

Programmation objet en Java

Xavier Crégut

<cregut@enseeiht.fr>

Septembre 2005

Objectifs du cours

Les objectifs de ce cours sont :

- la programmation (orientée) objet en l'illustrant avec le langage Java ;
- l'utilisation de la notation UML (Unified Modeling Language) pour représenter l'architecture statique du système (diagramme de classes) ;
- les exceptions ;
- la programmation par contrat ;
- les interfaces graphiques (Swing) et la programmation événementielle ;
- des éléments méthodologiques.

Remarque : Même si le langage cible est Java, les concepts présentés ici peuvent être appliqués dans le cadre d'autres langages objets (C++, Eiffel, etc.).

Références

- [1] Cay S. Horstmann and Gary Cornell. *Au cœur de Java 2*, volume 1. Campus Press, 2001.
- [2] Irène Charon. *Le langage Java, concepts et pratique*. Hermes, 2000.
- [3] David Flanagan. *Java in a Nutshell*. O'Reilly, 2001.
- [4] Bruce Eckel. *Thinking in Java*. Prentice-Hall, 2000. version PDF sur <http://www.bruceeckel.com>.
- [5] Sun. The Source for Java Technology. <http://java.sun.com>.
- [6] Bertrand Meyer. *Object-oriented software construction*. Prentice Hall, 2nd edition, 1997.
- [7] Pierre-Alain Muller and Nathalie Gaertner. *Modélisation objet avec UML*. Eyrolles, 2^e édition, 2000.
- [8] OMG. UML Resource Page. <http://www.omg.org/uml/>.

Plan du cours

- Exemple introductif : approche objet vs approche traditionnelle (C)
- La plateforme Java : caractéristiques et outils
- Algorithmique en Java : types, opérateurs, structures de contrôle
- Paquetages : structuration d'une application
- Encapsulation : Classes et objet
- Tableaux, String et autres classes de l'API Java
- Spécification de comportement : les interfaces
- Paramétrisation : la généricité
- Héritage et concepts associés : polymorphisme, liaison tardive, classes abstraites
- Responsabilité d'une classes : Exceptions et Programmation par contrat
- Quelques API Java
- Interfaces graphiques (autre support)

Approche objet : Exemple introductif

Exercice 1 : Équation du second degré

Comment résoudre (afficher les solutions de) l'équation du second degré :

- avec une approche traditionnelle (langage impératif ou fonctionnel) ;
- avec une approche objet.

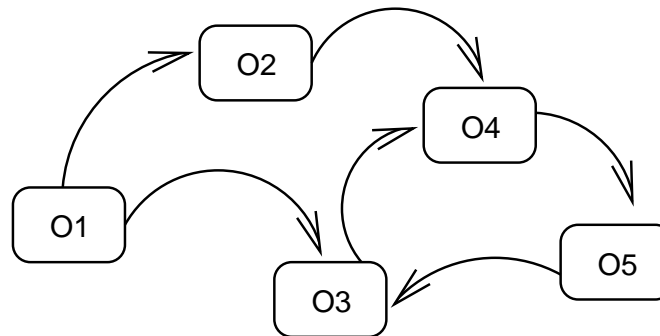
Remarque : On se limite au cas général d'une équation avec deux racines réelles.

Les équations : version « traditionnelle »

```
1 public class ÉquationTraditionnelle {
2     public static void main (String[] args) {
3         double a, b, c;          // coefficients de l'équation
4
5         // Initialiser les coefficients
6         a = 1;
7         b = 5;
8         c = 6;
9
10        // Calculer le discriminant
11        double delta = (b * b - 4 * a * c);
12
13        // Calculer les racines (sans contrôle de cohérence)
14        double x1 = (-b + Math.sqrt(delta)) / 2 / a;
15        double x2 = (-b - Math.sqrt(delta)) / 2 / a;
16
17        // Afficher les résultats
18        System.out.println("Racine_1:_:" + x1);
19        System.out.println("Racine_2:_:" + x2);
20    }
21 }
```

Approche objet : changement de façon de penser

Principe : Pour pouvoir penser une solution informatique, il faut substituer à la notion de PROGRAMME la notion d'ORGANISME, ensemble d'objets pouvant communiquer entre eux.



« Les systèmes logiciels sont caractérisés au premier chef par les objets qu'ils manipulent, non par la fonction qu'ils assurent.

Ne demandez pas **CE QUE FAIT LE SYSTÈME.**

Demandez **À QUI IL LE FAIT ! »**

Bertrand Meyer

Les équations : version (plus) objet

```
1  /** Modélisation d'une équation du second degré et de sa résolution.
2   * @author Xavier Crégut
3   * @version 1.3
4   */
5  class Équation {
6
7      /** Coefficients de l'équation */
8      double coeffA, coeffB, coeffC;
9
10     /** Solutions de l'équation */
11     double x1, x2;
12
13     /** Déterminer les racines de l'équation du second degré. */
14     void résoudre() {
15         double delta = // variable locale à la méthode résoudre
16             this.coeffB * this.coeffB - 4 * this.coeffA * this.coeffC;
17         this.x1 = (- this.coeffB + Math.sqrt(delta)) / 2 / this.coeffA;
18         this.x2 = (- this.coeffB - Math.sqrt(delta)) / 2 / this.coeffA;
19     }
20 }
```

Attention : Cette classe Équation est simpliste !

Les équations : version (plus) objet

Une équation est caractérisée par ses coefficients, ses racines et le fait que la résoudre consiste à calculer ses racines en fonction de ses coefficients.

On en déduit :

- les *attributs* : l'état d'un objet Équation (les coefficients et les racines) ;
- les *méthodes* : les « actions » qui peuvent être réalisées sur un objet de type Équation (résoudre l'équation).

Remarque : On ne s'intéresse qu'au cas général (deux solutions) de l'équation de second degré à coefficients et valeurs dans les réels.

Attention : Outre la remarque précédente, cette classe est un exemple à ne pas suivre : il est seulement introductif !

Exercice 2 En deuxième lecture, expliquer pourquoi cette classe Équation constitue un exemple à ne pas suivre.

Programme principal manipulant les équations

```
1 public class RésolutionÉquation
2 {
3     /** Méthode principale */
4     public static void main (String[] args) {
5         Équation uneÉquation;           // une poignée sur une Équation
6         uneÉquation = new Équation();    // création d'un objet Équation
7
8         // Initialiser les coefficients
9         uneÉquation.coeffA = 1;
10        uneÉquation.coeffB = 5;
11        uneÉquation.coeffC = 6;
12
13        // Calculer les racines de l'équation
14        uneÉquation.résoudre();
15
16        // Afficher les résultats
17        System.out.println("_Racine_1_:_" + uneÉquation.x1);
18        System.out.println("_Racine_2_:_" + uneÉquation.x2);
19    }
20 }
```

Exercice 3 Dessiner l'évolution de la mémoire.

Quelques constatations

- Une classe ressemble à un enregistrement dans lequel on peut mettre à la fois des champs (attributs) et des fonctions (méthodes).
- En Java, on les appelle respectivement attributs et méthodes.
En UML on utilise les termes attributs et opérations.
- Les objets sont toujours créés dans le tas (allocation dynamique) par l'opérateur **new**.
- Les objets ne sont accessibles que par l'intermédiaire de poignées (ou variables d'objet) équivalentes à des pointeurs. Le type de la poignée conditionne les objets qui peuvent lui être attachés.
- Il n'y a pas de `delete` car Java intègre un ramasse-miettes.
- La mémoire des variables locales (types élémentaires ou poignées) est allouée dans la pile (gérée par le compilateur).
- On utilise la notation pointée pour accéder à l'attribut d'un objet ou lui appliquer une méthode.

Autre version en langage C

Une approche traditionnelle intégrant la notion de *type abstrait* nous conduit à identifier deux constituants pour ce petit exercice :

- un **module** décrivant l'équation (le type `Equation` et les opérations associées) ;
- un **programme principal** correspondant à la résolution d'une équation particulière.

En langage C, le module se traduit en deux fichiers :

- un fichier d'entête `equation.h` qui contient la spécification (interface) du module ;
- un fichier d'implantation `equation.c` qui contient l'implantation du module ;

Le programme principal est dans le fichier `test_equation.c`.

Le fichier d'entête du module : equation.h

```
1  /*****
2  *  Objectif :  Modélisation d'une équation du second degré
3  *              et de sa résolution.
4  *  Auteur   :  Xavier CRÉGUT <cregut@enseeiht.fr>
5  *  Version  :  1.2
6  *****/
7
8  #ifndef EQUATION__H
9  #define EQUATION__H
10
11
12  /* Définition du type Equation */
13  struct Equation {
14      double coeffA, coeffB, coeffC; /* coefficients de l'équation */
15      double x1, x2;                /* racines de l'équation */
16  };
17
18  typedef struct Equation Equation;
19
20
21  /* Déterminer les racines de l'équation du second degré.  */
22  void resoudre(Equation *eq);
23
24  #endif
```

Le fichier d'implantation du module : `equation.c`

```
1
2 #include <math.h>
3
4 #include "equation.h"
5
6 void resoudre(Equation *eq)
7 {
8     double delta = /* variable locale à la fonction resoudre */
9                     eq->coeffB * eq->coeffB - 4 * eq->coeffA * eq->coeffC;
10
11     eq->x1 = (- eq->coeffB + sqrt(delta)) / 2 / eq->coeffA;
12     eq->x2 = (- eq->coeffB - sqrt(delta)) / 2 / eq->coeffA;
13 }
```

Exercice 4 Comparer la fonction résoudre en C et la méthode résoudre en Java.

Le programme principal : test_equation.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include "equation.h"
5
6  int main()
7  {
8      Equation uneEquation;      /* notre équation */
9
10     /* Initialiser les coefficients */
11     uneEquation.coeffA = 1;
12     uneEquation.coeffB = 5;
13     uneEquation.coeffC = 6;
14
15     /* Calculer les racines de l'équation */
16     resoudre(&uneEquation);
17
18     /* Afficher les résultats */
19     printf("_Racine_1_:_%f\n", uneEquation.x1);
20     printf("_Racine_2_:_%f\n", uneEquation.x2);
21
22     return EXIT_SUCCESS;
23 }
```

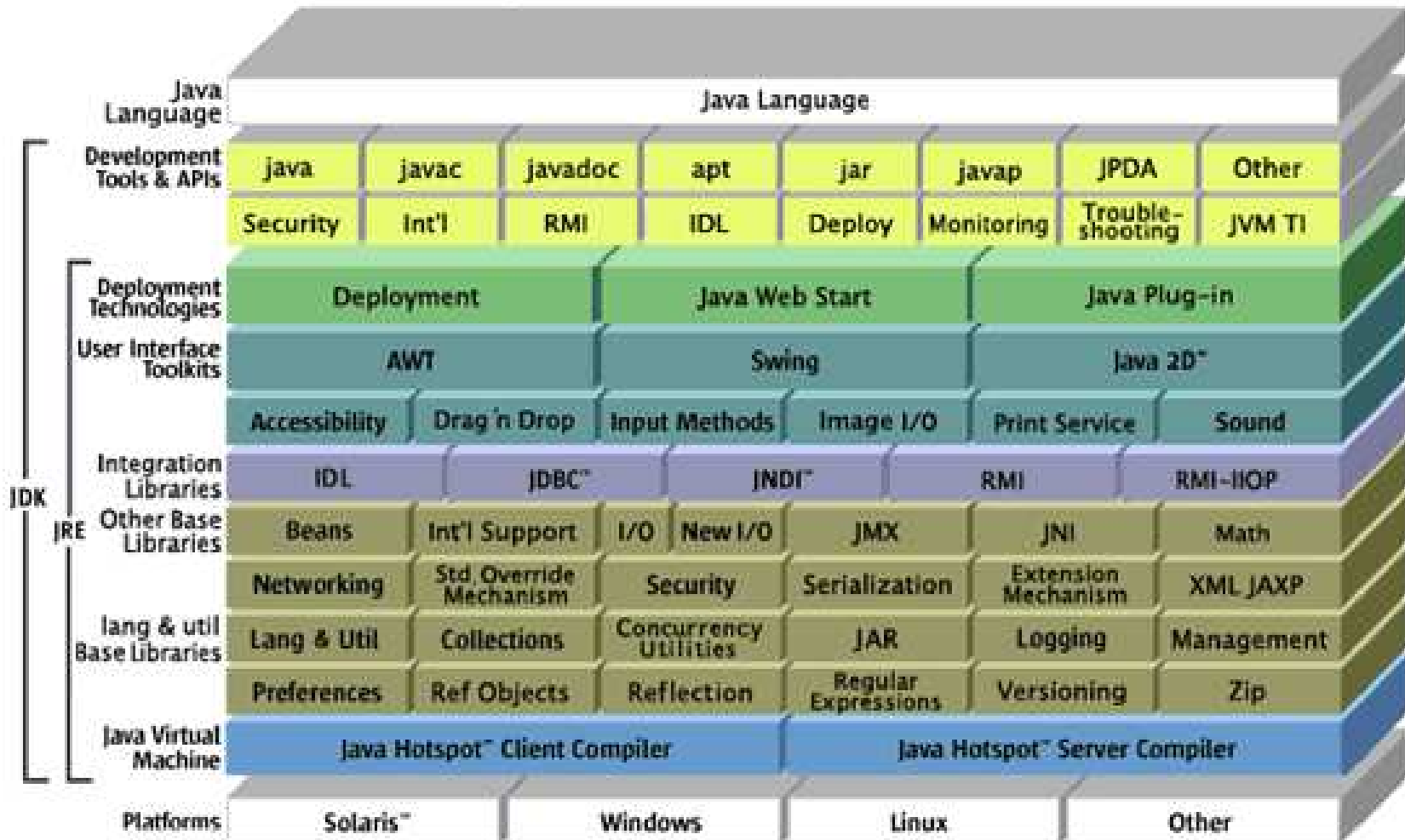
De nouvelles constatations

- On peut avoir une approche objet même avec un langage non objet !
Attention : On n'aura pas tous les bénéfices d'une approche objet... sauf à faire des choses très (trop !) compliquées.
- Dans la version C, il y a séparation entre la spécification (interface) et l'implantation (corps) du module alors qu'en Java tout est dans **une même construction syntaxique** (la classe), dans **un seul fichier**.
- La fonction `resoudre` est à l'extérieur de l'enregistrement.
- Le paramètre `eq` de `resoudre` a disparu en Java. Il est devenu implicite. Pour y faire référence, on utilise le mot-clé **this**.
- Dans la version C, on utilise **#include** `<math.h>`. En Java on aurait pu faire **import** `java.lang.Math`; mais ceci est implicite.
- Le **new** de Java correspondrait à un `malloc` en C mais en Java, la mémoire est libérée automatiquement (pas de `delete` ou `free`).

Plate-formes Java proposées par Sun

- J2SE : Java 2 Platform Standard Edition
 - JRE (Java Runtime Environment) : Java API, JVM... pour exécuter une application/applet Java
 - JDK (J2SE Development Kit) : JRE + outils de développement (compilateur...)
- J2EE (Java 2 Platform Enterprise Edition) : développement d'application multi-couches orientées composants (Entreprise JavaBeans), web services (servlet, JSP, XML)...
- J2ME (Java 2 Platform Micro Edition) : Java pour les téléphones mobiles, PDA et autres appareils embarqués. Optimisé pour la mémoire, la puissance de traitement et les E/S.
- Java Card : fournir un environnement sûr sur des cartes avec de faibles mémoires et capacités de traitement.

Java™ 2 Platform Standard Edition 5.0



Premier programme Java

```
1  /** Un programme minimal qui, de manière classique, affiche « bonjour ».
2     * En Java, tout est défini dans une classe même dans le cas du
3     * programme le plus simple !
4     */
5  public class Bonjour {
6     // Méthode principale : c'est la méthode exécutée lorsque
7     // l'utilisateur demande d'exécuter la classe Bonjour.
8     // args : les arguments de la ligne de commande (hors java Bonjour)
9     /** Dire bonjour à tout le monde.
10     * @param args ceux à qui dire bonjour
11     */
12     public static void main(String[] args) {
13         for (int i = 0; i < args.length; i++) {
14             // Afficher sur la sortie standard
15             System.out.println("Bonjour_" + args[i]);
16         }
17         System.out.println("Bonjour_tout_le_monde_!");
18         /* Les commentaires à la C sont aussi disponibles */
19     }
20 }
```

Principaux outils du J2SE Developer Kit (JDK)

Le JDK (J2SE Developer Kit) est l'environnement de développement (minimal) pour Java proposé par Sun.

- javac : compilateur de sources Java
- java : machine virtuelle (interprète code binaire)
- javadoc : générateur de documentation HTML
- jar : constructeur d'archives
- appletviewer : interprète des Applet
- javap : désassembleur code binaire
- jdb : outil de mise au point (interfacé avec DDD)
- ... et bien d'autres !

Remarque : Il existe de nombreux IDE (Integrated Development Environments) pour Java.

Le compilateur : javac

Le **compilateur** javac produit du code intermédiaire (byte code).

```
javac Bonjour.java                (==> produit Bonjour.class)
```

Java est donc un **langage compilé**. Le compilateur vous aidera à détecter et corriger les erreurs dans vos programmes (messages relativement clairs).

Remarque : Depuis la version 1.2, javac calcule les dépendances et compile tous les fichiers nécessaires. *Il suffit donc de compiler la classe principale.*

```
licorne> javac -verbose RésolutionÉquation.java
----- extraits de l'affichage -----
[parsing started RésolutionÉquation.java]
[checking RésolutionÉquation]
[loading ./Équation.java]
[parsing started ./Équation.java]
[wrote RésolutionÉquation.class]
[checking Équation]
[loading /opt/j2sdk1.4.0/jre/lib/rt.jar(java/lang/Math.class)]
[wrote ./Équation.class]
[total 1148ms]
```

Compilation de RésolutionÉquation.java

```
licorne> javac -verbose RésolutionÉquation.java
[parsing started RésolutionÉquation.java]
[parsing completed 305ms]
[loading /opt/j2sdk1.4.0/jre/lib/rt.jar(java/lang/Object.class)]
[loading /opt/j2sdk1.4.0/jre/lib/rt.jar(java/lang/String.class)]
[checking RésolutionÉquation]
[loading ./Équation.java]
[parsing started ./Équation.java]
[parsing completed 5ms]
[loading /opt/j2sdk1.4.0/jre/lib/rt.jar(java/lang/System.class)]
[loading /opt/j2sdk1.4.0/jre/lib/rt.jar(java/io/PrintStream.class)]
[loading /opt/j2sdk1.4.0/jre/lib/rt.jar(java/io/FilterOutputStream.class)]
[loading /opt/j2sdk1.4.0/jre/lib/rt.jar(java/io/OutputStream.class)]
[loading /opt/j2sdk1.4.0/jre/lib/rt.jar(java/lang/StringBuffer.class)]
[wrote RésolutionÉquation.class]
[checking Équation]
[loading /opt/j2sdk1.4.0/jre/lib/rt.jar(java/lang/Math.class)]
[wrote ./Équation.class]
[total 1148ms]
```

La machine virtuelle Java (JVM) : java

Le code produit par javac étant du code intermédiaire, il ne peut pas être directement exécuté par la machine.

⇒ Il faut donc utiliser une machine virtuelle Java : java dans le JDK :

```
licorne> java Bonjour Xavier  
Bonjour Xavier  
Bonjour tout le monde !
```

On donne en paramètre une *classe Java* (donc sans extension !) qui doit contenir la méthode principale main, suivi des éventuels arguments de la ligne de commande.

Attention : Ne pas mettre d'extension derrière le nom de la classe.

```
licorne> java Bonjour.class  
Exception in thread "main"java.lang.NoClassDefFoundError: Bonjour/class
```

```
licorne> java Bonjour.java  
Exception in thread "main"java.lang.NoClassDefFoundError: Bonjour/java
```

Le « . » est le séparateur de paquetages (équivalent des répertoires).

Outil d'extraction de documentation : javadoc

Le JDK contient un outil, javadoc, qui engendre automatiquement la documentation des classes Java à partir de leur code source.

```
javadoc *.java          (==> produit plein de fichiers HTML)
```

Intérêt : La documentation est directement rédigée dans le source Java, avec le code. Ceci facilite sa mise à jour et donc améliore (mais ne garantit pas !) sa cohérence.

javadoc permet une présentation standardisée de la documentation.

Moyen : C'est bien entendu au programmeur de fournir les éléments de la documentation. Il utilise alors des commentaires spécifiques `/** */` placés *avant* l'élément à documenter.

Attention : Par défaut, seules les informations publiques sont documentées.

Documentation engendrée

The screenshot shows a web-based Java documentation interface. On the left, a sidebar titled 'All Classes' contains a link for 'Bonjour'. The main content area has a navigation bar with links for 'Package', 'Class', 'Tree', 'Deprecated', 'Index', and 'Help'. Below this, there are links for 'PREV CLASS', 'NEXT CLASS', 'SUMMARY: NESTED | FIELD | CONSTR | METHOD', 'FRAMES', 'NO FRAMES', and 'DETAIL: FIELD | CONSTR | METHOD'. The main heading is 'Class Bonjour', followed by the package path 'java.lang.Object' and a class hierarchy diagram showing 'Bonjour' as a subclass of 'Object'. The source code for the class is shown: 'public class Bonjour extends java.lang.Object'. A descriptive paragraph follows: 'Un programme minimal qui, de manière classique, affiche << bonjour >>. En Java, tout est défini dans une classe même dans le cas du programme le plus simple !'. Below the text are two summary sections: 'Constructor Summary' with a link to 'Bonjour()' and 'Method Summary'.

All Classes
[Bonjour](#)

Package **Class** [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#) [FRAMES](#) [NO FRAMES](#)
[SUMMARY: NESTED | FIELD | CONSTR | METHOD](#) [DETAIL: FIELD | CONSTR | METHOD](#)

Class Bonjour

java.lang.Object
└─ **Bonjour**

```
public class Bonjour  
extends java.lang.Object
```

Un programme minimal qui, de manière classique, affiche << bonjour >>. En Java, tout est défini dans une classe même dans le cas du programme le plus simple !

Constructor Summary

[Bonjour\(\)](#)

Method Summary

Les commentaires

Java propose trois types de commentaires :

- Les commentaires à la C : utiles pour mettre en commentaire plusieurs lignes de code.

```
/* Un commentaire qui peut se continuer  
sur plusieurs lignes */
```

```
/* Exactement comme en C  
* et donc ils ne peuvent pas être imbriqués  
*/
```

Attention : Ces commentaires ne peuvent pas être imbriqués.

- Les commentaires à la C++ :

```
// Ce commentaire se termine avec la fin de la ligne
```

Avantage : On ne risque pas d'oublier de les fermer !

Un bon éditeur doit permettre de les ajouter et les enlever facilement.

- Les **commentaires structurés** exploités par l'outil javadoc.

Les commentaires structurés

Objectif : Les commentaires structurés sont extraits par l'outil javadoc pour produire la documentation de la classe au format HTML.

Ces commentaires peuvent contenir :

- des étiquettes spécifiques à javadoc : elles commencent par @ (@author, @param, @return, @see, etc.) ;
- des éléments HTML.

```
/** Les commentaires structurés commencent par une double étoile (**).  
 * Ils peuvent contenir des éléments <strong>HTML</strong>.  
 * Ils sont placés devant l'entité qu'ils décrivent : une classe, un  
 * attribut, une méthode  
 */
```

Principe de l'auto-documentation : le concepteur d'un module doit s'astreindre à exprimer toute l'information sur le module dans le module lui-même.

Lien avec le gestionnaire de fichiers : CLASSPATH

Par défaut, les outils du JDK cherchent les classes dans le répertoire courant.

Si les classes sont dans plusieurs répertoires, on utilise le « classpath » :

- soit avec l'option `-classpath` des outils du JDK ;
- soit avec la variable d'environnement `CLASSPATH`.

Remarque : Nécessaire dès qu'on utilise des bibliothèques (JUnit, Log4J...) qui sont dans des répertoires ou fichiers d'archive (.jar) propres.

