d'instances

Métaphoriquement, un objet agit en fonction des messages qu'il reçoit. Pour répondre à ces messages, l'objet envoie d'autres messages à d'autres objets ou à lui-même. Chaque envoi de message se traduit

- dans le code source : par un appel de méthode sur une cible, cette cible étant une expression d'un type référence ;
- à l'exécution : par l'exécution du corps de la méthode associée à l'objet dont la référence est, à cet instant, la valeur de l'expression cible.

Au niveau du langage, les méthodes réalisent le comportement des objets comme des routines particulières.

La différence fondamentale entre une méthode et une routine classique tient au fait qu'une méthode est appelée « sur un objet », alors qu'une routine est appelée sans objet sur lequel s'appliquer. L'objet sur lequel est appelée la méthode permettra de déterminer à l'exécution quelles sont les variables à considérer (celles de l'objet cible de l'appel), d'une part, et d'autre part quelles instructions exécuter (celles constituant le corps de la méthode présente dans le type générateur de l'objet cible).

En reportant au moment de l'exécution la connaissance complète du type de l'objet cible

et, donc, le choix de la méthode à exécuter, tout se passe comme si les objets étaient des êtres intelligents, choisissant eux-même le code à exécuter. Nous étudierons ce mécanisme, dit de liaison dynamique, dans le prochain chapitre.

3.2) Méthodes de classe

Il existe une autre catégorie de méthodes : les méthodes de classe.

Une méthode de classe est une méthode qui correspond à une fonctionnalité associée à une classe plutôt qu'à l'une de ses instances. Pour que cela ait un sens, il faut que cette méthode ne dépende d'aucune instance particulière de la classe. Dans ces conditions, il n'y a pas de différence entre une routine au sens classique et une méthode de classe.

Puisque la méthode à exécuter est ici connue dès la compilation, il est possible d'utiliser un mécanisme de liaison statique à l'invocation de la méthode. Pour les distinguer des méthodes d'instance dans le code source, et pour indiquer au compilateur l'usage de liaison statique, on marque les méthodes de classe avec le mot clé **`static`**.

3.3) Passage de paramètres

En Java, le passage des paramètres d'une méthode se fait toujours par valeur : « changer la valeur d'un paramètre formel » n'a donc pas d'effet sur le paramètre effectif (lorsque celui-ci est une variable). Lors du passage d'un objet toutefois, il faut bien garder à l'esprit que c'est la référence de l'objet qui est transmise, et une méthode peut tout à fait modifier l'état d'un objet dont elle reçoit la référence en argument.

4) Variables

4.1) Affectation d'un objet à une variable référence

Une variable référence est une variable dont les valeurs possibles sont uniquement des références d'objets qui varient au cours de l'exécution. Elle peut aussi ne contenir « aucune valeur » : elle contient en fait dans ce cas une valeur spéciale, appelée référence vide (**`null`** en Java), qui ne pointe vers aucun objet et représente ainsi l'absence d'objet. On dit alors que la variable est vide.

L'affectation d'un objet à une variable est l'action qui établit le lien décrit ci-dessus entre une variable et un objet. La variable est la cible de l'affectation et l'objet dont elle va recevoir la référence en est la source. Le passage d'un objet comme paramètre effectif d'un appel de méthode met lui aussi en œuvre une affectation au niveau du paramètre formel, considéré comme une variable locale, qui est initialisé avec le paramètre effectif.

4.2) Recyclage d'un objet

Lors d'une nouvelle affectation d'une variable, ou lors de son vidage, il se peut que l'objet auquel la variable était préalablement affecté devienne inaccessible (plus aucune variable ne le référence).

En Java, le programmeur est libéré de la contrainte de gérer lui-même la destruction des objets inaccessibles (il n'y a pas, comme en C, de fonction `free()` ou autre chose de ce

genre). Lorsqu'un objet n'est plus référencé par aucune variable, l'espace mémoire qu'il occupe est automatiquement recyclé par un processus dédié que l'on appelle le ramasse-miettes (*garbage collector* en anglais).

4.3) Catégories de variables

Toute variable doit être déclarée au sein d'une description de type. Une déclaration qui est située à l'intérieur d'une méthode définit une variable locale à cette méthode. En dehors de toute méthode, elle définit un attribut.

4.3.1) Attributs

Un attribut est une variable globale, mais il existe deux niveaux de globalité. Un attribut statique (variable de classe ou d'interface selon le cas) est une variable globale relative à une classe (ou à une interface) : il n'en existe qu'un seul exemplaire, situé dans la zone des méthodes attribuée à l'application. Un attribut non statique (variable d'instance) est une variable globale relativement à chaque objet : dans ce cas l'espace mémoire associé à chaque instance de la classe (situé dans le tas) possède en propre un champ personnel correspondant à cette variable.

Une variable d'instance, appartenant à un objet, est définie dans le tas ; sa durée de vie se confond avec celle de l'objet. De façon duale, la durée de vie d'une variable statique est celle de la classe ou de l'interface à laquelle elle appartient (généralement tout le temps d'exécution de l'application).

4.3.2) Variables locales et paramètres formels

Une variable locale est une variable déclarée dans le corps d'une méthode.

Un paramètre formel est une variable déclarée dans l'entête d'une méthode, on peut le considérer comme une variable locale qui sera systématiquement initialisée avec la valeur du paramètre effectif au moment de l'appel.

Variables locales et paramètres formels sont définis sur la pile d'appel et sont automatiquement détruits juste avant le retour à l'appelant.

4.4) Initialisation

En Java, toute variable est initialisée avant sa première utilisation :

- un attribut est initialisé automatiquement (on dit : de façon standard) par la *JVM* avec une valeur par défaut (booléen : `false`, caractère : `'\u0000'`, numérique : `0` et référence : `null`) ;
- un paramètre formel est initialisé automatiquement par la *JVM* avec la valeur du paramètre effectif lors de l'appel ;
- une variable locale n'est pas initialisée par la *JVM*, elle doit donc nécessairement l'être par le programmeur avant sa première utilisation.

5) Instanciation

En Java, l'opérateur d'instanciation est **`new`**. Utilisé lors d'un attachement il permet d'associer un nouvel objet à une variable.

On sait que chaque objet est fabriqué à partir d'une (et d'une seule) classe concrète, appelée classe génératrice de l'objet, qui fixe la structure et le comportement de cet objet. Cette classe est le type de l'objet.

Une instance est donc l'objet généré (par la *JVM)* au cours d'une instanciation, conformément à la description donnée par le texte de la classe utilisée lors de cette instanciation.

5.1) Création d'objets

Une expression de création d'objet de la forme **`new`** `X(args)` déclenche une instanciation lorsqu'elle est évaluée. Lors de cette évaluation il se produit un effet de bord qui consiste à réserver un emplacement mémoire adapté à l'objet créé et à l'initialiser. La valeur résultat de cette évaluation est la référence de l'objet ainsi créé.

5.2) Constructeur

Un constructeur est une procédure d'initialisation utilisée conjointement avec l'opérateur **`new`** lors d'une instanciation. Il permet d'effectuer une initialisation personnalisée des champs de l'objet construit.

Un constructeur porte toujours exactement le nom de la classe dans laquelle il est défini, ce n'est pas une méthode mais une procédure et il ne retourne jamais de valeur, donc son entête ne contient pas de type pour la valeur de retour (pas même **`void`**).

Si le programmeur ne définit pas de constructeur dans une classe, le compilateur en synthétise un tout de même : celui-ci n'aura pas d'argument, possédera le même modificateur de visibilité que sa classe et son corps sera réduit à un appel à **`super`**`()`.

Le constructeur n'est exécuté qu'une seule fois par objet, lors de la création de l'objet, et on ne peut pas l'exécuter de nouveau sur l'objet par la suite (on ne peut donc pas « réinitialiser » l'objet, en tout cas pas de cette manière).

5.2.1) Appel de constructeur

On peut appeler un constructeur de trois manières différentes. Dans chaque cas, c'est la liste des paramètres qui permet de sélectionner le bon constructeur :

1. depuis n'importe où, avec l'opérateur **`new`** suivi du nom de la classe et de la liste des paramètres ;
2. depuis le corps d'un autre constructeur de la même classe, avec la forme **`this`**`(...)` ;
3. depuis le corps d'un constructeur d'une sous-classe directe, avec la forme **`super`**`(...)`.

Attention, les appels récursifs ou cycliques de constructeurs sont interdits.

5.2.2) Effet d'un appel

Voici la séquence des actions qui ont lieu lors de l'évaluation de `new X(...)` :

- Action de `new` :
  1. allocation mémoire pour un objet de type `X`
  2. initialisation standard des variables d'instance
- Exécution du corps du constructeur `X(...)` :
  1. création et initialisation des paramètres formels avec les valeurs des paramètres effectifs
  2. si la première instruction est un appel à la forme `this(...)` exécuter le corps du constructeur correspondant de la classe dans laquelle on se trouve puis sauter au point 6
  3. si la première instruction est un appel à la forme `super(...)` exécuter le corps du constructeur correspondant de la super-classe puis continuer au point 4
  4. exécution des expressions d'initialisation situées dans les déclarations d'attributs
  5. exécution des blocs d'initialisation
  6. exécution du reste du corps du constructeur `X(...)`
- Retour de la référence de l'objet

5.3) Création de tableaux

Un tableau à une dimension est un objet créé suite à l'évaluation d'une expression de création de tableaux du genre `new X[]`, où `X` est le type des éléments de base du tableau.

On manipule un tableau de la manière suivante :

- Déclaration d'une variable tableau : `Integer[][] tab;`
- Création d'une instance de tableau :
  1. `tab = new Integer[3][];`
  2. `tab = new Integer[3][5];`
- Initialisation du contenu :
  1. par exemple une boucle `for` de variant `i` et de corps `tab[i] = new Integer[i + 1];` pour finir de créer les tableaux internes ; on obtient alors ici un tableau de forme triangulaire ;
  2. dans cette seconde forme les tableaux internes sont déjà créés, tous de même taille (5) et on obtient un tableau de forme rectangulaire.

Un initialiseur de tableau est une expression entre accolades qui donne à la fois la forme et le contenu du tableau, il s'utilise conjointement à une expression de création de tableau adaptée pour laquelle on ne précise les tailles d'aucune dimension (elles sont calculées par le compilateur en fonction de l'initialiseur).

6) Héritage

6.1) Définition

L'héritage est un mécanisme qui permet, au moment de définir un nouveau type, de réutiliser dans ce nouveau type le code de types déjà existants. On dit que le nouveau type hérite de l'ancien type, ou encore qu'il le dérive ou qu'il l'étend.

Dans le nouveau type, on récupère ainsi la structure (attributs) et le comportement (méthodes) définis dans un ou plusieurs types existants, sans qu'il soit nécessaire de les définir à nouveau. Bien entendu, on peut aussi ajouter du code dans le type en cours d'élaboration pour enrichir cette structure, ou adapter le comportement hérité.

Les mots clés utilisés en Java sont :

- **`extends`** pour signaler qu'une classe récupère le code d'une (seule) autre classe ou qu'une interface récupère le code d'une (ou plusieurs) autre(s) interface(s) ;
- **`implements`** pour signaler qu'une classe récupère le code[3] d'une (ou plusieurs) interface(s).

On remarquera qu'une interface peut hériter de plusieurs interfaces, mais qu'une classe bien qu'elle puisse implémenter plusieurs interfaces, ne peut hériter que d'une seule classe.

Si l'on considère les relations d'héritage entre toutes les classes disponibles (en excluant les interfaces), elles constituent un arbre dont la racine est la classe mère de toutes les classes : `Object`.

6.2) Héritage et constructeurs

Une classe dérivée n'hérite d'aucun constructeur de sa super-classe directe. Ce qui ne l'empêche pas de pouvoir les utiliser s'ils lui sont accessibles.

6.3) Héritage et caractéristiques

Une classe dérivée n'hérite d'aucune caractéristique privée ni de classe (**`static`**). Par contre elle hérite de sa super-classe directe toutes les caractéristiques d'instance publiques et protégées, ainsi que celles avec une accessibilité par défaut (*package private*) si la classe est située dans le même paquetage que sa super-classe.

Une classe dérivée peut déclarer un attribut de même nom que celui d'un attribut de sa super-classe : le nouveau nom masque alors l'ancien. L'accès à un attribut est déterminé à la compilation, à l'aide du type de l'expression représentant la cible de l'appel.

Une classe dérivée peut déclarer une méthode de même signature que celle de l'un de ses super-types directs : la nouvelle méthode implémente, redéfinit ou masque l'ancienne selon que cette dernière est une méthode abstraite (nécessairement d'instance), une méthode concrète d'instance ou une méthode (nécessairement concrète) de classe.

Dans le texte d'une classe héritière, on peut appeler les méthodes accessibles de la super-classe directe en utilisant pour cible le mot clé **`super`**, chose que l'on ne peut pas

3 En fait, des signatures de méthodes et des constantes statiques.

faire depuis le texte d'une classe non héritière.

L'entête[4] d'une méthode qui implémente ou redéfinit une méthode d'une super-classe, s'il déclare lever des exceptions, ne peut pas en lever de moins spécifiques que celles indiquées dans l'entête initial. Et il doit être associé à une visibilité au moins aussi étendue que celle indiquée dans l'entête initial.

Remarque : depuis Java 5, le type de retour d'une méthode implémentée ou redéfinie peut être remplacé par un sous-type Java du type de retour de l'entête initial (on dit que la redéfinition du type de retour est covariante).

6.4) Polymorphisme

6.4.1) Type d'une expression et type de sa valeur

Le type d'une variable est celui qui lui a été donné lors de sa déclaration. Le type d'une expression `e` (CT(`e`) pour *Compile-time Type*) est le type calculable à la compilation à partir du type des constantes, du type des variables et de la signature des opérateurs et méthodes qu'elle contient.

Le type de la valeur de cette expression (RT(`e`) pour *Runtime Type*), à un instant donné de l'exécution, est le type du résultat de son évaluation à cet instant. Par exemple le type de la valeur d'une variable référence, juste après une affectation, est le type générateur de l'objet qu'elle référence.

6.4.2) Polymorphisme et compatibilité entre types

On dit qu'une variable est polymorphe si elle peut référencer, à l'exécution, des objets d'un type différent de celui avec lequel elle a été déclarée.

On a vu qu'un type `S` est compatible (pour l'affectation) avec un type `T` si l'on peut affecter les éléments de `S` à toute variable de type `T`.

En Java, les seuls types compatibles avec un type référence `T` sont les types héritiers de `T`, ainsi que les types interfaces lorsque `T` est le type `Object`. Par conséquent le polymorphisme est contraint par la compatibilité des types. Dit autrement : pour toute variable `v`, RT(`v`) est sous-type de CT(`v`).

7) Mot clé `final`

Une classe déclarée avec le mot clé **`final`** ne peut pas être dérivée (par ex. : la classe `String`). Cela permet au concepteur de s'assurer que l'implantation de la classe ne pourra pas être modifiée par héritage.

Une méthode déclarée avec le mot clé **`final`** ne peut pas être redéfinie. Cela permet de verrouiller la réalisation d'un service dans les sous-classes (par ex. : la méthode `setTime(Date)` dans `java.util.Calendar`). On peut ainsi s'assurer qu'une requête réalisée à l'aide d'un attribut (requête de base) ne pourra pas être redéfinie dans une

4 Signature = nom de la fonction + séquence des types des arguments
Prototype = type de retour + signature
Entête = [modificateurs +] prototype [+ clause **`throws`**]

sous-classe, ce qui se ferait sinon au détriment de l'intégrité de la classe mère. On notera au passage que la classe mère peut alors faire référence à l'attribut ou à la requête indifféremment.

Une variable (attribut ou variable locale ou paramètre formel) déclarée avec le mot clé **`final`** ne peut plus être modifiée après son initialisation (elle est donc constante après initialisation). L'initialisation d'un attribut d'instance constant doit se faire dans un constructeur. Celle d'un attribut de classe constant doit se faire au moment de sa déclaration ou dans un bloc statique d'initialisation. Enfin, celle d'une variable locale doit se faire au moment de sa déclaration. Un paramètre déclaré constant dans l'entête d'une méthode ne peut pas être modifié dans le corps de cette méthode.

8) Erreurs et exceptions

8.1) Erreurs

Lorsqu'une situation d'erreur non récupérable survient dans le système, une instance (généralement indirecte) de `java.lang.Error` est levée. Dans ces circonstances, l'exécution du programme doit être arrêtée et celui-ci doit être corrigé.

8.2) Exceptions

Lorsqu'une situation anormale mais récupérable survient dans le système, une instance (généralement indirecte) de `java.lang.Exception` est levée. Dans ces circonstances, le programme peut gérer la situation en récupérant l'exception et en appliquant un code de remplacement grâce à l'instruction **`try-catch-finally`**.

8.2.1) Exceptions non contrôlées

En général, le compilateur Java oblige le programmeur à déclarer le type des exceptions pouvant survenir dans le corps des méthodes par ajout d'une clause **`throws`** (attention au **`s`**) dans l'entête de la méthode.

Toutefois, il tolère que l'on ne déclare pas les levées du type `RuntimeException` et ses types héritiers : ce sont les exceptions non contrôlées.

8.2.2) Usage des exceptions

Il n'y a que trois manières d'utiliser les exceptions :

1. On sait comment gérer le problème lors de l'exécution : on récupère l'exception et on la traite définitivement (à l'aide de **`try-catch-finally`**).
2. On ne sait pas comment gérer le problème lors de l'exécution : on laisse filer l'exception sans traitement ; l'entête doit contenir une clause **`throws`** si l'exception est contrôlée.
3. On veut effectuer un traitement avant de signaler le problème lors de l'exécution : on récupère l'exception, on la traite, puis on la laisse filer ou on en lève une nouvelle ; l'entête doit contenir une clause **`throws`** si l'exception qui file ou qui est nouvellement levée est contrôlée.

9) Programmation des classes

Pour produire du code de qualité, il est nécessaire de coder nos classes à la fois comme des modules et comme des implémentations de TDA, nous allons voir ci-après ce que cela signifie...

9.1) Paquetages

Un paquetage (*package*) est un système logique de regroupement de types en fonction des catégories de services qu'ils proposent.

En plus des types qu'il contient, un paquetage peut aussi contenir des sous-paquetages. Le nom complet d'un paquetage est donc constitué d'une séquence de noms de paquetages séparés par des points : `java.awt.event` pour le sous-paquetage `event`, du sous-paquetage `awt`, du paquetage `java`.

Lorsqu'un type fait partie d'un paquetage, on doit l'indiquer au système en plaçant une directive **`package`** suivie du nom complet du paquetage sur la première ligne du fichier source. Si, dans un fichier source, il n'y a pas de directive **`package`**, cela signifie que les types définis par ce fichier se situent dans un paquetage anonyme (hors de tout paquetage, en quelque sorte). Cette façon de faire n'est pas recommandée et vous ne la pratiquerez donc pas.

Lorsqu'un type fait partie d'un paquetage, son fichier compilé doit se trouver dans une hiérarchie de répertoires respectant le nom complet du paquetage. L'organisation physique des paquetages (arborescence de répertoires) coïncidera donc avec leur organisation logique (préfixes des noms complets de paquetages).

9.2) Rétention de l'information

Communiquer à ses clients le moins d'information possible sur sa propre structure améliore l'indépendance du code client vis à vis du code fournisseur. Bien sûr il ne s'agit pas de cacher notre code à nos clients, mais plutôt de rendre inaccessible sa structure interne ainsi que nos choix d'implémentation, en proposant un fonctionnement de type boite noire.

9.2.1) Principe d'encapsulation

Le principe d'encapsulation s'énonce ainsi : « Seul l'objet lui-même a accès à ses propres données ».

L'application de ce principe permet :

- de considérer les objets comme des boites noires, manipulables uniquement par l'intermédiaire des méthodes qu'ils proposent, sans pouvoir accéder directement à leurs champs ;
- d'interdire aux clients d'une classe l'accès à sa « cuisine interne », à la manière dont elle fonctionne, à ses choix d'implémentation ;
- d'interdire aux clients d'un paquetage l'accès aux classes utilitaires qu'il peut contenir.

Elle permet aussi de garantir la validité dans le temps d'une classe qui réalise un TDA et

sa spécification (voir plus bas, la section sur les TDA).

9.2.2) Module

Un élément du logiciel qui met en œuvre le principe d'encapsulation s'appelle un module.

Le langage Java offre un mécanisme de gestion de l'accessibilité qui permet de contrôler l'accès aux membres d'un type référence ou d'un paquetage depuis le code client.

Classes et paquetages Java sont des éléments du logiciel. Ce ne seront des modules que si le développeur respecte le principe d'encapsulation et le met en œuvre correctement à l'aide du mécanisme de gestion d'accessibilité disponible dans le langage.

9.2.3) Niveaux d'accessibilité

Le niveau d'accessibilité par défaut (c'est-à-dire sans modificateur d'accessibilité) d'un type est restreint au paquetage. Mais ce type peut être rendu accessible sans restriction à l'aide du modificateur `public`.

Le niveau d'accessibilité par défaut d'un constructeur ou d'un membre d'une classe est restreint au paquetage. Cela signifie que l'on peut y accéder depuis tout code situé dans le même paquetage que le type qui définit ce constructeur ou ce membre.

Ce niveau peut être altéré par l'un des modificateurs suivants :

- `private` restriction au texte de la classe uniquement ;
- `protected` restriction au texte des classes héritières ou membres du même paquetage ;
- `public` aucune restriction.

Le niveau d'accessibilité est calculable à la compilation puisqu'il est situé dans le texte qui décrit la classe. On remarquera que sa granularité est la classe (non l'objet).

On remarquera aussi que, les membres d'une interface étant obligatoirement publics, on ne peut pas modifier leur niveau d'accessibilité.

9.3) Spécification

9.3.1) TDA

Un type de données abstrait (ou TDA) est un modèle mathématique qui définit en même temps un ensemble de valeurs et des opérations applicables sur ces valeurs sans aucune information sur la manière dont les valeurs ou les opérations devraient être implantées en machine. La définition des opérations contient uniquement leur spécification (leur « mode d'emploi »). L'intérêt d'un TDA tient à sa structure algébrique, qui devrait permettre de prouver formellement que toutes ses valeurs se comporteront toujours comme prévu.

Un TDA est utilisable au sein d'un algorithme, ce n'est pas une sorte de type Java. Pour pouvoir utiliser un TDA dans un programme Java, on doit lui substituer un type référence qui respecte à la fois le principe d'encapsulation et la spécification du TDA[5].

5 Voir le polycopié de POO1 pour plus de détails...

9.3.2) Programmation par contrats

Cette technique correspond à l'introduction de la spécification d'un TDA dans le code d'un type référence. Selon ce principe, toute méthode doit être accompagnée d'un contrat qui précise comment elle doit être utilisée par les clients :

- des préconditions indiquent les conditions nécessaires avant que la méthode puisse s'exécuter ;
- des postconditions indiquent les effets produits par l'exécution de la méthode, sous réserve que les préconditions soient satisfaites au préalable et que la méthode se termine.

À cela il faut rajouter les invariants de type qui sont des propriétés que l'on peut systématiquement observer sur les instances de la classe lorsqu'elles sont dans un état stable (c'est-à-dire quand elles sont au repos, qu'elles ne sont pas en train de changer d'état pendant le traitement d'un message).

Ainsi, une classe `C` est correcte si elle respecte le principe d'encapsulation et que les schémas pré-post suivants sont vérifiés :

- $\{pre_C\}$ `new C(params)` $\{post_C \wedge inv_C\}$ pour tout constructeur de `C` et tout jeu de paramètres adéquats `params`.
- $\{pre_m \wedge inv_C\}$ `x.m(params)` $\{post_m \wedge inv_C\}$ pour toute instance `x` de `C`, toute méthode `m` (correspondant au traitement d'un message) de `x`, et tout jeu de paramètres adéquats `params` pour `m`.

9.3.3) Tests unitaires

L'utilisation de tests unitaires permet de s'affranchir de l'implantation *in situ* des postconditions et des invariants. En effet, pour chaque méthode `m` correspondant à un message, on lance une batterie de procédures de test sur une instance `x` de la classe à tester, pour valider la conformité de la réalisation de `m` à sa spécification (qui dit, je le rappelle, que $\{pre_m \wedge inv_C\}$ `x.m(params)` $\{post_m \wedge inv_C\}$) :

- le test construit `x` et un jeu de paramètres pour `m` de sorte que $\{pre_m \wedge inv_C\}$ soit vrai
- le test exécute `x.m(…)`
- le test vérifie que $\{post_m \wedge inv_C\}$ est vrai

Un test unitaire est donc capable de tester que pour des préconditions valides, les postconditions et les invariants le sont aussi après exécution de chaque méthode ; mais attention il ne peut pas affirmer que les préconditions seront toujours respectées dans le code client ! C'est pourquoi les préconditions doivent être codées, alors que les postconditions et les invariants non.

Enfin, un test n'est pas une preuve formelle : il permet au mieux d'augmenter le niveau de confiance du développeur en la correction de la classe...

10) Application Java

10.1) Définition

Une application Java est un logiciel constitué de types Java regroupés en paquetages. Le point d'entrée d'une application est une procédure qui contient l'instruction de démarrage du programme. En Java, son entête est `public static void main(String[] args)`.