```
javac -classpath /usr/local/java/junit/junit.jar:. TestMaClasse.java
java -classpath /usr/local/java/junit/junit.jar:. TestMaClasse
```

Les classes sont cherchée dans `junit.jar` puis dans le répertoire courant.

Attention : Ne pas oublier le répertoire courant !

En utilisation `CLASSPATH` avec (t)csh :

```
setenv CLASSPATH /usr/local/java/junit/junit.jar:.
javac TestMaClasse.java
java TestMaClasse
```

Java est un langage simple et familier

- familier par sa parenté avec C :
 - structures de contrôle ;
 - types primitifs.
- simple par rapport à C/C++ :
 - pas de fichier d'entête, pas de préprocesseur ;
 - pas d'arithmétique des pointeurs (ni de syntaxe) ;
 - ni structures, ni unions, ni énumérations (ajoutées dans Java 1.5) ;
 - pas de surcharge des opérateurs ;
 - pas de conversions de types sans contrôles ;
 - pas de sémantique de la valeur pour les objets ;
 - gestion automatique de la mémoire.

Attention : Java est un langage à objets. Il nécessite donc un changement de point de vue et contient des aspects subtils (héritage, liaison tardive...).

Java est un langage robuste

But : Éliminer les risques d'erreur (contexte logiciel embarqué ou mobile)

Moyen :

- Mécanisme sophistiqué de gestion des erreurs
 - Mécanisme de typage « fort » (détection statique des erreurs)
 - L'éditeur de lien utilise les informations de typage
 - Conversions de type contrôlées
 - Détection dynamique des dépassements des bornes de tableaux
- Mécanisme sophistiqué de gestion mémoire
 - Contrôle de l'accès à la mémoire (pas de risque d'écrasement)
 - Libération « automatique » de la mémoire (ramasse-miettes)

Java : Une architecture neutre

But : Exécuter du code mobile dans un environnement hétérogène ou, plus simplement, exécuter le programme sur des machines différentes.

Principe : Éviter les dépendances vis-à-vis :

- du matériel
- des couches logicielles : réseau, système d'exploitation, environnement graphique

Moyen :

- Utilisation d'une machine virtuelle (processeur abstrait)
- Définition d'une bibliothèque standard abstraite instanciée pour chaque environnement (et tendre vers du pur Java)

Java : Un langage portable

But :

- Un même code compilé sur toutes les architectures produit le même résultat
- Minimiser les modifications liées au portage machine virtuelle

Moyen :

- Bibliothèque indépendante
- Définition (sémantique) précise du langage :
 - taille, organisation physique données
 - valeurs par défaut, minimales, maximales
 - effets des opérateurs sur les données
 - ordre des calculs
 - effets des instructions sur la mémoire

Java est un langage à objets

Bénéficiaire de la meilleure technologie logicielle disponible à ce jour

Concepts-clés : Deux concepts majeurs :

- Modularité : encapsulation et masquage d'information (notion de classe)
- Extensibilité (relation d'héritage)

Technique :

- Classes, interfaces, paquetages
- Tout est classe (enveloppe – *wrapper* – possible pour types primitifs)
- Les objets sont accessibles par des poignées (pointeurs)
- Héritage simple des classes
- Héritage multiple d'interfaces
- Polymorphisme et liaison tardive
- Bibliothèque très riche

Java : Un langage performant

- Byte code adapté pour être compilé à la volée (*Just In Time*) pour produire puis réutiliser le code associé à chaque instruction
- Cache mémoire pour éviter le chargement (et la vérification) multiple d'une même classe
- Compilation classique pour engendrer un programme propre à une architecture donnée avec édition de lien classique (perte mobilité)
- Ramasse-miettes : Processus indépendant de faible priorité

La performance d'un langage ne se mesure pas qu'à sa vitesse d'exécution mais aussi au temps de développement requis. Les atouts de Java sont :

- Simplicité du langage (à nuancer !)
- Vérification statique et dynamique forte
- Bibliothèque standard très complète

Autres points forts de Java

Java possède d'autres points forts qui ne sont pas développés dans ce cours. Ainsi Java est aussi un langage...

- **distribué** : exploiter simplement les ressources INTERNET (code mobile avec les applets, servlets, RMI, Corba) ;
- **sécurisé** : éviter les intrusions liées au code mobile (« bac à sable », API de sécurité) ;
- **dynamique (interprété)** : accélérer le cycle de développement (édition des liens dynamique), introspection ;
- **parallèle** : Processus légers (exécution parallèle dans le même espace d'adressage) ;

Java : un langage mais pas seulement !

En fait, Java c'est trois éléments :

1. la **machine virtuelle Java** (JVM) dont la spécification est stable depuis 1995 ;
2. le **langage Java** par lui-même. Il a connu des évolutions jusqu'à la version 1.2, un peu à la 1.4 (assert) et 1.5 (généricité, foreach, énumération, ellipse...);
3. la **bibliothèque standard Java**. C'est elle qui évolue le plus et provoque les changements de version actuels.

Notons par exemple le modèle de gestion des événements, swing...

Remarquons que pour améliorer sa portabilité, la bibliothèque est principalement développée en Java.

Origine et historique

- Projet SunSoft : Langage pour **applications embarquées**
 - 1991 : Projet Green, Produit *7, Langage Oak (futur Java) \implies Échec !
- Évolution vers INTERNET : **Applications mobiles**
 - 1994-1995 : navigateur WWW novateur, futur *HotJava*
 - 1995 : intégration de Java dans Netscape
- Les principales versions de Java :
 - 1996 : première version de Java publiée (Java 1.02)
 - 1997 : Java 1.1 (modèle des événements, internationalisation, *Java Beans...*)
 - 1998 : Java 1.2 (Swing optimisation de la JVM, ...).
Sun appelle *Java 2* les versions de Java.
 - 2001 : Java 1.4 (assertions, amélioration JDBC, regexp...)
 - 2004 : Java 1.5 (généricité, énumération, foreach) : J2SE 5.0

Algorithmique en Java

La méthode principale

Par définition, lorsqu'on lance la JVM avec une classe, c'est sa « méthode principale » qui est exécutée. Elle se déclare de la manière suivante :

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

Attention : Tous les mots (sauf args) sont importants !

- Les qualificatifs **static** (T. 97) et **public** (T. 68) sont obligatoires !
- Elle a pour type de retour **void** (pas de valeur retournée).
- Elle prend un seul paramètre de type tableau (T. 109) de chaînes de caractères (T. 117), les arguments de la ligne de commande.
- Chaque classe peut définir **sa** méthode principale.
- Une classe peut ne pas avoir de méthode principale.

Les types primitifs

Les types primitifs comprennent :

- les entiers : **byte**, **short**, **int**, **long**.

Ce sont des entiers signés (il n'y a pas d'entiers non signés).

- les nombres à virgule flottante : **float**, **double**.

- les booléens : **boolean** (avec les deux valeurs **false** et **true**).

- les caractères : **char** (en Unicode)

Quelques propriétés des types primitifs :

- Ils ont tous une valeur par défaut (en général 0) utilisée pour les initialisations automatiques.

- Les entiers ne sont pas compatibles avec les booléens.

- Entiers et caractères sont compatibles (attention à $3 + 'a'$).

Les types primitifs : tableau récapitulatif

Type	# bits	Défaut	Minimum	Maximum	Exemples
byte	8	0	-128	+127	-10, 0, 10
short	16	0	-32768	32767	-10, 0, 10
int	32	0	-2^{31}	$2^{31} - 1$	-10, 0, 10
long	64	0	-2^{63}	$2^{63} - 1$	-10L, 0L, 10L
float	32	0.0	IEEE 754	IEEE 754	3.1F, 3.1f, 31e-1f
double	64	0.0	IEEE 754	IEEE 754	3.1, 3D, 1e-4
boolean	1	false	false	true	
char	16	'\u0000'	'\u0000'	'\uFFFF'	'a', '\n', '\'

Remarque : Java autorise les conversions entre types à condition qu'il n'y ait pas de perte d'information (ou précision).

```
float x1 = 4;           // OK : coercion
float x2 = 4.21;       // Erreur de compilation !
```

Identificateurs

- Les identificateurs en Java sont de la forme : lettre (lettre | chiffre) *
Les lettres sont a-z, A-Z, _, ainsi que tout caractère unicode qui correspond à une lettre dans la langue utilisée (caractères accentués).
- Java comme C et C++ distingue majuscules et minuscules
- **Conseil** : Toujours choisir un identificateur significatif.
- Mettre en évidence les différents mots d'un identificateur. La convention Java est de mettre l'initiale des mots suivants en majuscule (exemple : idEnPlusieursParties).
- **Remarque** : Le caractère \$ fait partie des « chiffres » mais est réservé pour des identificateurs engendrés automatiquement. **Ne pas l'utiliser !**

Opérateurs relationnels

<code>==</code>	<i>// égalité (conversion si types différents)</i>	<code>3 == 3</code>
<code>!=</code>	<i>// différence (non égalité)</i>	<code>4 != 5</code>
<code>></code>	<i>// plus grand que</i>	<code>4 > 3</code>
<code>>=</code>	<i>// plus grand que ou égal</i>	<code>4 >= 3</code>
<code><</code>	<i>// plus petit que</i>	<code>3 < 4</code>
<code><=</code>	<i>// plus petit que ou égal</i>	<code>3 <= 4</code>

Attention : Ne pas confondre = (affectation) et == (égalité) !

Opérateurs arithmétiques

<code>+ -</code>	<i>// addition et soustraction (op. binaires)</i>	<code>10 + 5 == 15</code>
<code>* /</code>	<i>// multiplication et division</i>	<code>10 / 3 == 3</code>
<code>%</code>	<i>// modulo : le reste de la division entière</i>	<code>10 % 3 == 1</code>
<code>+</code>	<i>// opérateur unaire</i>	<code>+10 == 10</code>
<code>-</code>	<i>// opérateur unaire (opposé de l'opérande)</i>	<code>-10</code>
<code>++</code>	<i>// pré-incrémentation (++x) ou post-incrémentation (x++)</i>	
<code>--</code>	<i>// pré-décrémentaion (--x) ou post-décrémentaion (x--)</i>	

Opérateurs logiques (booléens)

```
&&    // ET logique      expr1 && expr2  
||    // OU logique     expr1 || expr2  
!     // NON logique    ! expr1
```

Remarque : Les opérateurs logiques sont évalués en court-circuit (évaluation partielle) : dès que le résultat est connu, l'évaluation s'arrête.

```
true || expr    // vrai sans avoir à évaluer expr  
false && expr   // faux sans avoir à évaluer expr
```

Formulations équivalentes : (la seconde est préférable)

```
A == true    est équivalent à  A  
A == false   est équivalent à  !A
```

Opérateurs sur les bits

```
expr1 & expr2 // ET bit à bit
expr1 | expr2 // OU bit à bit
expr1 ^ expr2 // XOR bit à bit
~ expr1       // inverse les bits de l'opérande
expr << nb    // décalage de expr à gauche de n bits (fois 2)
expr >> nb    // décalage de expr à droite de n bits (div 2)
expr >>> nb   // idem >> (sans préservation du signe)
```

Opérateur conditionnel (si arithmétique)

```
condition ? valeur_vrai : valeur_faux
```

Si la condition est vraie, le résultat est valeur_vrai, sinon c'est valeur_faux.

```
status = (age >= 18) ? "majeur" : "mineur"
```

Attention : Peut être difficile à lire !

Priorité et associativité des opérateurs

Priorité	Opérateurs
1D	++, -, +, -, ~, !, (...)
2G	*, /, %
3G	+, -
4G	<<, >>, >>>
5G	<, <=, >, >=, instanceof
6G	==, !=
7G	&
8G	^
9G	
10G	&&
11G	
12G	? :
13D	=, *=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, =

`a + b + c + d` // G : associativité à gauche $((a + b) + c) + d$
`x = y = z = t` // D : associativité à droite $x = (y = (z = t))$

Instructions et structures de contrôle

Instructions « simples »

- Déclaration de variables
- Affectation
- Instruction nulle
- Instruction composée

Structures de contrôle

- Conditionnelles : **if** et **switch**
- Répétitions : **while**, **do**, **for** et **foreach**

Instructions à ne pas utiliser

Instructions simples : affectation

Affectation : Donner une nouvelle *valeur* à *une variable*.

```
int i, j, k;  
i = 2;           // affectation avec une expression constante  
j = i * (i - 1); // la valeur est une expression quelconque  
j = k = i+j;    // l'affectation renvoie une valeur !
```

Variantes de l'affectation : L'opérateur d'affectation peut être combiné avec la plupart des opérateurs :

```
x += y    //    x = x + y  
x -= y    //    x = x - y  
x %= y    //    x = x % y  
x |= y    //    x = x | y  
...
```

Pré- et post-opérateurs : ++ et --

Incrémenter (++) ou décrémenter (--) une variable de 1.

Instructions simples (suite)

Instruction nulle :

```
; // instruction nulle : ne fait rien !
```

Bloc ou instruction composée : grouper des instructions avec des accolades pour qu'elles soient considérées comme une seule.

```
{ // instruction composée
  nbAnnées++;
  capital = capital * (1 + taux);
}
```

Instructions simples : déclaration de variables

La déclaration de variables est une instruction.

⇒ Les déclarations n'ont plus à être groupées au début d'un bloc.

Déclaration d'une variable :

```
<type> <nom> [= <valeur_initiale>]; // rôle de la variable
```

Exemples :

```
int age, numéro, montant; // éviter les déclarations multiples !  
int px = 0, py = 0; // avec initialisation  
double x = 55, y = x*x; // utilisation d'une expression calculée
```

Conseil : Ne déclarer une variable que quand on est capable de l'initialiser.

Portée : Une variable (locale) est accessible de sa déclaration jusqu'à la fin du bloc dans lequel elle est définie.

Attention : Impossible d'utiliser un même identificateur dans un sous-bloc.

Remarque : Le mot-clé `final` permet de définir une (variable) constante !

```
final int MAX = 10;
```

Conditionnelles : if

```
if (<condition>)  
    <instruction>;
```

```
if (<condition>) {  
    <instructions1>;  
} else {  
    <instructions2>;  
}
```

Remarque : si on veut mettre plusieurs instructions, il faut utiliser les { } pour en faire une instruction composée.

Règle : Toujours mettre les { }.

Attention : En Java, toutes les conditions sont (et doivent être) booléennes !

```
if (n > max) {  
    max = n;  
}
```

```
if (n1 == n2) {  
    res = "égaux";  
} else {  
    res = "différents";  
}
```

Remarque : Pour représenter un **SinonSi**, utiliser **else if**.

Conditionnelles : switch (choix multiples)

```
switch (<expression>) {  
  case <expr_cste1>:  
    <instructions1>;  
    break;  
  ...  
  case <expr_csten>:  
    <instructionsn>;  
    break;  
  default:  
    <instruction>;  
    break  
}
```

```
switch (c) { // c caractère  
  case 'o':  
  case 'O':  
    res = "Affirmatif";  
    break;  
  case 'n':  
  case 'N':  
    res = "Négatif";  
    break;  
  default:  
    res = "!!?!?!?!";  
    break;  
}
```

Principe : L'expression est évaluée et l'exécution continue à la première instruction qui suit la 1^{re} expression constante lui correspondant (ou **default**). Si un **break** est rencontré, l'exécution se poursuit à la fin du **switch**.

Conséquence : Si le même traitement doit être fait pour plusieurs cas, il suffit de lister les différents **case** correspondants consécutivement.

Conseil : Mettre un **break** après chaque groupe d'instructions d'un **case**.

Répétitions : while

```
while (<condition>) {  
    <instructions>;  
}
```

```
// nb d'années pour atteindre l'objectif  
double taux = 0.03;  
double capital = 5000;  
double objectif = 10000;  
int nbAnnées = 0;  
while (capital < objectif) {  
    nbAnnées++;  
    capital = capital * (1 + taux);  
}
```

Sémantique : Tant que <condition> est vraie, <instructions> (simple ou bloc) est exécutée.

Remarque : <instructions> peut ne pas être exécutée.

Répétitions : do ... while

```
do {  
    <instructions>;  
} while (<condition>);
```

```
// Calculer la racine carrée de a  
final double EPSILON = 1.e-4;  
double a = 2;  
double un = a;  
double up; // valeur précédente de  $u_n$   
do {  
    up = un;  
    un = (un + a/un)/2;  
} while (Math.abs(un - up) > EPSILON);
```

Sémantique : <instructions> est exécutée puis, tant que <condition> est vraie, <instructions> est exécutée.

Remarque : <instructions> est exécutée au moins une fois.

Répétitions : for

```
for (<init>; <cond>; <incr>) {  
    <instructions>;  
}
```

```
for (int i = 1; i < 10; i++) {  
    System.out.println(i);  
}
```

Sémantique : <init> (initialisation) est exécutée puis, tant que <cond> est vraie <instructions> et <incr> (incrémentations) sont exécutées.

```
{ // réécriture du for à l'aide du while  
    <init>; // initialisation  
    while (<cond>) { // condition de continuation  
        <instructions>; // traitement  
        <incr>; // incrémentations  
    }  
}
```

Conséquence : Une variable déclarée dans <init> n'est visible que dans le bloc du **for** (cas du **int** `i = 1`, par exemple).

Conseil : Conserver la sémantique du **Pour** algorithmique : on sait à l'avance combien de fois la boucle doit être exécutée.

Répétitions : foreach

```
for (<type> <var> : <col>) {  
    <instructions>;  
}  
|  
public static void main(String[] args) {  
    for (String nom : args) {  
        System.out.println("Bonjour_" + nom);  
    }  
}
```

Vocabulaire : On dit « Pour chaque <var> dans <col> » (foreach ... in ...).

Sémantique : <col> est soit un *tableau*, soit une *collection* (en fait un itérable, T. 295).

Les instructions sont exécutées pour <var> prenant chaque valeur de <col>.

Avantage : Écriture simple conservant la sémantique du **Pour** algorithmique.

Limites : <instructions> ne doit pas modifier le parcours de la collection (détecté à l'exécution `ConcurrentModificationException`). Règle du 80–20 !

Instructions à ne pas utiliser

Les instructions suivantes sont proposées par Java :

- **break** : arrêter l'exécution d'une boucle ou d'un **switch** ;
- **continue** : arrêter l'itération actuelle d'une boucle et passer à la suivante.

Attention : Ces deux instructions **ne doivent pas être utilisées** car elles violent les principes de la programmation structurée.

Exception : Bien sûr, le **break** peut et doit être utilisé dans un **switch** !

Remarque : En Java, on peut étiqueter les répétitions.

```
première: for (int i = 0; i < 7; i++) { // for étiqueté « première »
    System.out.println("i_=_ " + i);
    for (int j = 0; j < 4; j++) {
        System.out.println("_____j_=_ " + j);
        if (j == i) {
            continue première;           // ==> passer au i suivant !
        }
    }
}
System.out.println("Fin_!");
```

Paquetages

Programme = ensemble de classes (et interfaces) organisées en paquetages.

```
package fr.enseeiht.cregut;    // paquetage d'appartenance  
class A { ... }              // texte Java de la classe
```

Une classe ne peut appartenir qu'à un seul paquetage.

Remarque : La structure des paquetages s'appuie sur le système de fichiers : les paquetages sont des répertoires et les classes les fichiers.

Remarque : Si le paquetage d'appartenance n'est pas précisé, la classe appartient au *paquetage anonyme* (le répertoire courant).

Convention : Le nom des paquetages est en minuscules : java.lang, java.util, fr.enseeiht.cregut (qui correspondent respectivement à java/lang/, java/util/ et fr/enseeiht/cregut).

Paquetages : utiliser une classe d'un paquetage

- qualifier complètement la classe :

```
java.awt.Color c;      // La classe Color du paquetage java.awt
```

- importer une classe (on a accès à la classe sans la qualifier) :

```
import java.awt.Color; // en début de fichier
```

```
...
```

```
Color c;              // La classe Color du paquetage java.awt
```

- importer le contenu d'un paquetage (on a accès à toutes ses classes) :

```
import java.awt.*;    // en début de fichier
```

```
...
```

```
Color c;              // La classe java.awt.Color
```

```
Point p;              // La classe java.awt.Point
```

Attention aux conflits si un même nom de classe est utilisé dans deux paquetages ! Le conflit doit être résolu en utilisant le nom qualifié.

- `java.lang.*` est importé par défaut : il contient les classes de base, les classes `System`, `Math`, `String`, etc.

Paquetage : intérêts

Intérêt : L'intérêt des paquetages est de :

- structurer l'application en regroupant ses constituants ;
- éviter les conflits de noms : un même nom de classe peut être utilisé dans deux paquetages différents (un paquetage définit un espace de nommage).

Conseil : Pour éviter toute ambiguïté, il est recommandé de toujours utiliser la forme complètement qualifiée des classes !

Exemple : `java.awt.Color` plutôt que `Color`.

Remarque : Nous parlons maintenant des paquetages car ils sont nécessaires pour comprendre certains éléments de Java (droit d'accès) ou certains messages d'erreurs avec les outils du JDK.

Attention : Les paquetages sont très importants pour la structuration d'un programme, ou d'une bibliothèque. Cependant, nous n'insisterons pas sur cet aspect dans la suite de ce cours.

Classes, objets et envois de message

- Les objets
- Les poignées
- Les classes
- Les attributs
- Les méthodes
- Les constructeurs
- Les attributs et méthodes de classe

Les objets

Un objet est caractérisé par :

- un *état* : la valeur des attributs (coeffA, coeffB, etc.) ;
- un *comportement* : les méthodes qui peuvent lui être appliquées (résoudre) ;
- une *identité* qui identifie de manière unique un objet (par exemple son adresse en mémoire).

Remarque : Un objet n'a de réalité qu'à l'exécution du programme.

Exercice 5 Quel est l'état d'une fraction ? Quel est son comportement ?

Attention : Les objets ne sont accessibles que par l'intermédiaire de poignées : Un objet est *attaché* à une poignée.

Les poignées

Les objets sont alloués dynamiquement (dans le tas) : opérateur **new**.

```
new Équation();           // création d'un objet Équation
```

Il retourne l'*identité* de l'objet créé. Elle est conservée dans une *poignée*.

Poignée : Une poignée est une variable dont le type est le nom d'une classe.