Au sein de l'application, la classe publique qui contient le point d'entrée est appelée la classe racine. On peut dire alors qu'une application est l'ensemble de types Java engendré à partir

- d'un élément distingué : la classe racine, qui constitue le type qui sera chargé en premier par la machine Java lors de l'exécution de l'application ;
- d'une relation binaire sur les types Java, définie par « X $\mathfrak{R}$ Y ↔ le type Y est cité dans le texte du type X ».

Le code que représente le texte de la classe racine et celui des types en relation par $\mathfrak{R}$ constitue la totalité du code de l'application.

10.2) Compilation

Les fichiers de code source Java ont pour extension « .java », ils contiennent chacun la description d'un ou de plusieurs types Java dont un, au plus, est public. S'il y a un type Java public dans un fichier source, le nom du fichier coïncide alors avec ce type Java. Si le fichier ne contient pas de type public, son nom peut être quelconque.

Les fichiers de code compilé Java ont pour extension « .class », ils contiennent le code compilé d'un seul type Java, qu'il soit public ou non, et portent exactement le même nom que le type qu'ils contiennent. La compilation du code source produit du code intermédiaire (*bytecode*) qui est :

- indépendant du système hôte et du processeur utilisé ;
- exécutable sur n'importe quel ordinateur doté d'une machine virtuelle Java (*JVM*).

Le compilateur Java est accessible par l'intermédiaire de la commande « javac » disponible sur tout système pour lequel le kit de développement Java (*JDK*) a été installé.

10.3) Exécution

La machine Java est accessible par l'intermédiaire de la commande « java » disponible sur tout système pour lequel le *JDK* ou l'environnement d'exécution (*JRE*) a été installé.

L'exécution de l'application par la machine virtuelle Java charge le code intermédiaire en mémoire en commençant par la classe racine, puis en chargeant de proche en proche tous les autres éléments qui constituent l'application. Elle se poursuit par l'exécution du point d'entrée et se terminera à la mort de tous les fils d'exécution qu'elle aura produits. Le chargement des classes est en fait le reflet de la structure engendrée par la classe racine de l'application et la relation binaire $\mathfrak{R}$; toutefois, il peut se faire de manière laxiste, la *JVM* attendant alors d'avoir effectivement besoin d'une classe avant de la charger réellement.

Chapitre 2 : Compléments

Avec la version 6 de Java apparaissent un certain nombre de nouveautés par rapport à la version 1.4 utilisée en deuxième année de licence...

1) Classes enveloppantes et auto-(un)boxing

Une classe enveloppante est une classe de l'*API* Java permettant d'encapsuler dans des objets non mutables[6] les valeurs d'un type primitif :

```
boolean   →   Boolean
byte      →   Byte
short     →   Short
int       →   Integer
long      →   Long
char      →   Character
float     →   Float
double    →   Double
```

Java 6 met en place un mécanisme de conversion automatique à la compilation, qui permet de passer d'un type primitif à sa classe enveloppante (*auto-boxing*), et réciproquement (*auto-unboxing*), sans action particulière de la part du programmeur.

Pour des raisons de performance, n'utiliser la conversion automatique que pour stocker ou récupérer des valeurs primitives dans ou depuis une collection (comme une liste ou un ensemble, par exemple).

2) Boucle *for* étendue

Depuis sa version 5, Java dispose d'une syntaxe simplifiée pour les boucles `for` dont la variable de contrôle parcourt un tableau ou un `java.lang.Iterable` :

```
for (T v : iterable) instruction;
```

`iterable` est un tableau ou un `Iterable` dont les éléments constitutifs sont de type `T`, et `v` est le nom de la variable locale à la boucle `for`. Cette variable prend la valeur d'un nouvel élément de `iterable` à chaque tour de boucle.

L'interface `java.lang.Iterable` impose à ses descendants de définir une méthode `Iterator iterator()` qui fournit un itérateur sur les éléments de l'objet itérable, c'est-à-dire un objet qui en délivre les éléments un par un, jusqu'à épuisement.

3) Classe `Object`

La classe `Object`, on l'a vu, est à la racine de la hiérarchie d'héritage des classes. Cette classe ne définit aucune variable d'instance ni de classe. Son seul constructeur est sans argument et de corps vide. Mais cette classe assure qu'un certain nombre de méthodes sont bien présentes dans toute classe.

6 C'est-à-dire dont on ne peut pas modifier l'état.

3.1) Méthode `getClass`

Les instances de la classe `java.lang.Class` représentent, lors de l'exécution, les différents types constituant l'application.

Par exemple, le type `Integer` est représenté par une instance de `Class` dont la référence est égale à la constante littérale `Integer.class`.

La méthode `getClass()` retourne l'instance de `Class` qui représente le type générateur de l'objet cible de l'appel, c'est-à-dire la classe dont l'objet cible est une instance directe (vous me suivez ?).

3.2) Équivalence

3.2.1) Méthode `equals`

La méthode **`boolean`** `equals(Object)` implante une relation d'équivalence[7] sur les instances d'une même classe. Sa présence tient essentiellement au fait que les collections (que nous verrons plus loin) l'utilisent pour retrouver les objets qu'elles stockent.

Toute classe, héritant de `Object`, possède donc une méthode `equals`. Telle que définie dans la classe `Object`, cette dernière réalise l'identité de référence (au sens de `==`). Si nécessaire, cette méthode peut être redéfinie pour implanter une relation d'équivalence plus grossière. Ce faisant, vous devrez respecter son contrat tel qu'il est défini dans la classe `Object` : la méthode `equals` doit réaliser une relation d'équivalence, consistante dans le temps (si `x` et `y` ne changent pas d'état dans le temps, alors `x.equals(y)` ne change pas de valeur non plus) et telle que `x.equals(`**`null`**`)` retourne toujours **`false`**.

Coder une méthode `equals` respectant ce contrat n'est pas difficile tant qu'on ne prend pas en charge les conséquences de l'héritage. C'est pourquoi redéfinir `equals` dans une classe finale se fera simplement ainsi :

```
final class A {
    private int a;
    public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof A) {
            A that = (A) other;
            result = (this.a == that.a);
        }
        return result;
    }
}
```

Mais s'il est possible d'hériter, la situation n'est plus si simple et la meilleure façon[8] de définir `equals` est la suivante :

```
class A {
    private int a;
    public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof A) {
```

7 C'est-à-dire une relation binaire réflexive, symétrique et transitive.
8 Voir la présentation faite en cours...

```
            A that = (A) other;
            result = that.canEquals(this) && (this.a == that.a);
        }
        return result;
    }
    public boolean canEquals(Object other) {
        return other instanceof A;
    }
}
class B extends A {
    private int b;
    public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof B) {
            B that = (B) other;
            result = super.equals(that) && (this.b == that.b);
        }
        return result;
    }
    public boolean canEquals(Object other) {
        return other instanceof B;
    }
}
```

Attention : définir `public boolean equals(A other)` reviendrait en fait à surcharger `equals` et non à la redéfinir (voir plus loin la définition de surcharge).

3.2.2) Méthode `hashCode`

Cette méthode réalise une fonction de hachage, comme `equals` elle est définie pour tous les objets.

Cette méthode est utilisée en interne par les classes qui mettent en œuvre la notion de hachage, conjointement avec `equals`, pour stocker et retrouver leurs éléments. C'est le cas de toutes les collections qui contiennent le terme *hash* : `HashSet`, `HashMap`, etc. C'est pourquoi `equals` et `hashCode` sont interdépendantes. La documentation Java donne les recommandations suivantes :

1. si deux instances sont `equals` alors il faut que les valeurs retournées par `hashCode` soient égales ;
2. deux appels de `hashCode` sur un même objet n'ayant pas changé d'état entre ces appels doivent retourner la même valeur.

Remarque : Deux instances qui ne sont pas `equals` pourraient avoir même valeur de `hashCode` (mais ce n'est pas souhaitable).

3.2.3) Équivalence de tableaux

Il y a deux façons de comparer les tableaux entre eux : au sens de l'identité de référence (1) ou au sens de l'équivalence de leur contenu (2, 3 et 4)

1. `t1.equals(t2)` a pour sémantique `t1 != null && t1 == t2`
2. `Arrays.equals(t1, t2)` (lorsque les éléments sont d'un type primitif non flottant) a pour sémantique

`t1 == t2`, ou alors
`t1 != null && t2 != null && t1.length == t2.length`
`&& {∀ i : (t1[i] == t2[i])}`

3. `Arrays.equals(t1, t2)` (lorsque les éléments sont d'un type primitif flottant, `TF` étant la classe enveloppante correspondante) a pour sémantique[9]
   `t1 == t2`, ou alors
   `t1 != null && t2 != null && t1.length == t2.length`
   `&& {∀ i : (new TF(t1[i]).equals(new TF(t2[i]))}`
4. `Arrays.equals(t1, t2)` (lorsque les éléments sont d'un type référence) a pour sémantique
   `t1 == t2`, ou alors
   `t1 != null && t2 != null && t1.length == t2.length`
   `&& {∀ i : (t1[i] == null && t2[i] == null)`
   `|| (t1[i] != null && t1[i].equals(t2[i]))}`

3.3) Clonage

3.3.1) Profondeur du clonage

Un objet pouvant contenir des attributs qui référencent d'autres objets, sa structure interne est représentée par un graphe (et il peut y avoir des cycles).

Le clonage d'un objet consiste à produire un autre objet en tout point semblable au premier. La question est : jusqu'à quel niveau faut-il dupliquer l'état de l'objet de départ ? Le premier niveau (duplication de la structure de l'objet et de ses valeurs) est appelé clonage de surface, les autres niveaux constituent des clonages plus ou moins profonds.

3.3.2) Méthode `clone`

La méthode `Object clone()` permet de dupliquer un objet ; elle doit retourner une copie indépendante de l'objet courant et vérifiant :

- `x.clone() != x`
- `x.clone().getClass() == x.getClass()`
- `x.clone().equals(x) == true`

Attention, le comportement par défaut interdit le clonage ; si l'on souhaite pouvoir cloner les instances d'une classe, il faudra l'indiquer explicitement dans le code de la classe, de la manière suivante :

1. implanter l'interface `java.lang.Cloneable`
2. redéfinir la méthode `clone` dans la classe courante en terminant la chaîne des appels aux méthodes `clone` par celle de la classe `Object`
3. capturer l'exception `CloneNotSupportedException` susceptible d'être levée par un appel à `Object.clone`

on peut de plus (optionnellement) :

9 Deux valeurs NaN sont ainsi considérées comme identiques, mais les valeurs 0.0 et -0.0 ne le sont pas.

- élever le niveau de visibilité de `protected` à `public` (si l'on veut que les clients aussi puissent cloner)
- cloner les éléments mutables de l'état interne du clone (si l'on veut un clonage profond plutôt que de surface)

Remarque : la méthode protégée `Object.clone` effectue un clonage de surface.

4) Types énumérés

4.1) Type énuméré Java

Un type énuméré est un type référence non dérivable dont les seules instances possibles sont celles qui sont énumérées dans la définition de ce type.

Par exemple : `enum Civ { INC, M, MME, MLLE }` définit une classe non dérivable, `Civ`, héritière de `java.lang.Enum`, dont les seules instances sont celles attachées aux quatre constantes déclarées. Toute constante de `Civ` est implicitement publique statique (et donc finale) et on ne peut ni ajouter d'autres instances, ni les cloner.

Les instances d'un type énuméré sont comparables entre elles, elles sont ordonnées selon l'ordre dans lequel elles sont déclarées et elles sont utilisables dans un `switch`. On peut aussi facilement itérer sur les instances d'un type énuméré (voir la méthode `values` plus bas). Enfin, on peut personnaliser les caractéristiques de ces instances et en adapter le comportement comme dans une classe normale...

4.2) Méthodes standard

- `String toString()` donne l'identificateur de la constante attachée à l'instance (son nom, par défaut)
  Ex. : `INC.toString()` → `"INC"`
- `static Civ valueOf(String)` donne l'instance de `Civ` associée à l'identificateur de constante passé en paramètre
  Ex. : `Civ.valueOf("INC")` → `INC` (l'instance)
- `static Civ[] values()` donne un tableau de toutes les instances de `Civ` (pour itérer)
  Ex. (si `tabCiv` a été initialisée avec `Civ.values()`) :
  ```
  for (Civ c : tabCiv) System.out.print(c + " ");
  ```

4.3) Méthodes spéciales

Ces méthodes sont normalement sans intérêt pour les clients, elles peuvent néanmoins être utilisées par le développeur du type énuméré.

- `final String name()` donne l'identificateur tel que défini lors de la déclaration
  Ex. : `INC.name()` → `"INC"`
  La méthode `toString()` retourne la valeur de `name()` par défaut, mais elle peut être redéfinie
- `final int ordinal()` donne la valeur du rang de l'instance dans la séquence de déclaration

Ex. : `INC.ordinal()` → `0`
Utilisée dans les classes `EnumMap` et `EnumSet`

5) Relations de sous-typage

5.1) Sous-typage Java

5.1.1) Définitions

En Java, on dit que `S` est un sous-type (Java) direct de `T` (noté `S` <' `T`) lorsque :

- `S` est une classe qui étend directement la classe `T`
- `S` est une classe qui implante directement l'interface `T`
- `S` est une interface qui étend directement l'interface `T`
- `S` est une interface qui n'étend aucune interface et `T` est `Object`
- `S` est `A[]` et `T` est `B[]`, `A` et `B` sont des types références et `A` <' `B`
- `S` est `Object[]` et `T` est `Object`, `Cloneable` ou `Serializable`
- `S` est `p[]`, `p` est un type primitif et `T` est `Object`, `Cloneable` ou `Serializable`

Aux situations précédentes, on ajoute **`byte`** <' **`short`** <' **`int`** <' **`long`** <' **`float`** <' **`double`** et **`char`** <' **`int`**.

On dit que `S` est un sous-type de `T` (noté `S` <: `T`) si `S` et `T` sont identiques ou bien s'il existe un autre type `Z` tel que `S` soit sous-type direct de `Z` et que `Z` soit sous-type de `T`.

Enfin, on dit que `S` est sous-type propre de `T` (noté `S` < `T`) si `S` est sous-type de `T`, différent de `T`.

5.1.2) Graphes d'héritage et de sous-typage

Il est pratique de représenter les relations entre types à l'aide de schémas.

Le graphe d'héritage complet d'un type Java est un graphe orienté dont les sommets représentent le type lui-même ainsi que ses super-classes et ses super-interfaces, et les arêtes représentent (à l'aide de flèches dont la tête est un triangle blanc) les relations d'héritage entre les différents types (les relations **`extends`** et **`implements`**). Lorsque le schéma ne contient que des classes, il s'agit d'un arbre ; lorsqu'il contient en plus les interfaces, il s'agit d'un graphe orienté sans cycle.

Le graphe de sous-typage complet d'un type Java est un graphe orienté qui se construit à partir du graphe d'héritage de ce type. L'ensemble des sommets est exactement le même que celui du graphe d'héritage pour ce type. Ensuite on ajoute une arête (une flèche à tête normale) pour chaque flèche d'héritage du graphe d'héritage, en conservant la même orientation. Puis on représente la réflexivité de la relation de sous-typage par une flèche qui boucle sur chaque type. Puis on indique la propriété qu'ont les interfaces qui n'héritent d'aucune autre interface d'être des sous-types de `Object`. Puis on ajoute la propriété de transitivité de la relation de sous-typage.

Pour des raisons de llisibilité, on simplifie parfois le graphe de sous-typage en n'y indiquant pas les propriétés de réflexivité et de transitivité.

5.1.3) Compatibilité des types pour l'affectation

5.1.3.1) Type compatible pour l'affectation

En Java, la compatibilité pour l'affectation est complètement définie par la relation de sous-type Java : si `v` est une variable de type `T` et `e` une expression de type `S`, l'affectation `v = e` n'est possible que si `S` <: `T`.

5.1.3.2) Type potentiellement extensible en un type compatible pour l'affectation

Il serait trop restrictif de n'autoriser que les affectations entre types références pour lesquelles `S` <: `T`. Par exemple, l'affectation `Integer v = e;`, où `e` est de type `Object`, pourrait se révéler tout à fait possible si l'évaluation de `e` produisait un `Integer` à l'exécution.

Ainsi, bien qu'incompatible avec `T`, un type `S` peut être potentiellement extensible en un type compatible avec `T` (c'est le cas de `Object` dans l'exemple précédent, qui est potentiellement extensible en `Integer`, qui est compatible avec lui-même). Bien entendu tous les types ne sont pas potentiellement extensibles en un type compatible avec un autre : par exemple `String` n'est pas potentiellement extensible en un type compatible avec `Integer`... voyez-vous pourquoi ?

La notion d'extensibilité potentielle s'exprime ainsi : un type `S` qui n'est pas compatible avec un type `T` est potentiellement extensible en un type compatible avec `T` s'il y a une chance qu'il existe à l'exécution un type `Z` tel que `Z` <: `S` et `Z` <: `T`. Une telle occasion peut survenir lorsque :

- `S` est un super-type Java de `T` (`Z` pourrait être `T`)
- `S` est une classe non finale et `T` est une interface (`Z` pourrait être une sous-classe de `S` qui implémente `T`)
- `S` est une interface et `T` est une classe non finale (`Z` pourrait être une sous-classe de `T` qui implémente `S`)
- `S` et `T` sont des interfaces (`Z` pourrait être une classe qui implémente `S` et `T`)
- `S` est l'interface `Cloneable` ou `Serializable` et `T` est un type tableau (`Z` pourrait être un sous-type Java de `T[]`)
- `S` et `T` sont des types tableaux et le type de base de `S` est potentiellement extensible en un type compatible avec le type de base de `T` (`Z` pourrait être `Y[]`, où `Y` pourrait être un sous-type Java de `S` et de `T`).

Dans le cas où `e` est de type `S` et `S` est potentiellement extensible en un type compatible avec `T`, cela signifie que le type de la valeur de `e` a des chances d'être compatible avec `T`. Dans ces conditions, le compilateur accepte l'affectation de la valeur de `e` à `v`, de type `T`, à condition que le programmeur force le type de `e` à `T` (par transtypage). Par exemple, si `S` est `Comparable` et `T` est `Number`, l'affectation suivante est compilable, bien que `Comparable` ne soit pas sous-type Java de `Number`, car il pourrait exister un type `Z` qui soit à la fois sous-type Java de `S` et de `T` (en fait il en existe même plusieurs : `Integer`, `Double`, ...) :

```
Number v = (Number) e;
```

Mais à l'exécution, cette affectation ne réussira que si, effectivement, le type de la valeur de `e` est bien sous-type Java de `S` et de `T` (par exemple, `Integer`). Dans le cas contraire, une `ClassCastException` sera levée.

5.1.4) Conversions et transtypage

On parle de conversion d'un type S vers un type T lorsque le compilateur remplace une valeur de S par une valeur adaptée de T lorsque S et T sont des types primitifs, ou si l'un des deux est primitif et l'autre le type enveloppant correspondant. Dans le cas où S et T sont deux types références, il y a conversion lorsque le compilateur traite une expression de type S comme si elle était de type T, bien qu'il n'y ait pas de remplacement de valeur.

Ces conversions surviennent dans le cadre d'une affectation ou d'un passage de paramètre lors d'une invocation de méthode ou d'activation de constructeur.

Elles peuvent être faites automatiquement par le compilateur (*upcast* seulement), ou à la demande du programmeur lors d'un transtypage.

5.1.4.1) Types primitifs

Par exemple lorsqu'on veut stocker la constante littérale 5 (de type **`int`**) dans une variable d'un type flottant, une conversion entre type entier et type flottant est effectuée.

Dans ce cas, le compilateur vérifie que les types sont compatibles et produit une erreur si ce n'est pas le cas. La conversion ne lèvera jamais d'exception à l'exécution.

Conversions élargissantes

Certaines conversions sont dites élargissantes. On peut considérer qu'elles correspondent à un élargissement du type, c'est-à-dire que le domaine des valeurs du type d'arrivée est plus vaste que celui des valeurs du type de départ. Il s'agit de :

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

Ces conversions sont exécutées automatiquement. Sans action spécifique du programmeur, elles sont donc implicites. Par exemple l'instruction `double x = 5;` provoque une conversion élargissante (implicite) de **`int`** vers **`double`**.

Elles n'entraînent aucune perte de précision lorsqu'on passe d'un type entier à un autre type entier, mais peuvent en entraîner une lorsqu'on passe d'un type entier à un type flottant.

Conversions rétrécissantes

Certaines conversions sont dites rétrécissantes. On peut voir cela comme un rétrécissement du domaine des valeurs du type de départ. Il s'agit de :

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

Ces conversions nécessitent une action particulière de la part du programmeur, elles sont donc rendues explicites au moyen d'une opération de transtypage (*cast*). Par exemple l'instruction `int x = (int) 5.0;` provoque une conversion rétrécissante qui est explicitement demandée par l'opération de transtypage `(int)`.

Ce genre de conversion peut entraîner des pertes de précision, voire de magnitude.

5.1.4.2) Types références

Lors d'une affectation `v = e;`, le compilateur vérifie toujours la compatibilité (ou éventuellement la compatibilité potentielle) du type de `e` avec celui de `v`. Notons S le type de `e` et T celui de `v`.

Si le compilateur découvre que les types sont compatibles (S <: T), la vérification s'arrête là, et on parle de conversion élargissante.

Si le compilateur peut prouver que S n'est ni compatible avec T ni potentiellement extensible en un type compatible avec T, la compilation s'arrêtera sur une erreur.

Enfin, si S est potentiellement extensible en un type compatible avec T (par exemple parce que le développeur a forcé le type de `e` par une opération de transtypage idoine), le compilateur insère une instruction `checkcast` dans le *bytecode* afin que soit vérifiée, à l'exécution, la compatibilité effective du type de la valeur de `e` avec T. Si cette instruction devait échouer, une `ClassCastException` serait levée à l'exécution.

Les règles de conversion sont :

1. Une conversion élargissante de S vers T a toujours lieu (au moins implicitement) lorsque S <: T
2. Une conversion rétrécissante de S vers T est possible (explicitement) lorsque S est potentiellement extensible en un type compatible avec T

5.1.5) Opérateur **`instanceof`**

Soient `A` un type Java et `e` une expression. À l'exécution `e instanceof A` vaut **`true`** si et seulement si `e != null` et l'évaluation de `(A) e` ne lèvera pas d'exception de type `ClassCastException`, c'est-à-dire si RT(`e`) <: `A`.

Si le type de `e` n'est pas compatible avec `A`, ni même potentiellement extensible en un type compatible avec `A`, alors l'opérateur **`instanceof`** ne peut pas être utilisé (erreur de compilation).

L'intérêt de cet opérateur, vous l'aurez compris, est de garantir que le transtypage sera possible à l'exécution. Néanmoins, quel que soit le type `A`, l'expression **`null`** `instanceof A` vaudra toujours **`false`**.

5.2) Sous-typage en théorie des types

5.2.1) Définition

En théorie des types, il existe une notion de sous-type qui n'a rien à voir avec le mécanisme d'héritage parce qu'on ne se place pas au même niveau conceptuel. L'héritage concerne la réutilisation du code et il se trouve qu'en Java, on définit aussi une notion de sous-type (que nous avons appelé « sous-type Java ») relativement à l'héritage. Or la notion de sous-type en théorie des types concerne, elle, la substituabilité (propriété de l'héritage) et la compatibilité des spécifications (propriété indépendante de l'héritage).

Le programmeur habile concevra donc des types de données qui utilisent l'héritage d'une manière compatible avec la relation de sous-typage de la théorie des types. En clair, il ne faut pas faire n'importe quoi comme, par exemple, dire que le type VéhiculeÀMoteur hérite du type Moteur sous prétexte que l'on va ainsi récupérer du code déjà fourni et que c'est un gain de temps : un véhicule n'est pas une sorte de moteur puisqu'il ne pourrait pas être employé directement en tant que tel dans un autre véhicule.

À côté de la notion de sous-type Java, apparaît donc celle de sous-type de la théorie des types, relation que nous appellerons « sous-typage (au sens) de Liskov » car c'est Barbara Liskov qui l'a caractérisée efficacement avec son principe de substitution (1993).

5.2.2) Principe de substitution de Liskov

Un type `S` est un sous-type de Liskov d'un type `T` (notation : $S \subset T$) si, dans tout programme `P`, on peut utiliser une valeur de `S` partout où une valeur de `T` est attendue, sans modifier la sémantique de `P`.

5.2.3) Utilisation de l'héritage

Une relation d'héritage entre deux types Java n'est autorisée que si elle ne contrevient pas au PSL. Formellement, cette règle s'énonce ainsi :

$$S \not<: T \Rightarrow \neg(S <: T)$$

En pratique, pour que deux types Java S et T tels que S <: T respectent le PSL, il faudra donc que l'invariant de S restreigne celui de T, et que les contrats des méthodes de S soient des raffinements des contrats hérités de T.

Explications :

- inv(S) restreint inv(T) : $inv(S) \Rightarrow inv(T)$
- contrat(m_S) raffine contrat(m_T) : m_S redéfinit m_T avec $(pre_{mT} \Rightarrow pre_{mS}) \wedge (post_{mS} \Rightarrow post_{mT})$

Cette règle permet de décider quand utiliser le mécanisme d'héritage et quand ne pas l'utiliser. Elle est contraignante mais il existe d'autres moyens de réutiliser du code, comme la délégation par exemple.

6) Mécanisme d'invocation de méthode

6.1) Compilation d'une instruction d'appel

Dans le code source, une instruction d'appel de méthode n'est compilable que si la signature de la méthode existe au niveau du type CT(cible) (parce qu'elle a été définie dans ce type, ou bien dans l'un de ses super-types) et qu'elle est accessible depuis le site d'appel.

Le compilateur insère alors dans le byte-code une instruction d'invocation de méthode en rapport avec ce type et la signature de la méthode appelée. Notez bien que s'il s'agit d'un appel de méthode de classe, la cible est une classe et son type est... elle-même.

Voici l'algorithme de recherche de la signature lors de la compilation d'une instruction d'appel :

1. Notons T_0 le type de la cible
2. Notons T_1, ..., T_n les types des arguments
3. On recherche le n-uplet de types $(S_1,\ldots,S_n)$ défini par :
   $(S_1,\ldots,S_n) = \min\{(X_1,\ldots,X_n) \mid T_1<:X_1,\ldots,T_n<:X_n, m(X_1,\ldots,X_n)\in T_0\}$
4. On obtient ainsi la signature $SIG = m(S_1,\ldots,S_n)$ ou une erreur si aucune signature n'a été trouvée

6.2) Exécution d'une méthode

À l'exécution, la méthode effectivement sélectionnée sera celle dont la signature a été précisément déterminée à la compilation. Si une telle méthode n'est pas trouvée au moment de l'exécution, une erreur (`NoSuchMethodError`) se produit.

Pour une méthode d'instance qui n'est ni privée ni appelée avec **`super`**, la machine Java exécute toujours la méthode la plus spécifique qu'elle peut trouver pour l'objet, c'est-à-dire la méthode définie dans le type « le plus bas possible » dans la hiérarchie d'héritage du type de l'objet. Ce mécanisme s'appelle la liaison dynamique, et ces méthodes sont dites virtuelles.

Dans le cas d'une méthode non virtuelle (statique, ou privée, ou appelée avec **`super`**), la méthode exécutée correspond exactement à celle qui a été trouvée lors de la compilation.

6.3) Surcharge de méthode

La surcharge de méthode est la possibilité de définir plusieurs méthodes avec le même nom, au sein d'une même classe. Comme ces méthodes doivent se différencier par leur signature, chaque méthode impliquée dans la surcharge doit posséder une séquence de types d'arguments différente de celle des autres méthodes de la surcharge. Pour une surcharge portant sur le nom `m`, on dit par abus de langage que « la méthode `m` est surchargée » (en réalité, c'est le nom `m` qui est surchargé).

Du fait du mécanisme de liaison dynamique, la présence de surcharge peut donner des résultats qui surprendraient un œil non averti (voir exemples en cours et exercices en TD) !

Il peut aussi y avoir une ambiguïté lors de la compilation car `min{(X1,…,Xn) | T1<:X1, …,Tn<:Xn,m(X1,…,Xn)∈T0}` n'est pas toujours bien défini (voir la présentation faite en cours).

7) Retour sur le mot clé `protected`

Des quatre niveaux de contrôle d'accès, **`protected`** est le plus délicat à cerner. La description donnée au précédent chapitre est celle que l'on trouve généralement dans la littérature ; elle n'est pas fausse mais elle est incomplète.

En réalité, un membre **`protected`** `m` défini dans une classe `A`, elle-même définie dans un paquetage `p`, est accessible sous la forme `x.m` si cet appel se situe dans le texte d'une classe du paquetage `p`, ou bien s'il se situe (hors de `p`) dans le texte d'une sous-classe `B` de `A` qui est « responsable de l'implémentation de `x` ». Les cas où `B` est responsable de l'implémentation de `x` sont ceux pour lesquels `x` est la variable **`this`** ou la variable **`super`**, ou bien tels que CT(`x`) <: `B`.

On se convaincra facilement que cette définition est compatible avec celle donnée plus haut, et qu'elle est un peu moins restrictive.

Chapitre 3 : Flux

1) Définition

Un flux est un objet transportant des données séquentiellement, en quantité indéterminée, pour permettre au programme de communiquer avec l'extérieur du système, ou avec le système, par échange de données de manière standardisée.

Les objets avec lesquels le programme communique peuvent être :

- une ressource externe (fichiers)
- de la mémoire (chaînes, tampons, tableaux, ...)
- un connecteur réseau (*socket*)
- un tube de communication interprocessus (*pipe*)
- un autre flux

Il y a énormément de classes de flux en Java, il est donc intéressant de les classer pour pouvoir mieux les appréhender.

2) Taxonomie

Il y a trois manières de classer les flux : selon leur mode d'accès aux données (lecture ou écriture), selon le type de données qu'ils transportent (caractères ou octets) et selon leur mode opératoire (flux primaires ou de traitement).

2.1) Mode d'accès

Les flux d'entrée sont des flux permettant au programme de lire des données issues d'une source de données.

Une zone de mémoire accédée en lecture ou un fichier physique ouvert en lecture sont des exemples concrets de sources de données : ce sont les éléments qui fournissent les données au flux lecteur qui les utilise. L'usage de ces sources de données à travers un flux lecteur uniformise de fait l'accès aux données qu'elles contiennent.

Les flux de sortie sont des flux permettant au programme d'écrire des données dans un collecteur de données.

Une zone de mémoire accédée en écriture ou un fichier physique ouvert en écriture sont des exemples concrets de collecteurs de données : ce sont des éléments qui récoltent les données que fournit le flux écrivain qui les utilise. Ici encore, l'usage de ces collecteurs de données à travers un flux écrivain uniformise de fait la production des données.

2.2) Type de données véhiculées

Les flux de caractères sont des flux permettant au programme de traiter des données textuelles par manipulation de caractères.

Un caractère est une valeur entière codée sur deux octets (en UTF-16) ; la valeur -1, qui

ne représente aucun caractère, indique la fin du flux.

Les flux binaires sont des flux permettant au programme de traiter des données brutes par manipulation d'octets.

Un octet est une valeur entière non signée comprise entre 0 et 255. Toutefois, la fin d'un flux binaire est indiquée par la valeur -1, qui n'est pas une valeur permise dans cet intervalle. C'est pourquoi (voir plus loin) les méthodes abstraites de lecture et d'écriture d'octets retournent un `int` ou prennent un `int` en argument : si la valeur représentée correspond bien à un octet (et pas à la marque de fin de flux), elle est stockée sur les bits de poids faible de l'entier.

2.3) Mode opératoire

Les flux primaires sont des flux permettant au programme d'accéder à une ressource fournie directement par le système.

De telles ressources seront par exemple des tableaux de caractères ou d'octets, des chaînes de caractères, des fichiers, des tubes, etc.

À l'opposé, les flux de traitement sont des flux permettant au programme d'ajouter un traitement supplémentaire à un autre flux, lors de son utilisation, par connexion sur ce dernier d'un flux de même mode d'accès.

Par exemple, si l'on veut faire une lecture bufferisée dans un fichier (pour économiser les accès au disque physique) on commence par créer un flux de lecture dans un fichier, qui est un flux primaire connecté à une source de données fournie par le système (ici un descripteur de fichier). Ensuite on crée un flux de traitement de même nature que le premier (un lecteur, donc) qui prend pour source de données le flux précédent. Finalement, lorsqu'on commandera la lecture d'une donnée :

- soit le second flux l'a déjà lue dans le premier et il nous la délivre immédiatement ;
- soit le second flux ne l'a pas encore lue et il effectue la lecture d'un bloc de données dans le premier, ne nous retournant que la première donnée et gardant les suivantes du bloc en réserve pour les prochaines commandes de lecture.

Les flux primaires ne possèdent que des constructeurs admettant pour argument des ressources systèmes (nom de fichier, tableau, chaîne de caractères, etc.) alors que les constructeurs des flux de traitement n'admettent que des flux de même mode d'accès que le leur.

3) Onomastique

Les flux se répartissent en quatre grandes familles qui se déduisent de la combinaison des axes de classement selon le mode d'accès et selon la nature des données :

- `java.io.Reader` : classe mère des flux d'entrée de caractères ;
- `java.io.Writer` : classe mère des flux de sorties de caractères ;
- `java.io.InputStream` : classe mère des flux d'entrée binaires ;
- `java.io.OutputStream` : classe mère des flux de sortie binaires.

Ces quatre classes sont abstraites.

Pour se repérer dans la jungle des flux, il est utile de comprendre que leur nom est construit sur le modèle « FonctionCatégorie ». La fonction détermine la particularité du flux et la catégorie est le nom de l'une des quatre classes principales. On peut ainsi très facilement détecter le rôle des différents flux seulement par leur nom.

Par exemple, le nom d'un flux écrivain véhiculant des données binaires contiendra le mot « *OutputStream* » alors qu'il contiendra le mot « *Writer* » s'il s'agit d'un flux écrivain véhiculant des caractères.

Ainsi `BufferedReader` est le type des flux d'entrée de caractères qui utilisent un buffer interne pour la lecture, `FileInputStream` est le type des flux binaires qui lisent leurs données dans un fichier, `StringWriter` est le type des flux qui écrivent des caractères dans une chaîne de caractère, `DataOutputStream` est le type des flux binaires qui écrivent la représentation mémoire des valeurs primitives, etc.

3.1) Les flux d'entrée binaires

- Flux primaires
  - ressources : mémoire
    `ByteArrayInputStream` : lecture à partir d'un tableau de **`byte`**
  - ressource : descripteur de fichier
    `FileInputStream` : lecture à partir d'un fichier (binaire) précisé par son nom (`String`) ou une abstraction de son chemin d'accès (`File`)
  - ressource : tube de communication (écrivain)
    `PipedInputStream` : lecture d'octets à partir d'un tube écrivain (`PipedOutputStream`)
- Flux de traitement
  - traitement : tampon de lecture
    `BufferedInputStream`
  - traitement : réintégration possible d'octets
    `PushbackInputStream`
  - traitement : reconstruction de données
    `DataInputStream` (types primitifs ou chaînes)
    `ObjectInputStream` (objets)
  - traitement : concaténation de flux
    `SequenceInputStream`
  - traitement : classe de factorisation pour les flux de traitement
    `FilterInputStream`

3.2) Les flux d'entrée de caractères

- Flux primaires
  - ressources : mémoire
    `CharArrayReader` lecture à partir d'un tableau de **`char`**
    `StringReader` lecture à partir d'une `String`
  - ressources : descripteur de fichier
    `FileReader` lecture à partir d'un fichier (de texte) précisé par son nom (`String`) ou une abstraction de son chemin d'accès (`File`)

  - ressources : tube de communication (écrivain)
    `PipedReader` lecture de caractères à partir d'un tube écrivain (`PipedWriter`)

- Flux de traitement
  - traitement : lecture bufferisée
    `BufferedReader` (tampon de lecture ligne par ligne)
    `LineNumberReader` (... avec comptage des lignes)
  - traitement : conversion d'un `InputStream` en `Reader`
    `InputStreamReader`
  - traitement : tampon de retour (réintégration possible de caractères)
    `PushbackReader`
  - traitement : classe de factorisation pour les flux de traitement
    `FilterReader`

3.3) Les flux de sortie binaires

- Flux primaires
  - ressources : mémoire
    `ByteArrayOutputStream` écriture dans un tableau de **`byte`**
  - ressources : fichier
    `FileOutputStream` écriture dans un fichier (binaire) précisé par son nom (`String`) ou une abstraction de son chemin d'accès (`File`)
  - ressources : tube de communication
    `PipedOutputStream` écriture d'octets dans un tube lecteur (`PipedInputStream`)

- Flux de traitement
  - traitement : écriture bufferisée (tampon d'écriture)
    `BufferedOutputStream`
  - traitement : déconstruction de données
    `DataOutputStream` (types primitifs, chaînes)
    `ObjectOutputStream` (objets)
  - traitement : filtrage
    `FilterOutputStream`

3.4) Les flux de sortie de caractères

- Flux primaires
  - ressources : mémoire
    `CharArrayWriter` écriture dans un tableau de **`char`**
    `StringWriter` écriture dans une `String`
  - ressources : fichier
    `FileWriter` écriture dans un fichier (de texte) précisé par son nom (`String`) ou une abstraction de son chemin d'accès (`File`)
  - ressources : tube de communication
    `PipedWriter` écriture de caractères Unicode dans un tube lecteur (`PipedReader`)

- Flux de traitement
  - traitement : écriture bufferisée (tampon d'écriture)
    `BufferedWriter`
  - traitement : conversion d'un `OutputStream` en `Writer`
    `OutputStreamWriter`
  - traitement : écriture formatée de valeurs de tout type
    `PrintWriter`
  - traitement : filtrage
    `FilterWriter`

4) Utilisation

Attention, la plupart des opérations sur les flux (sauf les méthodes `print`) lèvent une `IOException` lorsque survient une erreur d'entrée/sortie.

4.1) Ouverture et fermeture

Il n'est pas nécessaire d'ouvrir un flux car le simple fait de construire l'objet ouvre le flux.

En revanche, il est nécessaire de fermer un flux après utilisation, pour libérer les ressources qui lui ont été allouées par le système et pour assurer le vidage des tampons en écriture. Les classes de flux implémentent donc l'interface `Closeable`.

Toutefois, certains flux ne doivent jamais être fermés, il s'agit des canaux standards d'entrée sortie : `System.out`, `System.in` et `System.err`.

4.2) Lecture

Une opération de lecture se fait à l'aide de la méthode `read` qui est surchargée plusieurs fois dans chaque classe de base (`Reader` et `InputStream`).

L'une de ces versions représente l'opération élémentaire de lecture, il s'agit de `int read()` pour `InputStream` et de `int read(char[], int, int)` pour `Reader`. Ces deux méthodes sont abstraites dans leur classe de base et devront être réalisées en fonction des spécificités de chaque sous-classe de flux lecteur. Toutes les autres versions de ces méthodes sont concrètes dans leur classe de base et elles délèguent la lecture effective à l'opération élémentaire. Si les sous-classes ont besoin de plus d'efficacité, elles peuvent toujours redéfinir l'opération élémentaire, impactant ainsi toutes les versions de `read` qu'elles contiennent.

4.2.1) Lecture d'octets (`InputStream`)

- `abstract int read() throws IOException`
  - lit un seul octet dans le flux, et le stocke dans les huit bits de poids faible de l'entier retourné
  - retourne -1 si la fin du flux est atteinte
- `int read(byte[] buf, int offset, int length) throws IOException`
  - lit au plus `length` octets dans le flux et les range dans `buf` entre les positions `offset` et (au plus) `offset` + `length` - 1

  - les éléments de `buf` entre 0 et `offset` - 1 sont inchangés
  - retourne le nombre d'octets lus, ou -1 si la fin du flux est atteinte
- `int read(byte[] buf) throws IOException`
  - équivaut à `read(buf, 0, buf.length)`

L'extrait de code suivant est une manière (il y en a d'autres) de lire des octets par paquets de `max` dans un `InputStream` :

```
int max = ... // taille du tableau buf
InputStream input = ... // création et ouverture
byte[] buf = new byte[max];
try {
    int n = input.read(buf);
    while (n != -1) {
        for (int i = 0; i < n; i++) {
            ... // traitement de buf[i]
        }
        n = input.read(buf);
    }
} catch (IOException e) {
    // problème lors de la lecture
} finally {
    try {
        input.close();
    } catch (IOException e) {
        // problème lors de la fermeture
    }
}
```

4.2.2) Lecture de caractères (`Reader`)

- `int read() throws IOException`
  - lit un seul caractère dans le flux, et le stocke dans les deux octets de poids faible de l'entier retourné
  - retourne -1 si la fin du flux est atteinte
  - Rq : cette méthode n'est pas très efficace car elle appelle `read(buf, 0, 1)` !
- `abstract int read(char[] buf, int offset, int length) throws IOException`
  - lit au plus `length` caractères dans le flux et les range dans `buf` entre les positions `offset` et (au plus) `offset` + `length` - 1
  - les éléments de `buf` entre 0 et `offset` - 1 sont inchangés
  - retourne le nombre de caractères lus, ou -1 si la fin du flux est atteinte
- `int read(char[] buf) throws IOException`
  - équivaut à `read(buf, 0, buf.length)`

L'extrait suivant est une manière (il y en a d'autres) de lire des caractères un à un dans un `Reader` :

```
Reader input = ... // création et ouverture
try {
    int n = input.read();
    while (n != -1) {
        char c = (char) n;
        ... // traitement de c
```

```
            n = input.read();
        }
    } catch (IOException e) {
        // problème lors de la lecture
    } finally {
        try {
            input.close();
        } catch (IOException e) {
            // problème lors de la fermeture
        }
    }
```

4.2.3) Lecture de lignes de texte (`BufferedReader`)

On peut lire des lignes entières (séparées par des marques de fin de ligne) dans un `BufferedReader` avec la méthode `String readLine() throws IOException`.

```
BufferedReader input = ... // création et ouverture
try {
    String line = input.readLine();
    while (line != null) {
        ... // traitement de line (sans marque de fin de ligne)
        line = input.readLine();
    }
} catch (IOException e) {
    // problème lors de la lecture
} finally {
    try {
        input.close();
    } catch (IOException e) {
        // problème lors de la fermeture
    }
}
```

4.3) Écriture

Une opération d'écriture se fait à l'aide de la méthode `write` qui est, elle aussi, surchargée plusieurs fois dans chaque classe de base (`Writer` et `OutputStream`).

Comme dans le cas de la lecture, l'une des versions (abstraite) représente une opération élémentaire d'écriture (`void write(char[], int, int)` pour `Writer` et `void write(int)` pour `OutputStream`) et les autres y font appel.

4.3.1) Écriture d'octets (`OutputStream`)

- `abstract void write(int b) throws IOException`
  - écrit dans le flux l'octet de poids faible de b
- `void write(byte[] buf, int offset, int length) throws IOException`
  - écrit dans le flux les octets de `buf` à partir de `offset` et sur une longueur de `length`
- `void write(byte[] buf) throws IOException`
  - équivaut à `write(buf, 0, buf.length)`

L'extrait de code suivant est une manière (il y en a d'autres) d'écrire des octets par paquets de `max` dans un `OutputStream` :

```
int max = ... // taille du tableau buf
OutputStream output = ... // création et ouverture
try {
    byte[] buf = new byte[max];
    while (...) { // tant qu'il y a des données à écrire
        int n = ... // nb de données à écrire (n <= max)
        ... // initialiser buf[0..n-1]
        output.write(buf, 0, n);
    }
} catch (IOException e) {
    // problème lors de l'écriture
} finally {
    try {
        output.close();
    } catch (IOException e) {
        // problème lors de la fermeture
    }
}
```

4.3.2) Écriture de caractères (`Writer`)

- `void write(int c) throws IOException`
  - écrit dans le flux les deux octets de poids faible de `c`
- `abstract void write(char[] text, int offset, int length) throws IOException`
  - écrit dans le flux les caractères de `text` à partir de `offset` et sur une longueur de `length`
- `void write(char[] text) throws IOException`
  - équivaut à `write(text, 0, text.length)`
- `void write(String s, int offset, int length) throws IOException`
  - écrit dans le flux les caractères de `s` à partir de `offset` et sur une longueur de `length`
- `void write(String s) throws IOException`
  - équivaut à `write(s, 0, s.length())`

L'extrait suivant est une manière (il y en a d'autres) d'écrire des caractères un à un dans un `Writer` :

```
Writer output = ... // création et ouverture
try {
    int c;
    while (...) { // tant qu'il y a des données à écrire
        ... // initialiser c
        output.write(c);
    }
} catch (IOException e) {
    // problème lors de l'écriture
} finally {
    try {
```

```
        output.close();
    } catch (IOException e) {
        // problème lors de la fermeture
    }
}
```

4.3.3) Écriture formatée dans un `PrintWriter`

- **`void`** `print({`**`boolean`**`|`**`char`**`|`**`int`**`|`**`long`**`|`**`float`**`|`**`double`**`|`**`char`**`[]|String|Object})`
  - écrit son argument dans le flux sous forme de texte
- **`void`** `println([`**`boolean`**`|`**`char`**`|`**`int`**`|`**`long`**`|`**`float`**`|`**`double`**`|`**`char`**`[]|String|Object])`
  - écrit son argument dans le flux sous forme de texte et ajoute une marque de fin de ligne

Dans l'exemple suivant, `output` est connecté à un fichier de texte dont les caractères constitutifs s'interprètent comme une séquence de nombres :

```
PrintWriter output = ... // création et ouverture
double d;
while (... && !output.checkError()) {
    ... // donner une valeur à d
    output.println(d);
}
if (output.checkError()) {
    // problème lors de l'écriture
}
try {
    output.close();
} catch (IOException e) {
    // problème lors de la fermeture
}
```

4.4) Remarques

1. L'accès à un fichier doit toujours être bufferisé pour accélérer le traitement.
2. La seule façon de s'assurer que le système a terminé d'écrire dans un collecteur consiste à vider le tampon système (**`void`** `flush()`), ou à fermer le flux (**`void`** `close()`).
3. Dans un `PrintWriter` et dans un `PrintStream` :
   - on peut commander un *flush* automatique après chaque ligne écrite
   - il n'y a jamais de levée d'`IOException` (mais il faut tester le bon déroulement des opérations par appel à `checkError`).
4. L'entrée standard doit être manipulée comme un flux de caractères pour pouvoir être correctement traitée (`System.in` est un `InputStream`).
   - Pour lire des caractères :
     `new` `InputStreamReader(System.in)`
   - Pour lire des lignes :
     `new` `BufferedReader(`**`new`**` InputStreamReader(System.in))`

5. La sortie standard est un `PrintStream` mais il suffit de la manipuler comme un `PrintWriter` pour y écrire des caractères.
   - Pour écrire des caractères (avec *flush* automatique en fin de ligne) : `new PrintWriter(System.out, true)`

5) Classe `Console`

La classe `java.io.Console` existe depuis Java SE 6, elle permet d'accéder à la console associée à la *JVM*. Celle-ci est une instance accessible depuis la classe `java.lang.System` par appel à la méthode statique `Console console()`, qui retourne `null` lorsque aucune console n'est attachée à la *JVM* (par ex. sous Éclipse !).

5.1) Lecture sur la console

On obtient le flux d'entrée de la console par appel de la méthode `Reader reader()` qui retourne `System.in` s'il n'y a pas eu redirection de l'entrée.

On peut lire directement sur la console au moyen de diverses méthodes :

- `String readLine()`
  lit une ligne de texte sur l'entrée de la console
- `String readLine(String fmt, Object... args)`
  affiche un prompt (suivant les règles de formatage définies par les paramètres) puis lit une ligne de texte sur l'entrée de la console
- `char[] readPassword()`
  lit un mot de passe sans écho sur la console
- `char[] readPassword(String fmt, Object... args)`
  affiche un prompt (suivant les règles de formatage définies par les paramètres) puis lit un mot de passe sans écho sur la console

5.2) Écriture sur la console

On obtient le flux de sortie de la console par appel à `PrintWriter writer()` qui retourne `System.out` s'il n'y a pas eu redirection de la sortie.

On peut écrire directement sur la console par appel à la méthode :

```
Console format(String format, Object... args)
```

qui écrit son argument sous forme de texte respectant les règles de format imposées. Cette méthode retourne l'objet cible pour pouvoir chaîner les appels de la manière suivante : `csl.format(...).format(...)`.

La notation `Object...` signale un nombre variable d'arguments. Une telle notation ne peut apparaître, dans un entête, qu'une seule fois et en dernière position. Dans le corps de la méthode `format`, `args` est un tableau d'éléments de type `Object`, toujours différent de `null`, et éventuellement de longueur 0 (si pas d'argument lors de l'appel).

Par exemple :

```
String fm = "Salut %s ! Bien réveillé en ce %2$te/%2$tm/%2$tY ?";
Calendar c = new GregorianCalendar();
String name = "Mr Py";
System.console().format(fm, name, c);
```

affichera sur la console : `Salut Mr Py ! Bien réveillé en ce 04/09/2023 ?`

6) Classe `File`

La classe `java.io.File` est la classe des chemins abstraits. Une instance de cette classe définit un chemin abstrait pouvant représenter un fichier ou un répertoire, qui existe déjà ou pas encore. On peut donc le manipuler au sein du programme, et ceci de façon indépendante de la plate-forme.

Par exemple, l'expression de création :

```
new File("/usr/java/jdk1.6.0_14/docs/api/index.html")
```

retourne un objet qui représente l'*i-node* du fichier index de la documentation Java sur un système Linux. On obtiendrait un objet équivalent sous Windows avec l'une des deux expressions suivantes :

```
new File("C:\\PROGRA~1\\Java\\jdk1.6.0_14\\docs\\api\\index.html")
new File("file:///C:/Program%20Files/Java/jdk1.6.0_14/docs/api/index.html")
```

Une fois l'instance de `File` créée, on peut physiquement créer le fichier ou le répertoire correspondant à l'aide de diverses méthodes (`createNewFile`, `mkdir`, `mkdirs`, etc.).

La classe `File` regorge d'outils en tous genres : renommage, suppression, lecture ou modification d'attribut, gestion des répertoires, espace disponible sur les disques, création de fichiers temporaires, ...