```
Équation eq;              // déclarer une poignée eq de type Équation  
eq = new Équation();      // créer un objet et l'attacher à la poignée
```

La **valeur par défaut** d'une poignée est **null**. Elle indique qu'aucun objet n'est *attaché* à la poignée.

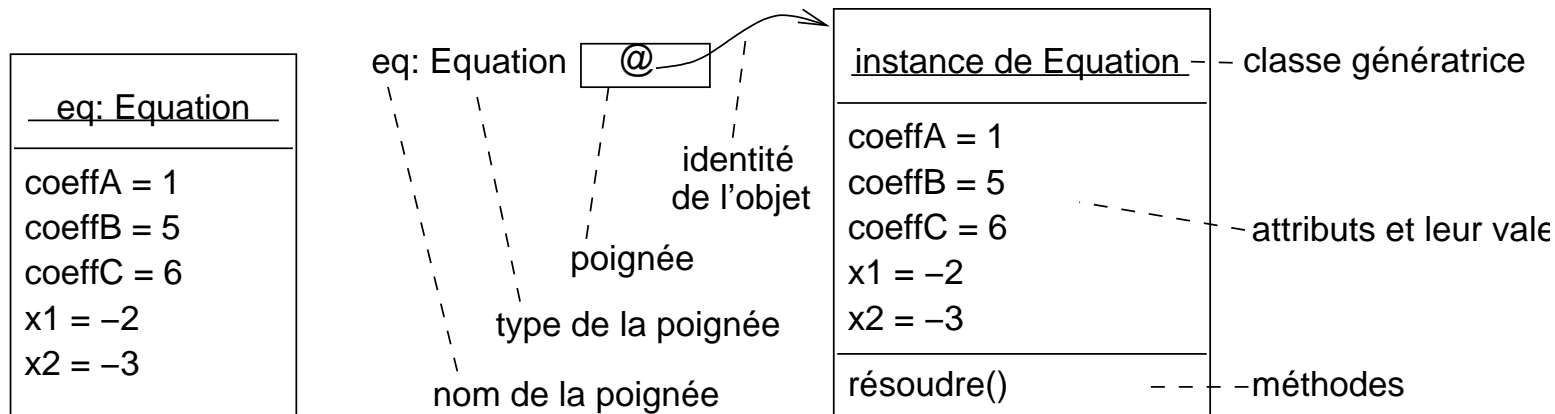
Remarque : On peut regrouper déclaration de la poignée et initialisation :

```
Équation eq = new Équation(); // en une ligne pour ne pas oublier  
// d'initialiser la poignée
```

Conseil : Préférer la version où déclaration de la poignée et création de l'objet sont faites en même temps.

Notation graphique d'un objet

```
Équation eq = new Équation();
... // initialisation, résolution, etc.
```



en UML

représentation abstraite (en mémoire)

Remarque : En UML, on peut omettre le nom de l'objet ou de la classe. C'est donc le souligné qui indique qu'il s'agit d'un objet !

Remarque : En UML, les méthodes ne sont pas données car tous les objets d'une même classe possèdent les mêmes méthodes. C'est implicite.

Et après ?

Nous avons vu ce qu'est un objet, comment le créer et l'attacher à une poignée.

Mais :

- Où sont définies les caractéristiques des objets ? Dans une **classe** !
- Comment peut-on accéder aux caractéristiques des objets ? Par l'**envoi de messages** (ou **appel de méthodes**) !
- Comment initialiser un objet (nous ne savons que le créer) ? C'est l'objectif des **constructeurs** !

Les classes

Une classe définit :

- un **MODULE** : elle regroupe la déclaration des attributs et la définition des méthodes associées dans une même construction syntaxique ;

```
class NomDeLaClasse {  
    // Définition de ses caractéristiques  
}
```

Les *attributs* permettent le stockage d'information (état de l'objet).

Les *méthodes* sont des unités de calcul (fonctions ou procédures).

La classe est donc l'unité d'**encapsulation** et un **espace de nommage**.

- un **TYPE** qui permet de :
 - créer des objets ;
 - déclarer des poignées auxquelles seront attachés ces objets.

Pourront être appliqués à ces objets (par l'intermédiaire des poignées), les méthodes et attributs définis sur la classe comme **MODULE**.

Liens entre objet, poignées, classe et type

Voici quelques affirmations :

- Un objet est une instance d'une classe.
- Un objet est instance d'une et une seule classe.
- Une poignée a un type qui est le nom d'une classe.
- Un objet a pour type le nom de sa classe mais peut avoir d'autres types (voir héritage T. 171 et interfaces T. 134).
- On peut initialiser une poignée avec toute expression dont le type est le même que celui de la poignée (ou un sous-type, voir héritage T. 171 et interfaces T. 134).

Droit d'accès (accessibilité) des caractéristiques

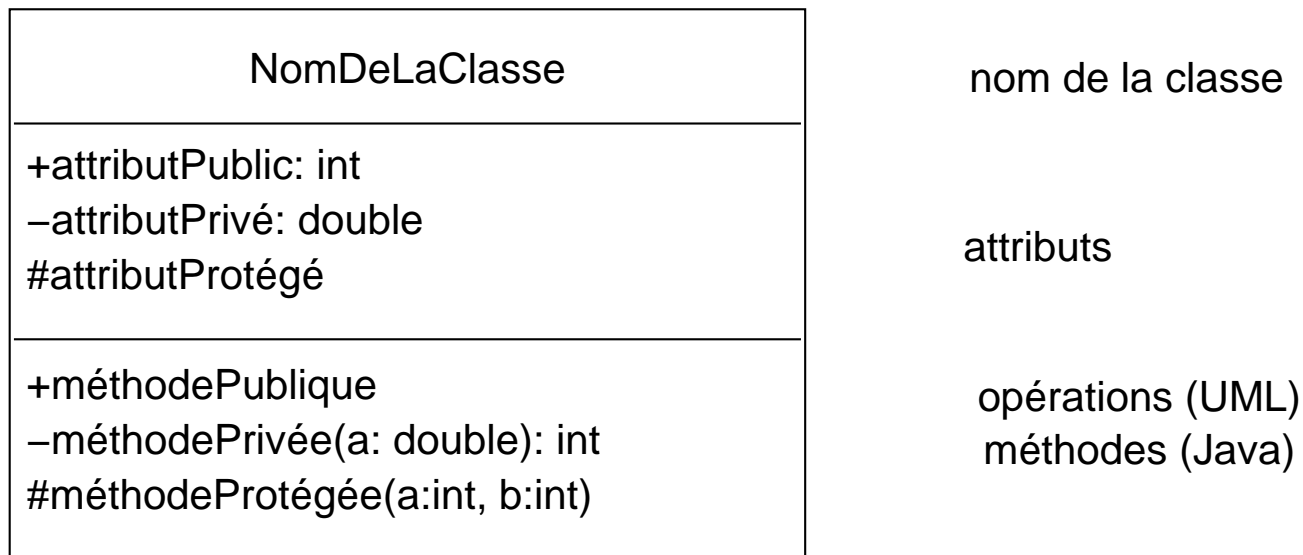
Chaque caractéristique a un droit d'accès. Il existe quatre niveaux de droit d'accès en Java :

- **public** : accessible depuis toutes les classes ;
- **private** : accessible seulement de la classe et d'aucune autre ;
- absence de modificateur de droit d'accès : *droit d'accès de paquetage*, accessible depuis toutes les classes du même paquetage ;
- **protected** : accessible du paquetage **et** des classes dérivées (cf héritage).

Accessible depuis une méthode définie dans	Droit d'accès/Visibilité			
	public	protected	défaut	private
La même classe	oui	oui	oui	oui
Une classe du même paquetage	oui	oui	oui	non
Une sous-classe d'un autre paquetage	oui	oui	non	non
Une autre classe d'un autre paquetage	oui	non	non	non

Intérêt : Les droits d'accès permettent le **masquage d'information**.

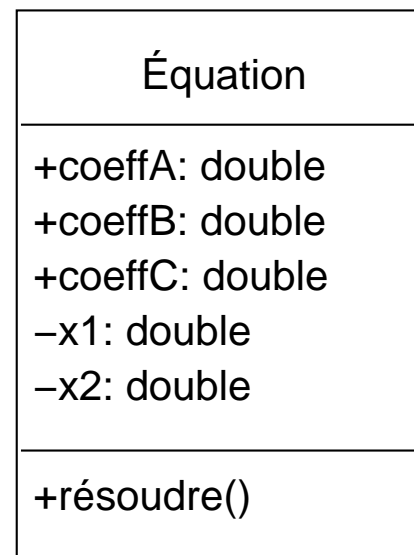
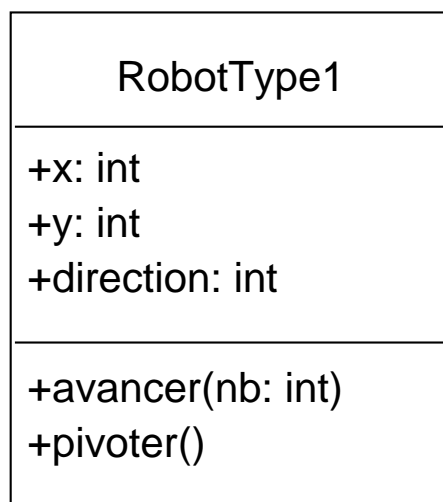
Représentation UML d'une classe



La première partie contient le nom de la classe, la seconde les attributs et la troisième les méthodes (qui sont appelées opérations en UML).

Les droits d'accès sont symbolisés par les signes + - # correspondant respectivement à **public**, **private** et **protected**.

Exemple de classes UML



Exercice 6 Comment faire dans la « classe principale » pour afficher les racines de l'équation ?

Exercice 7 Proposer la description UML d'une classe Fraction qui représente les fractions rationnelles.

Attributs

Les attributs permettent de stocker les informations spécifiques d'un objet.

Ils se déclarent nécessairement à l'intérieur d'une classe :

```
/** Documentation javadoc de l'attribut */  
<modifieurs> Type idAttribut [, idAttribut]* ;  
  
/** coefficient de  $x^2$  */  
public double coeffA;
```

Les modifieurs incluent la définition des droits d'accès.

Accès à un attribut : *poignée.attribut*

```
Équation uneÉquation = new Équation();  
double sol1 = uneÉquation.x1;     // accès en lecture  
uneÉquation.coeffA = 1;           // accès en modification
```

Attention : Erreur de compilation si droit d'accès insuffisant !

```
Équation eq = null;           // pas d'objet attaché à eq  
double sol2 = eq.x2;         // --> NullPointerException
```

Attention : Si la poignée est **null** une erreur est signalée à l'exécution.

Attributs : règle sur le droit d'accès

Règles : Un attribut devrait toujours être défini **private** pour :

- respecter le principe d'accès uniforme et
- permettre à l'auteur de la classe de garantir l'intégrité des objets de la classe.

Exemple : L'attribut `x1` de `Équation`.

Exercice 8 Comment accéder à la valeur de `x1` si `x1` est **private** ?

Exercice 9 Définir une date avec jour, mois, année et quantième.

Méthodes

Définition : Une méthode est une unité de calcul (fonction au sens de C) qui exploite l'état d'un objet (en accès et/ou en modification).

- Une méthode est identifiée par sa classe, son nom, le nombre et le type de ses paramètres (surcharge T. 78).
- Elle possède également un type de retour qui est `void` si elle ne retourne rien (procédure).
- Elle a un code entre accolades.

Syntaxe

```
/** Documentation javadoc de la méthode décrivant l'objectif de la méthode.  
 * Les paramètres et le retour de la méthode sont également documentés :  
 * @param nomParamètre description de nomParamètre  
 * @return description de l'information retournée (si non void)  
 */  
<modifieurs> TypeRetour idMéthode( [Type1 p1[, Type p]*]) {  
    ...  
}
```

Méthodes : exemple

Définir les méthodes set et delta sur la classe Equation.

```
/** Initialiser une Équation à partir de la valeur de ses coefficients.  
* @param a coefficient de  $x^2$   
* @param b coefficient de  $x$   
* @param c coefficient  
*/  
public void set(double a, double b, double c) {  
    this.coeffA = a;  
    this.coeffB = b;  
    this.coeffC = c;  
}  
  
/** Obtenir le discriminant de l'équation.  
* @return le discriminant de l'équation  
*/  
private double delta() {  
    return this.coeffB * this.coeffB - 4 * this.coeffA * this.coeffC;  
}
```

Remarque : delta() est **private** car elle n'a pas à être utilisée de l'extérieur.

Méthodes : utilisation

Une méthode est toujours appliquée sur une poignée (comme un attribut) et est exécutée sur l'objet associé à cette poignée.

```
poignée.méthode(p1, ..., pn); // Forme générale (envoi de message)

Équation eq = new Équation(); // Créer un objet Équation attaché à eq
eq.set(1, 5, 6); // Initialiser l'équation (méthode void)
double delta = eq.delta(); // Utiliser une méthode non void
eq.résoudre();
eq.delta(); // Valide mais quel intérêt ?

Équation eq2 = null; // pas d'objet attaché à eq
eq2.set(1, 4, 4); // ==> NullPointerException
```

Remarque : Appliquer une méthode sur une poignée `null` provoque une exception (`NullPointerException`).

Liaison statique : Le compilateur accepte l'appel `p.m(a1, ..., an)` ssi il existe, dans la classe définissant le type de la poignée `p`, une méthode `m` d'arité `n` telle que les types de `a1, ..., an` sont compatibles avec sa signature.

Méthodes : le paramètre implicite `this`

Une méthode est appliquée à un objet, appelé *récepteur*, (généralement à travers une poignée) et manipule l'état de cet objet.

Le récepteur est un paramètre implicite car il n'apparaît pas dans la signature de la méthode. À l'intérieur du code de la méthode, on peut y faire référence en utilisant le mot-clé `this`.

```
public void set(double a,  
                double b, double c) {  
    this.coeffA = a;  
    this.coeffB = b;  
    this.coeffC = c;  
}
```

```
public void set(double a,  
                double b, double c) {  
    coeffA = a; // this implicite !  
    coeffB = b;  
    coeffC = c;  
}
```

Remarque : `this` peut être nécessaire si un paramètre a le même nom qu'un attribut. Mais il est **plus sûr** de changer le nom du paramètre !

Exercice 10 Écrire une méthode supérieur (>) si inférieur (<) est définie.

Définir une classe

Exercice 11 : Compteur

Un compteur a une valeur (entière) qui peut être incrémentée d'une unité. Elle peut également être remise à 0. On suppose également qu'il est possible d'initialiser le compteur à partir d'une valeur entière positive.

- 11.1 Modéliser en utilisant la notation UML la classe Compteur.
- 11.2 Écrire un programme de test de la classe Compteur.
- 11.3 Écrire en Java la classe Compteur.
- 11.4 Comment être sûr que la valeur du compteur est toujours positive ?

Surcharge

Défi nition : En Java, indiquer le nom d'une méthode n'est pas suffisant pour l'identifier. Il faut préciser :

- la classe à laquelle elle appartient ;
- son nom ;
- son nombre de paramètres ;
- le type de chacun de ses paramètres.

Exemple : Les méthodes suivantes sont toutes différentes :

```
class A {  
    void afficher()           // afficher sans paramètre  
    void afficher(int i)     // afficher un entier  
    void afficher(long i)    // afficher un entier long  
    void afficher(String str) // afficher une chaîne  
    void afficher(String s, int largeur); // sur une certaine largeur  
    void afficher(String s, int largeur, char mode); // mode == 'c', 'g', 'd'  
    void afficher(int nbFois, String str); // contre-exemple !  
}
```

Surcharge : résolution

Le même nom peut être utilisé pour nommer des méthodes différentes.

⇒ Pour résoudre un appel de méthode, le compilateur s'appuie également sur le nombre et le type des paramètres effectifs :

```
// On suppose que l'on est dans le corps d'une méthode de la classe A  
afficher(10);           // afficher(int)  
afficher(10L);         // afficher(long)  
afficher("Bonjour", 20, 'c'); // afficher(String, int, char)  
afficher("Bonjour");   // afficher(String)  
afficher();            // afficher()  
afficher(true);        // Erreur à la compilation  
afficher(20, "Bonjour"); // afficher(int, String)
```

Intérêt : Éviter de multiplier les noms (afficherInt, afficherLong, etc.).

Conseil : Respecter Le sens sous-entendu par le nom de la méthode.

Exercice 12 Expliquer comment la surcharge permet de simuler des valeurs par défaut aux paramètres de méthodes (à la C++).

Exercice 13 Quelles méthodes de Fraction pourraient être surchargées ?

Exemples de surcharge

Exercice 14 On considère la classe suivante. Indiquer pour chaque appel dans la méthode main qu'elle est la méthode réellement appelée.

```
class TestSurcharge {
    static void m1(double d, double d2)    { System.out.println("m1(d,d)"); }
    static void m1(double d, int i)        { System.out.println("m1(d,i)"); }
    static void m1(int i, double d)        { System.out.println("m1(i,d)"); }
    static void m1(int i, int i2)          { System.out.println("m1(i,i)"); }

    static void m2(double d, int i)        { System.out.println("m1(d,i)"); }
    static void m2(int i, double d)        { System.out.println("m1(i,d)"); }

    public static void main(String[] args) {
        m1(1, 1);                          m2(1, 1);
        m1(2, 2.0);                        m2(2, 2.0);
        m1(3.0, 3);                        m2(3.0, 3);
        m1(4.0, 4.0);                      m2(4.0, 4.0);
    }
}
```

Surcharge et espace de nommage

Il est possible de définir la même méthode (même nom, même nombre de paramètres, et mêmes types de paramètres) dans deux classes différentes.

```
class A {
    /** afficher en commençant
     * par un décalage */
    void afficher(int décalage);
}

class B {
    /** afficher nb fois */
    void afficher(int nb);
}
```

Pour savoir quelle méthode choisir, le compilateur s'appuie sur le type du récepteur (le type de la poignée sur laquelle la méthode est appliquée).

⇒ La classe définit un **espace de nommage**.

```
A x1;    // poignée x1 de type A
B x2;    // poignée x2 de type B
...      // les initialisations de x1 et x2
x1.afficher(5);    // afficher(int) de la classe A
x2.afficher(10);   // afficher(int) de la classe B
```

Remarque : Pour certains auteurs, il s'agit de surcharge. Pour nous, ce sont des espaces de nommage différents.

Méthodes : passage de paramètres

Java n'a qu'un seul mode de passage de paramètres : le *passage par valeur*.

Remarque : Si le paramètre est un « objet », c'est donc un passage par valeur de la poignée qui est équivalent à un passage par référence de l'objet.

Conséquences : On peut en déduire que :

- les modifications apportées à un paramètre de type primitif ne sont pas visibles du programme appelant (équivalent de variables locales initialisées) ;
- les modifications apportées à un objet attaché à un paramètre de type poignée sont visibles du programme appelant ;
- les modifications de la valeur de la poignée (réaffectation) ne seront pas visibles du programme appelant.

Remarque : Ceci est également vrai pour le retour d'une fonction !

Passage de paramètres : exemple

Exercice 15 : Comprendre le passage de paramètres en Java

Étant donnée la classe Compteur définie dans l'exercice 11 et dont le texte est donné listing 2, nous définissons la classe TestParametres (listing 1).

Son exécution donne le résultat suivant :

```
a = 10  
c = 11
```

Expliquer les résultats obtenus lors de l'exécution de TestParametres.

Listing 1 – Le fichier TestParametres.java

```
public class TestParametres {  
  
    public static void incrémenter(int n) {  
        n++;  
    }  
  
    public static void incrémenter(Compteur c) {  
        c.incrémenter();  
    }  
  
    public static void main(String[] args) {  
        int a = 10;  
        incrémenter(a);  
        System.out.println("a_=_ " + a);           // Valeur de a ?  
  
        Compteur c = new Compteur();  
        c.set(10);  
        incrémenter(c);  
        System.out.println("c_=_ " + c.getValeur()); // valeur de c ?  
    }  
}
```

Listing 2 – Le fichier Compteur.java

```
/** Définition d'un compteur avec incrémentation.
 * @author      Xavier Crégut
 * @version     1.2 */
public class Compteur {
    private int valeur;          // valeur du compteur

    /** Augmenter d'une unité le compteur */
    public void incrémenter()    { this.valeur++; }

    /** Obtenir la valeur du compteur.
     * @return la valeur du compteur.
     */
    public int getValeur()       { return this.valeur; }

    /** Remettre à zéro le compteur */
    public void raz()            { this.set(0); }

    /** Modifier la valeur du compteur.
     * @param valeur_ la nouvelle valeur du compteur
     */
    public void set(int valeur_) { this.valeur = valeur_; }
}
```

Constructeurs

Rappel : La création d'un objet nécessite en fait deux étapes :

1. *La réservation de la zone mémoire nécessaire.* Ceci est entièrement réalisé par le compilateur (à partir des attributs de la classe).
2. *L'initialisation de la zone mémoire.* Le compilateur ne peut, *a priori*, faire qu'une initialisation par défaut des attributs.

Il y a alors risque d'avoir une initialisation incorrecte, ou non conforme aux souhaits de l'utilisateur (exemple : Fraction, Date, etc.).

Les **constructeurs** permettent alors :

- au programmeur d'indiquer comment un objet peut être initialisé ;
- au compilateur de vérifier que tout objet créé est correctement initialisé.

Exemple : Une équation peut (doit !) être initialisée à partir de la donnée des valeurs de ses coefficients.

Les constructeurs en Java

En Java, un constructeur ressemble à une méthode mais :

- il a nécessairement le même nom que la classe ;
- il ne peut pas avoir de type de retour ;

Exemple : Un constructeur pour les équations

```
/** Initialiser une équation à partir de la donnée de ses coefficients */  
public Équation(double a, double b, double c) {  
    this.coeffA = a;  
    this.coeffB = b;  
    this.coeffC = c;  
}
```

Remarque : Un constructeur a un droit d'accès (idem attributs et méthodes).

Attention : Mettre un type de retour supprime le caractère « constructeur ».

On a alors une simple méthode !

Constructeur et surcharge

Même si le nom d'un constructeur est imposé, la surcharge permet de définir plusieurs constructeurs pour une même classe.

Exemple : On peut souhaiter initialiser une équation à partir de la somme et du produit de ses racines.

```
public Équation(double somme, double produit) {
    this(1, -somme, produit);
    // Appel au constructeur Équation(double, double, double)
    // Cet appel est nécessairement la première instruction !
}
```

Remarque : Un constructeur peut appeler un autre constructeur en utilisant `this(...)` comme toute première instruction. Les paramètres de `this` permettent de sélectionner l'autre constructeur.

Exercice 16 Peut-on définir un constructeur qui initialise une équation à partir de ses deux solutions ? Pourquoi ?

Création d'un objet

La création d'un objet en Java est alors :

```
new <Classe>(<paramètres effectifs>);
```

Les paramètres effectifs sont fournis par l'utilisateur de la classe et sont utilisés par le compilateur pour sélectionner le constructeur à appliquer (surcharge).

Si aucun constructeur n'est trouvé, le compilateur signale une erreur.

```
new Équation();           // Incorrect !  
new Équation(1, 5, 6);   // OK    $x^2 + 5x + 6$   
new Équation(2, 1);     // OK    $x^2 - 2x + 1$   
new Équation(10);      // Incorrect !
```

Conséquence : Le constructeur permet de rendre atomique la réservation de la mémoire et son initialisation.

Le constructeur par défaut

On appelle **constructeur par défaut** le constructeur qui ne prend pas de paramètres.

Justification : C'est le constructeur utilisé si aucun paramètre n'est fourni lors de la création d'un objet.

Règle : Le constructeur par défaut est régi par deux règles :

1. Si **aucun** constructeur n'est défini sur une classe, le système synthétise un constructeur par défaut (qui ne fait rien), le *constructeur prédéfini*.
2. Dès qu'un constructeur est défini sur une classe, le constructeur par défaut synthétisé par le système disparaît.

Remarque : Le programmeur peut toujours définir un constructeur par défaut... mais y a-t-il intérêt ?

Un constructeur n'est pas une méthode

Même si un constructeur ressemble à une méthode, ce n'est pas une méthode :

- il n'a pas de type de retour ;
- il a une syntaxe d'appel spécifique (associé à l'opérateur **new**) ;
- il ne peut pas être appliqué sur un objet (sauf lors de sa création) ;
- il ne peut pas être redéfini dans une classe dérivée (cf héritage T. 170).
- le caractère de « constructeur » ne s'hérite pas (cf héritage T. 170).

Autres manières d'initialiser un objet

Outre les constructeurs, Java permet au programmeur de définir :

- des *valeurs par défaut pour les attributs*. Elles remplacent alors les valeurs par défaut du langage ;
- des *initialiseurs*. Ce sont des instructions mises entre accolades. S'il y a plusieurs initialiseurs, ils sont exécutés dans l'ordre d'apparition.

Exemple : La classe Fraction.

```
public class Fraction {
    private int num;        // valeur par défaut de Java : 0
    private int dén = 1;   // valeur par défaut du programmeur : 1
    { // initialiseur
        num = 0;
        dén = 5;
    }
    { // initialiseur 2
        dén = 1;
    }
} // dén prendra successivement les valeurs : 0, 1, 5 et 1
```

Initialisation d'objet : Bilan

Voici ce que fait Java lors de la création d'un objet :

1. Initialisation des attributs avec la valeur par défaut de leur type ;
2. Utilisation des valeurs par défaut fournies par le programmeur ;
3. Exécution des initialiseurs dans leur ordre d'apparition ;
4. Exécution du constructeur :
 - si aucun constructeur n'est défini sur la classe, c'est le constructeur prédéfini qui est utilisé. Le programmeur ne doit pas fournir de paramètre au constructeur ;
 - si au moins un constructeur est défini sur la classe, le programmeur doit fournir des paramètres (éventuellement aucun !) qui permettent au compilateur de choisir l'un des constructeurs de la classe.

Conseil : Préférer les constructeurs explicites aux autres initialisations (valeurs par défaut, initialiseurs, constructeur par défaut synthétisé).

Destructeurs

Destructeur : méthode appelée automatiquement quand un objet disparaît (quand sa mémoire est libérée). Il est le pendant du constructeur.

Conséquence : Son code contient les traitements à réaliser lors de la disparition de l'objet : libération des ressources utilisées (mémoire...), etc.

Attention : Il ne peut y avoir qu'un seul constructeur par classe.

En Java : En Java, le destructeur est :

```
protected void finalize()
```

En Java, le ramasse-miettes rend généralement inutile la définition du destructeur.

Attention : En raison du ramasse-miettes, aucune garantie n'existe en Java sur quand le destructeur sera appelé... ou s'il sera réellement appelé.

⇒ Définir une méthode explicite... et dire aux utilisateurs de penser à l'appeler !!!

Les attributs et méthodes de classe

Exercice 17 On considère une classe Date qui a pour attributs l'année, le numéro du mois et le numéro du jour dans le mois. On souhaite définir sur cette classe une méthode incrémenter qui fait passer la date au lendemain.

Comment écrire le code de cette méthode ?

Attributs et méthodes de classe : on les a déjà vus !

Que penser de l'expression suivante :

```
Math.sqrt(4)    // racine carrée de 4
```

`sqrt(double)` est bien une méthode mais elle n'est pas appliquée à un objet mais à la classe `Math`. C'est une **méthode de classe**.

On constate que `sqrt(double)` travaille exclusivement sur son paramètre.

Que penser de l'instruction suivante :

```
System.out.println("Que_suis-je_?");
```

`out` est un attribut (puisque'il n'est pas suivi de parenthèses) mais il est appliqué à une classe (`System`). C'est parce qu'il n'est pas spécifique d'un objet particulier : la sortie standard est la même pour tout le monde !

C'est en fait un **attribut de classe**.

Attributs et méthodes de classe : en Java

On distingue :

- attributs et méthodes d'instance : toujours appliqués à un objet (éventuellement `this`). On les appelle simplement attributs et méthodes ;
- attributs et méthodes de classe : appliqués à une classe non à un objet.

Syntaxe : C'est le modifieur `static` qui indique si un attribut ou une méthode est de classe ou non.

Droit d'accès : Les mêmes que pour les attributs et méthodes d'instance.