Chapitre 4 : Généricité

1) Notion de généricité

La généricité représente une avancée importante pour la sûreté du typage ; elle facilite aussi grandement l'écriture du code source.

Elle est apparue avec Java 5. Auparavant, on pouvait la simuler en utilisant la compatibilité universelle de la classe `Object`, mais ce n'était qu'un pis-aller qui ne pouvait en aucun cas se mesurer à la notion de généricité.

1.1) Définition

La généricité est le principe qui rend possible la définition d'éléments (types, méthodes), dans le code source, paramétrés par des variables représentant des types inconnus au moment de l'écriture de leur définition.

1.2) Intérêt

Cela permet au programmeur d'augmenter sensiblement la fiabilité, la modularité et la lisibilité de son code.

2) Types génériques

Pour dire les choses rapidement, un type générique est un type défini relativement à un ou plusieurs paramètres de types placés entre chevrons après le nom du type, comme dans `Stack<E>`. La définition de `Stack<E>` est celle d'un type « à trous », dont les trous ont tous `E` pour étiquette, ce symbole représentant, dans la définition, un type Java fixe (ce type est inconnu pour l'instant mais ce sera le même pour toutes les occurrences de `E`).

On peut ensuite construire autant de types de piles que l'on veut à partir de `Stack<E>` en remplaçant `E` par n'importe quel type (référence) existant ; on aura ainsi des piles d'entiers (`Stack<Integer>`) des piles de chaînes de caractères (`Stack<String>`), etc. Ces derniers types sont appelés des types paramétrés, ils correspondent au type `Stack<E>` dans lequel on a remplacé toute occurrence de `E` par le type donné en paramètre (`Integer`, `String`, etc).

2.1) Type générique

Un type générique est donc un élément logiciel qui représente une famille de types, il est décrit dans un texte paramétré par une ou plusieurs variables de types (des paramètres formels de type), chacune de ces variables représentant un type inconnu dans cette définition. Sa forme générale est `G<T1, ..., Tn>` où `G` est le type de base et les `Ti` sont les paramètres formels.

2.2) Type paramétré

2.2.1) Instanciation générique

Lorsqu'on veut utiliser un type générique, il faut remplir ses trous : on remplace les paramètres formels de types par des noms de types références existants. Ce faisant, on effectue une « instanciation générique » qui produit un type paramétré, c'est-à-dire un nouveau type référence, obtenu d'un type générique par substitution de paramètres effectifs aux paramètres formels.

2.2.2) Paramètres admissibles

Lors d'une instanciation générique, un paramètre effectif peut être :

- une classe abstraite, ou
- une classe concrète, ou
- une interface, ou
- un joker (borné ou non), ou
- un autre type paramétré, ou
- un paramètre de type du type générique englobant lorsqu'il y en a un.

2.2.3) Instanciation

Il ne faudra pas confondre le terme d'« instanciation générique » avec celui d'« instanciation » utilisé jusqu'à présent dans les chapitres précédents. Le premier parle de la production d'un type (un type paramétré) à partir d'une famille de types (son type générique), alors que le second parle de la production d'un objet (une instance) à partir d'un type (sa classe génératrice).

En Java 6, il faut obligatoirement indiquer les paramètres effectifs après le constructeur lors de l'appel de ce dernier. Par contre, lors de la définition du constructeur du type générique, son entête ne prend aucun paramètre de type.

2.3) Type brut

2.3.1) Effacement

Pour des raisons de rétro-compatibilité, les concepteurs du langage Java ont décidé que toute marque de généricité devait disparaître durant l'étape de compilation du code. Mais attention, cette disparition ne doit pas se faire au détriment de la sécurité du typage apportée par la généricité !

Cette opération, appelée effacement, est donc effectuée par le compilateur. Elle consiste à supprimer, dans les fichiers compilés, tout ce qui concerne la généricité rencontrée dans les fichiers source, en remplaçant les types génériques ou paramétrés par leur type brut (voir ci-après), les paramètres formels des types génériques par l'effacement du membre gauche de leur contrainte générique (voir plus loin), et les paramètres effectifs des types paramétrés par des instructions de transtypage adaptées partout où cela est nécessaire.

2.3.2) Type brut

Un type brut (*raw type*) est un type référence correspondant au type de base d'un type générique (le `G` dans `G<E>`).

Par exemple, le type brut correspondant à `Stack<String>` est le type `Stack`. Ce dernier représente les piles dont les éléments sont des `Object`. Son utilisation ne permet aucune vérification du type des éléments de la pile (on pourrait donc y mêler des `String` et des `Integer`, par exemple).

Le code intermédiaire produit est rigoureusement identique à celui que produisait le compilateur pour le type `Stack` avant Java 5. La *JVM* ne connaît donc pas la généricité. Elle n'utilise pas `Stack<E>` ni `Stack<String>` mais `Stack` uniquement.

2.3.3) Notifications "unchecked"

Lorsqu'on utilise des types paramétrés dans le code source, le compilateur introduit dans le code compilé des instructions de vérification du type des éléments (*checkcast*).

Il se peut qu'une opération de transtypage ne soit pas sûre. Par exemple `(Stack<String>) obj` ne peut pas être vérifiée le compilateur quand la variable `obj` est de type `Object`. Dans ce cas le compilateur génère une notification *unchecked* (*unchecked warning*) pour signaler qu'il n'a pas assez d'informations pour pouvoir insérer une instruction *checkcast* dans le code intermédiaire.

De même, lorsqu'on utilise des types bruts à la place des types paramétrés dans le code source, le compilateur génère des notifications *unchecked*. L'utilisation des types bruts dans votre code source le rend compatible avec le code source et les compilateurs des versions antérieures à Java 5 (*legacy code*), mais elle n'est pas sûre du point de vue du typage, c'est pourquoi il est recommandé de ne pas utiliser cette possibilité. Les types bruts ne doivent être utilisés dans le code source que lorsqu'on interagit avec des bibliothèques développées dans une version de Java antérieure à la version 5, c'est-à-dire n'utilisant pas la généricité.

2.4) Comparaison des trois catégories

Un type générique représente une famille de types, paramétrée par des variables de types, qui n'est utilisable par ses clients que par instanciation générique, et qui n'est mis à disposition du code intermédiaire que sous la forme d'un type brut (effacement de la généricité).

Un type paramétré peut être vu comme obtenu d'un type générique dans lequel on a remplacé toutes les occurrences d'un paramètre de type par un même type référence, et ceci pour chaque paramètre. Il n'est utilisable que dans le code source, c'est-à-dire qu'on ne le retrouve pas dans le code intermédiaire. En revanche, le compilateur insère des instructions *checkcast* dans le code intermédiaire pour compenser la perte d'information sur les types qu'entraîne l'effacement de la généricité.

Un type brut est un type chargé par la JVM, utilisable par l'intermédiaire d'une instance de `Class`. Des trois catégories, c'est la seule réellement présente à l'exécution. Son usage dans le code source empêche le programmeur de typer précisément ses variables, ce que le compilateur signale d'une notification *unchecked*.

2.5) Cas des Exceptions

Une classe générique ne peut pas hériter (directement ou pas) de la classe `Throwable`, ce qui signifie que les exceptions ne peuvent pas être génériques, et les erreurs non plus. En effet, si cela était permis, on pourrait se retrouver avec plusieurs clauses **`catch`** qui, après effacement de la généricité, seraient identiques, ce que ne permet pas le langage.

3) Usage des paramètres de type

3.1) Usages valides

Dans le code source d'un type générique, on peut utiliser les paramètres de types (les `Ti` de la définition), ou des types tableaux basés sur ces paramètres (`Ti[]`), pour typer des variables ou des valeurs de retour de méthodes.

3.2) Usages inopérants

À cause de l'effacement de la généricité, il ne sert à rien de transtyper une expression avec un paramètre de type puisque le compilateur ne pourra pas insérer dans le code intermédiaire l'instruction *checkcast* correspondante. Dans ce cas, le compilateur émet une notification *unchecked*.

3.3) Usages interdits

On ne peut pas instancier un paramètre de type (`T`) puisque le nom du type qu'il faudrait réellement instancier à l'exécution n'est pas connu à la compilation.

Pour la même raison, on ne peut pas utiliser les constructions suivantes : `T.class`, **`new`** `T[]`, `x` **`instanceof`** `T`, `x` **`instanceof`** `T[]`, **`class`** `C` **`extends`** `T` et **`new`** `G<T>[n]`.

4) Membres statiques

Puisqu'un type générique ne produit qu'un seul type (brut) à la compilation, les membres statiques de ce type générique ne peuvent se trouver que dans un seul endroit : le type brut correspondant.

Par conséquent, les membres statiques d'un type générique ne peuvent pas dépendre des paramètres de types ni y accéder.

Un appel statique correct ne peut donc se faire que sur le type brut :

- appel correct : `G.methodeStatique()`
- appels incorrects : `G<Integer>.methodeStatique()` ou `G<E>.methodeStatique()`

5) Héritage et généricité

5.1) Sous-typage Java

5.1.1) Incompatibilité horizontale

Nous formulons sous le nom de règle d'incompatibilité horizontale le fait que deux instanciations génériques d'un même type générique ne peuvent pas être en relation de sous-typage Java :

Si `G<E>` est un type générique alors `G<B>` n'est pas sous-type de `G<A>` (même si `B` <: `A`)

5.1.2) Compatibilité verticale

Nous formulons sous le nom de règle de compatibilité verticale le fait que deux types paramétrés sont en relation de sous-typage s'ils ont les mêmes paramètres effectifs et si leurs types de base sont en relation de sous-typage Java :

Pour deux types génériques `G<E>` et `H<E>`, tels que `G<E>` hérite de `H<E>` alors `G<A>` <: `H<A>` (pour tout `A`)

5.1.3) Sous-types d'un type générique

Un sous-type d'un type générique est générique, avec autant ou plus de paramètres de types que son super-type :

- `class F<T> extends G<T>`
- `class F<T, S> extends G<T>`

Un sous-type d'un type non générique peut être générique, mais pas nécessairement :

- `class X extends G<String>`
- `class F<T> extends G<String>`
- `class X extends Y`
- `class F<T> extends Y`

5.2) Méthodes pont

Une méthode pont (*bridge method*) est une méthode synthétisée[10] par le compilateur au cours de l'étape d'effacement de la généricité et insérée dans le *bytecode*. Elle fournit ainsi le complément d'implantation parfois nécessaire à une sous-classe qui hérite d'un type paramétré.

À titre d'exemple, considérons l'interface `Comparable<E>` qui possède une seule méthode, de prototype `int compareTo(E)`. Après effacement, le type `Comparable` contient toujours une seule méthode, mais de prototype `int compareTo(Object)`. Considérons maintenant une classe (non générique) qui implémente l'interface `Comparable<String>`. Après compilation de cette nouvelle classe, on obtient un type brut doté d'une méthode de prototype `int compareTo(String)` conformément à l'interface `Comparable<String>` indiquée dans le code source. Mais on trouve dans le *bytecode* une méthode de prototype `int compareTo(Object)` conforme à ce que l'on

10 C'est-à-dire créée par le compilateur, pas par le développeur (elle n'apparaît pas dans le code source).

trouve dans l'interface brute `Comparable`. Or cette dernière méthode devrait être abstraite (car son corps n'a pas été défini dans le code source) si le compilateur n'en synthétisait pas lui-même le corps. Voici donc la méthode pont que synthétise le compilateur dans le *bytecode* :

```
public int compareTo(Object obj) {
    return compareTo((String) obj);
}
```

L'intérêt de ces méthodes est de permettre la rétrocompatibilité du code Java avec les versions du langage qui ne possédaient pas la généricité...

6) Généricité contrainte

Les paramètres d'un type générique peuvent être bornés par un type référence comme, par exemple, dans `E extends Number` où `E` est borné par `Number`, ce qui signifie que le type référence qui sera utilisé à la place de `E` lors d'une instanciation générique devra absolument hériter de `Number`.

On parle de généricité contrainte lorsque des paramètres de type sont bornés par d'autres types que `Object`.

6.1) Contrainte générique

On peut imposer plusieurs bornes sur un même paramètre. Les contraintes sont exprimées dans la déclaration du paramètre de type, par l'usage du mot clé **`extends`** et sont séparées entre elles par le caractère `&`.

Attention : il y ne peut pas y avoir plus d'une classe dans la contrainte, et dans ce cas elle doit obligatoirement apparaître en première position : `G<X extends {C|I|T} {& I}*>`. Dans cette expression, `C` représente une classe, `I` une interface ; et `X` et `T` des paramètres de types. Si plusieurs interfaces sont présentes, leurs effacements doivent être distincts deux à deux.

6.2) Borne générique

Toute classe ou interface apparaissant dans une contrainte générique à droite du mot clé **`extends`** est une borne générique du paramètre de type. Cette borne restreint les types possibles, lors des instanciations génériques futures, à la borne ou à l'un de ses sous-types.

Notez qu'une borne ne peut pas être un type primitif ni tableau d'aucune sorte, mais qu'elle peut très bien être un paramètre de type.

6.3) Effacement de la généricité contrainte

Lors de l'effacement de la généricité, chaque paramètre de type est remplacé par l'effacement de la première borne (en partant de la gauche) de sa contrainte générique.

L'effacement de la généricité non contrainte est cohérent puisqu'il remplace le paramètre non contraint par la classe `Object`, qui peut être considérée comme l'unique borne d'un

paramètre non contraint.

6.4) Cas des exceptions

Bien que les exceptions ne puissent pas être génériques, elles peuvent néanmoins servir de contrainte. Par exemple `<T extends Throwable>` est tout à fait licite, et on pourra alors utiliser `T` dans une clause `throws`.

7) Joker de type

7.1) Motivation

Dans certaines situations, il est souhaitable de s'affranchir de la règle d'incompatibilité horizontale : par exemple lorsqu'on veut paramétrer une méthode pour qu'elle accepte tout type de pile en entrée... Dans ce cas on utilisera un joker de type (*wildcard*) noté « `?` ».

7.2) Définition

Le joker de type est un symbole permettant de construire un super-type commun pour toutes les instanciations d'un même type générique.

Par exemple, `G<?>` représente le super-type de toutes les instanciations génériques de `G<T>`. Il n'est pas instanciable mais il permet de typer des variables et des valeurs de retour de méthodes, on le considère en fait comme un type paramétré abstrait.

`Stack<?>` et `Stack<Object>` ne représentent pas le même type.

`Stack<Object>` est un type paramétré, dont le paramètre effectif est `Object`. Il équivaut structurellement au type brut `Stack`. Leur différence tient à ce que le premier s'insère dans le système de types génériques, alors que le second non.

`Stack<?>` est un type tel que n'importe quelle instanciation générique de `Stack<E>` est compatible avec lui. Dans le système de type Java, c'est le super-type de toute instanciation générique de `Stack<E>`, et il admet le type brut `Stack` comme super-type direct.

7.3) Inconvénients

L'inconvénient du joker de type, c'est qu'on n'a aucune information sur le type qu'il représente. Par conséquent, la seule valeur compatible avec `?` est `null`, et `?` n'est compatible qu'avec le seul type `Object`.

7.4) Joker borné

De même qu'un paramètre de type peut être borné, un joker peut, lui aussi, être borné.

7.4.1) Borne supérieure

« `? extends A` » est la notation joker borné supérieurement, qui, combinée avec un type de base `G`, permet de construire un super-type pour tous les types paramétrés de type de

base `G` et dont le paramètre serait `A` ou n'importe quel sous-type de `A`.

La seule valeur compatible avec ce joker est **`null`**, et ce joker n'est compatible qu'avec `A` et ses super-types.

7.4.2) Borne inférieure

« `? super A` » est la notation joker borné inférieurement, qui, combinée avec un type de base `G`, permet de construire un super-type pour tous les types paramétrés de type de base `G` et dont le paramètre serait `A` ou n'importe quel super-type de `A`.

Les valeurs compatibles avec ce joker sont toutes celles des sous-types de `A` et ce joker n'est compatible qu'avec le seul type `Object`.

8) Méthodes génériques

8.1) Motivation

L'apport de la généricité au niveau des types permet de les paramétrer en fonction d'autres types encore inconnus au moment de la compilation. De la même manière, il est souhaitable de pouvoir paramétrer des méthodes avec des arguments dont le type est inconnu au moment de la compilation.

8.2) Définition

Une méthode générique est une méthode dotée de paramètres de types :

[`modifs`**]** `<T1,...,Tn> TypeDeRetour signature` **[**`corps`**]**

pour laquelle les types des arguments de la méthode et/ou le type de retour font référence aux `Ti`.

Attention : une méthode générique peut se trouver aussi bien dans un type générique que dans un type non générique.

8.3) Inférence des paramètres de types

Un appel de méthode générique complet est de la forme `obj.<`*`ParamEff`*`>p(...)` : on précise la valeur du ou des paramètres effectifs de type entre l'opérateur point et le nom de la méthode. Toutefois, le compilateur est capable (dans la plupart des cas) d'inférer, c'est-à-dire de calculer automatiquement, la valeur de ces paramètres effectifs de types en fonction du type des arguments et du contexte de l'appel.

On a donc le choix d'indiquer ou pas la valeur des paramètres effectifs de types lors de l'appel. En général, on laisse faire le compilateur, mais dans certains cas ambigus on est obligé de les donner explicitement pour lever le doute.

L'algorithme d'inférence qu'utilise le compilateur calcule, pour une méthode générique `p` de paramètres génériques T_1, ..., T_n, le n-uplet de types (A_1, ..., A_n) spécifié ainsi :

- pour un entête `<`DT_1`, ..., `DT_n`> R p(`F_1`, ..., `F_m`)`

(avec DT_i → T_i `extends` $U_{i,1}\&...\&U_{i,h_i}$)

- et un appel $p(e_1, ..., e_m)$ (avec CT(e_j) = E_j)
- on obtient en sortie (A_1, ..., A_n) = (X_1, ..., X_n) le plus spécifique pour les contraintes :

  $\forall\ 1 \leq j \leq m$, $E_j <: F_j[T_1|X_1, ..., T_n|X_n]$

  et dans un second temps s'il reste des T_{i_α} non résolus, en ajoutant les contraintes :

  $R[T_1|X_1, ..., T_n|X_n] <: S$ (S déterminé par le contexte d'appel)

  $\forall\ 1 \leq h \leq h_{i_\alpha}$, $X_{i_\alpha} <: U_{i_\alpha,h}[T_1|X_1, ..., T_n|X_n]$

8.4) Généricité ou usage du joker ?

La règle de base s'exprime ainsi :

- On doit utiliser une méthode générique lorsqu'on veut exprimer des dépendances entre le type des arguments de la méthode et/ou le type de valeur retournée.
- On doit utiliser le joker lorsqu'on ne veut exprimer que la flexibilité de typage de la méthode, c'est-à-dire lorsque le type de valeur retournée est indépendant du type des arguments de la méthode.

Pour l'utilisation du joker borné, voici quatre règles de bon usage :

1. Le type de valeur retournée par une méthode ne doit pas être un type à joker borné.
2. Utiliser « `? extends X` » pour typer des arguments dont le rôle est de fournir passivement de l'information, auxquels on ne veut accéder qu'en lecture...
3. Utiliser « `? super X` » pour typer des arguments dont le rôle est de consommer activement de l'information, à l'intérieur desquels on veut écrire...
4. Ne pas utiliser de joker pour les types dont les instances jouent les deux rôles précédents.

8.5) Capture de joker

Ce mécanisme (*wildcard capture*) intervient lors d'un appel de méthode générique, au cours de l'inférence des types.

Il permet au compilateur de remplacer chaque joker présent dans les types effectifs des arguments de l'appel par un nouveau nom de type, et d'inférer ce nouveau nom pour le paramètre de type correspondant dans la méthode générique. Ce mécanisme n'est possible que si :

1. le type de l'argument effectif de l'appel de méthode est de la forme `H<?>` ;
2. le type de l'argument formel correspondant dans l'entête de la méthode générique est de la forme `G<T>`.

où `H` et `G` sont des types de base tels que `H` <: `G`.

9) Classe `Class<E>`

Le type générique `Class<E>` est défini comme le type des objets qui représentent « la-classe-des-objets-de-type-`E` » (`E` pouvant prendre toutes les valeurs de type référence possibles).

Chaque instanciation générique de `Class<E>` est une classe qui ne contient qu'une et une seule instance. Par exemple, à l'exécution l'objet désigné par la constante littérale `String.class` est une instance de `Class`, la seule qui puisse être typée dans le code source par `Class<String>`. Rendre la classe `Class` générique permet donc de rendre plus précis le typage de ces constantes dans le code source.

Le prototype de la méthode `getClass()` (dans la classe `Object`) indique retourner une valeur de type `Class<?>`. En réalité, le code de cette méthode s'efforce de retourner un objet que l'on pourrait essayer de typer plus précisément par `Class<? extends |X|>` où `X` est le type avec lequel on a déclaré la variable à qui on envoie le message `getClass()` et où `|X|` représente l'effacement de `X`. Malheureusement, il est impossible de dénoter `X` ou `|X|` dans le code source et c'est pourquoi le prototype de la méthode `getClass()` n'est pas aussi précis qu'il pourrait l'être.

10) Classe `Enum<E>`

La classe `Enum<E>` est définie dans le paquetage `java.lang`, c'est une classe abstraite spéciale que le programmeur n'a pas le droit d'étendre (au sens de l'héritage). C'est le compilateur qui, lors de la définition d'un type énuméré `X`, synthétise une nouvelle classe étendant le type paramétré `Enum<X>`.

L'entête complet de la classe `Enum` est :

```
public abstract class Enum<E extends Enum<E>>
        extends Object
        implements Comparable<E>, Serializable
```

Quelques compléments sont donnés en cours pour comprendre le sens de cette déclaration...

Chapitre 5 : Collections Java

1) Définition

Une collection est un objet doté d'un comportement qui lui permet de gérer le stockage d'autres objets par ajout ou suppression d'éléments, recherche ou énumération du contenu.

En Java, les collections sont génériques. Elles se déclinent en deux grandes catégories : les collections générales (listes, ensembles et files d'attente) associées à l'interface `Collection`, et les collections indexées par des clés (tables associatives) associées à l'interface `Map`.

2) Méthodes optionnelles

Une méthode optionnelle est une méthode définie dans une interface mais qui ne sera pas forcément supportée dans les classes d'implantation ; elle pourra :

- réussir (en modifiant ou pas l'instance) ;
- échouer (en levant une `UnsupportedOperationException`).

Voici un exemple de méthode optionnelle, tiré de l'interface `Collection` : la méthode `boolean add(E e)`. La postcondition de cette méthode optionnelle déclare que la collection contient `e` et qu'un appel retourne `true` si et seulement si l'état de la collection a changé. Toutefois, cette méthode est implantée de manière fondamentalement différente dans les trois classes suivantes de l'*API* :

- `ArrayList<E>` : ajoute toujours `e` dans la liste (à la fin) et retourne toujours `true` ;
- `HashSet<E>` : ajoute `e` dans l'ensemble s'il n'y était pas déjà, retourne `true` si l'ensemble a été modifié et `false` sinon ;
- `HashMap.KeySet` : échoue en levant une `UnsupportedOperationException`.

Le seul avantage des méthodes optionnelles tient à ce que le nombre de classes et d'interfaces nécessaires pour définir l'ensemble de toutes les collections reste raisonnable.

Mais ce type de méthodes pâtit d'un double inconvénient :

1. elles rendent généralement le contrat d'une interface inutilisable puisqu'une méthode optionnelle fait... ou ne fait pas !
2. du coup il est nécessaire de connaître le type précis d'une collection pour pouvoir l'utiliser correctement.

Conseil : si possible, n'utilisez pas la notion de méthode optionnelle dans vos propres classes.

3) Itérateurs

3.1) Définition

Un itérateur est un objet qui permet à ses clients d'énumérer les éléments d'une collection,

sans que ceux-ci aient connaissance de la structure interne de cette collection. En revanche, l'itérateur est un objet qui, de par son fonctionnement, dépend fortement de la structure interne de la collection sur laquelle il opère. Il permet généralement[11] d'accéder une et une seule fois à chaque élément de la collection à laquelle il est associé.

3.2) Itérateurs

Les itérateurs sont spécifiés par l'interface `java.util.Iterator` :

- `E next()` : retourne l'élément courant de la collection sous-jacente et passe au suivant ;
- `boolean hasNext()` : retourne `true` tant qu'il y a un élément suivant (non encore obtenu) dans la collection sous-jacente ;
- `void remove()` (méthode optionnelle) : supprime l'élément retourné par le dernier appel à `next()` de la collection sous-jacente.

Un appel à `next()` alors qu'on a déjà obtenu tous les éléments (`hasNext() == false`) lève une `NoSuchElementException`.

Enfin, la méthode `remove()` doit lever une `UnsupportedOperationException` si elle n'est pas supportée (puisqu'elle est optionnelle). Lorsqu'elle est supportée, cette méthode ne peut pas être appelée deux fois de suite (sinon elle devra lever une `IllegalStateException`).

3.3) Itérateurs à échec rapide

Un itérateur à échec rapide (*fail-fast*) est un type d'itérateur dont les méthodes `next()` et `remove()` lèveront une `ConcurrentModificationException` si le contenu de l'objet itérable associé a été modifié structurellement par un autre objet que l'itérateur lui-même pendant l'énumération.

Ce genre d'itérateur permet d'arrêter une énumération proprement et au moment de la modification concurrente, plutôt que d'attendre que la modification entraîne une erreur ultérieurement... La plupart des itérateurs fournis avec les collections de l'API sont à échec rapide.

Supposons `intList` une variable de type `List<Integer>`, qui est un type itérable et dont les itérateurs sont a échec rapide. Dans le code ci-dessous, l'itérateur `it` lèvera une `ConcurrentModificationException` car la structure de `intList` a été modifiée (`add(5)`) après la création de `it` :

```
Iterator<Integer> it = intList.iterator();
intList.add(5);
System.out.println(it.next());
```

alors que dans le code suivant, tout se passe bien puisque c'est `it` lui-même qui modifie `intList` :

```
for (Iterator<Integer> it = intList.iterator(); it.hasNext();) {
    Integer n = it.next();
```

11 Sauf dans le cas des itérateurs bidirectionnels sur les listes (voir l'interface `ListIterator`).

```
    if (n <= 5) {
        it.remove();
    }
}
```

3.4) Itérables

Une instance de `Iterable` est un objet capable de fournir à qui le lui demande un itérateur d'éléments. Toute classe implantant `Iterable<T>` doit donc fournir une méthode concrète `Iterator<T> iterator()` permettant aux clients de parcourir les instances à l'aide d'une boucle **`for`** généralisée (qui utilise en fait un `Iterator<T>` en interne).

Par exemple `List<E>` implémente `Iterable<E>` donc elle fournit une méthode `Iterator<E> iterator()`. Supposons que `lst` soit une variable initialisée, de type `List<String>`. Voici comment parcourir les éléments de `lst` à l'aide d'un itérateur (remarquez que c'est `lst` qui construit et fournit l'itérateur sur ses éléments) :

```
for (Iterator<String> it = lst.iterator(); it.hasNext();) {
    String s = it.next();
    // traitement de s
}
```

Puisqu'une liste est itérable, on peut aussi l'utiliser dans une boucle for étendue :

```
for (String s : lst) {
    // traitement de s
}
```

4) Relations d'ordre

Les relations d'ordre disponibles en Java permettent de définir un ordre total sur les éléments du type T (leur paramètre générique). Elles sont disponibles sous la forme de deux interfaces

- `java.lang.Comparable` : permet de rajouter à une classe la propriété que ses éléments sont comparables entre eux, c'est pourquoi on parle ici d'ordre naturel ;
- `java.util.Comparator` : permet de définir un type d'objets capables de comparer deux à deux les éléments d'une autre classe (pour lesquels il n'existe pas forcément d'ordre préalable, ou pour définir un ordre supplémentaire à l'ordre naturel).

4.1) Ordre naturel

Il est défini par l'interface

```
public interface Comparable<T> {
    int compareTo(T obj);
}
```

La méthode `compareTo` doit retourner une valeur

- `< 0` si **`this`** est avant `obj`
- `== 0` si **`this`** est équivalent à `obj`
- `> 0` si **`this`** est après `obj`

Dans la documentation Java, il est fortement recommandé[12] que `compareTo` soit cohérente avec `equals`, c'est-à-dire que pour toute instance `obj` de la même classe que **`this`** : **`this`**`.compareTo(obj) == 0` ⇔ **`this`**`.equals(obj) == true`.

4.2) Autres ordres

Il sont définis par l'interface

```
public interface Comparator<T> {
    int compare(T obj1, T obj2);
    boolean equals(Object obj);
}
```

La méthode `compare` doit retourner une valeur

- `< 0` si obj1 est avant obj2
- `== 0` si obj1 est équivalent à obj2
- `> 0` si obj1 est après obj2

La méthode `equals` est redéclarée dans l'interface `Comparator` pour spécifier qu'il faudra la redéfinir de telle sorte que deux comparateurs sont équivalents s'ils induisent la même relation d'ordre sur les éléments de `T`.

Il est tolérable que la méthode `compare` ne soit pas cohérente avec la méthode `equals` (celle de `T` !) ; mais dans ce cas il faudra l'indiquer clairement dans la documentation afin que les clients n'utilisent pas cette relation d'ordre avec des collections ordonnées.

5) Les collections

Les collections (autres que les tables associatives) sont spécifiées par l'interface `java.util.Collection`. Une instance de `Collection` gère un regroupement d'éléments, ordonnés ou pas, avec ou sans doublons. C'est le type de collections le moins contraint possible, c'est-à-dire que tout type de collection implante cette interface. Il n'en existe toutefois pas d'implantation directe ; elle sert uniquement à typer les collections en toute généralité, indépendamment de leur spécificité.

La documentation de l'*API* préconise de doter chaque classe de collection `X<E>` d'au moins deux constructeurs :

- `X()` : crée une collection de type `X<E>`, vide ;
- `X(Collection<? extends E> c)` : crée une collection de type `X<E>`, initialisée avec les éléments de la collection `c`.

12 Comme nous le verrons plus loin, dans une collection ordonnée, il est vital qu'une relation d'ordre soit cohérente avec la relation d'équivalence.

5.1) Comportement

Voici la liste des méthodes spécifiées dans l'interface `Collection` (toutes les méthodes précédées d'un astérisque sont optionnelles) :

- `*boolean add(E e)`
- `*boolean addAll(Collection<? extends E> c)`
- `*void clear()`
- `boolean contains(Object o)`
- `boolean containsAll(Collection<?> c)`
- `boolean isEmpty()`
- `Iterator<E> iterator()`
- `*boolean remove(Object o)`
- `*boolean removeAll(Collection<?> c)`
- `*boolean retainAll(Collection<?> c)`
- `int size()`
- `Object[] toArray()`
- `<T> T[] toArray(T[] a)`

Les méthodes `add` et `addAll` permettent d'ajouter un ou plusieurs éléments. Les méthodes `clear`, `remove`, `removeAll` et `retainAll` permettent de supprimer un ou plusieurs éléments. Les méthodes `contains` et `containsAll` permettent de tester la présence d'un ou de plusieurs éléments. La méthode `isEmpty` permet de tester la vacuité de la collection, la méthode `size` permet d'en connaître le nombre d'éléments présents et la méthode `iterator` retourne un itérateur sur les éléments de la collection. Enfin, les méthodes `toArray` retournent un tableau contenant les éléments de la collection.

5.1.1) Remarques

1. La méthode `contains` a un paramètre de type `Object` et non pas `E`, ceci pour permettre la manipulation d'expressions dont le type est différent de `E` :
```
Collection<Number> cn = ...;
Object  e = 5;
Integer i = 3;
boolean ok = cn.contains(e); // peut valoir true !
ok = cn.contains(i);         // peut valoir true !
```
2. La méthode `containsAll` a un paramètre de type `Collection<?>` et non pas `Collection<E>`, ceci pour permettre la manipulation d'expressions dont les types statiques ne sont pas compatibles :
```
Collection<String> cs = ...;
Collection<Number> cn = ...;
ok = cs.containsAll(cn); // false évidemment !
```
3. Les méthodes `toArray` retournent le contenu de la collection dans un tableau conservant l'ordre des éléments tel qu'il est défini dans la collection ; le tableau retourné est indépendant de la collection (modifier le contenu de la collection ne modifie pas celui du tableau, ni inversement).
4. Voici l'algorithme (un peu particulier) de la méthode `<T> T[] toArray(T[] a)` :

Si `a.length >= size()` Alors
    stocker les éléments de la collection en tête de `a`
    Si `a.length > size()` Alors `a[size()]` ← **`null`** FinSi
    retourner `a`
Sinon
    créer un tableau `tmp` de type `T[]` et de taille `size()`
    remplir `tmp` avec les éléments de la collection
    retourner `tmp`
FinSi

5.2) Onomastique

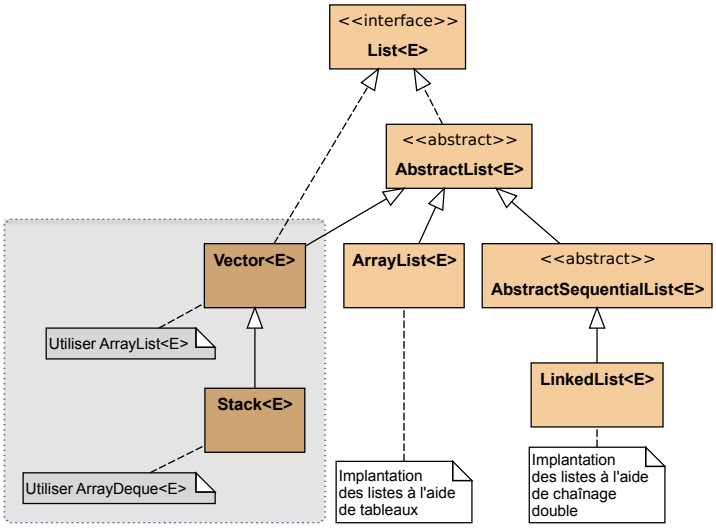
Les diverses collections sont implantées à l'aide de différentes structures de données que l'on peut deviner juste à la lecture du nom de la collection :

- implantation par tableau (`ArrayList`, `ArrayDeque`, ...)
  - accès par position : *O(1)*
  - insertion/suppression : *O(n)*
- implantation par double chaînage (`LinkedList`, `LinkedHashMap`, ...)
  - accès par position : *O(n)*
  - insertion/suppression : *O(1)*
- implantation par table de hachage (`LinkedHashMap`, `HashSet`, ...)
  - accès par clé : *O(1)*
  - insertion/suppression : *O(1)*
- implantation par arbre binaire de recherche (`TreeMap`, `TreeSet`, ...)
  - accès par clé : *O(log n)*
  - insertion/suppression : *O(log n)*

La méthode `contains(Object)` permet de rechercher un élément dans une collection. Pour sa recherche, elle utilise :

- `equals` (dans les types qui ne contiennent ni `Hash` ni `Tree`) ;
- `equals` et `hashCode` (dans les types qui contiennent `Hash`) ;
- `compareTo` (dans les types qui contiennent `Tree`).

Seule la méthode `contains` du type `IdentityHashMap` déroge à cette règle puisqu'elle n'utilise que `==`.



6) Les listes

L'interface `java.util.List` spécifie les collections d'éléments ordonnés par leur rang.

6.1) Comportement

En plus des méthodes héritées de `Collection`, on trouve notamment

les méthodes :

- `E get(int)`
- *`E set(int, E)`
- *`E remove(int)`
- *`void add(int, E)`
- `int indexOf(Object)`
- `int lastIndexOf(Object)`
- `List<E> subList(int, int)`
- `ListIterator<E> listIterator()`
- `ListIterator<E> listIterator(int)`
- *`boolean addAll(int, Collection<? extends E>)`

Les méthodes `listIterator` permettent d'effectuer des itérations bidirectionnelles, permettant donc de « rembobiner » le parcours.

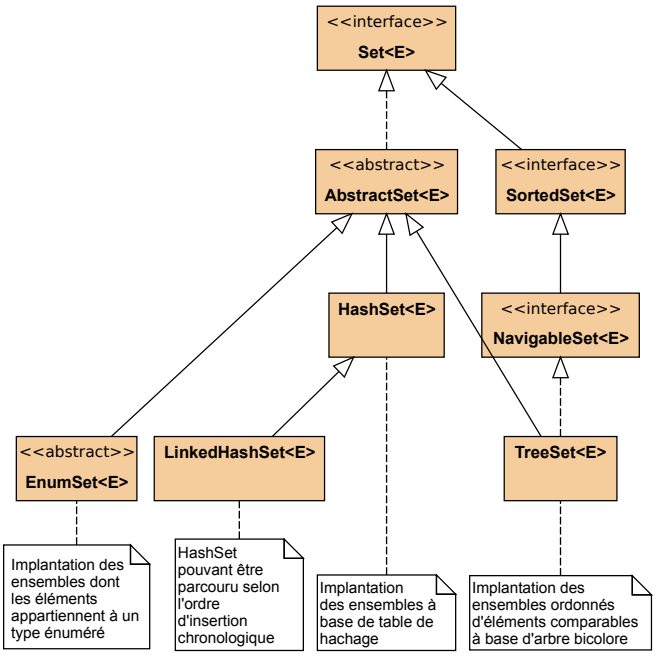
6.2) Implantation

La classe `java.util.ArrayList<E>` représente les tableaux de longueur variable, l'accès à un élément se fait en temps constant, les insertions et les suppressions avec décalage vers le début sont linéaires.

La classe `java.util.LinkedList<E>` représente les listes doublement chaînées, les insertions et les suppressions avec décalage vers le début se font en temps constant mais l'accès, bien qu'optimisé, est linéaire.

La classe `java.util.Vector<E>` date de l'époque où le cadre des collections était peu fourni et incomplètement spécifié. Cette classe ne devrait plus être utilisée ; elle est avantageusement remplacée par `ArrayList<E>`.

La classe `java.util.Stack<E>` ne devrait plus être utilisée non plus ; pour manier une pile il faut instancier `ArrayDeque<E>` et l'utiliser à travers l'interface `Queue<E>` comme on le verra plus loin.



7) Les ensembles

L'interface `java.util.Set` spécifie les collections d'éléments non ordonnés, sans doublons (au sens de `equals`).

L'interface `java.util.SortedSet` spécifie les ensembles dont les éléments sont munis d'un ordre total et l'interface `java.util.NavigableSet` ceux dont on peut chercher l'élément le plus proche d'un élément donné.

Une remarque importante concernant les ensembles ordonnés : toutes les opérations de

comparaison d'éléments se font à l'aide de la méthode `compareTo` (ou `compare` selon le cas), c'est pourquoi l'ordre naturel (ou induit par un comparateur) doit être cohérent avec `equals`. Par exemple, si `a` et `b` sont des instances d'une classe `C` comparable, telles que `a.equals(b) == false` tandis que `a.compareTo(b) == 0`, la situation suivante pourrait alors prêter à confusion :

```
Set<C> x = new TreeSet();
System.out.println(x.add(a));
System.out.println(x.add(b));
```

car la ligne 1 affichera **`true`** et la 2 affichera **`false`**, alors que le contrat de `Set` voudrait que la ligne 2 affiche **`true`** (puisque `!a.equals(b)`).

7.1) Comportement

7.1.1) Ensembles quelconques

L'interface `Set` ne définit pas plus de méthodes que `Collection`. mais elle en complète la spécification ainsi :

- Une insertion (`add`, `addAll`) ne modifie l'ensemble (retourne **`true`**) que si les éléments ajoutés n'étaient pas déjà présents.
- Une suppression (`remove`, `removeAll`, `retainAll`) ne modifie l'ensemble (retourne **`true`**) que si les éléments retirés étaient déjà présents.

7.1.2) Ensembles ordonnés

L'interface `SortedSet` permet d'accéder aux éléments extrémaux de l'ensemble :

- `E first()` : le premier élément ;
- `E last()` : le dernier élément ;

et de gérer des intervalles d'éléments :

- `SortedSet<E> headSet(E)` : intervalle s'étalant du premier élément jusqu'à l'argument exclus ;
- `SortedSet<E> subSet(E, E)` : intervalle s'étalant du premier argument inclus jusqu'au second exclus ;
- `SortedSet<E> tailSet(E)` : intervalle s'étalant de l'argument inclus jusqu'au dernier élément inclus.

Enfin, l'itérateur d'un `SortedSet` parcourt les éléments de l'ensemble en ordre croissant.

L'interface `NavigableSet`, par rapport à la précédente, autorise en plus le retrait des éléments extrémaux :

- `E pollFirst()` : supprime le premier élément ;
- `E pollLast()` : supprime le dernier élément ;

et la gestion fine des intervalles d'éléments :

- `NavigableSet<E> headSet(E, boolean)` : intervalle s'étalant du premier

élément jusqu'au premier argument, ce dernier étant inclus ou exclus selon la valeur du second argument ;
- `NavigableSet<E> subSet(E, boolean, E, boolean)` : intervalle s'étalant du premier argument jusqu'au second, l'inclusion des bornes est contrôlée par les argument booléens ;
- `NavigableSet<E> tailSet(E, boolean)` : intervalle s'étalant de l'argument jusqu'au dernier élément, le premier élément étant inclus ou exclus selon la valeur du second argument.

De plus, elle permet de naviguer en recherchant la correspondance la plus proche d'un élément donné :

- `E ceiling(E)` : le plus petit élément plus grand que l'argument, ou l'argument s'il est présent ;
- `E floor(E)` : le plus grand élément plus petit que l'argument, ou l'argument s'il est présent ;
- `E heigher(E)` : le plus petit élément strictement plus grand que l'argument, ou `null` s'il n'existe pas ;
- `E lower(E)` : le plus grand élément strictement plus petit que l'argument, ou `null` s'il n'existe pas.

Elle permet aussi le parcours en ordre inverse :

- `NavigableSet<E> descendingSet()` : ensemble ordonné selon l'ordre inverse des éléments ;
- `Iterator<E> descendingIterator()` : itérateur parcourant l'ensemble dans l'ordre inverse des éléments.

7.2) Implantation

La classe `java.util.HashSet<E>` représente les ensembles non ordonnés.

La classe `java.util.LinkedHashSet<E>` représente celle des ensembles ordonnés par l'ordre chronologique, c'est-à-dire que chaque instance maintient à jour une `LinkedList` de ses éléments, rangés au fur et à mesure de leur insertion.

La classe `java.util.TreeSet<E>` représente les ensembles munis d'un ordre total sur leurs éléments (ordre naturel ou ordre induit par un `Comparator`). La création d'un ensemble ordonné pourra donc se faire ainsi :

- si `C` implante `Comparable<C>`
  `new TreeSet<C>()`
  `new TreeSet<C>(Comparator<C>)`
- si `C` n'implante pas `Comparable<C>`
  `new TreeSet<C>(Comparator<C>)`

Pour les deux premières classes d'implantation, les insertions et les suppressions se font en temps constant. Ces opérations sont légèrement moins efficaces dans la seconde que dans la première puisqu'il faut y mettre à jour une liste chaînée.

En revanche, une itération complète se fera en un temps proportionnel au nombre d'éléments, plus la capacité de la table de hachage sous-jacente, pour les instances de la

première classe, alors qu'elle sera seulement proportionnelle au nombre d'éléments pour une instance de la seconde.

Les recherches sont constantes, en moyenne.

Pour la troisième classe, les opérations d'insertion, de suppression, de recherche ou de passage au suivant sont toutes en *log n*.

7.3) Types énumérés

Le type `java.util.EnumSet` est une implantation de `Set` spécialement adaptée pour contenir les éléments d'un type énuméré. L'avantage de ce type tient à ce que ses instances occupent un faible encombrement mémoire (vecteur de bits) et la plupart des opérations prennent un temps constant.

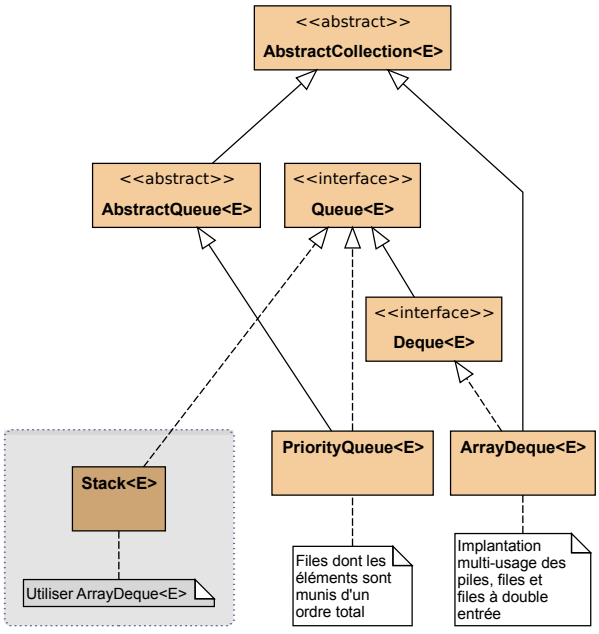
`EnumSet` est une classe abstraite, on ne crée donc pas les instances soi-même : on utilise plutôt les diverses méthodes statiques disponibles dans cette classe. Donnons-nous par exemple le type énuméré :

```
Taille { INFIME, MINUS, PETIT, NORMAL, GRAND, ENORME, GIGANT }
```

À partir de `Taille`, nous pouvons alors fabriquer tous les ensembles que nous souhaitons, de la manière suivante :

```
EnumSet<Taille> tous = EnumSet.allOf(Taille.class);
EnumSet<Taille> petit = EnumSet.range(INFIME, PETIT);
EnumSet<Taille> std = EnumSet.range(PETIT, GRAND);
EnumSet<Taille> peuCourant = EnumSet.of(MINUS, ENORME);
EnumSet<Taille> petitOuStd = EnumSet.copyOf(petit);
petitOuStd.addAll(std);
EnumSet<Taille> tresGrand = EnumSet.complementOf(petitOuStd);
EnumSet<Taille> raisonnablementPetit = EnumSet.copyOf(petit);
raisonnablementPetit.retainAll(std);
```

Notez encore qu'il n'est possible ni de mélanger les valeurs de plusieurs types énumérés dans un même ensemble, ni d'y ajouter **`null`**.



8) Les files d'attente

L'interface `java.util.Queue` spécifie les files à simple entrée, qui sont des collections linéaires d'éléments (non **`null`**) à traiter dans un ordre spécifique (file d'attente) par ajout à la queue (un bout) avec retrait à la tête (l'autre bout).

Exemples de files d'attente à simple entrée :

- Pile (stratégie LIFO) : ordre de récupération anti-chronologique
- File (stratégie FIFO) : ordre de

récupération chronologique
- File de priorité : ordre de récupération dépendant d'une fonction de priorité

L'interface `java.util.Deque` spécifie les files à double entrée, qui sont des collections linéaires d'éléments qui supportent l'ajout et le retrait d'éléments en leurs deux bouts.

8.1) Comportement

8.1.1) Files à simple entrée (`Queue`)

Lorsqu'une file est vide, on ne peut pas retirer d'élément ni consulter l'élément courant. Les classes d'implantation pourront aussi modéliser des files bornées, pouvant échouer lors de l'ajout d'un élément. Il y a donc deux types de méthodes :

- celles qui lèvent une exception en cas d'échec :
  - `boolean add(E)` : ajoute le paramètre en queue de file ;
  - `E remove()` : retire l'élément en tête de file ;
  - `E element()` : retourne (sans le retirer) l'élément en tête de file ;
- celles qui retourne `false` ou `null` en cas d'échec :
  - `boolean offer(E)` : ajoute le paramètre en queue de file ;
  - `E poll()` : retire l'élément en tête de file ;
  - `E peek()` : retourne (sans le retirer) l'élément en tête de file.

Ces seules méthodes permettent de manipuler indistinctement des piles, des files ou des files de priorité. Pour changer la stratégie de base d'une file `x` (FIFO) en celle d'une pile (LIFO), il suffit de l'encapsuler dans un objet proxy : `Collections.asLifoQueue(x)`.

8.1.2) Files à double entrée (`Deque`)

En plus des méthodes de `Queue` (dont les spécifications sont inchangées), l'interface `Deque` spécifie les méthodes suivantes

- avec levée d'exception en cas d'échec :
  - `boolean addFirst(E)` : ajoute le paramètre en tête de file ;
  - `E removeFirst()` : retire l'élément en tête de file (équivalente à `remove`) ;
  - `E getFirst()` : retourne (sans le retirer) l'élément en tête de file (équivalente à `element`) ;
  - `boolean addLast(E)` : ajoute le paramètre en queue de file (équivalente à `add`) ;
  - `E removeLast()` : retire l'élément en queue de file ;
  - `E getLast()` : retourne (sans le retirer) l'élément en queue de file ;
- avec retour de valeur spécifique (`false` ou `null`) en cas d'échec :
  - `boolean offerFirst(E)` : ajoute le paramètre en tête de file ;
  - `E pollFirst()` : retire l'élément en tête de file (équivalente à `poll`) ;
  - `E peekFirst()` : retourne (sans le retirer) l'élément en tête de file (équivalente à `peek`) ;
  - `boolean offerLast(E)` : ajoute le paramètre en queue de file (équivalente à `offer`) ;

- `E pollLast()` : retire l'élément en queue de file ;
- `E peekLast()` : retourne (sans le retirer) l'élément en queue de file.

Plutôt que d'utiliser la classe `Stack`, on utilisera `ArrayDeque` avec les équivalences suivantes :

- `boolean addFirst(E)` : équivaut à `push` ;
- `E removeFirst()` : équivaut à `pop` ;
- `E peekFirst()` : équivaut à `peek`.

8.2) Implantation

La classe `java.util.ArrayDeque<E>` est « LA » classe générale d'implantation des piles, files et autres monstres polycéphales et multicaudes.

Voici des exemples de manipulation de toutes les sortes de file d'attente :

- Pile (stratégie LIFO) :
```
Queue<Person> stack = Collections.asLifoQueue(new ArrayDeque<Person>());
// stack est une pile manipulée par add, remove, element
```
- File (stratégie FIFO) :
```
Queue<Person> queue = new ArrayDeque<Person>();
// queue est une file manipulée par add, remove, element
```
- File de priorité :
```
Queue<Person> pqueue = new PriorityQueue<Person>();
// pqueue est une file de priorité manipulée par add, remove, element
// remove retire l'élément de plus haute priorité
```
- File à double entrée :
```
Deque<Person> deque = new ArrayDeque<Person>();
// les personnes peuvent être ajoutées ou retirées aux deux bouts
```

La plupart des opérations se font en temps constant. La recherche est linéaire.