```
public class Math {  
    public static double sqrt(double) {}  
}  
|  
public class System {  
    public static PrintStream out;  
}
```

Utilisation : `NomClasse.attribut` OU `NomClasse.méthode(...)`

Si `NomClasse` est la classe « courante », elle peut être omise.

Question : Que penser de la méthode principale ?

Attribut de classe

Définition : Un attribut de classe est un attribut qui n'est pas spécifique d'une instance particulière de la classe. Il est commun à (et partagé par) tous les objets de la classe.

Intérêt : Équivalent à une variable globale dont la portée est la classe.

Exemple : Compter le nombre d'équations créées.

Le compteur est une information relative à la classe mais qui doit être mise à jour par chacune des instances (dans chacun des constructeur de la classe).

```
public class Équation {
    private static int nbCréées = 0;        // initialisation explicite

    public Équation(double a, double b, double c) {
        nbCréées++;                        // idem : Équation.nbCréées++;
        ...
    }
}
```

Initialisation des attributs de classe

Les attributs de classe sont initialisés au chargement de la classe.

Ils sont initialisés avec la valeur par défaut de leur type, puis la valeur par défaut fournie par le programmeur, et enfin par l'initialiseur statique (un bloc d'instructions précédé de **static**).

```
class UneClasse {  
    static public int i;        // initialisé à 0 (valeur par défaut des int)  
    public static int j = 10; // valeur par défaut du programmeur  
    public static int k;  
  
    static {                    // initialiseur statique  
        k = 1; // il est utile pour les initialisations complexes (tableaux)  
        j = 5; // 5 remplace la valeur 10 fournie comme défaut  
    }  
}
```

Règle : Comme ceux d'instance, déclarer les attributs de classe **private**.

Méthodes de classe

Définition : Une méthode de classe est une méthode indépendante de toute instance de la classe. Elle est donc appliquée à une classe et non à un objet.

Conséquence : Une méthode de classe n'a pas de paramètre implicite (**this**) et ne peut donc pas utiliser les attributs et méthodes d'instance de sa classe.

```
public class Date {
    private int jour, mois, année;
    static private int nbCréées = 0;    // nombre de dates créées

    public static boolean estBissextile(int a) {
        return ((a % 4) == 0)           // divisible par 4
            && ((a % 100 != 0)         // et non divisible par 100
                || (a % 400 == 0));    // sauf si divisible par 400
    }

    public static int getNbCréées() {
        return nbCréées;
    }
}
```

Exercice 18 Comment savoir si une méthode doit être de classe ?

Importation statique

Il existe une variante de la clause `import` qui permet d'accéder directement aux attributs et méthodes de classe.

```
1  import static java.lang.System.out; // importe une caractéristique
2  import static java.lang.Math.*;     // ou toutes
3
4  public class ImportationStatiqueMath {
5      public static void main(String[] args) {
6          double pi = PI; // java.lang.Math.PI
7          out.println(sqrt(pi));
8          System.out.println(Math.sqrt(pi));
9      }
10 }
```

Intérêt : Il est relativement faible :

- écrire `sqrt` plutôt que `Math.sqrt` ;
- utiliser directement des constantes `RED` au lieu `Color.RED` ;

En fait, il vise à **éviter de mauvaises pratiques** : hériter (T. 170) des classes `Math` ou `java.awt.Color`.

Les classes du point de vue des utilisateurs

Jusqu'à maintenant, nous avons défini une classe en nous plaçant du point de vue du programmeur chargé de la réaliser. Il est intéressant de se placer du point de vue des programmeurs qui l'utiliseront (les « utilisateurs »).

Point de vue du programmeur : une classe est composée de :

- *attributs* : stockage d'informations (conserver l'état de l'objet) ;
- *méthodes* : unités de calculs.

Point de vue de l'utilisateur : une classe est un ensemble de :

- *requêtes* : informations qui peuvent être demandées à la classe ;
- *commandes* : services réalisés par la classe.

Une requête correspond soit à un attribut (information stockée), soit à une méthode (information calculée).

L'utilisateur de la classe doit accéder de la même manière à une information, qu'elle soit calculée ou stockée. C'est le **principe de l'accès uniforme**. En Java, il est donc nécessaire de définir des méthodes d'accès.

Comment définir une classe

Pour définir une classe, je conseille de suivre les étapes suivantes :

1. Définir l'interface utilisateur :
 - Identifier les requêtes ;
 - Identifier les commandes ;
 - Identifier les constructeurs.
2. Choisir une représentation (choix des attributs).

Les attributs sont définis **private**.
3. Implanter les requêtes et les commandes sous forme de méthodes.

Ce peut être une simple méthode d'accès pour les requêtes qui correspondent à un attribut.
4. Tester au fur et à mesure !

Conventions de codage : règles de nommage

Ces conventions de codage peut être trouvées à l'URL :

<http://java.sun.com/docs/codeconv/index.html>

Règles de nommage :

- paquetage : tout en minuscule (`java.lang`) ;
- classe : les initiales en majuscule (`Equation`, `MaClasse`) ;
- méthode : en minuscule sauf l'initial des mots internes en majuscule (`résoudre`, `setCoeffA`, `maMéthode`) ;
- attribut : comme les méthodes ;
- constante : tout en majuscule avec les mots séparés par des soulignés (`_`) (`MAX`, `MA_CONSTANTE`).

Conventions de codage : structure d'un fichier

```
package ...;    // définition du paquetage d'appartenance

import ...;    // importation des classes utilisées

/**
 * Commentaire de documentation de la classe
 * @version
 * @author      Prénom Nom
 */

public class MaClasse { // ==> Le fichier est MaClasse.java

    /* commentaire d'implémentation de la classe */

    // variables (attributs) de classe

    // variables (attributs) d'instance

    // constructeurs

    // méthodes. Les méthodes sont regroupées par thème
    //           et non par droit d'accès.

}
```

Conventions de codage : règles diverses

- **Indentation** : 4 espaces, la tabulation est fixée à 8.
- **Longueur des lignes** : 80 caractères maximum. Les lignes sont coupées de préférence après une virgule ou devant un opérateur.
- Déclarer une seule variable par ligne (facilite sa documentation) :

```
int maVariable;           // sa documentation
```

Déclarer les variables en début de bloc (bof!).

- Ne pas mettre de () après **return** (sauf si améliorent la compréhension).
- Pas d'espace entre méthode et (. Mettre un espace entre mot-clé et (, autour d'un opérateur, devant une {, après , et ;.

```
while_(Math.sqrt(x)>_y)_ {  
    _____  
}
```

Conventions de codage : programmation (1/2)

- Ne pas mettre les attributs (de classe ou d'instance) publics.

Définir *éventuellement* des accesseurs et des modifieurs.

- Utiliser la classe et non un objet pour accéder à un attribut ou méthode de classe.

```
MaClasse.méthodeDeClasse();    // OUI
unObjet.méthodeDeClasse();     // À ÉVITER
```

- Ne pas utiliser directement des constantes littérales (sauf pour -1, 0 et 1).
- Éviter les affectations multiples (`a = b = c;`) ou dans des expressions
`a = (a = b + c) + r;`
- Utiliser les parenthèses dans des expressions mélangeant plusieurs opérateurs.

Conventions de codage : programmation (2/2)

- Faire que la structure du programme ressemble à votre intention.

À éviter

```
if (condition) {  
    return true;  
} else {  
    return false;  
}
```

```
if (condition) {  
    return x;  
}  
return y;
```

À préférer

```
return condition;
```

```
return (condition ? x : y);
```

- Mettre `XXX` dans un commentaire pour signaler un code maladroit mais qui fonctionne et `FIXME` pour un code qui ne fonctionne pas.

Les tableaux

Les tableaux en Java se rapprochent beaucoup des objets :

- ils sont accessibles par une poignée ;

```
int[] tab1;    // tab1 est une poignée sur un tableau (tab1 == null)
int tab2[];   // on peut déclarer les tableaux comme en C
Type[] tab;   // une poignée tab sur un tableau de Type
```

- ils sont créés dynamiquement en utilisant l'opérateur **new**.

```
tab1 = new int[5]; // création d'un tableau de 5 entiers attaché à tab1
tab = new Type[capacité]; //création d'un tableau de capacité Type
```

Mais ce ne sont pas des objets :

- ils ont une syntaxe spécifique (les crochets) ;
- ils sont le seul type générique de Java (paramétré par un type, celui des éléments du tableau) en attendant la version 1.5 !

Caractéristiques d'un tableau

- Un tableau non créé ne peut pas être utilisé (NullPointerException)
- La capacité du tableau est obtenue par l'« attribut » length.

```
double tab[] = new double[10];  
int nb = tab.length;           // nb == 10 (nb de cases allouées)
```

- L'accès à un élément se fait par les crochets ([]).

Les indices commencent à 0 (premier élément) et se terminent à length-1 (dernier élément) \implies ArrayIndexOutOfBoundsException.

```
for (int i = 0; i < tab.length; i++) {  
    tab[i] = i;  
}  
int p = tab[0];           // premier élément  
int d = tab[tab.length-1]; // dernier élément  
int e = tab[tab.length];  // ArrayIndexOutOfBoundsException
```

- L'affectation de deux tableaux est une affectation de poignée (partage).

On peut utiliser `System.arraycopy(...)`.

Caractéristiques d'un tableau (suite)

- Lorsqu'un tableau est créé (**new**), chacun de ses éléments est initialisé avec la valeur par défaut de son type (**false**, **0**, **null**).
- Il est possible d'initialiser explicitement le tableau :

```
int[] petitsNbPremiers = { 3, 5, 7, 11, 13 };  
int nb = petitsNbPremiers.length;           // nb == 5  
String[] nomJours = { "lundi", "mardi", "mercredi", ..., "dimanche" };  
nb = nomJours.length;                       // nb == 7
```

- Les tableaux ne peuvent pas être redimensionnés (`length` ne peut pas changer de valeur). Voir `java.util.Vector`.
- À l'exception des tableaux de caractères, l'affichage par `System.out.println` n'est pas lisible.

Tableaux d'objets

On peut bien sûr créer des tableaux d'objets. Tout fonctionne comme les tableaux de types élémentaires sauf que le contenu d'une case est une poignée sur un objet du type précisé.

```
1  public class TableauÉquations {
2      public static void main (String args []) {
3          Équation[] système = new Équation[3];
4          système[0] = new Équation(1, 5, 6);
5          système[1] = new Équation(4, 4);
6          for (int i = 0; i < système.length; i++) {
7              système[i].résoudre();
8              System.out.print("Équation_" + (i+1) + "_:_");
9              système[i].afficher();
10             System.out.println("x1_=" + système[i].x1);
11             System.out.println("x2_=" + système[i].x2);
12         }
13     }
14 }
```

Résultat de l'exécution

Équation 1 : $1.0 \cdot x^2 + 5.0 \cdot x + 6.0 = 0$

x1 = -2.0

x2 = -3.0

Équation 2 : $1.0 \cdot x^2 + -4.0 \cdot x + 4.0 = 0$

x1 = 2.0

x2 = 2.0

Exception in thread "main" java.lang.NullPointerException
at TableauÉquations.main(TableauÉquations.java:7)

Autre manière d'écrire le programme (Java 1.5)

```
1 public static void main (String args []) {
2     Équation[] système = { new Équation(1, 5, 6),
3         new Équation(4, 4), null };
4     for (Équation eq : système) {
5         eq.résoudre();
6         System.out.print("Équation_:_"); // sans numéro !
7         eq.afficher();
8         System.out.println("x1_=_ " + eq.x1);
9         System.out.println("x2_=_ " + eq.x2);
10    }
11 }
```

Tableaux à plusieurs dimensions

Les tableaux à deux dimensions (ou plus) sont en fait des tableaux de tableaux (de tableaux...). Ils peuvent être initialisés de deux manières.

1. Allocation de toutes les cases en une seule fois

```
1  int[][] matrice = new int[2][3];
2  for (int i = 0; i < 2; i++)
3      for (int j = 0; j < 3; j++)
4          matrice[i][j] = i+j;
5  afficher(matrice);
```

Remarque : On a un tableau de tableaux. On peut donc manipuler une « ligne » par l'intermédiaire d'une poignée.

```
1  // permuter les deux premières lignes
2  int[] ligne = matrice[0];
3  matrice[0] = matrice[1];
4  matrice[1] = ligne;
5  afficher(matrice);
```

Tableaux à plusieurs dimensions (suite)

2. Allocation individuelle de chacun des (sous-)tableaux

```
1 //           Le triangle de Pascal
2 // Créer le tableau de lignes
3 int[][] triangle = new int[10][];
4 // Construire la première ligne
5 triangle[0] = new int[2];
6 triangle[0][0] = triangle[0][1] = 1;
7 // Construire les autres lignes
8 for (int i = 1; i < triangle.length; i++) {
9     // Création de la (i+1)ème ligne du triangle
10    triangle[i] = new int[i+2];
11    triangle[i][0] = 1;
12    for (int j = 1; j < triangle[i].length - 1; j++) {
13        triangle[i][j] = triangle[i-1][j-1] + triangle[i-1][j];
14    }
15    triangle[i][i+1] = 1;
16 }
17
18 afficher(triangle);
```

Tableaux à plusieurs dimensions : utilisation

Un tableau à plusieurs dimensions est un tableau de tableaux dont :

- toutes les cases (« lignes ») n'ont pas nécessairement la même capacité ;
- les cases peuvent ne pas être allouées (**null**).

Exemple : Afficher un tableau à deux dimensions d'entiers.

```
1  static void afficher(int[][] mat) { // On suppose mat != null
2      for (int i = 0; i < mat.length; i++) {
3          if (mat[i] == null) { // mat[i] non alloué
4              System.out.println();
5          } else {
6              System.out.print(+ mat[i][0]);
7              for (int j = 1; j < mat[i].length; j++) {
8                  System.out.print(", " + mat[i][j]);
9              }
10             System.out.println();
11         }
12     }
13 }
```

La classe String

Attention : Les « String » sont des objets. Elles sont donc accessibles au moyen de poignées.

```
String s0;           // Une poignée non initialisée (éventuellement null)
String s1 = null;   // Une poignée initialisée à null
String s2 = "";     // Une chaîne de caractères vide
String s3 = "Bonjour";
String s4 = new String("Bonjour"); // équivalent mais plus long !
String s5 = s3 + "Xavier";         // concaténation
```

- s0 et s1 ne sont pas des chaînes de caractères mais des poignées nulles !
- Les chaînes de caractères littérales se notent entre guillemets.
- l'opérateur + correspond à la concaténation des chaînes de caractères (si l'un des paramètres est une chaîne).

```
int valeur = 5;
String s6 = "Total_=_"+ valeur + '.';           // Total = 5.
String s7 = "Total_=_"+ (valeur + '.');        // Total = 51
```

Attention : Les parenthèses changent l'évaluation !

Les « String » sont des objets

```
1 // Les « String » sont des objets
2 String s1 = "Bonjour";
3 int lg = s1.length(); // la longueur de la chaîne : 7
4 char initiale = s1.charAt(0); // 'B'
5 char erreur = s1.charAt(lg); // -> StringIndexOutOfBoundsException
6 // les indices sur les chaînes vont de 0 à length()-1.
7
8 String s2 = s1.substring(0, 3); // "Bon"
9 String s3 = s1.substring(3, 7); // "jour"
10 // substring(int début, int fin) : sous-chaîne comprise
11 // entre les indices début inclu et fin exclu.
12
13 String s4 = s1.toUpperCase(); // BONJOUR
14 String s5 = s1.toLowerCase(); // bonjour
15 String s6 = s1.replace('o', '.'); // B.nj.ur
16 int p1 = s1.indexOf("on"); // 1
17 int p2 = s1.indexOf("on", 2); // -1 (non trouvé !)
18
19 String s7 = "Un_texte_avec_des_blancs";
20 String s8 = s7.replaceAll("\\s+", " "); // remplace blancs par espace
21 String[] mots = s7.split("\\s+");
22 // mots == { "Un", "texte", "avec", "des", "blancs" }
23 // replaceAll et split ont une expression régulière comme paramètre
```

Comparaisons de chaînes (String)

Deux types d'égalité :

- l'**égalité physique** : deux chaînes correspondent au même objet en mémoire. C'est l'égalité de poignée (`s1 == s2`).

```
String s1 = "N7_2TR";  
boolean estN7 = s1.substring(0, 2) == "N7"; // généralement faux !
```

- l'**égalité logique** : deux chaînes sont composées des mêmes caractères.

```
boolean estN7 = s1.substring(0, 2).equals("N7"); // toujours vrai !
```

Autre comparaison logique :

```
int res = s1.compareTo(s2); // comparer suivant l'ordre lexicographique  
// négatif si s1 < s2 ; nul si s1.equals(s2) ; positif si s1 > s2
```

Rq : On a équivalence entre `s1.equals(s2)` et `s1.compareTo(s2) == 0` !

Attention : Une chaîne de caractères (String) ne peut pas être altérée (*immutable* !). Il est donc nécessaire de construire de nouvelles chaînes. (voir StringBuffer.)

La classe `StringBuffer`

Un « `StringBuffer` » est une chaîne de caractères (comme « `String` ») qui peut être modifiée !

En plus de la majorité des méthodes de `String`, `StringBuffer` propose

- `append` : ajouter à la fin de la chaîne ;
- `insert` : ajouter à une position spécifiée de la chaîne.

Ces deux méthodes sont largement surchargées !

Passage de `String` à `StringBuffer` et inversement :

```
StringBuffer t1 = new StringBuffer("Bonjour");
StringBuffer t2 = new StringBuffer();    // Chaîne de longueur 0 !
String s2 = t1.toString();
StringBuffer t3 = new StringBuffer(s2);
StringBuffer t4 = "toto";                // Interdit !
```

Pourquoi String et StringBuffer ?

- Les « String » sont inaltérables. Ainsi, l'affectation entre chaînes de caractères peut se faire par affectation des poignées, sans copies explicites.
 - ⇒ gain de mémoire et de temps.
- Le caractère inaltérable des « String » fait que toute modification nécessite la création d'une nouvelle chaîne, donc de l'allocation de mémoire !
 - ⇒ StringBuffer permet de faire des modifications sur la même zone mémoire et évite donc les réallocations.

Remarque : StringBuffer est utilisée par le compilateur pour implanter la concaténation des chaînes de caractères. L'instruction :

```
x = "a" + 4 + "c"
```

est transformée par le compilateur en :

```
x = new StringBuffer().append("a").append(4).append("c").toString()
```

String ou StringBuffer ? Un exemple !

```
public class ConcatenerString {
    public static void main(String[] args) {
        String chaîne = "";
        for (int i = 0; i < 100000; i++) {
            chaîne = chaîne + 'x';
        }
        System.out.println("Longueur_de_chaîne_=" + chaîne.length());
    }
}

public class ConcatenerStringBuffer {
    public static void main(String[] args) {
        StringBuffer tampon = new StringBuffer();
        for (int i = 0; i < 100000; i++) {
            tampon.append('x');
        }
        String chaîne = tampon.toString();
        System.out.println("Longueur_de_chaîne_=" + chaîne.length());
    }
}
```

String ou StringBuffer ?

Temps d'exécution :

```
> time java ConcatenerString
Longueur de chaîne = 100000
real 1335.64
user 1302.43
sys 26.20
```

```
> time java ConcatenerStringBuffer
Longueur de chaîne = 100000
real 0.70
user 0.58
sys 0.06
```

Les concaténations entre String sont plus claires que les opérations sur StringBuffer, donc :

- Préférer les « StringBuffer » si de nombreuses modifications doivent être apportées à une chaîne (par exemple dans une boucle).
- Préférer les « String » pour des affectations simples de chaînes (le compilateur fera la transformation en StringBuffer pour vous !).

Les classes enveloppes

- Les types primitifs (`int`, `double`, `boolean`...) ne sont pas des objets.
- Cependant, pour chacun des types primitifs, il existe une classe correspondante, appelée *classe enveloppe* (*wrapper*) dans `java.lang`.
- Chaque classe enveloppe permet de construire un objet à partir d'une valeur du type primitif, et possède une méthode d'accès retournant la valeur du type primitif.
- Les instances des classes enveloppes sont des objets *non altérables* !
- Les classe enveloppes fournissent les valeurs minimales (`MIN_VALUE`) et maximales (`MAX_VALUE`) du type.
- **Intérêt** : Les classes enveloppes n'ont d'intérêt que si l'on a besoin de considérer les types primitifs comme des objets (T. 155).
- Depuis Java 1.5, la conversion entre types primitifs et classes enveloppes est automatique (*auto boxing/unboxing*).

Exemple : caractéristiques de la classe enveloppe Integer

Les constantes de classe :

```
public static final int MAX_VALUE;           // Plus grande valeur entière  
public static final int MIN_VALUE;          // Plus petite valeur entière
```

Les constructeurs :

```
public Integer(int);  
public Integer(String);
```

Les méthodes de classe

```
static int parseInt(String s, int radix);  
static int parseInt(String);  
static Integer decode(String); // plus générale : decode("0xFF")
```

Les méthodes de conversion vers d'autres types

```
int intValue();  
float floatValue();  
double doubleValue();  
String toString();
```

Exemple d'utilisation de la classe Integer (Java 1.4)

```
1 Integer n1 = new Integer(15);           // construire depuis un int
2 Integer n2 = new Integer("100");       // construire depuis String
3
4 // int i1 = n1;           // Erreur : types différents !
5 int i1 = n1.intValue();
6 double d2 = n2.doubleValue(); // d2 == 100.0
7
8 int i3 = Integer.parseInt("421");      // i3 == 421
9 Integer n4 = Integer.decode("0xFFF");  // n4 == 4095
10
11 boolean test;
12 // test = n1 == i1;      // Types incompatibles
13 // test = i1 == n1;      // Types incompatibles
14
15 test = n1.intValue() == i1; // true
16 test = n1 == new Integer(i1); // false
17
18 // Convertir le premier paramètre de la ligne de commande.
19 if (args.length > 0) {
20     Integer n5 = Integer.decode(args[0]);
21     int i5 = Integer.parseInt(args[0]);
22     // Attention à NumberFormatException !
23 }
```

Exemple d'utilisation de la classe Integer (Java 1.5)

```
1 Integer n1 = new Integer(15);           // construire depuis un int
2 Integer n2 = new Integer("100");       // construire depuis String
3
4
5 int i1 = n1;                            // en fait : i1 = n1.intValue()
6 double d2 = n2.doubleValue();          // d2 == 100.0
7
8 int i3 = Integer.parseInt("421");       // i3 == 421
9 Integer n4 = Integer.decode("0xFFF");   // n4 == 4095
10
11 boolean test;
12 test = n1 == i1;                        // true
13 test = i1 == n1;                        // true
14
15 test = n1.intValue() == i1;             // true
16 test = n1 == new Integer(i1);          // false
17
18 // Convertir le premier paramètre de la ligne de commande.
19 if (args.length > 0) {
20     Integer n5 = Integer.decode(args[0]);
21     int i5 = Integer.parseInt(args[0]);
22     // Attention à NumberFormatException !
23 }
```

Les énumérations

Depuis Java 1.5, il est possible de définir des types énumérés.

```
1 public enum Fruit { POMME, POIRE, ORANGE, PRUNE };
1 public enum Couleur { JAUNE, VIOLET, ORANGE };
1 class ExempleEnum {
2     static Couleur getCouleur(Fruit f) {
3         Couleur resultat = null;
4         switch (f) {
5             case POMME: resultat = Couleur.JAUNE; break;
6             case ORANGE: resultat = Couleur.ORANGE; break;
7             case PRUNE: resultat = Couleur.VIOLET; break;
8         }
9         return resultat;
10    }
11    public static void main(String[] args) {
12        for (Fruit f : Fruit.values()) {
13            System.out.println(f + "_est_" + getCouleur(f));
14        }
15    }
16 }
```

Les énumérations : propriétés

- Une énumération est une classe.
Chaque « valeur » est équivalente à un attribut de classe constant.
- Le même nom (ORANGE) peut être utilisé dans deux énumérations (espaces de noms différents).
- On peut utiliser un **switch** avec une expression de type énumération.

Intérêt :

- Évite d'avoir à définir des constantes entières.
- Contrôle de type fort (à la compilation !).
- Les constantes ne sont pas compilées dans le code client.
- On peut obtenir (`name()`) et afficher le nom d'une valeur.

POMME est JAUNE
POIRE est **null**

ORANGE est ORANGE
PRUNE est VIOLET

- On peut ajouter des méthodes et des attributs dans une classe énumérée.

Relations entre classes

On dit qu'il y a relation de dépendance entre une classe A et une classe B si la classe A fait référence à la classe B dans son texte.

Cette relation peut être momentanée si B apparaît comme

- un paramètre d'une méthode ;
- une variable locale ;
- le retour d'une méthode.

Cette relation est structurelle si elle dure, c'est généralement le cas quand B est un attribut. En UML, on fait apparaître une relation entre les classes.

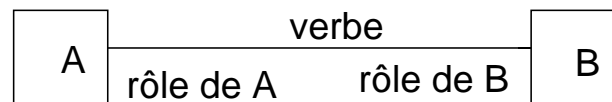
En Java, cette relation correspond, par défaut, à un partage d'objets sauf si le programmeur réalise explicitement des copies.

En UML elle peut être précisée : association, agrégation ou composition.

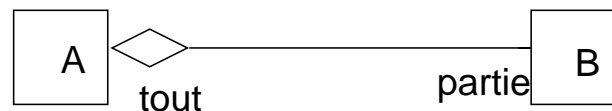
Relation entre classes

Une application est composée de plusieurs classes dont le couplage est caractérisé par des relations (d'utilisation) :

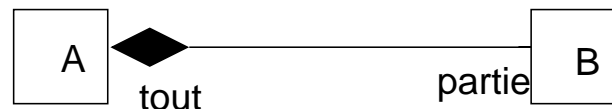
- **association** : couplage faible correspondant à une relation symétrique entre objets relativement indépendants (durées de vie non liés) ;



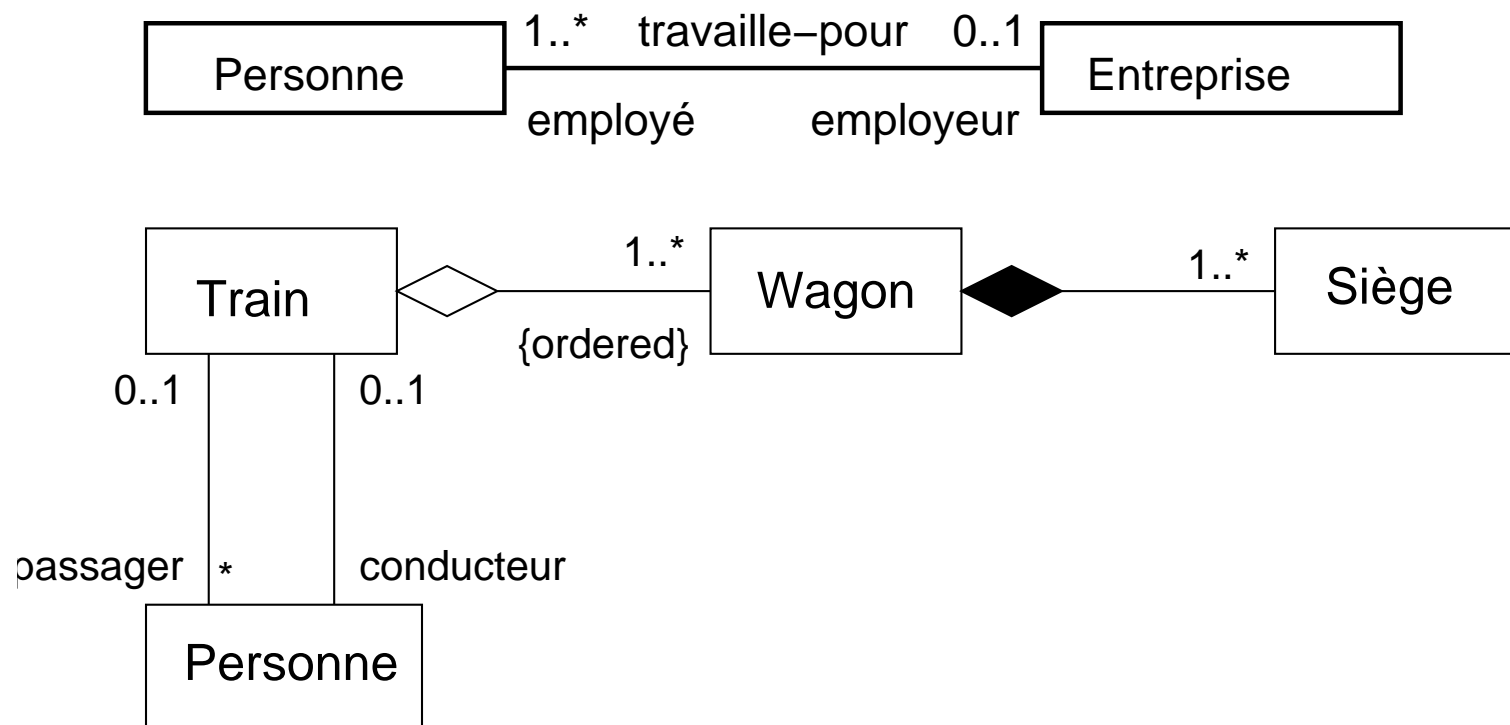
- **agrégation** : association non symétrique avec couplage plus fort, relation de subordination. C'est une relation de type tout-partie ;



- **composition** : agrégation forte (par valeur). La durée de vie des objets « partie » est liée à celle du « tout ». Pas de partage possible.



Relations entre classes : exemples



Exercice 19 Dessiner un diagramme de classes faisant apparaître un site web, des pages HTML et un « webmaster ».

Traduction en Java



```
public class Train {
    private Personne conducteur;
    // Pour une multiplicité de 1 ou 0..1 (null ?)

    private Personne[] passagers;
    // Au lieu d'un tableau, il est préférable
    // d'utiliser une structure de données adaptée.

    ...
}
```

- La flèche indique le sens de navigation de la relation : d'un objet train on peut obtenir le conducteur ou les passagers mais pas l'inverse.
- Agrégation et association se représentent de la même façon en Java.
- Pour la composition, plusieurs stratégies sont possibles (voir TD).

Interfaces

Exercice 20 : Liste de réels

On considère une liste de réels (**double**) offrant les opérations suivantes :

- connaître la taille de la liste (son nombre d'éléments) ;
- obtenir l'élément à la position i de la liste ;
- remplacer le i^{e} élément de la liste ;
- ajouter un élément dans la liste en position i ;
- supprimer l'élément à la position i de la liste.

On doit avoir $0 \leq \text{indice} < \text{taille}$ sauf pour ajouter car ajouter à taille signifie ajouter en fin. **On ne traitera pas les cas d'erreur.**

20.1 Dessiner le diagramme UML de la classe Liste.

20.2 Écrire une méthode dans une classe `OutilsListe` qui calcule la somme des réels d'une liste.

20.3 On hésite entre stocker les éléments de la liste dans un tableau ou les chaîner. Ceci remet-il en cause le travail fait ?

Constatations

- On sait ce qu'est une liste :
 - on peut donner la signature de chaque méthode ;
 - on peut en donner la spécification (commentaire javadoc) ;

⇒ On peut donc utiliser une liste.
- **mais** on ne sait pas :
 - où sont stockés les éléments ;
 - ni, donc, écrire le code (implantation) des méthodes ;

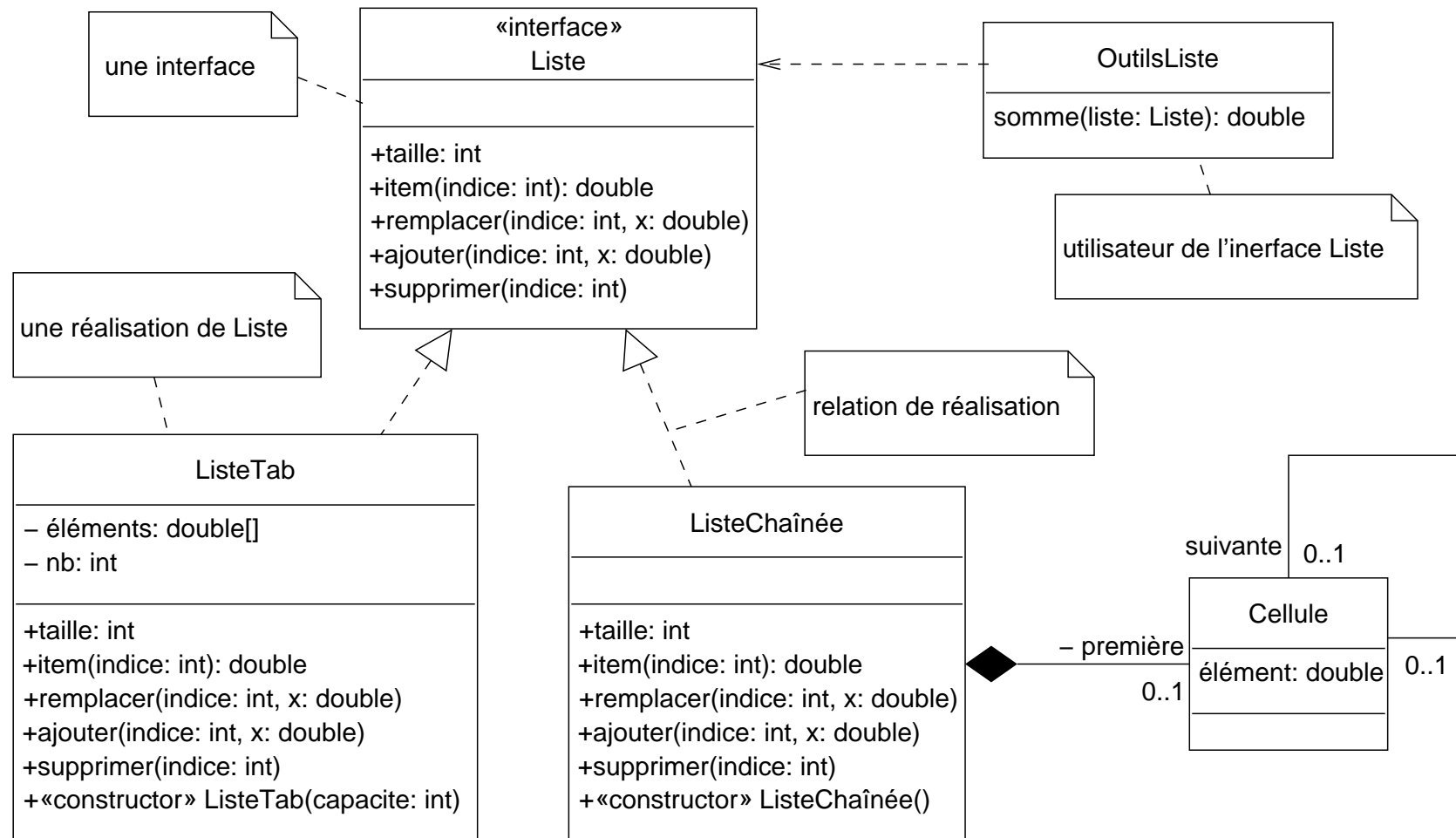
⇒ **On ne peut pas écrire une classe.**

Solution : Écrire une **interface** `Liste` (spécifier le comportement).

Remarque : Un jour, il faudra coder les méthodes de la liste et donc faire un choix de représentation. On pourra ainsi avoir plusieurs réalisations :

- les éléments sont stockés dans un tableau : `ListeTab` ;
 - les éléments sont chaînés entre eux : `ListeChaînée`.
- ⇒ `ListeTab` et `ListeChaînée` sont des **réalisations** de l'interface `Liste`.

Notation UML



Qu'est ce qu'une interface ?

Définition : Une interface est le point de jonction entre des classes utilisant l'interface et des classes la réalisant.

Exemple : L'interface `Liste` est le point de jonction entre `OutilsListe` (en fait, `OutilsListe.somme`) et `ListeTab` ou `ListeChaînée`.

Essentiel : Une interface définit un comportement qui doit être respecté par ses réalisations et sur lequel peuvent s'appuyer ses utilisateurs.

Conséquence : La documentation est essentielle ! La signature des méthodes ne suffit pas : utilisateurs et implémenteurs ne doivent pas pouvoir avoir des interprétations incompatibles (ou contradictoires).

Intérêt : Une interface définit un contrat entre utilisateurs et réalisateurs :

- les implémenteurs doivent le respecter ;
- les utilisateurs savent comment utiliser une interface.

Exemple : l'interface Liste

```
1  /** Spécification d'une liste. @version 1.4 */
2  public interface Liste {
3      /* Obtenir la taille de la liste.
4       * @return nombre d'éléments dans la liste */
5      int taille();
6
7      /* Obtenir un élément de la liste.
8       * @param indice position de l'élément */
9      double item(int indice);
10
11     /* Remplacer un élément de la liste.
12      * @param indice indice de l'élément à remplacer
13      * @param x nouvelle valeur */
14     void remplacer(int indice, double x);
15
16     /* Ajouter un élément dans la liste.
17      * @param indice indice où doit se trouver le nouvel élément
18      * @param x élément à insérer */
19     void ajouter(int indice, double x);
20
21     /* Supprimer un élément de la liste.
22      * @param indice indice de l'élément à supprimer */
23     void supprimer(int indice);
24 }
```

Quelques contraintes sur les interfaces

Une *interface* ressemble à une classe **mais** :

- Tous les éléments d'une interface doivent nécessairement (et ont implicitement) un droit d'accès **public**.

Justification : Quel intérêt de spécifier quelque chose d'inaccessible ?

- Une interface ne peut pas contenir de code : seule la signature et la spécification des méthodes sont données. Jamais le code !

Justification : Une interface est une spécification et ne doit donc pas :

- (trop) contraindre les choix de ses implémenteurs ;
- surcharger les utilisateurs de détails inutiles.
- Une interface ne peut pas contenir d'attributs (sauf constants, **final**).
- Il n'est pas possible de spécifier de constructeur.

Justification : Intérêt d'un constructeur sans son implantation ?

Instances d'une interface ?

Constatation : Une interface décrit seulement un comportement, une abstraction (elle contient la spécification et non le code).

⇒ Une interface décrit une *notion abstraite*.

Conséquence : Impossible de créer une instance à partir d'une interface.

Justification : Quel sens aurait une telle instance ?

```
1  /** Une interface ne peut pas être instanciée ! */
2  public class TestListeInstanceErreur {
3      public static void main(String[] args) {
4          Liste une_liste = new Liste();
5          une_liste.ajouter(0, 3.14);    // Quel sens ?
6      }
7  }
```

```
TestListeInstanceErreur.java:4: Liste is abstract; cannot be instantiated
    Liste une_liste = new Liste();
                        ^
```

Conséquence : Définir des classes qui réalisent l'interface et qui pourront, *elles*, avoir des instances (ListeTab et ListeChaînée).

Réalisation d'une interface

Définition : On appelle *réalisation d'une interface* une classe qui s'engage à définir les méthodes spécifiées dans cette interface.

Notation : En Java, on utilise le mot-clé `implements` pour dire qu'une classe *réalise* une interface :

```
class ListeTab implements Liste {  
    ...  
}
```

Remarque : Une même classe peut réaliser un nombre quelconque d'interfaces.

```
class A implements I1, I2, I3 {  
    ...  
}
```

Attention : Si une classe ne définit pas toutes les méthodes des interfaces qu'elle réalise, elle est dite *abstraite* (voir T. 190).

Exemple de réalisation

```
1  /** Réalisation de la Liste en stockant les éléments dans un tableau. */
2  public class ListeTab implements Liste {
3      private double[] elements; // les éléments de la liste
4      private int nb; // la taille de la liste
5      /** Construire une liste vide.
6          * @param capacite capacité initiale de la liste
7          */
8      public ListeTab(int capacite) {
9          this.elements = new double[capacite];
10         this.nb = 0; // la liste est initialement vide
11     }
12
13     public int taille() {
14         return this.nb;
15     }
16
17     public double item(int index) {
18         return this.elements[index];
19     }
20
21     public void remplacer(int index, double x) {
22         this.elements[index] = x;
23     }
```

```

24
25     public void ajouter(int index, double x) {
26         if (this.nb >= this.elements.length) { // tableau trop petit !
27             // agrandir le tableau : pourrait être plus efficace !
28             double[] nouveau = new double[this.nb+2]; // 2 arbitraire !
29             System.arraycopy(this.elements, 0, nouveau, 0, this.nb);
30             this.elements = nouveau;
31         }
32         // décaler les éléments à partir de index
33         System.arraycopy(this.elements, index,
34             this.elements, index+1, this.nb-index);
35         // ranger le nouvel élément
36         this.elements[index] = x;
37         this.nb++;
38     }
39
40     public void supprimer(int index) {
41         System.arraycopy(this.elements, index+1,
42             this.elements, index, this.nb-index-1);
43         this.nb--;
44     }
45 }

```

Exemple de programme utilisant les listes

```
1  /** Programme utilisant les listes ListeTab et ListeChaine. */
2  public class TestListes {
3      public static void main(String[] args) {
4          ListeTab l1 = new ListeTab(5);
5          ListeChaine l2 = new ListeChaine();
6          l1.ajouter(0, 3.14);           // quel ajouter ?
7          l2.ajouter(0, 3.14);         // quel ajouter ?
8          double v1 = l1.item(0);
9          double v2 = l2.item(0);
10     }
11 }
```

Remarque : Quel ajouter ? Voir T. 81.

Interfaces et poignées

Question : On ne peut pas créer d'instances d'une interface... Quel intérêt alors ?

Réponse : Une interface permet de déclarer des poignées.

Propriété : Une interface définit un type (par exemple, le type `Liste`).

Intérêt : Écrire des méthodes (et classes) qui s'appuient sur des interfaces sans en connaître les réalisations actuelles... ni futures !

```
1  /** Quelques méthodes utiles sur les listes. */
2  public class OutilsListe {
3      /* Calculer la somme des réels d'une liste.
4       * @param l la liste dont on veut sommer les valeurs
5       * @return la somme des valeurs de l */
6      static public double somme(Liste l) {
7          double resultat = 0;
8          for (int i = 0; i < l.taille(); i++) {
9              resultat += l.item(i);
10         }
11         return resultat;
12     }
13 }
```

Question : Mais que peut bien désigner le paramètre `l` de type `Liste` ?

Sous-type et principe de substitution

Sous-type : Si une classe C réalise une interface I alors le type C est un sous-type du type I.

Substitution : Si un type T1 est un sous-type de T2 alors partout où T2 est déclaré, on peut utiliser un objet de type T1.

Conséquence : Une poignée de type interface peut être initialisée avec un objet instance d'une classe réalisant cette interface.

Attention : L'inverse est faux ! \implies *Toujours* refusé par le compilateur !

```
Liste l1 = new ListeTab(5);      // principe de substitution
Liste l2 = new ListeChaine();   // principe de substitution
ListeTab lt1 = new ListeTab(10);
double s = OutilsListe.somme(lt1); // principe de substitution
// ListeTab lt1 = l1;           // Interdit : mauvais sens !
```

Remarque : Tout comme en français, « liste » est un terme général, abstrait qui désigne soit une liste avec tableau, soit une liste chaînée.

Liaison tardive (ou dynamique)

```
Liste p;           // Déclaration de la poignée p de type liste
p = new ListeTab(9); // Créer une ListeTab et l'attacher à p
p.ajouter(0, 3.14); // Appel de la méthode ajouter(int, double) sur p
```

Type apparent : Type de (déclaration de) la poignée (Liste).

Type réel : Classe de l'objet attaché à la poignée (ListeTab).

Rappel : La classe d'un objet **ne peut pas changer** ! Mais, à une même poignée, on peut attacher des objets de classes différentes.

Principe : Quand une méthode est appliquée sur une poignée :

1. le compilateur vérifie que la méthode est déclarée dans le type apparent de la poignée (sinon erreur de compilation) ;
2. la méthode effectivement exécutée est celle qui est définie sur le type réel de l'objet attaché à la poignée. C'est la *liaison tardive*.

Exercice 21 Si I est une interface qui spécifie la méthode m() et p une poignée non nulle déclarée de type I, est-on sûr que p.m() a un sens ?

Retour sur ListeTab

Exercice 22 : Questions sur ListeTab

Concernant la classe `ListeTab` (T. 143), répondre aux questions suivantes.

22.1 Pourquoi certaines méthodes n'ont pas de commentaires de documentation ?

22.2 Dans la méthode `ajouter`, agrandir le tableau est-il nécessaire ?

22.3 Que se passe-t-il quand la machine virtuelle ne trouve pas assez de mémoire pour agrandir le tableau ?

22.4 Pourrait-on ne pas agrandir le tableau ? Comment faire alors ?

Liste et al. dans la bibliothèque Java

L'interface `Liste` et les classes `ListeTab` et `ListeChaînée` existent dans la bibliothèque Java. Elles s'appellent respectivement :

- `java.util.List` (interface) ;
- `java.util.ArrayList` (classe réalisant `List`) ;
- `java.util.LinkedList` (classe réalisant `List`) ;

Remarque : Il existe également :

- une classe historique `java.util.Vector` proche de `java.util.ArrayList` ;
- et bien d'autres structures de données (voir T. 289).

Responsabilité d'une classe

.

Exercice 23 Le comportement de la liste est décrit informellement dans les commentaires de documentation. Expliquer comment décrire plus formellement ce comportement.

Remarque : Les contrats définis sur une interface (ici `Liste`) s'appliquent à toutes ses réalisations (ici `ListeTab`, `ListeChaînée...`).

Solutions pour formaliser un comportement

Formaliser le comportement, c'est définir les responsabilités des classes/interfaces (voir T. 223).

Essentiellement, deux techniques sont possibles :

- la programmation par contrat (T. 258) ;
- l'utilisation des exceptions (T. 229).