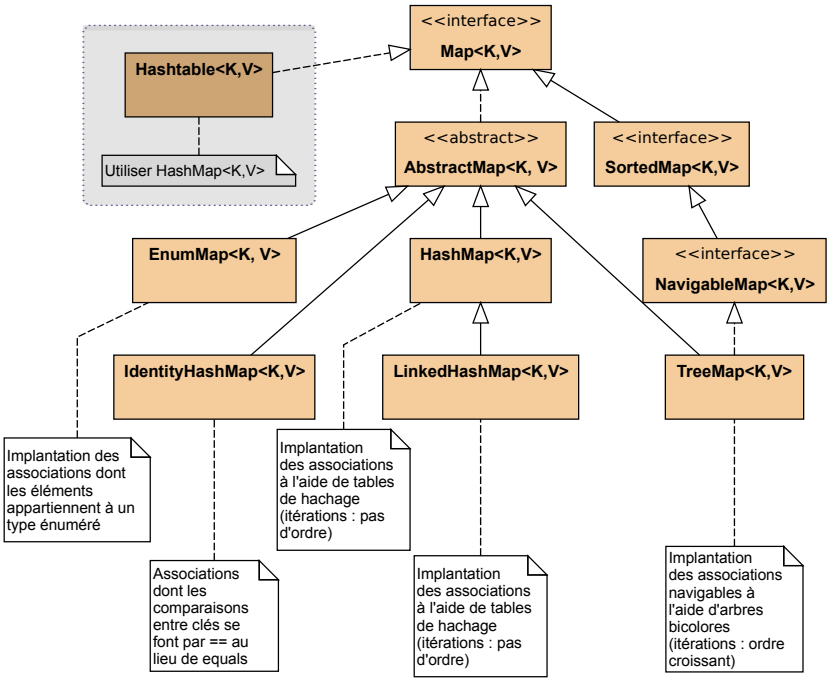
La classe `java.util.PriorityQueue<E>` permet de fixer l'ordre dans lequel on souhaite récupérer les éléments (ordre naturel ou via un `Comparator`). Les opérations d'ajout et de retrait sont en *log n*, la recherche est linéaire et les opérations de récupération sont constantes.

9) Les tables associatives

L'interface `java.util.Map` spécifie les tables associatives, c'est-à-dire les ensembles de couples clé-valeur dans lesquels les clés sont uniques (au sens de `equals`). La représentation interne des couples au sein d' une table associative doit respecter la spécification donnée par l'interface `java.util.Map.Entry`. `K` est le type des clés et `V` celui des valeurs.

L'interface `java.util.SortedMap` spécifie les tables associatives dont les clés sont munies d'un ordre total (naturel ou *via* un `Comparator`).

L'interface `java.util.NavigableMap` spécifie les tables associatives ordonnées dont on peut chercher une valeur dont la clé est la plus proche d'une clé donnée.



9.1) Comportement

9.1.1) Clés sans ordre

Retraits de couples de l'ensemble :

- `void clear()`
- `V remove(Object)`

Requêtes :

- `int size()`
- `V get(Object)`
- `boolean isEmpty()`
- `boolean containsKey(Object)`
- `boolean containsValue(Object)`

Ajouts de couples dans l'ensemble :

- `V put(K, V)`
- `void putAll(Map<? extends K, ? extends V>)`

Récupération des collections sous-jacentes :

- `Set<K> keySet()` : accès à l'ensemble de toutes les clés de l'association
- `Collection<V> values()` : accès à la collection de toutes les valeurs présentes dans l'association
- `Set<Map.Entry<K, V>> entrySet()` : accès à l'ensemble de tous les couples de l'association

Retirer un élément d'une des collections sous-jacentes peut supprimer un ou plusieurs couples de l'association, et inversement.

9.1.2) Clés ordonnées

L'interface SortedMap spécifie les 6 méthodes supplémentaires :

- `K firstKey()`
- `K lastKey()`
- `Comparator<? super K> comparator()`
- `SortedMap<K, V> subMap(K, K)`
- `SortedMap<K, V> headMap(K)`
- `SortedMap<K, V> tailMap(K)`

L'interface NavigableMap spécifie les 18 méthodes supplémentaires :

- `Map.Entry<K, V> {first|last}Entry()`
- `Map.Entry<K, V> poll{First|Last}Entry()`
- `Map.Entry<K, V> {ceiling|floor}Entry(K)`
- `Map.Entry<K, V> {heigher|lower}Entry(K)`
- `NavigableMap<K,V> subMap(K, boolean, K, boolean)`
- `NavigableMap<K,V> {head|tail}Map(K, boolean)`
- `K {ceiling|floor}Key(K)`
- `K {heigher|lower}Key(K)`
- `NavigableSet<K> navigableKeySet()`
- `NavigableSet<K> descendingKeySet()`
- `NavigableMap<K, V> descendingMap()`

9.2) Implantation

La classe `java.util.HashMap<K, V>` représente les associations simples, sans ordre sur les clés.

La classe `java.util.LinkedHashMap<K, V>` représente les associations au sein desquelles les couples sont ordonnés selon l'ordre chronologique des clés.

La classe `java.util.TreeMap<K, V>` représente les associations au sein desquelles les couples sont ordonnés selon l'ordre (naturel ou via `Comparator`) des clés.

Pour les deux premières classes, les insertions, suppressions et recherches de clés se font en temps constant en moyenne. Les recherches de valeurs se font en temps linéaire en le nombre d'éléments pour la seconde et linéaire en le nombre d'éléments plus la taille de la table de hachage interne pour la première.

Pour la troisième classe, les opérations se font en temps logarithmique sauf la recherche de valeurs qui est en *O(n log n)*.

La classe `java.util.IdentityHashMap<K, V>` représente les associations sans ordre sur les clés, mais ici les comparaisons lors des recherches se font avec `==`.

9.3) Types énumérés

Le type `java.util.EnumMap` est une implantation particulière de `Map`, qui tient compte du fait que ses clés sont les éléments d'un type énuméré : la table associative est basée sur un tableau de valeurs, indicé par les valeurs ordinales des constantes du type énuméré, ce qui permet une réalisation très efficace.

Chaque instance de la classe `EnumMap` doit pouvoir faire référence à l'ensemble des constantes du type de ses clés, ne serait-ce que pour connaître la taille du tableau de valeurs qu'elle utilisera. Mais comment procéder quand ce type s'appelle `E` dans le code source et qu'il disparaîtra à l'exécution ? La seule solution possible est de faire en sorte que l'on récupère le type des clés lors de l'exécution (une instance de `Class`) pour lui demander l'ensemble des constantes qui y sont définies par envoi du message `getEnumConstants`. Voici pourquoi on doit passer le type de la valeur des clés comme argument du constructeur de `EnumMap`.

Par exemple, si l'on veut utiliser une `EnumMap` dont les clés sont des `Civ` et les valeurs

des entiers on commandera :

```
Map<Civ, Integer> elements = new EnumMap<Civ, Integer>(Civ.class);
```

et le constructeur de `EnumMap` fera quelque chose du genre :

```
Enum[] universe = cls.getEnumConstants();
Object[] vals = new Object[universe.length];
```

où `cls` est l'argument du constructeur représentant le type des clés (`Civ.class` dans cet exemple). Ainsi `elements` pourra accéder au besoin à toutes les constantes de la classe des clés (tableau `universe`) ainsi qu'à l'emplacement de la valeur associée à la clé `k` par `vals[k.ordinal()]`.

Mais attention, le tableau interne `vals` est initialisé avec des valeurs **`null`** à chaque indice, indiquant qu'il n'y a pas d'association réalisée pour la constante énumérée d'ordinal cet indice. De ce fait, les couples d'entrée dans la table dont la valeur est **`null`** ne sont pas acceptés dans une `EnumMap`. Il va de soi qu'une `EnumMap` est vide à la création.

Chapitre 6 : Types imbriqués

1) Définition

Un type imbriqué (*nested type*) est un type dont la déclaration se situe à l'intérieur d'un autre type (dit « englobant ») plutôt que dans un paquetage.

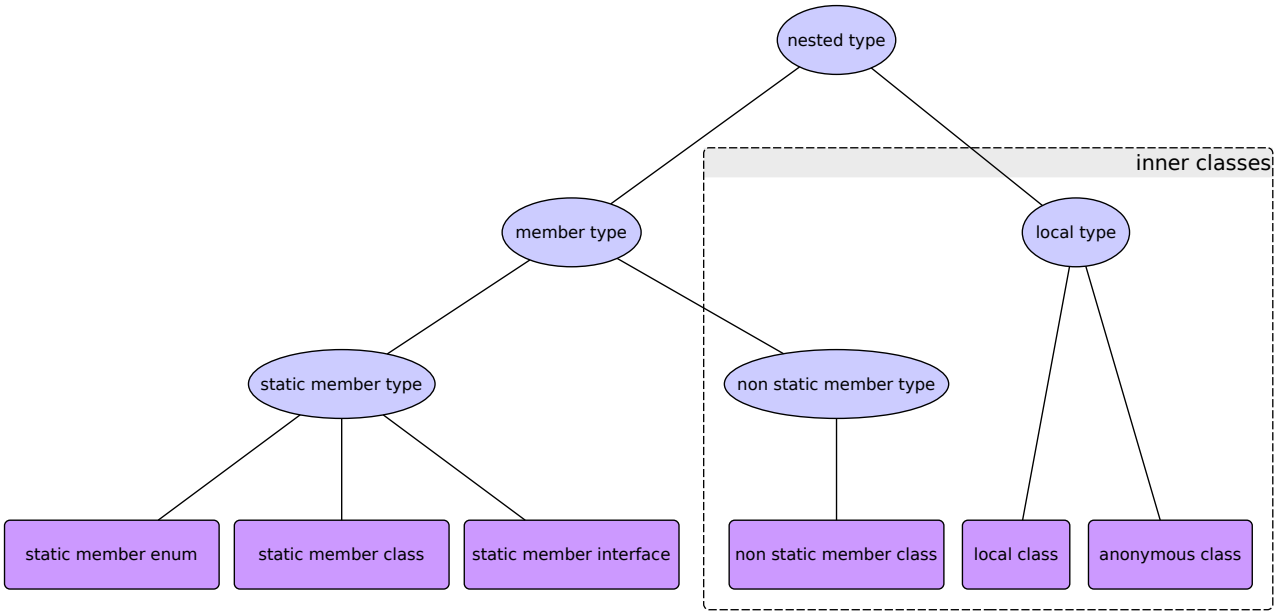
Dans l'exemple ci-dessous, les classes `B` et `C` sont imbriquées dans la classe `A` :

```
class A {
    class B { ... }
    void m() {
        class C { ... }
        C x = new C();
        ...
    }
}
```

mais alors que la classe `B` est une classe membre (définie au même niveau qu'un attribut) la classe `C` est une classe locale (définie au même niveau qu'une variable locale).

L'usage des types imbriqués permet d'améliorer l'encapsulation du code et de créer de nouveaux espaces de nommage. Cela peut améliorer la lisibilité du code, mais il faut être prudent car cela peut aussi provoquer l'effet inverse...

2) Classification



Les types imbriqués se séparent en deux catégories

- types membres : types imbriqués définis au même niveau que les membres de leur type englobant ;
- types locaux : types imbriqués définis dans un bloc de code (corps de méthode ou

bloc d'initialisation).

et les types membres se séparent encore en deux catégories

- types statiques : le type membre est préfixé du mot-clé **`static`** ;
- types non statiques : le type membre n'est pas préfixé du mot-clé **`static`**.

3) Particularités

3.1) Types membres

Un type membre peut être une classe, un type énuméré ou une interface. Il peut de plus être qualifié par les mots-clés : **`public`**, **`protected`**, **`private`**, **`abstract`**, **`final`** ou **`static`**.

La portée du nom simple d'un type membre est restreinte au type englobant, au-delà, on y accède (s'il est accessible) en qualifiant son nom simple avec celui du type englobant.

Les membres d'un type membre, même ceux qui sont privés, sont toujours accessibles depuis le code du type englobant.

3.2) Types membres statiques

Le code du type membre statique peut accéder aux membres statiques (même privés) du type englobant mais pas à ses membres d'instances. Par défaut, et lorsqu'il n'y a pas d'ambigüité possible, on peut omettre le nom du type englobant lors de l'accès à ses membres statiques depuis le code du type imbriqué.

En fait un type membre statique est un type normal mais situé dans un espace de nommage particulier : un autre type, plutôt qu'un paquetage.

Enfin, une interface ne peut englober que des types membres statiques (**`static`** est implicite dans la définition des types imbriqués).

Exemple (les crochets indiquent les parties optionnelles, que l'on peut omettre lorsqu'il n'y a pas d'ambigüité) :

```
public class A {
    private static int x = 0;
    public static class B {
        void m() {
            [A.]x = 1;
        }
    }
}
------------><8
A.B v = new A.B();
v.m(); // A.x vaut 1 en sortie
```

3.3) Types non statiques

Un type imbriqué non statique est nécessairement une classe définie ailleurs que dans

une interface. Une instance d'un tel type ne peut être créée qu'à partir d'une instance de son type englobant, appelée son instance englobante. Cela permet à l'instance imbriquée d'accéder aux membres d'instances (même privés) de l'instance englobante. Bien entendu, le code du type imbriqué peut aussi accéder aux membres statiques du type englobant.

Par défaut, et lorsqu'il n'y a pas d'ambigüité possible, on peut omettre la référence à l'instance englobante lors de l'accès à ses différents membres depuis le code du type imbriqué.

3.3.1) Classes membres non statiques

En général, une classe membre non statique n'est pas publique car elle permet essentiellement d'encapsuler un ensemble de fonctionnalités utiles à la classe englobante dans une structure qui nécessite un accès direct à sa structure interne. Néanmoins, au cas où elle serait accessible depuis l'extérieur de la classe englobante, on notera bien la syntaxe particulière utilisée pour la création des instances de la classe imbriquée :

```
public class A {
    private int x;
    public class B {
        private int y;
        void m() {
            [A.this.]x = 1;
            [this.]y = 2;
            B w = [A.this.]new B();
        }
    }
}
------------><8
A u = new A();
A.B v = u.new A.B(); // v dépend de l'instance englobante u
v.m(); // u.x vaut 1 et v.y vaut 2 en sortie
```

3.3.2) Types locaux

Un type local est obligatoirement une classe qui peut être déclarée **`abstract`** ou **`final`** (mais jamais **`public`**, **`protected`**, **`private`** ni **`static`**).

La portée d'une classe locale est restreinte au bloc dans lequel elle est définie ; au-delà, on ne peut pas y accéder. Par conséquent un type local accède à tous les membres de son type englobant, mais l'inverse est faux. De plus, le code du type local peut accéder à toute variable locale (ou paramètre) à condition que cette variable soit déclarée **`final`**.

Exemple de classe locale (nommée) :

```
public class A {
    private int x;
    public void m(final int i) {
        final int y = 1;
        class B {
```

```
            public int m() {
                return x + i + y;
            }
        }
        B v = new B();
        ...
    }
}
```

Mais une classe locale peut être anonyme si elle est instanciée immédiatement à l'endroit où elle est définie. Exemple de classe locale anonyme (avec instanciation) :

```
X obj = new X(a1,...,an) {
    ... // corps de la classe dérivée de la classe X
};
```

Dans l'exemple ci-dessus, on crée une classe locale que l'on instancie simultanément à sa création ; la référence de l'instance produite est stockée dans `obj`. Le type de la valeur de `obj` est une classe (dont le nom n'est pas connu dans le code source, c'est pour cela qu'elle est dite anonyme) qui dérive la classe `X`. Sous la forme de l'exemple, `X` doit être une classe possédant un constructeur `X(a1,...,an)` (`n` pouvant être égal à 0) ; mais si `X` était une interface, `n` serait nécessairement égal à 0, et l'exemple deviendrait

```
X obj = new X() {
    ... // corps de la classe dérivée de l'interface X
};
```

3.4) Classes internes

Un type imbriqué non statique (type membre ou type local) est nécessairement une classe, appelée classe interne (*inner class*).

Toute instance d'une classe interne est reliée, comme on l'a vu, à une instance de son type englobant, disons `X`. Dans le code de la classe interne, pour distinguer la référence à l'instance interne de celle de l'instance englobante, on note **`this`** (référence interne) et `X.this` (référence englobante). De plus, lorsque la classe interne porte un nom, disons `Y`, on peut aussi écrire `Y.this` au lieu de **`this`** pour désigner l'instance interne (toujours dans le code de la classe interne).

Une instance de classe interne `B` ne peut être créée que par le biais d'une instance de sa classe englobante `A` : si `a` est une instance de `A`, on peut créer une instance de `B` reliée à `a` avec l'expression `a.new B()`.

4) Nommage des types imbriqués

Depuis l'extérieur d'un type englobant `A<E>`, dans le code source, on accède au nom d'un type membre par :

- `A.X` si `X` est un type membre statique et non générique
- `A.Y<F>` si `Y<F>` est un type membre statique et générique
- `A<E>.Z` si `Z` est un type membre non statique et non générique

- `A<E>.U<F>` si `U<F>` est un type membre non statique et générique

On peut ensuite importer un type membre dans la définition d'un type à l'aide de la directive **`import`** (sans indiquer les éventuels paramètres de type).

Le compilateur traduit un type imbriqué en un type normal, de nom spécifique :

- tout type membre de nom `X.Y` est compilé dans un fichier de nom `X$Y.class`
- toute classe locale `Y` d'un type englobant `X` est compilée dans un fichier de nom `X$nY.class`
- toute classe anonyme d'un type englobant `X` est compilée dans un fichier de nom `X$n.class`

Chapitre 7 : Introduction aux IHM en Java avec Swing

1) Présentation

Une application Java munie d'une interface graphique homme-machine (IHM, *GUI*) est une application pour laquelle l'échange d'informations entre la machine et l'utilisateur humain se fait par l'intermédiaire d'une portion d'écran appelée fenêtre. Cette fenêtre présente l'information sous forme graphique et reçoit des données sous forme textuelle (clavier) et de pointage (souris). Une telle application

- utilise des composants graphiques (fenêtres, boutons, champs de texte, ...) ;
- gère une file d'événements initiés par les périphériques d'entrée (utilisateur) ou par l'application elle-même ;
- est multitâche (*multithreaded*).

Java se veut « *WORA* » (*Write Once Run Anywhere*), c'est-à-dire qu'un programme écrit en Java doit avoir le même aspect et le même comportement quelle que soit la plate-forme sous-jacente. La contrainte WORA est difficile à réaliser mais ce sont les bibliothèques de composants graphiques (*AWT* et Swing), distribuées avec Java, qui en ont la responsabilité.

AWT (*Abstract Window Toolkit*) est l'ancienne bibliothèque de composants graphiques fournie par Sun depuis sa première version (1.0).

Swing est une nouvelle bibliothèque graphique, complètement intégrée à Java depuis la version 1.2, entièrement repensée mais qui s'appuie tout de même sur certains composants AWT.

JFC (*Java Foundation Classes*) est la bibliothèque standard qui intègre *AWT* et Swing (et d'autres bibliothèques) et qui permet de développer des applications Java dotées d'IHM professionnelles.

1.1) AWT

La bibliothèque *AWT* repose sur une architecture à base de composants pairs (*peer components*). Au sein de cette bibliothèque, un composant pair est un composant Java qui contient du code natif (c'est-à-dire du code machine qui n'est pas issu de la compilation de code source Java).

Chaque composant *AWT*, appelé aussi composant lourd (*heavyweight component*), est associé à un composant pair, qui est un widget dont l'affichage est géré directement par le système de fenêtrage de la plate-forme sous-jacente. L'avantage de cette technologie tient à la rapidité de l'affichage puisque ce dernier est effectué par le système. Mais les inconvénients sont importants : le *look and feel* (*LAF*) est complètement dépendant de la plate-forme et le catalogue de composants est réduit au plus petit dénominateur commun.

1.2) Swing

Swing est une bibliothèque basée sur *AWT*, c'est-à-dire qu'elle nécessite la présence de certains composants *AWT* pour fonctionner. En effet, quatre composants Swing (`JFrame`,

`JDialog`, `JWindow` et `JApplet`) étendent directement des composants *AWT* (resp. `Frame`, `Dialog`, `Window` et `Applet`) et les autres composants graphiques Swing héritent de `java.awt.Container`. De plus, Swing utilise la gestion des événements graphiques de *AWT* (telle que définie dans la version 1.1).

Les composants Swing sont (presque tous) basés sur un seul et même composant pair vide et indépendant de la plate-forme. Par opposition aux composants *AWT*, ils sont appelés composants légers (*lightweight component*) car leur affichage est géré par du code Java. Cela leur procure plusieurs avantages : ils sont du coup très portables puisque indépendants de la plate-forme et faciles à définir. Du coup, le catalogue de ces composants est très fourni. Reste un inconvénient : l'affichage, codé en Java, est comparativement plus lent que pour les composants lourds.

2) Programmation événementielle

Dans le style de programmation événementielle, chaque action effectuée par l'utilisateur génère un ou plusieurs événements. Il se peut aussi que l'application elle-même ou le système créent des événements. L'application donne l'impression qu'elle passe son temps à attendre la création d'événements, pour aussitôt effectuer le traitement qui leur correspond.

Une action utilisateur est définie comme une interaction de l'utilisateur avec la machine, dans le but de piloter l'application (frappe clavier ou clic souris sur un composant de l'une des fenêtres de l'application). Cette action est à l'origine d'une interruption matérielle, récupérée par le système, puis transmise à la *JVM*, qui la code sous forme d'objet événement.

2.1) Principe de la gestion des événements

Dans une application graphique, un composant graphique `s` est susceptible d'être activé par l'utilisateur humain à l'aide de la souris ou du clavier. Lorsque `s` est ainsi activé, une instance `e` d'événement Java est créée pour décrire tout le contexte de l'activation de `s`, puis `e` est « transmis » à un ou plusieurs objets ℓ_i préalablement enregistrés auprès de `s`. On dit que `e` est un événement, que `s` est la source de l'événement `e` et que les ℓ_i sont les écouteurs (*listener*) de la source `s` pour les événements du type de `e`. La « transmission » de `e` s'effectue par l'appel « automatique » d'une certaine méthode, sur chacun des écouteurs ℓ_i, avec `e` pour argument. L'exécution du corps de cette méthode constitue la réponse de l'application à l'action de l'utilisateur.

2.2) Les événements en Java

2.2.1) Événements et écouteurs

En Java, tout événement est une instance de `java.util.EventObject` qui contient les informations nécessaires à sa gestion, à commencer par la source de l'événement :

```
Object getSource()
```

Tous les événements que nous traiterons ce semestre sont des instances de

`java.awt.AWTEvent`, la sous-classe des événements graphiques Java. Mais il existe aussi d'autres types d'événements que ceux qui rendent compte de l'action de l'utilisateur, et dont la source n'est pas nécessairement un composant graphique.

Les écouteurs sont spécifiés dans des interfaces présentes dans les paquetages `java.awt.event` ou `javax.swing.event` : `ActionListener`, `MouseListener`, `ItemListener`, etc.

2.2.2) Mécanisme de gestion des événements

Quand le programmeur décide qu'un composant `c` sera source d'événements de type `XxxEvent`, il enregistre auprès de `c` un écouteur (au moins) de type `XxxListener`. Pour que cela soit possible, il faut que le type de `c` possède une méthode `void addXxxListener(XxxListener)` permettant d'effectuer cet enregistrement. Il doit aussi posséder une méthode `void removeXxxListener(XxxListener)` pour retirer des écouteurs et peut posséder une méthode `XxxListener[] getXxxListeners()` permettant de les consulter.

Ensuite, durant l'exécution du programme, à chaque fois que `c` sera activé, un événement `e` de type `XxxEvent` sera créé, dont `c` sera la source. Juste après sa création, `e` est rangé dans une file d'attente instance de `java.awt.EventQueue`. Cette file est alimentée en événements par la *JVM* et un processus dédié (*EDT*, *Event Dispatch Thread*) la vide en traitant, un par un, les événements qu'elle contient : pour chaque événement traité, l'ensemble des écouteurs enregistrés sur la source de cet événement sont notifiés par envoi d'un message adapté qui transmet l'événement en argument.

Remarque : si un composant est la source d'un événement alors qu'aucun écouteur de ce type ne lui a été associé, il ne se passera rien.

2.2.3) Taxonomie des événements

Un `java.awt.event.ActionEvent` indique qu'un composant a été activé (par exemple un clic sur un bouton). En conséquence, la méthode `actionPerformed` de chaque `ActionListener` préalablement associé à la source est exécutée.

Les composants susceptibles d'être source d'`ActionEvent` sont :

- `JButton`, `JCheckBox`, `JRadioButton`, `JToggleButton`, `JMenuItem` ;
- `JComboBox` ;
- `JFileChooser` ;
- `JFormattedTextField`, `JPasswordField`, `JTextField`.