```

1  /** Spécification d'une liste. @version 1.4 */
2  public interface Liste {
3      //@ public invariant taille() >= 0;
4
5      /** L'itérateur qui permet de réaliser un parcours de la liste. */
6      Iterateur iterateur();
7
8      /* Obtenir la taille de la liste.
9       * @return nombre d'éléments dans la liste */
10     /*@ pure @*/ int taille();
11
12     /* Obtenir un élément de la liste.
13     * @param indice position de l'élément */
14     //@ requires 0 <= indice && indice < taille();
15     /*@ pure @*/ double item(int indice);
16
17     /* Remplacer un élément de la liste.
18     * @param indice indice de l'élément à remplacer
19     * @param x nouvelle valeur */
20     //@ requires 0 <= indice && indice < taille(); // indice valide
21     //@ ensures item(indice) == x; // élément remplacé
22     //@ ensures (\forallall int i; // autres éléments inchangés
23     //@ 0 <= i && i < taille();
24     //@ i == indice || item(i) == \old(item(i)));
25     //@ ensures taille() == \old(taille()); // pas d'ajout !
26     void remplacer(int indice, double x);
27
28     /* Ajouter un élément dans la liste.

```

```

29     * @param indice indice où doit se trouver le nouvel élément
30     * @param x élément à insérer */
31     //@ requires 0 <= indice && indice <= taille(); // indice valide
32     //@ ensures item(indice) == x; // élément ajouté
33     //@ ensures (\forallall int i; // éléments avant indice inchangés
34     //@           i >= 0 && i < indice;
35     //@           item(i) == \old(item(i)));
36     //@ ensures (\forallall int i; // éléments après indice décalés
37     //@           i > indice && i < taille();
38     //@           item(i) == \old(item(i-1)));
39     //@ ensures taille() == \old(taille()) + 1; // un élément de moins
40     void ajouter(int indice, double x);
41
42     /* Supprimer un élément de la liste.
43     * @param indice indice de l'élément à supprimer */
44     //@ requires 0 <= indice && indice <= taille(); // indice valide
45     //@ ensures taille() == \old(taille()) - 1; // un élément de moins
46     //@ ensures (\forallall int i; // éléments avant indice inchangés
47     //@           i >= 0 && i < indice;
48     //@           item(i) == \old(item(i)));
49     //@ ensures (\forallall int i; // éléments après indice décalés
50     //@           i >= indice && i < taille();
51     //@           item(i) == \old(item(i+1)));
52     void supprimer(int indice);
53 }

```

Généricité

Exercice 24 : Généraliser les listes

Dans l'exercice 20 nous avons défini la notion de liste de réels avec une interface (`Liste`) et deux réalisations (`ListeTab` et `ListeChaînée`).

Expliquer comment faire pour avoir des listes d'éléments de types différents comme par exemple une liste de fractions, une liste de personnes, une liste de listes de livres, etc.

Solutions

Principe : On constate que seul le type des éléments de la liste change. Les algorithmes (donc le code des méthodes) restent identiques.

Solution 1 (mauvaise) : Faire du copier/coller. Il faut :

- Renommer tous les fichiers (`Liste.java`, `ListeTab.java`...) en prenant un nouveau nom (`ListeDouble`, `ListeTabDouble`, `ListeFraction`... par exemple).
- Remplacer le type `double` par le nouveau type.

Inconvénients : Même si ces transformations peuvent être automatisées elles sont fastidieuses !

Solution 2 (mauvaise) : Celle utilisée avant java 1.5. Voir T. 206.

Solution 3 (la bonne) : Utiliser la généricité (à partir de java 1.5 !)

Version générique de l'interface Liste

```
1  /** Spécification d'une liste. @version 1.4 */
2  public interface Liste<T> {
3      /* Obtenir la taille de la liste.
4       * @return nombre d'éléments dans la liste */
5      int taille();
6
7      /* Obtenir un élément de la liste.
8       * @param indice position de l'élément */
9      T item(int indice);
10
11     /* Remplacer un élément de la liste.
12      * @param indice indice de l'élément à remplacer
13      * @param x nouvelle valeur */
14     void remplacer(int indice, T x);
15
16     /* Ajouter un élément dans la liste.
17      * @param indice indice où doit se trouver le nouvel élément
18      * @param x élément à insérer */
19     void ajouter(int indice, T x);
20
21     /* Supprimer un élément de la liste.
22      * @param indice indice de l'élément à supprimer */
23     void supprimer(int indice);
24 }
```

Le programme de test

```
1  /** Programme utilisant les listes ListeTab et ListeChaine. */
2  public class TestListes {
3      public static void main(String[] args) {
4          ListeTab<Double> l1 = new ListeTab<Double>(5);
5          ListeChaine<Double> l2 = new ListeChaine<Double>();
6          Liste<String> ls = new ListeTab<String>(3);
7          ListeTab<Liste<Double>> ll = new ListeTab<Liste<Double>>(2);
8          l1.ajouter(0, 3.14);           // quel ajouter ?
9          l2.ajouter(0, 3.14);           // quel ajouter ?
10         ls.ajouter(0, "PI");
11         ll.ajouter(0, l1);
12         ll.ajouter(1, l2);
13         double v1 = ll.item(0);
14         double v2 = ll.item(1).item(0);
15         String vs = ls.item(0);
16     }
17 }
```

Remarque : Le type des éléments des listes est précisé entre < >.

Remarque : On utilise `Double` et non `double` car il faut impérativement un objet. L'auto *boxing/unboxing* rend ceci transparent.

Explications

Une classe générique est une classe paramétrée par un (ou plusieurs) types.

```
public class Liste<T> { ... }  
public class Paire<A, B> { ... }
```

Vocabulaire : T, A et B sont dits *variables de type*, *paramètre de type formels* ou *paramètre de généricité*.

Pour obtenir une « classe » (appelée *type paramétré*), il faut préciser les paramètres de généricité (les valeurs des variables de type).

```
Liste<Double>, Liste<Personne>...  
Paire<String, Livre>...
```

Attention : La valeur du paramètre de généricité ne peut pas être un type primitif

⇒ Il faut utiliser les classes enveloppes !

```

1  /** Définition d'une paire. */
2  public class Paire<A, B> {
3      public A premier;
4      public B second;
5
6      public Paire(A premier_, B second_) {
7          this.premier = premier_;
8          this.second = second_;
9      }
10 }

1  import java.awt.Color;
2  public class TesterPaire {
3      public static void main(String[] args) {
4          Paire<String, Integer> p1 = new Paire<String, Integer>("I", 1);
5          Paire<Integer, String> p2 = new Paire<Integer, String>(2, "II");
6          p1.premier = p2.second;
7          p1.second = p2.premier;
8
9          Paire<Color, String> p3 =
10             new Paire<Color, String> (Color.RED, "rouge");
11          p3.second = "VERT";
12          p3.premier = Color.GREEN;
13      }
14 }

```

```

1  /** Définition d'un élément simplement chaînable sur un type T. */
2  public class Cellule<T> {
3      private T element;
4      private Cellule<T> suivante;
5
6      public Cellule(T element_, Cellule<T> suivante_) {
7          this.element = element_;
8          this.suivante = suivante_;
9      }
10
11     public Cellule<T> getSuiivante() {
12         return this.suivante;
13     }
14
15     public T getElement() {
16         return this.element;
17     }
18
19     public void setElement(T nouveau) {
20         this.element = nouveau;
21     }
22
23     public void setSuiivante(Cellule<T> s) {
24         this.suivante = s;
25     }
26 }

```

```

1  /** Réalisation de la Liste en chaînant les éléments. */
2  public class ListeChaine<T> implements Liste<T> {
3      private Cellule<T> premiere;    // première cellule de la liste
4      private int nb = 0; // nb de cellules dans la liste
5
6      public int taille() {
7          return this.nb;
8      }
9
10     /** Obtenir une cellule de la liste chaînée.
11         * @param index position de la cellule (0 pour la première)
12         * @return cellule à la position index */
13     private Cellule<T> cellule(int index) {
14         Cellule<T> curseur = this.premiere;
15         for (int i = 0; i < index; i++) {
16             curseur = curseur.getSuivante();
17         }
18         return curseur;
19     }
20
21     public T item(int index) {
22         return this.cellule(index).getElement();
23     }
24
25     public void remplacer(int index, T x) {

```

```

26     this.cellule(index).setElement(x);
27 }
28
29 public void ajouter(int index, T x) {
30     if (index == 0) {          // insertion en tête
31         this.premiere = new Cellule<T>(x, this.premiere);
32     } else {                  // insertion après une cellule
33         Cellule<T> precedente = cellule(index-1);
34         Cellule<T> nouvelle = new Cellule<T>(x,
35             precedente.getSuivante());
36         precedente.setSuivante(nouvelle);
37     }
38     nb++;    // une cellule ajoutée
39 }
40
41 public void supprimer(int index) {
42     if (index == 0) {        // insertion en tête
43         this.premiere = this.premiere.getSuivante();
44     } else {                // insertion après une cellule
45         Cellule<T> precedente = cellule(index-1);
46         precedente.setSuivante(precedente.getSuivante().getSuivante());
47     }
48     nb--;    // une cellule supprimée
49 }
50 }

```

Exercice 25 Écrire une méthode qui échange deux éléments d'un tableau à des indices donnés.

Elle doit par exemple pouvoir être utilisée pour échanger deux éléments d'un tableau de chaînes de caractères, d'un tableau de points, etc.

Méthodes de classes génériques

```
1 public class GenericiteSwap {
2     /** Permuter deux éléments d'un tableau.
3     * @param tab le tableau
4     * @param i indice du premier élément
5     * @param j indice du deuxième élément */
6     public static <T> void swap(T[] tab, int i, int j) {
7         T tmp = tab[i];
8         tab[i] = tab[j];
9         tab[j] = tmp;
10    }
11
12    public static void main(String[] args) {
13        String[] ts = { "I", "II", "III", "IV", "V" };
14        swap(ts, 0, 1);
15        assert ts[0].equals("II");
16        assert ts[1].equals("I");
17
18        Integer[] ti = { 1, 2, 3, 4, 5 };           // et pas int !!!
19        swap(ti, 0, 1);
20        assert ti[0] == 2;
21        assert ti[1] == 1;
22    }
23 }
```

Exercice 26 Écrire une méthode qui permet d'obtenir le plus grand élément d'un tableau. Cette méthode doit fonctionner quelque soit le type des éléments du tableau.

Généricité contrainte : méthode `max`

```
1 public class GenericiteMax {
2     /** Déterminer le plus grand élément d'un tableau.
3     * @param tab le tableau des éléments
4     * @return le plus grand élément de tab */
5     public static <T extends Comparable<T>> T max(T[] tab) {
6         T resultat = null;
7         for (T x : tab) {
8             if (resultat == null || resultat.compareTo(x) < 0) {
9                 resultat = x;
10            }
11        }
12        return resultat;
13    }
14
15    /** Programme de test de GenericiteMax.max */
16    public static void main(String[] args) {
17        Integer[] ti = { 1, 2, 3, 5, 4 };
18        assert max(ti) == 5;
19
20        String[] ts = { "I", "II", "IV", "V" } ;
21        assert max(ts).equals("V");
22    }
23 }
```

Généricité contrainte : Explications

- Pour pouvoir déterminer le max, il faut comparer les éléments du tableau. Aussi, nous imposons qu'ils soient comparables (`java.lang.Comparable`) :

```
1 public class Comparable<T> {
2     /** Compares this object with the specified object for order.
3     * Returns a negative integer, zero, or a positive integer
4     * as this object is less than, equal to, or greater than
5     * the specified object... */
6     int compareTo(T o);
7 }
```

- Dans le code de la méthode `max`, on peut donc utiliser `compareTo` sur les éléments du tableau.
- `Integer` réalise l'interface `Comparable<Integer>` et `String` réalise `Comparable<String>`. On peut calculer le max des tableaux `ti` et `ts`.
- Autre exemple : `Dictionnaire<Clé extends Hashable, Donnée>` ;

Remarque : Contraintes multiples : `C<T extends I1 & I2>`

Relation d'héritage

Exemple introductif

Exercice 27 : Définition d'un point nommé

Un point nommé est un point caractérisé par une abscisse et une ordonnée et qui possède également un nom. Un point nommé peut être traduit en précisant un déplacement par rapport à l'axe des X (abscisses) et un déplacement suivant l'axe des Y (ordonnées). On peut obtenir sa distance par rapport à un autre point. Il est possible de modifier son abscisse ou son ordonnée. Enfin, ses caractéristiques peuvent être affichées.

27.1 Modéliser en Java cette notion de point nommé.

27.2 Que constatez-vous quand vous comparez le point nommé et le point du transparent suivant ?

La classe Point (1/2)

```
1  /** Modélisation d'un point cartésien */
2  public class Point
3  {
4      private double x;           // abscisse
5      private double y;         // ordonnée
6
7      /** Construction d'un point à partir de ses coordonnées. */
8      public Point(double vx, double vy) { this.x = vx; this.y = vy; }
9
10     /** Abscisse du point. */
11     public double getX()           { return this.x; }
12
13     /** Ordonnée du point. */
14     public double getY()          { return this.y; }
15
16     /** Changer l'abscisse du point. */
17     public void setX(double vx)    { this.x = vx; }
18
19     /** Changer l'ordonnée du point. */
20     public void setY(double vy)    { this.y = vy; }
```

La classe Point (2/2)

```
1  /** Afficher toutes les caractéristiques du point. */
2  public void afficher() {
3      System.out.print("(" + this.getX() + "," + this.getY() + ")");
4  }
5
6  /** Distance par rapport à un autre point. */
7  public double distance(Point autre) {
8      return Math.sqrt(Math.pow(autre.getX() - this.getX(), 2)
9          + Math.pow(autre.getY() - this.getY(), 2));
10 }
11
12 /** Translater le point de dx (axe des X) et de dy (axe des Y). */
13 public void translater(double dx, double dy) {
14     this.x += dx;
15     this.y += dy;
16 }
17 }
```

Héritage et concepts associés

Les points importants sont :

- Héritage : relation de spécialisation/généralisation
- Principe de substitution (induit par l'héritage)
- Adaptations de l'héritage : enrichissement
- Héritage et constructeurs
- Redéfinition et liaison dynamique ;
- Interrogation dynamique de type (affection renversée) ;
- Classes abstraites ;
- Interfaces ;
- Qu'est ce qu'un bon héritage ?
- Héritage vs utilisation.

Héritage

Buts : plusieurs objectifs (non liés) :

- définir une nouvelle classe :
 - comme **spécialisation** d'une classe existante ;
 - comme **généralisation** de classes existantes (factorisation des propriétés et comportement communs) ;
- définir une **relation de sous-typage** entre classes (substitutionnalité) ;
- copier virtuellement les caractéristiques de classes existantes dans la définition d'une nouvelle classe (héritage).

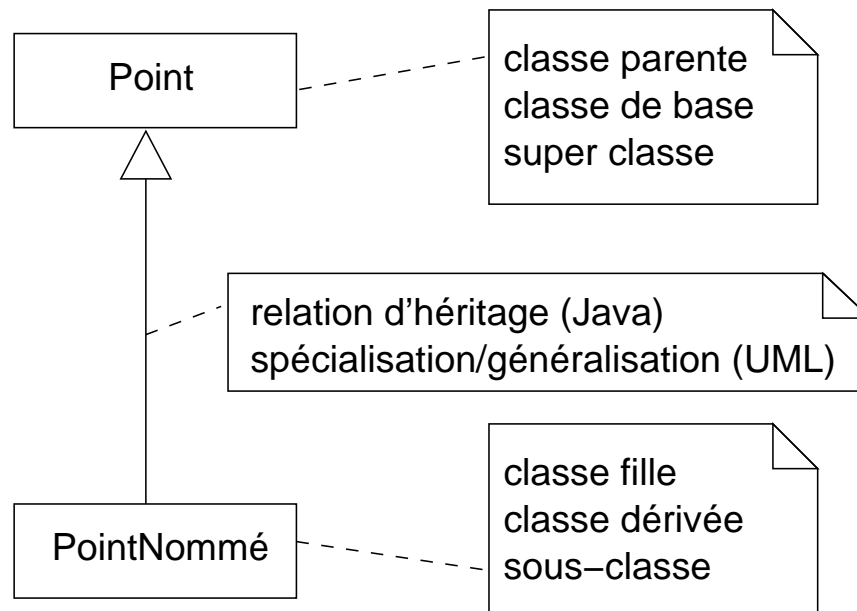
Moyen : La relation d'héritage en Java (appelée relation de généralisation/spécialisation en UML).

Exemples :

- PointNommé est une spécialisation de Point (un point avec un nom).
- Point est une généralisation de PointNommé, PointPondéré, PointColoré...

Notation et vocabulaire

Notation UML



Notation en Java

```
public class PointNommé
    extends Point                // relation d'héritage
{ ... }
```

Vocabulaire : On parle également d'ancêtres et de descendants (transitivité de la relation d'héritage)

Exemple d'héritage : la classe PointNommé (incomplète)

```
/** Un point nommé est un point avec un nom. Ce nom peut être changé. */
public class PointNomme
    extends Point          // héritage (spécialisation d'un point)
{
    private String nom;

    /** Initialiser à partir de son nom et de ses coordonnées cartésiennes */
    public PointNomme(String sonNom, double vx, double vy) {
        super(vx, vy); // appel au constructeur de Point (1ère instruction)
        nom = sonNom;
    }

    /** Le nom du point nommé */
    public String getNom() {
        return nom;
    }

    /** changer le nom du point */
    public void nommer(String nouveauNom) {
        nom = nouveauNom;
    }
}
```

Remarque : L'héritage évite de faire du copier/coller (toujours dangereux).

Attention : Ne pas confondre **extends** et **import** !

Enrichissement

Dans la classe dérivée, on peut ajouter :

- de nouveaux attributs (conseil : prendre de nouveaux noms !);

```
private String nom;
```

- de nouvelles méthodes.

```
public String getNom() { ... }
```

```
public void nommer(String nouveauNom) { ... }
```

Remarque : Ajouter une nouvelle méthode s'entend au sens de la *surcharge*. Par exemple, sur le `PointNommé` on peut ajouter :

```
public void afficher(String préfixe) {  
    System.out.println(préfixe);           // afficher le préfixe  
    afficher();                             // utiliser le afficher << classique >>  
}
```

qui surcharge la méthode `afficher()` de `Point`.

Héritage et constructeurs

Principe : Tout constructeur d'une classe dérivée doit appeler un des constructeurs de la classe de base.

Justification : Hériter d'une classe, c'est récupérer ses caractéristiques qui doivent donc être initialisées. Ds un `PointNommé`, il y a un `Point`... et un nom.

En pratique : On utilise `super` suivi des paramètres effectifs permettant au compilateur de sélectionner le constructeur de la classe parente.

```
public PointNomme(String sonNom, double vx, double vy) {  
    super(vx, vy); // appel au constructeur de Point (1ère instruction)  
    nom = sonNom;  
}
```

- L'appel à `super` doit être la première instruction du constructeur !
 - La classe parente est **toujours initialisée avant** la sous-classe.
 - Si aucun appel à `super` n'est fait, le compilateur appelle automatiquement le constructeur par défaut de la classe parente `super()`.
- ⇒ D'où le danger de définir un constructeur par défaut !

Utilisation de la classe PointNommé

Exemple d'utilisation la classe PointNommé :

```
1 PointNomme pn = new PointNommé("A", 1, 2,);
2           // créer un point "P" de coordonnées (1,2)
3 pn.afficher();           // (1,2) méthode de Point
4 pn.translater(-1, 2);    // méthode de Point
5 pn.afficher();           // (0,4) méthode de Point
6 pn.nommer("B");          // méthode de PointNomme
7 String n = pn.getNom();  // méthode de PointNomme
8 pn.afficher("Le_point_est"); // méthode de PointNomme (surcharge)
9           // Le point est (0, 4)
```

Remarque : La classe PointNommé a bien **hérité** de toutes les caractéristiques de Point.

Exercice 28 Lister toutes les caractéristiques de la classe PointNommé. Préciser les droits d'accès.

Remarque : afficher() ne donne pas le nom du PointNommé.

Héritage et droits d'accès

Il faut distinguer l'accessibilité pour (les méthodes de) la classe dérivée elle-même et l'accessibilité pour les classes utilisant la classe dérivée :

- Une méthode de la classe dérivée a accès aux caractéristiques de la classe de base déclarées **public**, **protected** ou à visibilité de paquetage.

Les primitives **private** sont inaccessibles.

- Les caractéristiques héritées conservent leur droit d'accès.

Remarque : La classe dérivée peut augmenter le droit d'accès des méthodes héritées (une méthode **protected** peut devenir **public**). Il suffit de la redéfinir avec le nouveau droit d'accès.

Attention : Elle ne peut pas la réduire (contrainte du sous-typage).

Remarque : Les caractéristiques qui étaient privées ne sont pas accessibles par les classes dérivées. Leur accessibilité ne peut pas être augmentée !

Redéfinition de méthode

La classe dérivée peut donner une nouvelle **version** (nouveau corps) à une méthode définie dans la classe de base (**adaptation** du comportement).

Exemple : Dans la classe PointNommé, on peut (on doit !) redéfinir « afficher » pour que le nom du point apparaisse également.

```
/** Afficher le point nommé sous la forme nom:(point) */  
public void afficher() {  
    System.out.print(getNom() + ":");  
    super.afficher(); // utilisation du afficher de Point  
}
```

Remarque : Dans le code de la méthode de la classe dérivée, on peut faire référence aux méthodes de la classe parente par l'intermédiaire de **super**.

```
pn.afficher(); // A:(0,4) méthode de PointNommé !
```

Attention : La méthode de la sur-classe n'est pas masquée mais reçoit une nouvelle implantation (les deux correspondent au même envoi de message).

Attention : La redéfinition n'est pas de la surcharge !

Principe de substitution

Principe : L'instance d'un descendant peut être utilisée partout où un ancêtre est déclaré.

Justification intuitive : Tout ce qui peut être demandé à la classe de base peut aussi l'être à la classe dérivée.

Attention : L'inverse est faux. L'instance d'un ancêtre ne peut pas être utilisée où un descendant est déclaré.

```
1 Point p1 = new Point(3, 4);
2 PointNomme pn1 = new PointNomme("A", 30, 40);
3 Point q;           // poignée sur un Point
4 q = p1; q.afficher(); // ??????
5 q = pn1; q.afficher(); // ??????
6
7 PointNomme qn;    // poignée sur un Point
8 qn = p1; qn.afficher(); // ??????
9 qn = pn1; qn.afficher(); // ??????
```

Remarque : Le principe de substitution est vérifié à la compilation.

Résolution d'un appel polymorphe

```
T p;           // Déclaration de la poignée (type apparent : T)
p = new X(...); // Affectation de la poignée (type réel : X)
...
p.m(a1, ..., an); // Appel de la méthode m() sur p
```

1. Résolution de la surcharge (vérification statique).

But : Identification de la signature de la méthode à exécuter.

La classe du type apparent de la poignée (classe T) doit avoir une méthode $m(T_1, \dots, T_n)$ dont les paramètres correspondent en nombre et en type aux paramètres effectifs a_1, \dots, a_n .

Si pas de signature trouvée **Alors** *erreur de compilation !*

Remarque : Le principe de substitution est utilisé sur les paramètres !

2. Liaison dynamique (à l'exécution, généralement).

Le système choisit la **version** de $m(T_1, \dots, T_n)$ à exécuter : c'est la dernière (re)définition rencontrée partant du type T et descendant vers le type réel de l'objet attaché à la poignée p (X).

Illustration de la liaison tardive

```
1 Point p1 = new Point(3, 4); // type type
2 PointNomme pn1 = new PointNomme("A", 30, 40); // apparent réel
3 Point q; // poignée sur un Point Point null
4 q = p1; q.afficher(); // (3,4) Point
5 q = pn1; q.afficher(); // A:(30,40) PN
6
7 PointNomme qn; // poignée sur un Point PN null
8 qn = p1; qn.afficher(); // INTERDIT ! Point
9 qn = pn1; qn.afficher(); // A:(30,40) PN
10
11 qn = q; // Possible ? (Le type réel de q est PointNommé)
12 qn.afficher() // ????
```

Remarque : Le principe est identique à celui vu avec les interfaces (T. 148) !

Interrogation dynamique de type

```
Point p = new PointNomme("A", 1, 2);
PointNomme q;
q = p; // Interdit par le compilateur
```

Le compilateur interdit cet appel car il s'appuie sur les types apparents.

Or, un PointNommé est attaché à p. L'affectation aurait donc un sens.

C'est le pb de l'**affectation renversée** résolu en Java par le « transtypage » :

```
q = (PointNomme) p; // Autorisé par le compilateur
```

Attention : Le transtypage ressemble au cast de C. Cependant cette conversion est vérifiée à l'exécution (exception `ClassCastException`).

Interrogation dynamique de type : opérateur `instanceof`

```
if (p instanceof PointNomme) {
    ((PointNomme) p).nommer("B");
}

if (p instanceof PointNomme) {
    PointNomme q = (PointNomme) p;
    q.nommer("B");
}
```

Liaison tardive : intérêt

Intérêt : Éviter des structures de type choix multiples :

- Les alternatives sont traitées par le compilateur (sûreté : pas d'oubli).
- L'ajout d'une nouvelle alternative est facilité (extensibilité).

En l'absence de liaison tardive, il faudrait tester explicitement le type de l'objet associé à une poignée pour choisir quelle méthode lui appliquer :

```
Point q = ...; // Afficher les caractéristiques de q
if (q instanceof PointNomme)           // tester le type réel
    // afficher q comme un PointNomme
else if (q instanceof PointColoré )    // tester le type réel
    // afficher q comme un PointColoré
    ...
else                                     // Ce n'est donc qu'un point !
    // afficher q comme un Point
```

Remarque : Permet de réaliser le **principe du choix unique** (B. Meyer) :

« Chaque fois qu'un système logiciel doit prendre en compte un ensemble d'alternatives, un et un seul module du système doit en connaître la liste exhaustive ».

Le modifieur final

Définition : Le modifieur **final** donne le sens d'immuable, de non modifiable.

Il est utilisé :

- pour une *variable locale* : c'est une constante (qui doit donc nécessairement être initialisée lors de sa déclaration) ;
- un *attribut d'instance ou de classe* (qui doit être nécessairement initialisé par une valeur par défaut) ;
- une *méthode* : la méthode ne peut pas être redéfinie par une classe dérivée. Elle n'est donc pas polymorphe ;
- une *classe* : la classe ne peut pas être spécialisée. Elle n'aura donc aucun descendant (aucune de ses méthodes n'est donc polymorphe).

La classe Object

En Java, si une classe n'a pas de classe parente, elle hérite implicitement de la classe Object. C'est l'**ancêtre commun** à toutes les classes.

Elle contient en particulier les méthodes :

- **public boolean** equals(Object obj); Égalité de **this** et obj (par défaut égalité des adresses). Elle a cependant le sens d'égalité logique et doit donc être redéfinie (String, etc.)
- **public String** toString(); chaîne de caractères décrivant l'objet. Elle est utilisée dans print, println et l'opérateur de concaténation + par l'intermédiaire de String.valueOf(Object).
- **protected void** finalize(); Méthode appelée lorsque le ramasse-miettes récupère la mémoire d'un objet.

...

Voir le transparent T. 302 pour d'autres caractéristiques de la classe Object.

Classe abstraite : exemple introductif

Exercice 29 Étant données les classes Point, PointNommé, Segment, Cercle... on souhaite formaliser la notion de dessin.

29.1 Comment peut-on modéliser cette notion de dessin en Java ?

29.2 Comment faire pour afficher ou translater un tel dessin ?

29.3 Comment faire si on veut ajouter un Polygone ?

29.4 Comment faire si on veut pouvoir agrandir (effet zoom) le dessin ?

Méthode retardée

Définition : Une méthode retardée (ou abstraite) est une méthode dont on ne sait pas écrire le code. Cette méthode est notée **abstract** (modificateur).

Exemple : Un objet géométrique peut être affiché et déplacé mais si on ne connaît pas le type d'objet on ne sait pas écrire le code de ces méthodes

```
/** Afficher les caractéristiques de l'objet géométrique. */  
abstract public void afficher();
```

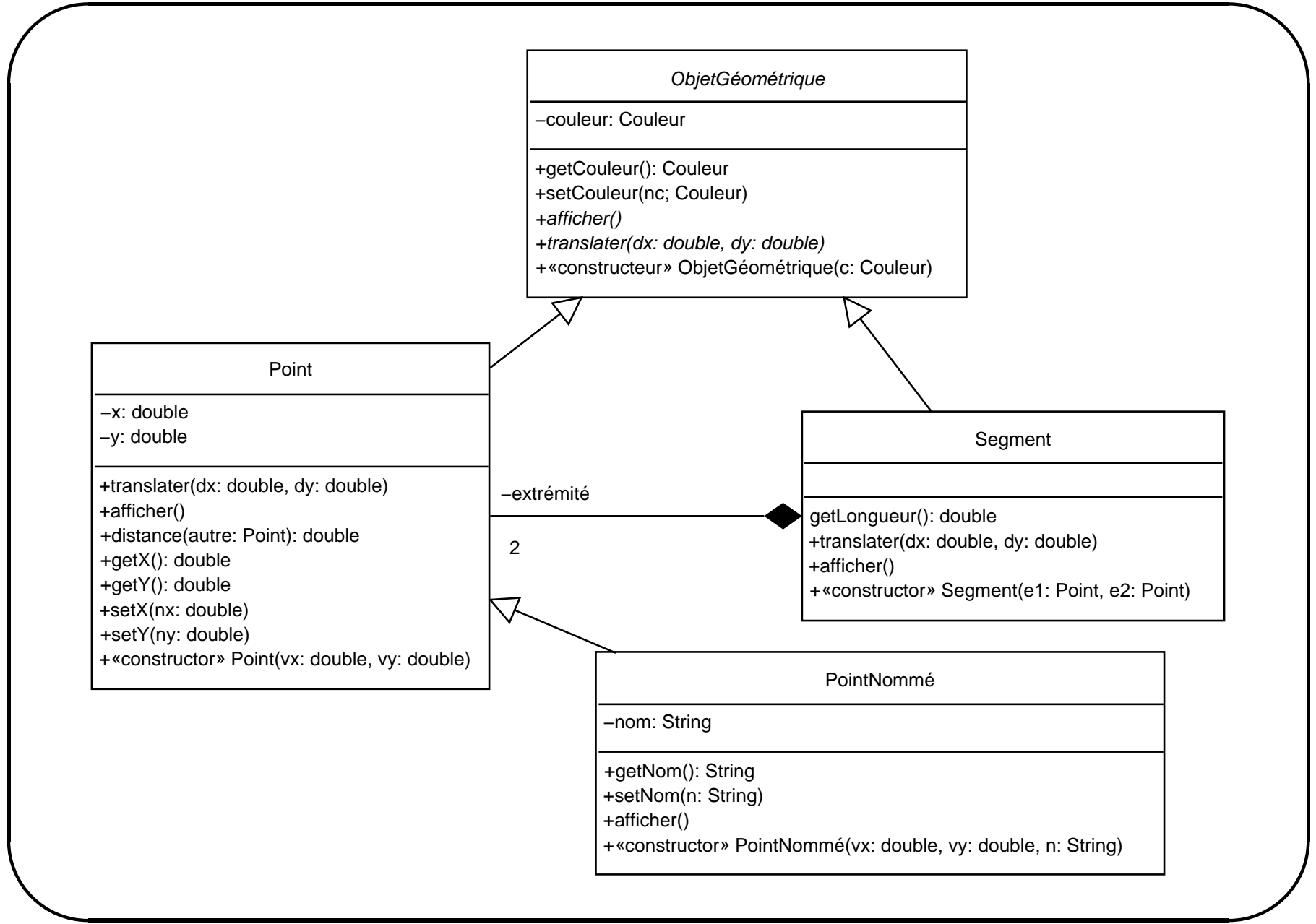
```
/** Translater l'objet.  
* @param dx déplacement suivant l'axe des X  
* @param dy déplacement suivant l'axe des Y */  
abstract public void translater(double dx, double dy);
```

Remarque : On ne met pas d'accolades après une méthode retardée !

Attention : **abstract** est incompatible avec **final** ou **static** : une méthode retardée est nécessairement une méthode d'instance polymorphe !

Classe abstraite : la classe **ObjetGéométrique**

```
1  /** Modélisation de la notion de objet géométrique. */
2  abstract public class ObjetGéométrique {
3      private java.awt.Color couleur;    // couleur du point
4
5      /** Construire une forme.
6          * @param c la couleur de la forme */
7      public ObjetGéométrique(java.awt.Color c)    { this.couleur = c; }
8
9      /** Obtenir la couleur de la forme.
10         * @return la couleur de la forme */
11     public java.awt.Color getCouleur()            { return this.couleur; }
12
13     /** Changer la couleur du point.
14         * @param c nouvelle couleur */
15     public void setCouleur(java.awt.Color c)    { this.couleur = c; }
16
17     /** Afficher sur le terminal les caractéristiques de la forme. */
18     abstract public void afficher();
19
20     /** Translater la forme.
21         * @param dx déplacement en X
22         * @param dy déplacement en Y */
23     abstract public void translater(double dx, double dy);
24 }
```



Commentaires sur la notation UML

- Essayer de placer les classes parentes au-dessus des sous-classes.
- Ne pas confondre la relation de généralisation/spécialisation avec une relation d'association avec contrainte de navigation !
- Le nom des classes abstraites et des méthodes retardées est noté en italique... Ce qui n'est pas toujours très visible !

Remarque : On peut également utiliser la contrainte {abstract}.

- Dans une sous-classe UML, on ne fait apparaître que les méthodes qui sont définies ou redéfinies dans la sous-classe.

Exemples : Point définit afficher et translater qui étaient spécifiées dans ObjetGéométrique.

Dans PointNommé on ne fait apparaître ni translater ni distance car ce sont celles de Point. En revanche, afficher est redéfinie.

- La relation entre Segment et Point est une relation de composition.

Comment la réaliser en Java ?

Classe abstraite

Défi nition : Une classe qui possède une méthode retardée (propre ou héritée) est nécessairement une classe abstraite.

Conséquence : Une classe abstraite ne peut pas être instanciée. Elle devra donc avoir des descendants qui définissent les méthodes retardées.

Remarque : En Java, on peut déclarer une classe abstraite même si elle ne possède pas de méthodes retardées.

Constructeurs : Une classe abstraite peut avoir des constructeurs et il est recommandé d'en définir (s'ils ont un sens).

Exercice 30 Quel est l'intérêt d'utiliser une méthode retardée plutôt que de définir une méthode avec un code vide (ou affichant un message d'erreur) qui serait redéfinie dans les sous-classes ?

Intérêt des classes abstraites

- **Factoriser**, même si on n'est pas capable de le coder, le comportement de plusieurs classes. Ce comportement peut alors être utilisé.

Exercice 31 Définir une classe Groupe (d'objets géométriques). Groupe est-elle abstraite ? Peut-on mettre un Groupe dans un Groupe ?

- **Classifier** les objets, par exemple les objets géométriques généraux, fermés, ouverts, etc.
- Permettre au compilateur de **vérifier** que la définition des méthodes retardées héritées est bien donnée dans les sous-classes.

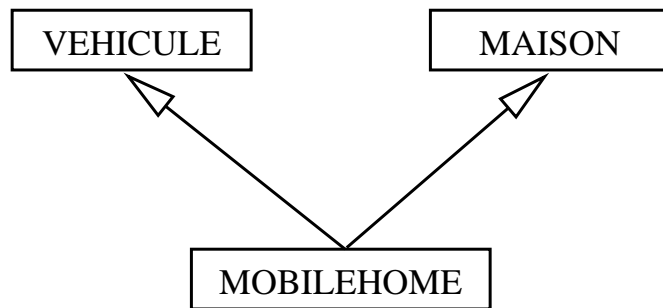
Remarque : Une sous-classe d'une classe abstraite peut être concrète ou abstraite suivant qu'elle donne ou non une définition à toutes ses méthodes (propres et héritées).

Exercice 32 Définir un menu textuel.

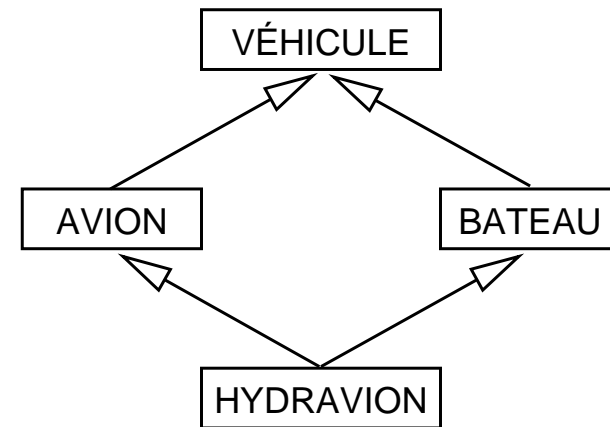
Héritage multiple

Définition : On dit qu'il a héritage multiple si une classe hérite d'au moins deux classes (parentes).

Exemple : Un mobilehome et un hydravion.



héritage multiple



héritage répété

Problèmes posés : Que deviennent les attributs et méthodes présents dans les deux classes parentes ? Représentent-ils la même notion (fusion) ou des notions différentes (duplication).

Solution proposée par Java : Interdire l'héritage multiple et introduire la notion d'interface.

Exercice 33

33.1 Comment faire pour écrire une méthode de tri d'un tableau qui soit générale ?

33.2 Comment faire pour écrire une méthode permettant de dessiner sur un écran graphique tous les éléments d'un tableau ?

33.3 Comment faire pour qu'un objet, un point par exemple, puisse apparaître à la fois comme élément du premier tableau et du deuxième tableau ?

Interfaces et héritage

Héritage multiple : Une interface peut spécialiser plusieurs interfaces.

```
interface I extends I1, I2 { ... }
```

Attention : Deux méthodes ayant même nom et même signature sont considérées comme la même méthode.

⇒ C'est une résolution syntaxique du pb de l'héritage multiple :-)

```
public interface Affichable {  
    /** Afficher avec un décalage de indentation espaces. */  
    void afficher(int indentation);  
}  
  
public interface MultiAffichable {  
    /** Afficher plusieurs fois. */  
    void afficher(int nb);  
}  
  
public class PbHeritageInterface implements Affichable, MultiAffichable {  
    public void afficher(int entier) { // Que signifie afficher ?  
        System.out.println("Entier_=_ " + entier);  
    }  
}
```

Interface vs Classe abstraite

Les classes abstraites ont pour avantage de pouvoir contenir un comportement partiellement défini (des attributs et le code de méthodes).

```
/** Comparer des objets avec une relation d'ordre total */
public abstract class OrdreTotal<T> {
    /** Strictement inférieur (lesser than) */
    public abstract boolean lt(T a);

    /** Supérieur ou égal (greater equal) */
    final public boolean ge(T a)          { return !this.lt(a); }

    /** Strictement supérieur (greater than) */
    final public boolean gt(T a)         { return a.lt(this); }

    /** Inférieur ou égal (lesser equal) */
    final public boolean le(T a)         { return !a.lt(this); }

    final public boolean equals(T a)
                                   { return !this.lt(a) && !a.lt(this); }
}
```

Attention : La liaison tardive fait que $a < b \equiv b > a$ peut être faux.

Interface vs Classe abstraite (suite)

La classe `OrdreTotal` permet de définir, à partir d'une méthode de comparaison, les autres méthodes usuelles. Malheureusement, si une classe hérite de `OrdreTotal`, elle ne pourra pas hériter d'une autre classe.

Pour transformer la classe abstraite en interface, il faut supprimer le code de toutes les méthodes. Il est alors préférable de les supprimer directement.

On arrive ainsi à l'interface `Comparable` de l'API Java :

```
interface Comparable {  
    /** Comparer cet objet avec l'objet spécifié. Le résultat est négatif,  
     * nul ou positif suivant que this est <, = ou > à l'objet spécifié */  
    public int compareTo(Object o);  
}
```

Attention : La documentation de `Comparable` recommande que `compareTo` soit cohérent avec `equals` mais ceci ne peut pas être forcé comme dans le cas de `OrdreTotal`.

Contraintes sur la (re)définition

Respect de la sémantique : La redéfinition d'une méthode doit préserver la sémantique de la version précédente : la nouvelle version doit fonctionner au moins dans les mêmes cas et faire au moins ce qui était fait (cf T. 272).

Preuve : Une méthode $f(B\ b, \dots)$ travaille sur une classe polymorphe B .

- Cette classe B contient au moins une méthode polymorphe g .
- L'auteur de f ne connaît que B (*a priori*). Il utilise donc la spécification de B pour savoir comment appeler g et ce qu'elle fait.
- En fait, la méthode f est appelée avec un objet de la classe A dérivée de B (principe de substitution) et redéfinissant g .
- En raison de la liaison tardive, c'est donc la version de g de la classe dérivée A qui est appelée.

Conclusion : la version de g dans A doit fonctionner dans les cas prévus dans la classe de base B et faire au moins ce qui était prévu dans B .

Test : Une classe dérivée doit réussir les tests de ses classes parentes.

Constructeur et méthode polymorphe

Règle : Un constructeur ne devrait pas appeler de méthode polymorphe.

Preuve : Considérons une classe A dont l'un de ses constructeurs utilise une méthode polymorphe m.

- Puisque m est une méthode est polymorphe, elle peut être redéfinie par une sous-classe B.
- La redéfinition de m dans B peut utiliser un attribut de type objet attr ajouté dans B (donc non présent dans A).
- L'ordre d'initialisation des constructeurs fait que le constructeur de A est exécutée avant celui de B, donc attr est **null**. Or le constructeur de A exécute m, donc en fait la version de B (liaison tardive) qui utilise attr non encore initialisé !

Réutilisation

La réutilisation se fait en Java au travers des classes.

Il s'agit de réutiliser des classes déjà écrites.

Il y a deux possibilités pour qu'une classe A « réutilise » une classe B :

– la *relation d'utilisation* (association, agrégation ou composition) :

```
class A {  
    B b;          // poignée sur un objet de B  
    ...  
}
```

Exemple : La classe segment (ré)utilise la classe Point.

– la *relation d'héritage* (spécialisation) :

```
class A extends B {    // A spécialise B  
    ...  
}
```

Exemple : La classe PointNommé (ré)utilise la classe Point.

La question est alors : « Quoi choisir entre utilisation et héritage ? »

Choisir entre héritage et utilisation

Règles simples :

- « a » \implies association (ou agrégation)
- « est composé de » \implies composition (ou agrégation)
- « est-un » \implies héritage

Attention : « est-un ... et ... » (utilisation) \neq « est-un ... ou ... » (héritage)

Remarque : ÊTRE, c'est AVOIR un peu !

On peut *toujours* remplacer l'héritage par l'utilisation.

L'inverse est faux. AVOIR, ce n'est pas toujours ÊTRE !

Deux règles :

- si on veut utiliser le polymorphisme \implies héritage
- si on veut changer de type dynamique \implies utilisation (poignée)

Exercice 34 Comment modéliser une équipe de football ?

Généricité vs Héritage

Exercice 35

35.1 Quelles sont les relations entre un ensemble de points, une liste de points et une liste chaînée de points ?

35.2 Quelles sont les relations entre une liste de points, une liste de personnes, une liste de livres, etc.

35.3 Définir une pile de points (de capacité fixe).

35.4 Comment adapter la pile de points pour avoir une pile de livres ou une pile de personnes ?

Généricité vs Héritage : schéma

Vocabulaire : La généricité s'appelle également *polymorphisme paramétrique* : transformer un type en paramètre.

Le polymorphisme de la programmation par objet s'appelle le *polymorphisme d'héritage* ou *sous-typage*.

Généricité vs Héritage : une pile fixe de points

```
/** Une pile de points
 * de capacité fixe. */
public class PileFixePoint {
    private Point[] éléments;
    private int nb;

    public PileFixePoint(int capacité) {
        éléments = new Point[capacité];
        nb = 0;
    }

    public boolean estVide() {
        return nb == 0;
    }

    public Point getSommet() {
        return éléments[nb-1];
    }

    public void empiler(Point elt) {
        éléments[nb++] = elt;
    }

    public void dépiler() {
        nb--;
    }
}
```

Attention : On ne fait aucun contrôle sur la validité des opérations (empiler, dépiler, sommet) : voir chapitre *Responsabilités d'une classe* (T. 223).

Généricité vs Héritage : utilisation de PileFixePoint

```
class TestPileFixePoint {
    public static void main (String args []) {
        PileFixePoint pile = new PileFixePoint(10);

        for (int i = 0; i < 10; i++) {
            pile.empiler(new Point(i, i));
        }

        while (!pile.estVide()) {
            pile.getSommet().afficher();
            pile.dépiler();
        }
    }
}
```

- Quelles sont les vérifications que peut faire le compilateur ?
- Quelles sont les vérifications qui ne peuvent être réalisées qu'à l'exécution ?

Une pile fixe générale : polymorphisme d'héritage

```
/** Une pile de capacité fixe générale */
 * polymorphisme d'héritage */
public class PileFixeObject {
    private Object[] éléments;
    private int nb;

    public PileFixeObject(int capacité) {
        éléments = new Object[capacité];
        nb = 0;
    }

    public boolean estVide() {
        return nb == 0;
    }

    public Object getSommet() {
        return éléments[nb-1];
    }

    public void empiler(Object elt) {
        éléments[nb++] = elt;
    }

    public void dépiler() {
        nb--;
    }
}
```

Attention : On ne fait aucun contrôle sur la validité des opérations (empiler, dépiler, sommet) : voir chapitre *Responsabilités d'une classe* (T. 223).

Généricité vs Héritage : utilisation de PileFixeObject

```
class TestPileFixeObject {
    public static void main (String args []) {
        PileFixeObject pile = new PileFixeObject(10);

        for (int i = 0; i < 10; i++) {
            pile.empiler(new Point(i, i));
        }

        while (!pile.estVide()) {
            ((Point)pile.getSommet()).afficher();
            pile.dépiler();
        }
    }
}
```

- Quelles sont les vérifications que peut faire le compilateur ?
- Quelles sont les vérifications qui ne peuvent être réalisées qu'à l'exécution ?

Une pile fixe générale : généricité

```
/** Une pile de capacité fixe :  
 * généricité */  
public class PileFixe<G> {  
    // G : paramètre générique  
  
    private G[] éléments;  
    private int nb;  
  
    public PileFixe(int capacité) {  
        éléments = new G[capacité];  
        nb = 0;  
    }  
  
    public boolean estVide() {  
        return nb == 0;  
    }  
  
    public G getSommet() {  
        return éléments[nb-1];  
    }  
  
    public void empiler(G elt) {  
        éléments[nb++] = elt;  
    }  
  
    public void dépiler() {  
        nb--;  
    }  
}
```

Attention : On ne fait aucun contrôle sur la validité des opérations (empiler, dépiler, sommet) : voir chapitre *Responsabilités d'une classe* (T. 223).

Généricité vs Héritage : utilisation de PileFixe

```
class TestPileFixe {
    public static void main (String args []) {
        PileFixe<Point> pile = new PileFixe<Point>(10);
        // instantiation du paramètre générique

        for (int i = 0; i < 10; i++) {
            pile.empiler(new Point(i, i));
        }

        while (!pile.estVide()) {
            pile.getSommet().afficher();
            pile.dépiler();
        }
    }
}
```

- Quelles sont les vérifications que peut faire le compilateur ?
- Quelles sont les vérifications qui ne peuvent être réalisées qu'à l'exécution ?

Généricité vs Héritage : conclusions

Utiliser l'héritage pour « simuler » la généricité, c'est masquer les types apparents par un type plus général (Object).

Ceci comporte plusieurs inconvénients :

- Le compilateur ne peut pas faire de contrôle (utile) de types. Il est possible de mettre n'importe quel type d'objet dans la pile !
- Le programmeur doit faire du transtypage quand il récupère les objets (contrôlé à l'exécution seulement, donc trop tard !).
- Les informations de type ne sont pas apparentes dans le source et doivent être remplacées par des commentaires : comment déclarer une pile de points ? Et une pile d'objets géométriques ?

Conclusion : L'héritage permet de simuler la généricité mais au détriment des contrôles qui peuvent être faits par le compilateur. Le programmeur doit être plus vigilant. Heureusement, Java 1.5 apporte la généricité !

Que faire si on ne dispose pas de la généricité ?

On peut allier polymorphisme d'héritage *et* contrôle de type en écrivant autant de classes « enveloppes » que d'instanciations possibles.

```
/** Une pile fixe de points                                }
 * en utilisant PileFixeObject
 * mais avec contrôle de type. */
public class PileFixePoint {
    private PileFixeObject points;

    public PileFixePoint(int capa) {
        points =
            new PileFixeObject(capa);
    }

    public boolean estVide() {
        return points.estVide();
    }

    public Point getSommet() {
        return (Point) points.getSommet();
    }

    public void empiler(Point elt) {
        points.empiler(elt);
    }

    public void dépiler() {
        points.dépiler();
    }
}
```

Remarque : C'est ce qu'engendre *automatiquement* le compilateur de java 1.5 (avec l'option `-source 1.5`) !

Conséquences

- D'un point de vue pratique, il n'existe qu'une seule classe, par exemple `Pile` et non `Pile<String>`, `Pile<Point>`... (différent de C++).
Si `G` est une classe Générique et `A` et `B` deux types, `G<A>` et `G` ont même classe `G`. L'information de généricité a disparu.
- Le compilateur engendre donc les transtypages nécessaires.
- Les informations de types sont uniquement connues du compilateur et non de la JVM. Dans le byte code, les informations de généricité ont disparu !
- On ne peut pas utiliser `instanceof` ni faire du transtypage avec type générique.

```

1  public class TestErreurTranstypage {
2      public static void main(String[] args) {
3          Paire<String, Integer> pp = new Paire<String, Integer>("I", 1);
4          Paire po = pp;
5          Object o = pp;
6
7          Paire<String, Integer> p2 = null;
8          p2 = po;                // unchecked conversion
9          p2 = (Paire) po;        // unchecked conversion
10         p2 = (Paire<String, Integer>) po; // unchecked cast
11
12         p2 = o;                // incompatible types
13         p2 = (Paire) o;        // unchecked conversion
14         p2 = (Paire<String, Integer>) o; // unchecked cast
15
16         Paire<String, Object> ps = (Paire<String, Object>) pp;
17             // inconvertible types
18             // Ce n'est pas vraiment du transtypage
19             // puisqu'il est interdit par le compilateur
20
21         assert po instanceof Paire<String, Integer>;
22             // llegal generic type for instanceof
23         assert po instanceof Paire;
24     }
25 }

```

Généricité et sous-typage

Exercice 36 Y a-t-il une relation de sous-typage entre `ListeTab<Object>` et `ListeTab<String>` ? Dans quel sens ?

Généricité et sous-typage

```
1 public class TestErreurGenericiteSousTypage {
2     public static void main(String[] args) {
3         ListeTab<String> ls = new ListeTab<String> (10);
4         ListeTab<Object> lo = ls;
5         lo.ajouter(0, "texte");
6         lo.ajouter(0, 15.5);    // en fait new Double(15.5);
7         String s = ls.item(0);
8         System.out.println(s);
9     }
10 }
```

```
TestErreurGenericiteSousTypage.java:4: incompatible types
found   : ListeTab<java.lang.String>
required: ListeTab<java.lang.Object>
    ListeTab<Object> lo = ls;
                        ^
```

Note: ./ListeTab.java uses unchecked or unsafe operations.

Note: Recompile with `-Xlint:unchecked` for details.

Remarque : Il y aurait une relation de sous-typage si ListeTab était immuable !

Exercice 37 On considère la méthode `afficher` et le programme de test ci-dessous.