Et tout composant Swing peut être écouté au moins sur les événements suivants :

- `ComponentEvent` (changement de taille, de position ou de visibilité)
- `FocusEvent` (gain ou perte du focus)
- `KeyEvent` (frappe de touche tandis que le composant a le focus)
- {`Mouse`,`MouseMotion`,`MouseWheel`}`Event` (événements liés à la souris, à son déplacement ou à l'activation de sa molette).

Mais un composant Swing peut éventuellement être source d'autres types d'événements :

- `HierarchyEvent` (indique l'ajout, la suppression ou le changement de visibilité à l'écran opéré sur l'un des parents du composant)
- `HierarchyBoundsEvent` (indique le déplacement ou le redimensionnement d'un parent du composant)
- `ContainerEvent` (indique l'ajout ou le retrait de composant du conteneur)
- etc.

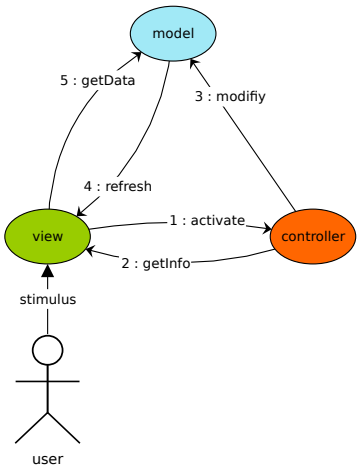
Nous verrons cela plus en détail au second semestre...

3) Architecture MVC

Quand on construit du logiciel objet qui présente ses données à travers une interface graphique, on a l'habitude d'organiser les classes qui le constituent en plusieurs parties, chacune dédiée à une fonctionnalité bien précise. Ceci dans le but d'optimiser les qualités de lisibilité, de simplicité, d'extensibilité et de maintenance du logiciel.

L'un des découpages possibles est l'architecture MVC, pour Modèle - Vue - Contrôleur. C'est cette architecture que nous adopterons systématiquement pour construire nos applications graphiques.

3.1) MVC à l'origine



Au sein d'une application développée selon une architecture MVC, le code est organisé en trois parties distinctes qui collaborent étroitement. Elles concernent respectivement la gestion des données (modèle), le rendu graphique (vue) et la communication entre l'application et l'utilisateur (contrôleur).

Le modèle gère la logique applicative du logiciel, c'est-à-dire qu'il encapsule les données nécessaires à l'application, le moyen d'y accéder et la manière de les modifier. Lorsque ses données changent de valeur, le modèle notifie les vues préalablement enregistrées afin qu'elles puissent se mettre à jour.

La vue gère le rendu graphique de l'application, c'est-à-dire qu'elle spécifie la manière dont il faut afficher les données que contient le modèle, auxquelles elle accède à travers les services que propose ce dernier. Elle se déclare auprès du modèle qui pourra ainsi la notifier lorsqu'une mise à jour sera nécessaire. Enfin, elle transmet les actions de l'utilisateur au contrôleur.

Le contrôleur définit le comportement de l'application. Il reçoit les actions de l'utilisateur qui lui sont transmises par la vue, récupère éventuellement des informations à partir de la vue et traduit ces actions, ainsi paramétrées, en modifications à apporter au modèle.

L'association entre une vue et un contrôleur est bidirectionnelle car ils doivent pouvoir communiquer l'un avec l'autre.

À un modèle on peut associer plusieurs vues qui le représenteront, chacune, totalement ou en partie seulement. Le mode de notification des vues par le modèle peut être de deux sortes :

- de type *push*, la notification contient toutes les informations nécessaires à la mise à

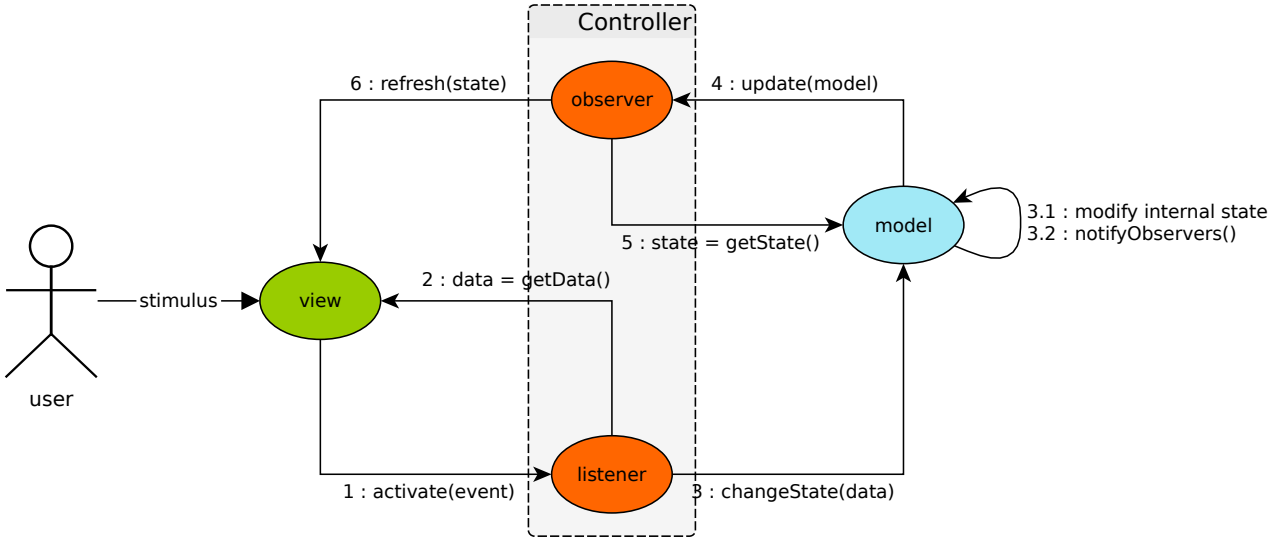
jour de la vue ;
- de type *pull*, la vue devra consulter le modèle pour y puiser les informations nécessaires à son rafraîchissement.

Si d'autres objets que le contrôleur peuvent modifier l'état du modèle, il est nécessaire d'implanter le mécanisme de notification à l'intérieur du modèle puisque c'est alors le seul moyen efficace d'assurer la cohérence entre l'état du modèle et l'apparence de la vue.

En dehors du mécanisme de notification, le modèle n'a pas lieu de communiquer avec ses vues ; il n'a par ailleurs aucune connaissance des contrôleurs qui leurs correspondent.

Cette description correspond à l'architecture d'origine, telle qu'elle était réalisée en Smalltalk-80. Dans ce langage, MVC était implanté directement dans des classes bas-niveau (View, Controller, Model et Object) qu'il suffisait de réutiliser par héritage.

3.2) MVC en Java



Dans un langage comme Java, les développeurs sont fortement incités à réutiliser cette architecture ; le mécanisme de notification (de type *pull*) est alors encapsulé dans la classe `java.util.Observable` dont devra dériver le modèle.

La dépendance entre le modèle et ses vues est externalisée dans l'interface `java.util.Observer` où elle est réduite à sa plus simple expression (une seule méthode : `update`). Concernant la relation vue-contrôleur, c'est la notion d'écouteur qui est utilisée. La notion de contrôleur se trouve donc éclatée en deux parties : un observateur (relation contrôleur-modèle) et un écouteur (relation vue-contrôleur).

On notera bien que la vue et le modèle sont ici complètement indépendants l'un de l'autre ; il suffit que la vue puisse gérer une séquence d'écouteurs (mécanisme implanté dans les composants graphiques de l'*API*) et le modèle une séquence d'observateurs (mécanisme implanté dans la classe `Observable`).

4) Composants non graphiques AWT

4.1) `java.awt.Color`

`Color` est la classe qui implante la notion de couleur définie par un système de couleurs. Le modèle par défaut est RGB, un système à trois composantes : Rouge, Vert et Bleu.

Chaque composante est un entier compris entre 0 et 255 ; à cela on peut ajouter une valeur alpha (opacité) fixée entre 0.0 et 1.0 (ou entre 0 et 255). L'opacité complète est représentée par 1.0 (ou 255).

Un grand nombre de constantes sont fournies en standard : `BLUE`, `WHITE`, `RED`, ...

4.2) `java.awt.Dimension`

`Dimension` est une classe qui implante les notions de largeur et de hauteur d'un composant. Elle propose une implantation sous forme de deux attributs entiers directement accessibles :

- **`int`** `width` : la largeur
- **`int`** `height` : la hauteur

Elle permet de fixer la taille des composants (en pixels) par appel à la méthode **`void`** `setSize(Dimension)`.

4.3) `java.awt.Point`

Cette classe implante la notion de coordonnées cartésiennes à deux dimensions. Elle propose une implantation sous forme de deux attributs entiers directement accessibles :

- **`int`** `x` : la coordonnée horizontale
- **`int`** `y` : la coordonnée verticale

On y trouve aussi quelques méthodes utiles :

- **`void`** `setLocation(int x, int y)` : fixe les coordonnées
- **`void`** `setLocation(Point p)` : fixe les coordonnées
- **`void`** `translate(int dx, int dy)` : déplace les coordonnées

4.4) `java.awt.Rectangle`

Cette classe implante la notion de rectangle défini par la position de son coin supérieur gauche (`x`, `y`) et de sa dimension (`width`, `height`). Elle propose une implantation sous forme d'attributs entiers directement accessibles :

- **`int`** `x` : coordonnée horizontale du coin supérieur gauche du rectangle
- **`int`** `y` : coordonnée verticale du coin supérieur gauche du rectangle
- **`int`** `width` : largeur du rectangle
- **`int`** `height` : hauteur du rectangle

Elle fournit aussi quelques méthodes utiles :

- `boolean contains(Rectangle)` : retourne `true` ssi le rectangle contient le paramètre
- `boolean intersects(Rectangle)` : retourne `true` ssi le rectangle possède une intersection non vide avec le paramètre
- `Rectangle union(Rectangle)` : construit puis retourne le plus petit rectangle contenant le rectangle et le paramètre

4.5) `java.awt.Toolkit`

Classe (abstraite) servant de lien entre les classes de composants graphiques *AWT* et les composants pairs du système sous-jacent.

Par exemple, le bouton pair nécessaire à l'existence d'un bouton *AWT* est créé par appel de la méthode `java.awt.peer.ButtonPeer createButton(Button)`.

Mais cette classe propose aussi des outils comme :

- `static Toolkit getDefaultToolkit()` : donne le *toolkit* de la plate-forme sous-jacente
- `void beep()` : émet un bip sonore
- `Dimension getScreenSize()` : retourne la taille de l'écran
- etc.

5) Classe `JComponent`

`javax.swing.JComponent` est la classe de base pour tous les composants graphiques Swing. Elle dérive de `java.awt.Container`, qui dérive elle-même de `java.awt.Component` de sorte qu'un composant Swing est toujours placé sur un conteneur parent (`getParent()`) et qu'il peut posséder des sous-composants (`getComponent(int)`). En pratique, certains composants Swing sont spécialement conçus pour être des conteneurs (comme `JPanel` par exemple) mais les autres n'en contiennent pas.

5.1) Comportement de base des composants graphiques

- `boolean isEnabled()` : donne l'état d'activation de ce composant (capacité à émettre des événements)
- `void setEnabled(boolean)` : fixe l'état d'activation de ce composant
- `boolean isVisible()` : donne l'état de visibilité de ce composant (et de ses sous-composants)
- `void setVisible(boolean)` : fixe l'état de visibilité de ce composant (et de ses sous-composants)
- `Color getBackground()` : donne la couleur de fond de ce composant
- `void setBackground(Color)` : fixe la couleur de fond de ce composant
- `Color getForeground()` : donne la couleur du texte de ce composant
- `void setForeground(Color)` : fixe la couleur du texte de ce composant

5.2) Comportement associé à la taille des composants

- `Dimension getSize()` et **`void`** `setSize(Dimension)` : donnent/fixent la taille actuelle de ce composant indépendamment de sa taille préférée
- `Dimension get{Minimum|Preferred|Maximum}Size()` : donnent la taille minimum, préférée ou maximum de ce composant
- **`void`** `set{Minimum|Preferred|Maximum}Size(Dimension)` : fixent la taille minimum, préférée ou maximum de ce composant

Attention, fixer la taille d'un composant avec `setSize` est bien souvent inefficace. En effet, comme on le verra plus loin, la gestion de la taille et de l'emplacement d'un composant est déléguée à un gestionnaire de répartition qui utilise, lui aussi, la méthode `setSize` systématiquement pour redimensionner le composant en fonction du contexte. Généralement, mieux vaut définir la taille préférée et laisser le gestionnaire de répartition s'en inspirer pour fixer la taille effective du composant.

6) Conteneurs

La classe `javax.swing.JPanel` est la classe Swing de base pour les conteneurs de composants légers. Pour pouvoir être affiché, un composant Swing doit être placé sur un panneau, lui même placé sur un panneau, et on remonte ainsi jusqu'au panneau principal de la fenêtre (accessible par `Container JFrame.getContentPane()`).

Nous utiliserons systématiquement la classe `JPanel` pour constituer les arborescences des composants graphiques de nos applications.

On peut créer un conteneur vide de composants graphiques à l'aide du constructeur sans argument de la classe `JPanel`. Ensuite, pour placer les composants sur ce `JPanel` on utilise une des deux méthodes disponibles `add(Component)` ou `add(Component, Object)` :

```
JPanel p = new JPanel();
p.add(unComposant[, ...]);
```

dans la version avec un second paramètre, il s'agit d'une contrainte de placement ; nous verrons cela plus loin, lors de l'étude des `BorderLayout`...

7) Fenêtres

Une application graphique Swing crée (au moins) une `javax.swing.JFrame` dans laquelle elle placera tous ses composants. La classe racine déclare donc un attribut de type `JFrame` et configure cette fenêtre selon ses besoins.

`JFrame` hérite de `java.awt.Frame` qui est un composant lourd : l'instance de `JFrame` sera donc chargée d'afficher tous les composants (légers) de l'application.

Pour créer une fenêtre initialement non affichée à l'écran (c'est-à-dire invisible) on dispose de deux constructeurs :

- `JFrame()` : une fenêtre sans titre
- `JFrame(String title)` : une fenêtre avec un titre

Ensuite, on peut la rendre visible ou invisible à l'écran :

- `void setVisible(boolean b)` : la fenêtre devient visible si `b == true`, et invisible sinon
- `boolean isVisible()`

interagir avec sa taille courante :

- `Dimension getSize()`
- `void setSize(Dimension d)`
- `void setSize(int width, int height)`

consulter ou modifier ses tailles minimum, préférée et maximum :

- `Dimension get{Minimum|Preferred|Maximum}Size()`
- `void set{Minimum|Preferred|Maximum}Size(Dimension d)`

et lui demander, ainsi qu'à tous ses sous-composants, de prendre leur taille initiale en accord avec la politique de placement des gestionnaires de répartition :

- `void pack()`

On peut aussi placer une fenêtre à l'écran avec :

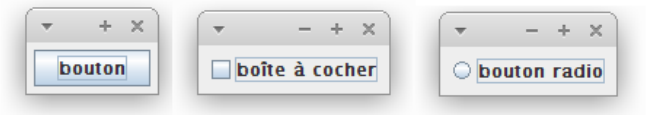
- `void setLocationRelativeTo(Component c)` : la fenêtre sera centrée à l'écran si `c == null || !c.isVisible()` ; et elle sera centrée sur `c` sinon

et indiquer le comportement attendu lorsque l'utilisateur ferme la fenêtre avec :

- `void setDefaultCloseOperation(int n)`
  - si `n == DO_NOTHING_ON_CLOSE` : ne fait rien, le programmeur doit implémenter cette action par ailleurs
  - si `n == HIDE_ON_CLOSE` : rend la fenêtre invisible (ne libère pas les ressources système associées) *valeur par défaut !*
  - si `n == DISPOSE_ON_CLOSE` : détruit la fenêtre mais ne quitte pas l'application (libère les ressources système associées)
  - si `n == EXIT_ON_CLOSE` : quitte l'application

8) Composants de base

8.1) Boutons



Il existe plusieurs classes de boutons :

- `JButton` : classe des boutons classiques (à cliquer).
- `JCheckBox`: classe des cases à cocher
- `JRadioButton` : classe des boutons radio

Elles descendent toutes de la classe abstraite `AbstractButton` et possèdent un grand nombre de méthodes, parmi lesquelles :

- `String getText()` : donne le texte de ce bouton
- `void setText(String)` : fixe l'étiquette de ce bouton

- `boolean isSelected()` : donne l'état de sélection de ce bouton
- `void setSelected(boolean)` : fixe l'état de sélection de ce bouton

Concernant les boutons, il existe aussi un composant non graphique (`ButtonGroup`) qui permet de regrouper plusieurs boutons afin d'assurer qu'un seul d'entre eux soit sélectionné : quand l'utilisateur sélectionne un bouton du groupe, celui qui était précédemment sélectionné est automatiquement désélectionné par le regroupement.

- `ButtonGroup()` : construit un groupement vide de boutons
- `void add(AbstractButton)` : ajoute un bouton au groupe
- `void remove(AbstractButton)` : retire un bouton du groupe
- `void clearSelection()` : assure qu'aucun bouton n'est sélectionné

Classiquement, on regroupe plutôt des boutons radio comme dans l'exemple ci-dessous :


```
final int n = 4;
final int max = 5;
ButtonGroup bg = new ButtonGroup();
JRadioButton[] radios = new JRadioButton[n];
for (int i = 0; i < radios.length; i++) {
    radios[i] = new JRadioButton(
            "Entre " + (i * max) + " et " + ((i + 1) * max)
    );
    parent.add(radios[i]);
    bg.add(radios[i]);
}
// aucun bouton n'est encore sélectionné
radios[0].setSelected(true);
// le bouton 0 est sélectionné,
// tous les autres sont automatiquement désélectionnés
```

8.2) Zones de texte

Toutes les zones de texte sauf `JLabel` héritent de `TextComponent`.

- `JLabel` : classe des textes sur une seule ligne, non modifiables par l'utilisateur
  - {get|set}Text
- `JTextComponent` : classe mère des composants qui permettent d'afficher du texte modifiable (ou pas) par l'utilisateur
  - {get|set}SelectedText
  - read
  - write
  - {get|set}SelectionStart
  - {get|set}SelectionEnd
  - {is|set}Editable
  - select
  - selectAll

  - `copy`
  - `cut`
  - `paste`

- `JTextField` : classe des textes modifiables, disposés sur une seule ligne.
  - {`get`|`set`}`Columns`
- `JPasswordField` : classe des textes modifiables, sur une seule ligne, masquée à l'utilisateur humain.
  - {`get`|`set`}`EchoChar`
  - `getPassword`
- `JFormattedTextField` : classe des textes modifiables, sur une seule ligne, vérifiant un certain format.
  - {`get`|`set`}`Value`
  - `isEditValid`
  - `commitEdit`
  - {`get`|`set`}`Formatter`
- `JTextArea` : classe des textes modifiables, sur plusieurs lignes
  - {`get`|`set`}`Columns`
  - {`get`|`set`}`Rows`
  - `insert`
  - `append`
- `JEditorPane` : classe des textes modifiables, sur plusieurs lignes, contenant du texte simple, HTML ou RTF.
  - {`get`|`set`}`Text`
  - {`get`|`set`}`Page`
  - {`get`|`set`}`ContentType`
- `JTextPane` : classe des textes modifiables, sur plusieurs lignes, contenant du texte stylisé structuré en paragraphes.
  - {`get`|`set`}`StyledDocument`

Exemple d'utilisation :

```
JLabel titre = new JLabel("Il y a bien longtemps dans une galaxie"
        + " lointaine, très lointaine....");
JTextField tf1 = new JTextField("Darky");
tf1.setEditable(false);
JTextField tf2 = new JTextField("Lucky");
tf2.setEnabled(false);
JTextField tf3 = new JTextField(10);
JTextArea ta1 = new JTextArea(15, 20);
ta1.setEditable(false);
ta1.setText("\n\n\nSi seulement tu connaissais\n"
        + ...);
JTextArea ta2 = new JTextArea(15, 20);
ta2.setText("J'apprendrai à maîtriser Java\n"
        + ...);
JTextArea ta3 = new JTextArea(15, 20);
...
```

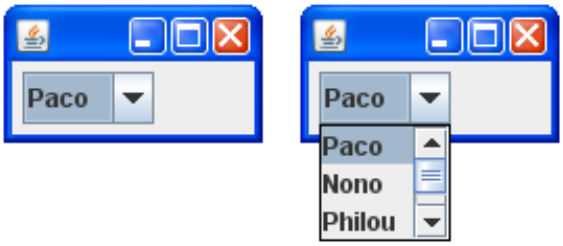
8.3) Listes déroulantes à choix

Les instances de `JComboBox` sont des composants qui résultent de l'association d'un bouton et d'un champ de texte éditable avec une liste déroulante.

- {add|remove}Item
- getItemAt
- getItemCount
- {get|set}SelectedItem
- {get|set}SelectedIndex
- {insert|remove}ItemAt
- {is|set}Editable
- {get|set}MaximumRowCount

Exemple :

```
JComboBox b = new JComboBox();
b.addItem("Paco");
b.addItem("Nono");
b.addItem("Philou");
b.addItem("Jeano");
b.setMaximumRowCount(3);
```

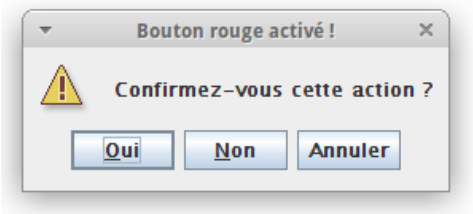


8.4) Boîtes de dialogue

On obtient une boîte de dialogue standard surgissante (pop-up) à partir de l'une des quatre méthodes publiques et statiques de la classe `JOptionPane` :

- `showConfirmDialog` : demande de confirmation
- `showInputDialog` : demande de données texte
- `showMessageDialog` : message (information, erreur, ...)
- `showOptionMessage` : dialogue hautement configurable

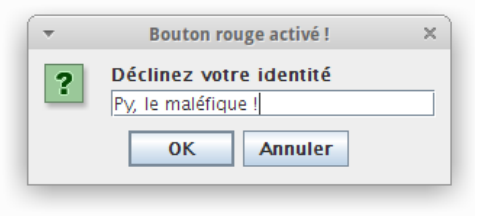
Exemple de confirmation :



```
int res = JOptionPane.showConfirmDialog(
        parent,
        "Confirmez-vous cette action ?",
        "Bouton rouge activé !",
        JOptionPane.YES_NO_CANCEL_OPTION,
        JOptionPane.WARNING_MESSAGE);
if (res == JOptionPane.YES_OPTION) {
    System.out.println("POUET !");
}
```

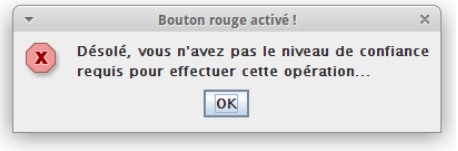
Exemple de récupération de donnée :

```
String res = JOptionPane.showInputDialog(
        parent,
        "Déclinez votre identité",
        "Bouton rouge activé !",
        JOptionPane.QUESTION_MESSAGE);
System.out.println(res);
```



Exemple d'affichage de message :

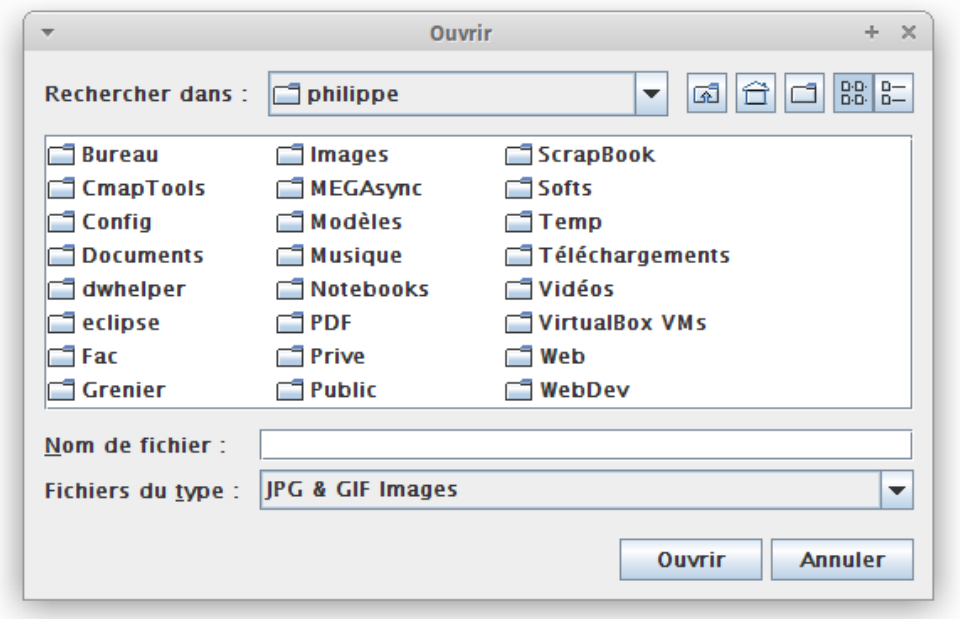
```
JOptionPane.showMessageDialog(
        parent,
        "Désolé, vous n'avez pas "
            + "le niveau de"
            + " confiance\n"
            + "requis pour effectuer "
            + "cette opération...",
        "Bouton rouge activé !",
        JOptionPane.ERROR_MESSAGE);
```



8.5) Sélection de fichiers

Les instances de `JFileChooser` sont des boîtes de dialogue surgissantes permettant à l'utilisateur de choisir un fichier présent dans le système de fichiers.

Exemple :



```
JFileChooser fc = new JFileChooser();
FileNameExtensionFilter flt =
    new FileNameExtensionFilter(
        "JPG & GIF Images", "jpg", "gif");
fc.setFileFilter(flt);
int res = fc.showOpenDialog(parent);
if (res == JFileChooser.APPROVE_OPTION) {
    System.out.println("Fichier ouvert : "
            + fc.getSelectedFile().getName());
}
```

8.6) Défilement des composants

Lorsqu'un composant est trop grand par rapport à la place disponible sur la fenêtre de l'application, l'usage d'un panneau de défilement (instance de `JScrollPane`) permet de ne visualiser qu'une fenêtre rectangulaire sur ce composant, et de déplacer la fenêtre.

8.7) Menus

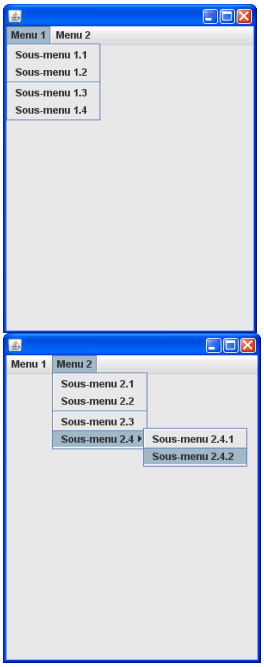
Une barre de menu (instance de `JMenuBar`) est une séquence de menus.

Un menu est la combinaison d'un bouton et d'une fenêtre surgissante contenant des items de menu : quand on clique sur le menu (bouton) son contenu (fenêtre surgissante) apparaît.

Un élément de menu se comporte comme un bouton : quand on clique sur lui, il est la source d'un événement d'action.

Exemple :

```
JMenuBar jmb = new JMenuBar(); {
    JMenu m = new JMenu("Menu 1"); {
        m.add(new JMenuItem("Sous-menu 1.1"));
        m.add(new JMenuItem("Sous-menu 1.2"));
        m.addSeparator();
        m.add(new JMenuItem("Sous-menu 1.3"));
        m.add(new JMenuItem("Sous-menu 1.4"));
    }
    jmb.add(m);
    m = new JMenu("Menu 2"); {
        m.add(new JMenuItem("Sous-menu 2.1"));
        m.add(new JMenuItem("Sous-menu 2.2"));
        m.addSeparator();
        m.add(new JMenuItem("Sous-menu 2.3"));
        JMenu sm = new JMenu("Sous-menu 2.4"); {
            sm.add(new JMenuItem("Sous-menu 2.4.1"));
            sm.add(new JMenuItem("Sous-menu 2.4.2"));
        }
        m.add(sm);
    }
    jmb.add(m);
```



```
}
frame.setJMenuBar(jmb);
```

9) Gestionnaires de répartition

Java capture la notion d'algorithme de répartition des sous-composants d'un conteneur grâce à l'interface `java.awt.LayoutManager` pour les gestionnaires sans contraintes (`FlowLayout` et `GridLayout`) et à son interface dérivée `java.awt.LayoutManager2` pour les gestionnaires avec contraintes (`BorderLayout`).

Ce n'est pas le programmeur qui s'occupe de placer les sous-composants d'un conteneur, c'est le gestionnaire de répartition associé au conteneur qui effectue tous les calculs nécessaires en fonction de sa politique de répartition.

Un gestionnaire de répartition effectue deux tâches importantes pour un conteneur :

- il fixe la taille de chacun des sous-composants en fonction de leur taille préférée, minimale et maximale, de la place disponible et des contraintes liées à la politique de répartition du gestionnaire ;
- il place les sous-composants sur le conteneur, en tenant compte de leur taille courante préalablement fixée.

Tout au long de l'exécution de l'application, la taille ou la disposition des composants graphiques sur un conteneur peut devenir obsolète, ceci pour plusieurs raisons :

- parce que la taille du conteneur a été modifiée ;
- parce que la taille ou la position de l'un de ses sous-composants ont été modifiées ;
- parce qu'un sous-composant a été ajouté, retiré, rendu visible ou invisible.

Dans chacun de ces cas, le conteneur est marqué comme non valide et la taille et la position de ses sous-composants doivent être recalculées *via* un appel à la méthode `layoutContainer` du `LayoutManager` associé. En général, Swing s'occupe de tout automatiquement (par exemple lorsqu'on retaille la fenêtre principale à la souris), mais dans certaines circonstances le programmeur doit s'occuper personnellement de lancer ces calculs (par exemple lorsqu'il ajoute un composant sur un conteneur déjà visible à l'écran). Dans ces cas, il faudra appeler la méthode `void revalidate()`, définie dans `JComponent`, pour que les calculs soient lancés le plus tôt possible.

Les gestionnaires de répartitions utilisent les tailles minimum/préférée/maximum des sous-composants du conteneur associé pour effectuer leurs calculs.

Pour tous les gestionnaire de répartition que nous étudierons, on peut définir l'espacement horizontal et vertical entre les composants dès l'instanciation (constructeurs) ou plus tard ({`get`|`set`}`Hgap`, {`get`|`set`}`Vgap`).

9.1) FlowLayout

Un `FlowLayout` accroche des composants en ligne, de gauche à droite, en leur donnant leur taille préférée. Les composants seront centrés sur la ligne (`CENTER`) ou calés à gauche (`LEFT`) ou à droite (`RIGHT`). Les sous-composants sont ajoutés un à un de la manière suivante :

```
parent.add(component);
```

La politique de répartition d'un `FlowLayout` est la suivante :

1. Démarrer la première ligne
2. Pour chaque composant, dans l'ordre d'ajout :
   1. donner sa taille préférée au composant
   2. si la ligne courante ne peut pas contenir ce composant (par rapport à la taille du conteneur) :
      placer la ligne courante sur le conteneur
      démarrer une nouvelle ligne
   3. ajouter ce composant à la ligne courante
3. Disposer la ligne courante sur le conteneur
4. Les lignes sont poussées vers le haut et les composants ne s'affichent pas complètement si la place est insuffisante.

Initialisation :

- `FlowLayout(int align, int hgap, int vgap)` : un gestionnaire dont la politique de répartition est définie par `align` (`CENTER`, `LEFT` ou `RIGHT`) et dont les espacements horizontaux (`hgap`) et verticaux (`vgap`) sont fixés
- `FlowLayout(int align)` : correspond à `FlowLayout(align, 5, 5)`
- `FlowLayout()` : correspond à `FlowLayout(CENTER, 5, 5)`

9.2) GridLayout

Un `GridLayout` partitionne la surface totale de son conteneur en une grille de cellules toutes de la même taille. On ne peut placer qu'un seul composant par cellule, si l'on veut plusieurs composants dans une même cellule, il faut les disposer sur un `JPanel` et placer ce panneau d'affichage sur la cellule. Chaque composant (ou panneau d'affichage) est étiré au besoin pour remplir totalement l'espace disponible dans sa cellule.

Les sous-composants sont ajoutés de gauche à droite, puis de haut en bas, ainsi :

```
parent.add(component);
```

La politique de répartition d'un `GridLayout` est la suivante :

1. La hauteur commune à toutes les cellules est la plus grande des hauteurs préférées des cellules
2. Si la somme des hauteurs est trop petite ou trop grande, la hauteur commune est recalculée comme le quotient de la hauteur du conteneur par le nombre de lignes
3. Mêmes calculs pour la largeur commune à toutes les cellules...
4. Les composants ne s'affichent pas complètement si la place est insuffisante

Initialisation :

- `GridLayout(int rows, int cols, int hgap, int vgap)` : un

gestionnaire doté de `rows` lignes et `cols` colonnes de cellules, dont l'espacement horizontal est fixé à `hgap` pixels et l'espacement vertical à `vgap` pixels
- `GridLayout()` : correspond à `GridLayout(1, 0, 0, 0)`
- `GridLayout(int rows, int cols)` : correspond à `GridLayout(rows, cols, 0, 0)`

Remarques :

1. si l'on donne `rows` > 0, alors la valeur de `cols` est ignorée
2. on peut placer plus de `rows × cols` composants : le nombre de cellules augmente automatiquement
3. si on veut fixer le nombre de colonnes, il faut donner à `rows` la valeur 0
4. on ne peut pas indiquer 0 à la fois pour `rows` et pour `cols`
5. on peut ne pas remplir complètement la grille

9.3) BorderLayout

Un `BorderLayout` répartit ses composants selon 5 zones spécifiées par les constantes : `NORTH`, `SOUTH`, `WEST`, `EAST`, `CENTER`. On ne peut placer qu'un seul composant par zone. Si l'on veut plusieurs composants dans une même zone, il faut les disposer sur un `JPanel` et placer ce panneau d'affichage sur la zone.

Les sous-composants sont ajoutés ainsi :

```
parent.add(component, BorderLayout.NORTH);
parent.add(component, BorderLayout.EAST);
...
```

La politique de répartition d'un `BorderLayout` est la suivante :

1. Les hauteurs des zones nord et sud sont toujours les hauteurs préférées de leur composant
2. Les largeurs des zones est et ouest sont toujours les largeurs préférées de leur composant
3. La zone centrale est ensuite étirée ou réduite dans chaque direction pour remplir le vide restant
4. Les composants ne s'affichent pas complètement si la place est insuffisante

Initialisation :

- `BorderLayout(int hgap, int vgap)` : un gestionnaire dont l'espacement horizontal est fixé à `hgap` pixels et l'espacement vertical à `vgap` pixels
- `BorderLayout()` : correspond à `BorderLayout(5, 5)`

9.4) Gestionnaires par défaut

Par défaut, un `JPanel` est muni d'un `FlowLayout` configuré pour centrer les composants.

Par défaut, le content pane d'une `JFrame` est doté d'un `BorderLayout`.

On peut changer le gestionnaire de répartition d'un conteneur à tout moment par appel à la méthode **`void`** `setLayout(LayoutManager)`.

9.5) Affichage des composants à l'écran

- `JComponent.isDisplayable` : la méthode **`boolean`** `isDisplayable()` indique si le composant cible est affichable ou pas.
- `JComponent.isVisible` : la méthode **`boolean`** `isVisible()` indique si le composant cible a été marqué comme visible ou pas. Initialement, une fenêtre est invisible mais tous ses sous-composants sont visibles.
- `JComponent.setVisible` : la méthode **`void`** `setVisible(`**`boolean`**`)` marque le composant cible comme visible ou non.

Un composant graphique est affichable lorsqu'il est relié à une arborescence graphique enracinée sous un composant associé à une ressource native du système de fenêtrage (une instance de `JFrame`, `JDialog`, `JApplet` ou `JWindow`). Il devient affichable au moment où il est ajouté à une arborescence graphique déjà affichable ou bien, s'il était déjà présent au sein d'une arborescence non affichable, lorsque celle-ci devient affichable. De même, il devient non affichable sitôt qu'il est retiré de son arborescence affichable directement ou non (appel de `dispose()` sur la fenêtre racine, par exemple).

Un composant graphique est visible lorsque il retourne **`true`** à l'appel `isVisible()`.

Finalement, pour être effectivement dessiné à l'écran, un composant doit être affichable et toute sa super-hiérarchie de composants graphiques doit être visible.

9.6) Validité des composants

- `JComponent.isValid` : la méthode **`boolean`** `isValid()` indique si la taille et la position du composant sont en accord avec la politique de répartition du conteneur parent. Si ce composant est en fait un conteneur, cela signifie aussi que tous les sous-composants de ce composant, dans l'arborescence graphique à laquelle il appartient, sont valides eux aussi.
- `JComponent.validate` : la méthode **`void`** `validate()` appliquée à un composant qui n'est pas un conteneur le marque comme valide. Elle est redéfinie dans la classe `Container` pour provoquer la (re)mise en place des sous-composants par les gestionnaires de répartition appropriés, puis marquer comme valides le conteneur lui-même ainsi que ses sous-composants.
- `JComponent.invalidate` : la méthode **`void`** `invalidate()` marque le composant cible comme non valide, de sorte qu'il sera candidat à une remise en place lorsque le gestionnaire de son conteneur parent appliquera sa politique de répartition. Marquer un composant comme non valide marque aussi tous ses

conteneurs parents comme non valides, jusqu'à la fenêtre racine de l'arborescence graphique à laquelle il appartient.

À la création d'un composant, celui-ci est initialement marqué comme invalide. Puis lorsque la racine de l'arborescence graphique à laquelle il appartient est associée à une ressource native du système de fenêtrage, le composant, ainsi que tous ceux de son arborescence graphique, sont marqués comme valides. Par la suite, chaque composant peut devenir valide ou invalide en fonction des événements qui surviendront au cours de la vie de l'application. Par exemple, les actions de redimensionnement de la fenêtre à l'aide de la souris sont susceptibles d'invalider un certain nombre de composants, qui seront ensuite automatiquement revalidés.

La méthode `void pack()` appliquée à une fenêtre réalise l'association de cette fenêtre à une ressource native du système de fenêtrage puis, elle valide récursivement tous les composants de l'arborescence graphique dont cette fenêtre est la racine. Tous ces composants se trouvent donc correctement dimensionnés, positionnés et validés, et ils sont donc affichables à l'écran (mais pas encore affichés...).

10) Création de nouveaux composants

10.1) Classe `Graphics`

La classe abstraite `java.awt.Graphics` définit la notion de contexte graphique grâce auquel les composants peuvent se dessiner. Elle encapsule un certain nombre de données :

- le composant sur lequel on dessine,
- le repère relatif à ce composant,
- ...

et donne accès à des primitives de dessin :

- `void draw{Arc|Ovale|Rect|String|Line}(...)` : pour dessiner au trait ;
- `void fill{Arc|Ovale|Rect}(...)` : pour dessiner une forme remplie de couleur.

Attention : l'origine du repère (0, 0) est située en haut à gauche. Par conséquent :

l'axe des abscisses est dirigé vers la droite
l'axe des ordonnées est dirigé vers le bas

10.2) Dessin d'un composant

Pour dessiner immédiatement un composant à l'écran il faut appeler sa méthode `public void paint(Graphics)`. Celle-ci fait elle-même appel à trois méthodes protégées :

- `protected void paintComponent(Graphics)` : peint le composant lui-même
- `protected void paintBorder(Graphics)` : peint la bordure du composant
- `protected void paintChildren(Graphics)` : peint les sous-composants du composant

Il vaut mieux demander la réactualisation de l'affichage avec `public void repaint()`,

qui commande une exécution de `paint` dès que possible.

10.3) Définition d'un nouveau composant

Pour définir un nouveau composant qui devra se dessiner, il faut dériver la classe qui convient. Normalement `JComponent`, en pensant à redéfinir les tailles du composant qui valent par défaut :

minimum = (0, 0)
préférée = (0, 0)
maximum = (`Short.MAX_VALUE`, `Short.MAX_VALUE`)

ou éventuellement `JPanel`. Très rarement une autre classe (`JButton`, ...).

Puis il faut redéfinir `paintComponent` ainsi :

```
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    // dessiner sur g
}
```

11) Recommandations pour le codage d'une application

11.1) Structure générale du code

Pour ce cours, j'exige que la mise en forme de votre code respecte les recommandations suivantes :

1. Ne pas afficher la fenêtre depuis le constructeur : on définit plutôt une méthode `public void display()` qui configure et affiche la fenêtre, cette méthode sera appelée sur l'instance qui représente l'application lorsqu'elle sera créée.
2. Si la création du modèle est complexe, créer le modèle dans une méthode dédiée `private void createModel()`. C'est la seule recommandation qui puisse ne pas être suivie, lorsque la création du modèle ne requiert qu'une création toute simple.
3. Créer les composants graphiques majeurs dans une méthode dédiée `private void createView()`.
4. Positionner les composants graphiques majeurs et mineurs sur la fenêtre dans une méthode dédiée `private void placeComponents()`.
5. Définir le contrôleur (écouteurs de la vue et observateurs du modèle) dans une méthode dédiée `private void createController()`. Utiliser des classes internes anonymes pour définir les observateurs et les écouteurs si le code n'est pas trop long.
6. Démarrer l'application en appelant la méthode `display` sur une nouvelle instance de l'application, par invocation de la « formule magique » :

```
public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
```

```
        public void run() {
            new Appli().display();
        }
    });
}
```

11.2) Visualisation de l'arborescence des composants graphiques

Les composants graphiques disposés sur une fenêtre constituent une arborescence dont la racine est une `JFrame`. Pour la bonne lecture du code, il est important de pouvoir visualiser cette arborescence graphique, dans la méthode `placeComponents`, soit à l'aide de la grammaire suivante :

topLevel → seq(`frame`)
seq(`p`) → elemP(`p`) seq(`p`) | elemC(`p`) seq(`p`) | ε
elemP(`p`) →

```
    [JPanel] q = new JPanel([lm_instance]);
    { //--
        [q.setLayout(lm_instance);]
        seq(q)
    } //--
    p.add(q[, args]);
```

elemC(`p`) →

```
    p.add(component[, args]);
```

soit à l'aide de règles de ré-écriture exécutées lors d'un parcours en profondeur de l'arborescence des composants graphiques (la variable ind représente le niveau d'indentation des lignes à afficher, initialement 0). Durant ce parcours, lorsqu'on arrive sur un nœud pour la première fois, on exécute la cellule adaptée de la colonne « arrivée » ci-dessous, et lorsqu'on quitte le nœud définitivement, on exécute la cellule adaptée de la colonne « départ » :

	arrivée	départ
racine	`private void placeComponents() {` ind ← ind + 1	ind ← ind - 1 `}`

noeud interne	`[JPanel ]p = new JPanel([...]);`
	`{ //--`
	ind ← ind + 1
		ind ← ind - 1
		`} //--`
		`parent.add(p[, ...]);`

feuille	*ne rien faire*	`p.add(c[, ...]);`

De plus, il faut toujours respecter les règles suivantes :

Lorsqu'on passe par un nœud interne :
- on utilise une variable locale d'une seule lettre (`p`, `q`, `r`, `s`, ...)
- on doit réutiliser le plus possible ces variables
- on crée toujours un nouveau conteneur (`new JPanel(...)`)

Lorsqu'on passe par une feuille :
- on n'utilise jamais de variable locale
- on utilise un composant graphique majeur

ou sinon on crée un composant graphique anonyme (forcément mineur)

Voici un exemple d'arborescence de composants graphiques (ici, à part la `JFrame`, les composants graphiques sont mineurs) :

```
JFrame
     JPanel à FlowLayout CENTER
          JButton de texte "1"
          JPanel à FlowLayout CENTER
               JButton de texte "2"
               JButton de texte "3"
          JButton de texte "4"
     JPanel à FlowLayout CENTER
          JLabel de texte "Nombre de clics : "
          JLabel de texte "0"
```

qui donne le code ainsi mis en forme :

```
private void placeComponents() {
    JPanel p = new JPanel();
    { //--
        p.add(new JButton("1"));
        // commentaire sur le rôle de ce panel
        JPanel q = new JPanel();
        { //--
            q.add(new JButton("2"));
            q.add(new JButton("3"));
        } //--
        p.add(q);
        p.add(new JButton("4"));
    } //--
    frame.add(p, ...);
    p = new JPanel();
    { //--
        p.add(new JLabel("Nombre de clics : "));
        p.add(new JLabel("0"));
    } //--
    frame.add(p, ...);
}
```

Notez que, sur cet exemple, tous les gestionnaires de répartition sont des `FlowLayout` (par défaut sur les panneaux d'affichage) à l'exception de celui de la fenêtre qui est un `BorderLayout`. Notez aussi que les méthodes `add` de `JFrame` sont des raccourcis qui permettent d'associer en réalité les panneaux secondaires au panneau principal de la fenêtre (celui que l'on obtient par appel à `getContentPane()`).

Appendice : Exécution de tâches régulièrement

Cet appendice n'est pas au programme, il est donné comme aide pour les projets que vous auriez à faire ce semestre dans d'autres matières, si vous souhaitez coder des animations (très limitées) en Java... en attendant le cours d'IHM du second semestre qui développera bien plus ces aspects de programmation concurrente.

1) Processus et *threads*

Un processus est un élément permettant de réaliser l'exécution concurrente de code au sein d'une même machine. Il est défini par :

- un ensemble d'instructions à exécuter
- un pointeur d'instruction (qui indique la prochaine instruction à exécuter)
- une zone de mémoire personnelle (pour les données)
- une pile d'exécution personnelle (pour les variables locales, ...)

Un processus ne peut accéder à aucune zone mémoire allouée à un autre processus.

Un thread (tâche, processus léger) est un élément créé par un processus, et permettant de réaliser l'exécution concurrente de code au sein d'une même machine. Il est défini par :

- un ensemble d'instructions à exécuter
- un pointeur d'instruction (qui indique la prochaine instruction à exécuter)
- une pile d'exécution personnelle (pour les variables locales)

Tout *thread* issu d'un processus partage la zone de mémoire personnelle de son père.

Le système réalise des exécutions concurrentes de code en partageant son temps CPU entre les différents *threads* et processus à l'aide d'un ordonnanceur.

La communication entre deux processus ou entre deux *threads* est possible :

- Deux processus communiquent entre eux par l'intermédiaire d'un mécanisme externe à l'application, géré par le système.
- Deux *threads* communiquent entre eux par l'intermédiaire des variables qu'ils partagent.

On suppose qu'entre deux instructions consécutives, un attribut ne change pas spontanément de valeur (c'est-à-dire que si on ne lui affecte pas une nouvelle valeur, elle n'a pas pu changer de valeur toute seule). Cette hypothèse est valide si le code est exécuté par un seul processus mais elle peut être fausse si le code est exécuté par un *thread* puisque plusieurs *threads* partagent le même espace mémoire (celui de leur père).

Lorsque le système donne la main à un processus ou à un *thread*, il doit en sauvegarder l'état courant avant de passer au suivant (on parle de changement de contexte). Pour un processus, le changement de contexte peut être très coûteux (beaucoup de mémoire à sauvegarder, éventuellement sur un disque). Pour un *thread* le changement de contexte est généralement très rapide (peu de mémoire à sauvegarder).

Le processus associé à l'exécution d'une *JVM* donne vie à un grand nombre de *threads* parmi lesquels :

- « main thread » : celui qui exécute la méthode main de l'application
- « event dispatch thread » : celui qui assure la gestion événementielle de Java ainsi que l'affichage des composants
- « toolkit thread » : celui qui traduit les événements système en événements Java
- les « user threads » créés par le code (définis par le programmeur)

2) Règle d'or n° 1

La plupart des appels de méthodes des composants Swing risquent de produire des erreurs d'inconsistance de la mémoire si les invocations sont faites à partir de plusieurs *threads* (*Swing is not thread-safe*).

Pour éviter tout problème de synchronisation une règle d'or s'impose :

« Sur EDT, les composants Swing tu manipuleras ! »

C'est pourquoi nos applications sont lancées par EDT de la manière suivante :

```
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        new Application().display();
    }
}
```

3) Règle d'or n° 2

EDT permet :

- de gérer les événements stockés dans la file d'attente
- d'activer les écouteurs
- de peindre les composants
- ...

Pour éviter de figer l'application il faut respecter la règle d'or :

« Des appels gourmands en temps CPU, EDT tu ne surchargeras point ! »

4) Décomposition en sous-tâches : les timers

Dans certains cas (typiquement les animations) on peut découper la tâche de départ en une séquence de tâches suffisamment courtes.

Par exemple l'animation d'un objet à l'écran peut être découpée en mouvements élémentaires exécutés à intervalles réguliers. On utilisera alors `javax.swing.Timer` qui génère des événements de type action à intervalles réguliers. Ces événements sont écoutés par un `ActionListener` dont la méthode `actionPerformed` est exécutée sur EDT, elle peut donc mettre à jour des composants graphiques à condition que la tâche soit assez rapide.

Exemple typique :

```
// Implémente un compte à rebours entre MAX et 0 dans le champs tf
// avec un délai de DELAY ms entre deux émissions d'ActionEvent
```

```
javax.swing.Timer timer = new javax.swing.Timer(
        DELAY,
        new ActionListener() {
            int cpt = MAX + 1;
            public void actionPerformed(ActionEvent e) {
                if (cpt > 0) {
                    cpt -= 1;
                    tf.setText(String.valueOf(cpt));
                } else {
                    // c'est terminé, on stoppe le timer
                    timer.stop();
                    // et on réarme le compteur
                    cpt = MAX + 1;
                }
            }
        }
);
...
timer.start();   // démarre
...              // stop au bout de (MAX + 1) * DELAY ms
timer.restart(); // redémarre
...
```

Tous les `DELAY` ms, un `ActionEvent` est généré par le timer et EDT exécute la méthode réflexe de l'`ActionListener` pour cet événement. Puisque ce code est court, il ne fige pas l'application.

Ce code contient sa propre condition d'arrêt : quand `cpt` devient nul, on stoppe le timer. On peut ensuite le relancer pour un nouveau cycle en appelant `timer.restart()`.