```
/** Afficher tous les éléments de ll. */
static public void afficher(Liste<Object> ll) {
    for (int i = 0; i < ll.taille(); i++) {
        System.out.println(ll.item(i));
    }
}

public static void main(String[] args) {
    Liste<Object> lo = new ListeTab<Object> (5);
    lo.ajouter(0, "deux");
    lo.ajouter(0, "un");
    afficher(lo);

    Liste<String> ls = new ListeTab<String> (5);
    ls.ajouter(0, "deux");
    ls.ajouter(0, "un");
    afficher(ls);
}
```

37.1 Que donne la compilation et l'exécution de ce programme ?

37.2 Proposer une nouvelle version de la méthode `afficher`.

Utilisation du type « *wildcard* »

- Erreur de compilation : `Liste<String>` n'est pas un sous-type de `Liste<Object>` ! Voir T. 218.
- Une solution consisterait à utiliser une méthode générique :

```
static public <T> void afficher(Liste<T> ll) {  
    for (int i = 0; i < ll.taille(); i++) {  
        System.out.println(ll.item(i));  
    }  
}
```

- Une meilleure solution consiste à d'utiliser un type « wildcard » : `<?>` :

```
static void afficher(Liste<?> ll) {  
    for (int i = 0; i < ll.taille(); i++) {  
        System.out.println(ll.item(i));  
    }  
}
```

`Liste<?>` est appelée « liste d'inconnus ». Le type n'est pas connu !

Cette solution est meilleure car `T` n'était jamais utilisé (ne servait à rien).

Rq : On peut utiliser des types « *wildcard* » contraints (`<? extends Type>`).

Limite des *wildcards*

```
/** Copier les éléments de tab dans la liste l. */  
public static void copier(Object[] tab, Liste<?> l) {  
    for (Object o : tab) {  
        l.ajouter(l.size(), o);  
    }  
}
```

- Le compilateur interdit d'ajouter un objet dans une liste d'inconnus !
- La solution consiste à prendre un paramètre de généricité :

```
public static <T> void copier(T[] tab, Liste<T> l) {  
    for (T o : tab) {  
        l.ajouter(l.size(), o);  
    }  
}
```

- Que donne alors les instructions suivantes ?

```
String[] tab = { "un", "deux" };  
copier(tab, new ListeTab<Object> (2));
```

Exercice 38 Écrire une méthode de classe qui copie une liste dans une autre.

Responsabilités d'une classe

Motivation : Un problème important dans le développement de logiciels est de savoir qui est responsable de quoi : **défi nition des responsabilités.**

C'est le problème que nous abordons ici.

Plan de la partie :

- Exemple introductif : les fractions
- Responsabilités de la classe Fraction
- Principe de protection en écriture des attributs
- Programmation défensive et exceptions
- Programmation par contrat
- Conclusions

Exemple : les fractions

Soit une classe Fraction définie par son numérateur et son dénominateur :

```
public class Fraction {  
    public int numérateur;  
    public int dénominateur;  
    public Fraction(int num, int dén) { ... }  
    public void normaliser() { ... }  
    ...  
}
```

- Est-il utile de normaliser la représentation d'une fraction ? Pourquoi ?
- Qui doit utiliser normaliser ?
- Peut-on créer une Fraction(-4, -12) ?
- Peut-on toujours calculer l'inverse d'une Fraction ? Comment est-on averti si une opération n'est pas possible ?
- Quels sont les avantages/inconvénients de pouvoir accéder aux attributs numérateur et dénominateur ?

Responsabilités de la classe Fraction

Définition : Les responsabilités d'une classe décrivent ce à quoi elle s'engage au moyen :

- d'**invariants** : ils lient les requêtes (indirectement l'état interne de l'objet). Un invariant doit toujours être vérifié par tous les objets ;
- d'**obligations** : elles décrivent les méthodes de la classe.

Invariant de la classe Fraction : Une fraction est toujours normalisée.

- Le dénominateur est strictement positif.
- Le numérateur et le dénominateur sont réduits.
- La fraction nulle est représentée par 0/1.

Obligations de la classe Fraction :

- L'inverse n'a de sens que pour une fraction non nulle.
- etc.

Principe de protection en écriture des attributs

Principe : Les attributs ne doivent pas être publics.

Justification : Si une classe donne accès en écriture à un attribut, elle ne peut plus garantir la cohérence de ses objets (invariants).

Conséquence : Définir des méthodes d'accès et des méthodes d'altération. Les méthodes d'altération contrôlent la cohérence de la modification.

En Java :

- Déclarer les attributs **private** ;
- Définir une méthode d'accès (si l'attribut fait partie des requêtes) ;
- Définir une méthode de modification (si dans les commandes).

Remarque : Pour des raisons de simplicité, nous ne définirons les modifieurs et accesseurs que dans le cas où ils sont **public**.

Fraction et principe de protection en écriture des attributs

```
public class Fraction {
    private int numérateur;
    private int dénominateur;
    public Fraction(int num, int dén)    { set(num, dén); }
    public int getNumérateur()          { return numérateur; }
    public int getDénominateur()        { return dénominateur; }
    public void set(int n, int d)        { ... }
    public void set(int n)                { set(n, 1); }
    ...
}
```

Remarques :

- Les deux attributs sont **private** ;
- Chaque attribut possède son accesseur ;
- Les modificateurs individuels des attributs ne sont pas définis. Ils auraient pu être définies comme **protected**.

Notation : La convention Java est de préfixer le nom de l'attribut par « get » et le modifieur par « set ».

Programmation défensive

Principe : Les cas anormaux sont testés dans le sous-programme.

Traitements possibles :

- signaler un message d'erreur et continuer l'exécution ;
- signaler un message d'erreur et arrêter l'exécution ;
- renvoyer un code d'erreur ;
- particulariser une valeur de retour pour indiquer l'erreur ;
- réaliser un traitement par défaut (renvoyer une valeur valide) ;
- appeler une fonction fournie par l'appelant ;

Questions :

- Qui détecte l'erreur (ou le problème) ?
- Qui est capable de le résoudre ?

Meilleure solution : Signaler l'erreur en levant une **exception**.

Exceptions

- Exemple introductif
- Intérêt des exceptions
- Classification des exceptions en Java
- Mécanisme des exceptions
- Les exceptions utilisateur
- Exemples d'utilisation des exceptions
- Une exception est une classe
- Conseils sur l'utilisation des exceptions

Exceptions : Exemple démonstratif

```
1  import java.io.*;           // io pour input/output
2  public class Moyenne {
3      /** Afficher la moyenne des valeurs réelles du fichier args[0] */
4      public static void main (String args []) {
5          try {
6              BufferedReader in = new BufferedReader(new FileReader(args[0]));
7              int somme = 0, nb = 0;    // somme et nb de valeurs lues
8              String ligne;           // une ligne du fichier
9              while ((ligne = in.readLine()) != null) {
10                 somme += Double.parseDouble(ligne);
11                 nb++;
12             }
13             in.close();
14             System.out.println("Moyenne_=_=" + (somme / nb));
15         } catch (IOException e) {
16             System.out.println("Problème_d'E/S_:_=" + e);
17         } catch (NumberFormatException e) {
18             System.out.println("Une_donnée_non_numérique_:_=" + e);
19         }
20     }
21 }
```

Exceptions : Exemples d'utilisation de Moyenne

```
> java Moyenne Données1          # execution nominale (10 15 20)
Moyenne = 15

> java Moyenne Données2          # (10 15 20 quinze)
Une donnée non numérique : java.lang.NumberFormatException: quinze

> java Moyenne Données3          # pas de valeurs dans le fichier
Exception in thread "main"java.lang.ArithmeticException: / by zero
    at Moyenne.main(Moyenne.java:14)

> java Moyenne Données4          # fichier contenant des lignes vides
Une donnée non numérique : java.lang.NumberFormatException: empty String

> java Moyenne Données5          # fichier inexistant
Problème d'E/S : java.io.FileNotFoundException: Données5 (Aucun fichier
ou répertoire de ce type)

> java Moyenne                    # pas de fichier en argument
Exception in thread "main"java.lang.ArrayIndexOutOfBoundsException
    at Moyenne.main(Moyenne.java:6)
```

Exceptions : Intérêts

Les exceptions permettent :

- de transférer le flôt de contrôle de l’instruction qui lève l’exception (qui détecte l’anomalie) vers la partie du programme capable de la traiter ;
- d’éviter de surcharger le code d’une méthode avec de nombreux tests concernant des cas anormaux ;
- de regrouper le traitement des cas anormaux et erreurs ;
- de différencier les anomalies (différents types d’exception) ;

Remarque : Les exceptions peuvent être considérées comme des « goto » disciplinés.

Attention : Il ne faut pas abuser des exceptions et les réserver aux cas réellement anormaux ou d’erreurs.

Exceptions : Principe

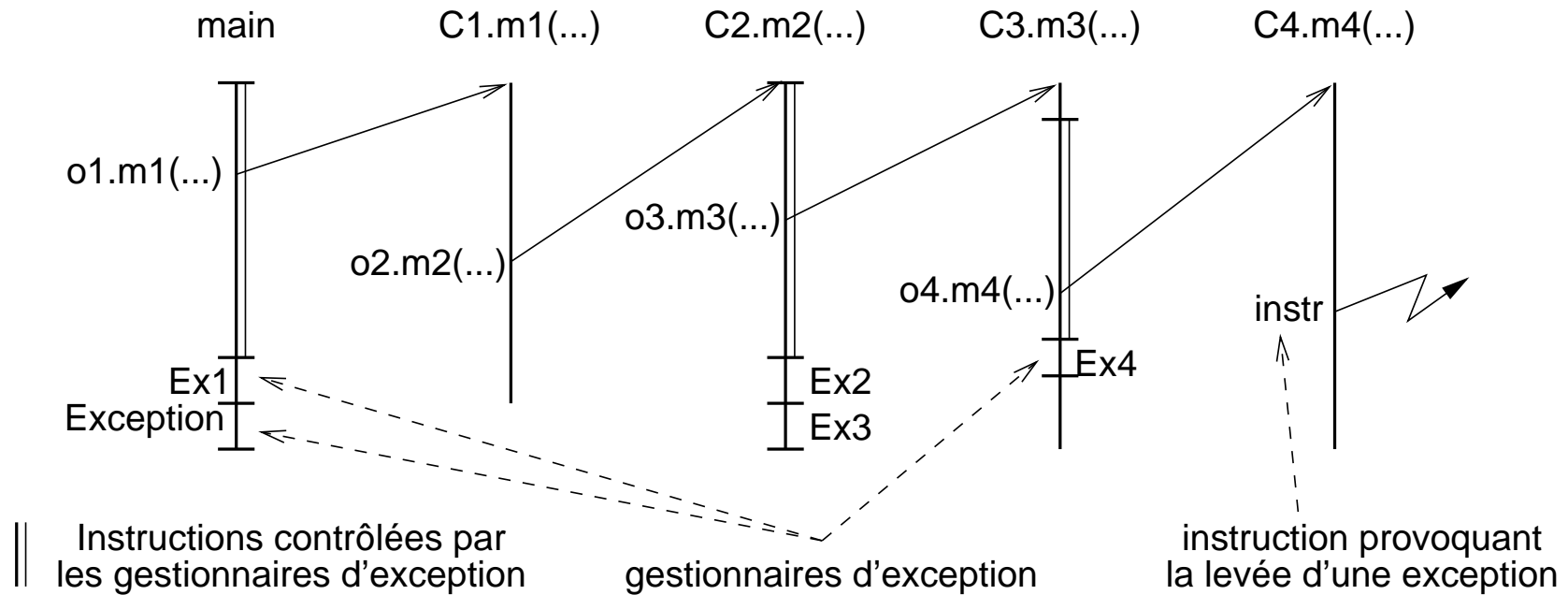
Motivation : Mécanisme pour le traitement des erreurs et/ou des cas anormaux.

Principe : Le mécanisme repose sur trois phases :

- une exception est *levée* quand une erreur ou anomalie est détectée ;
- l'exception est *propagée* : l'exécution séquentielle du programme est interrompue et le flot de contrôle est transféré aux gestionnaires d'exception ;
- L'exception est (éventuellement) *recupérée* par un gestionnaire d'exception. Elle est traitée et l'exécution « normale » reprend avec les instructions qui suivent le gestionnaire d'exception.

Remarque : Une exception non récupérée provoque l'arrêt du programme (avec affichage de la trace des appels de méthodes depuis l'instruction qui a levé l'exception jusqu'à l'instruction appelante de la méthode principale).

Mécanisme de propagation d'une exception



Exercice 39 Indiquer la suite du programme lorsque instr lève Ex4, Ex3, Ex1, Ex5 et Err.

Illustration du mécanisme de propagation

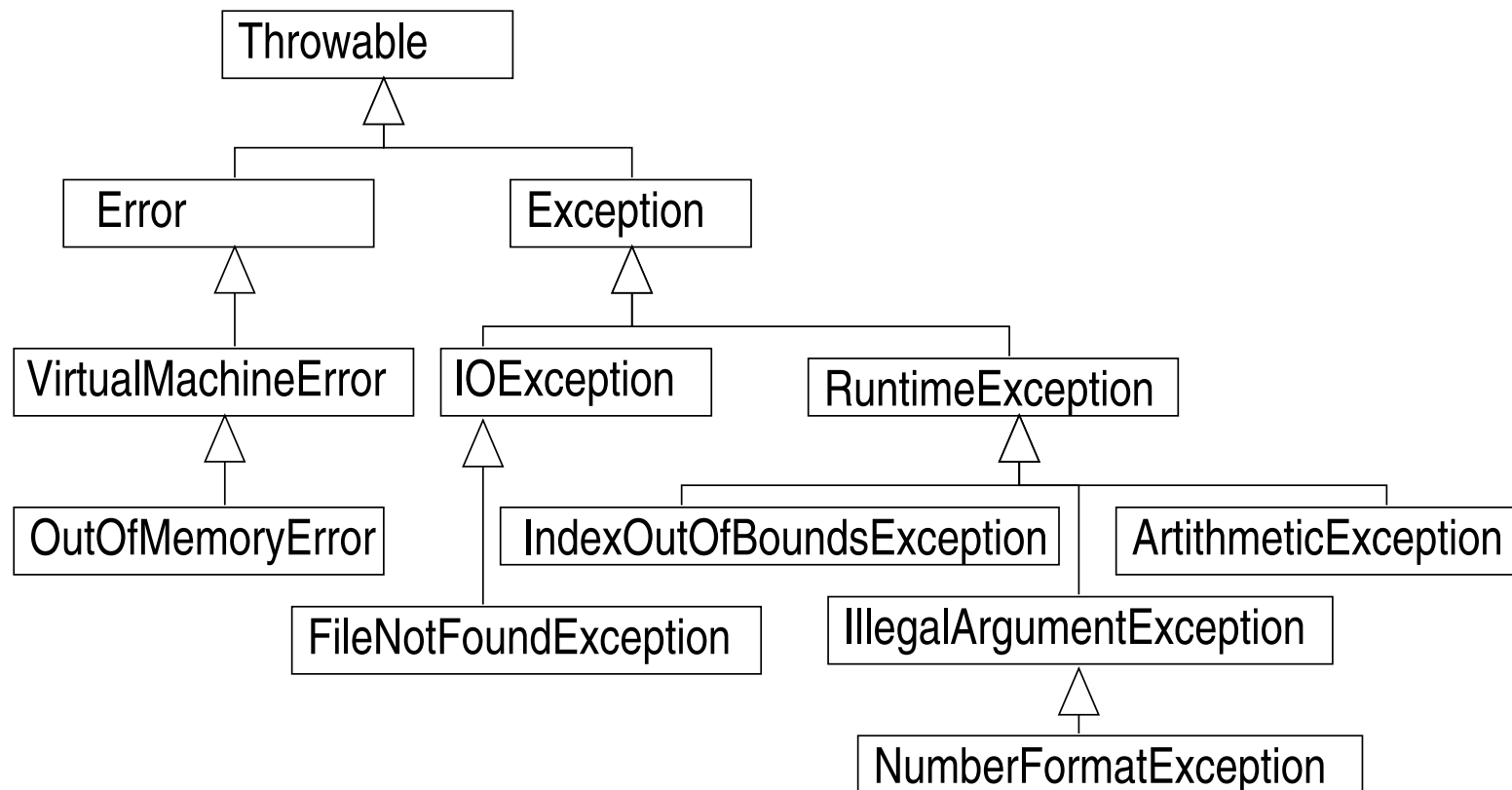
```
1  /** Illustrer le mécanisme de propagation des exceptions */
2  public class ExceptionPropagation {
3      public static void m1() {
4          m2();
5      }
6      public static void m2() {
7          m3();
8      }
9      public static void m3() {
10         int n = 1 / 0; // division par zéro !
11     }
12     public static void main(String[] args) {
13         m1();
14     }
15 }
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at ExceptionPropagation.m3(ExceptionPropagation.java:10)
    at ExceptionPropagation.m2(ExceptionPropagation.java:7)
    at ExceptionPropagation.m1(ExceptionPropagation.java:4)
    at ExceptionPropagation.main(ExceptionPropagation.java:13)
```

```
shell returned 1
```

Hiérarchie des exceptions en Java

Principe : Toutes les exceptions en Java héritent de la classe Throwable.
Voici un extrait de l'arbre d'héritage.



Classification des exceptions

Les exceptions sont forcément dérivées de la classe `Throwable`.

En Java, les exceptions sont classées en deux catégories :

– les **exceptions hors contrôle** :

– classes dérivées de `java.lang.Error` : ce sont des erreurs non accessibles, qui ne peuvent généralement pas être récupérées (`OutOfMemoryError`, `AssertionError`, `NoClassDefFoundError`...);

– classes dérivées de `java.lang.RuntimeException` : ce sont des erreurs de programmation ; elles ne devraient donc pas se produire (`NullPointerException`, `IndexOutOfBoundsException`...);

– les **exceptions sous contrôle**. Ce sont les classes dérivées de `Exception` (mais pas de `RuntimeException`), par exemple `java.io.IOException`.

Le compilateur veut être sûr que le programmeur en a tenu compte.

Elles correspondent à la notion de *robustesse*.

La classe `java.lang.Throwable`

- `Throwable(String message)` : constructeur avec un message expliquant la cause de l'exception ;
- `Throwable()` : constructeur par défaut (`message == null`) ;
- `Throwable(Throwable cause)` : constructeur avec cause ;
- `Throwable(String message, Throwable cause)` ;
- `printStackTrace()` : afficher la trace des appels de méthodes ;
- `getMessage()` : le message passé en paramètre du constructeur ;
- `getCause()` : la cause de l'exception ;
- ...

Les classes `java.lang.Exception` et `java.lang.Error`

- `Exception` et `Error` n'ajoutent aucune nouvelle caractéristique. Elles permettent simplement de classifier les anomalies.
- Elles définissent des constructeurs de même signature que `Throwable`.

Extrait des descendants de `java.lang.Error`

```
java.lang.AssertionError
java.lang.LinkageError
  +-- java.lang.ClassCircularityError
  +-- java.lang.ClassFormatError
      +-- java.lang.UnsupportedClassVersionError
  +-- java.lang.ExceptionInInitializerError
  +-- java.lang.IncompatibleClassChangeError
      +-- java.lang.AbstractMethodError
      +-- java.lang.IllegalAccessError
      +-- java.lang.InstantiationError
      +-- java.lang.NoSuchFieldError
      +-- java.lang.NoSuchMethodError
  +-- java.lang.NoClassDefFoundError
  +-- java.lang.UnsatisfiedLinkError
  +-- java.lang.VerifyError
java.lang.ThreadDeath
java.lang.VirtualMachineError
  +-- java.lang.InternalError
  +-- java.lang.OutOfMemoryError
  +-- java.lang.StackOverflowError
  +-- java.lang.UnknownError
```

Extrait des descendants de `java.lang.RuntimeException`

```
java.lang.ArithmeticException
java.lang.ArrayStoreException
java.lang.ClassCastException
java.lang.IllegalArgumentException
    +-- java.lang.IllegalThreadStateException
    +-- java.lang.NumberFormatException
java.lang.IllegalMonitorStateException
java.lang.IllegalStateException
java.lang.IndexOutOfBoundsException
    +-- java.lang.ArrayIndexOutOfBoundsException
    +-- java.lang.StringIndexOutOfBoundsException
java.lang.NegativeArraySizeException
java.lang.NullPointerException
java.lang.SecurityException
java.lang.UnsupportedOperationException
```

Extrait des autres descendants de Exception

```
java.lang.ClassNotFoundException  
java.lang.CloneNotSupportedException  
java.lang.IllegalAccessException  
java.lang.InstantiationException  
java.lang.InterruptedIOException  
java.lang.NoSuchFieldException  
java.lang.NoSuchMethodException
```

```
java.awt.AWTException  
java.util.zip.DataFormatException  
java.security.GeneralSecurityException  
java.beans.IntrospectionException  
java.io.IOException  
org.xml.sax.SAXException  
java.sql.SQLException  
org.omg.CORBA.UserException
```

Lever une exception

L'opérateur **throw** permet de lever une exception. Une exception est une instance d'une classe descendant de `Throwable` (souvent de `Exception`).

Forme générale :

```
if (<condition anormale>) {  
    throw new TypeException(<paramètres effectifs>);  
}
```

Exemples : En considérant la classe `Fraction` :

```
public Fraction(int num, int dén) {  
    if (dén == 0) {  
        throw new ArithmeticException("Division_par_zéro");  
    }  
    ...  
}  
public Fraction(Fraction autre) {  
    if (autre == null) {  
        throw new IllegalArgumentException("Poignée_nulle");  
    }  
    ...  
}
```

Récupérer une exception

Le bloc `try {...}` peut être suivi de plusieurs gestionnaires d'exception :

```
catch (TypeExc1 e) { // gestionnaire de l'exception TypeExc1
    // instructions à exécuter quand l'exception TypeExc1 s'est produite
} catch (TypeExc2 e) { // e ≈ paramètre formel (peu importe le nom)
    // instructions à exécuter quand l'exception TypeExc2 s'est produite
} catch (Exception e) { // toutes les exceptions (utiles au programmeur)
    // instructions à exécuter si une exception se produit
} catch (Throwable e) {
    // Toutes les erreurs et autres ! Utile?
}
```

Attention : L'ordre des `catch` est important (principe de substitution).

Remarque : Après l'exécution des instructions d'un `catch`, l'exécution continue après le dernier `catch` (sauf si une exception est levée).

Conseil : Ne récupérer une erreur que si vous savez comment la traiter (en totalité ou partiellement).

Traiter une exception

Si une exception est récupérée, c'est que l'on est capable de la traiter, au moins partiellement. Ce traitement est fait dans les instructions du **catch**. Il peut consister à :

- Réparer le problème et exécuter de nouveau l'opération (cf transparent suivant)
- Rétablir un état cohérent et continuer l'exécution sans recommencer
- Calculer un autre résultat remplaçant celui de la méthode
- Réparer localement le problème et propager l'exception

```
catch (TypeException e) {  
    faire des choses;    // par exemple rétablir la cohérence de l'état  
    throw e;             // propager l'exception  
    throw new ExcQuiVaBien(e); // OU Chaînage des exceptions (Java 1.4)  
}
```

- Réparer localement le problème et lever une nouvelle exception
- Terminer le programme

Traiter une exception : exemple du réessai

```
boolean reussi = false;    // est-ce que l'opération a réussi ?
do {
    try {
        instructions_qui_peuvent_échouer();
        reussi = true; // ne sera exécutée que si aucune exception ne se
                        // produit dans instructions_qui_peuvent_échouer();
    } catch (TypeExc1 e) {
        traiter_TypeExc1(); // exemple : expliquer l'erreur à l'utilisateur
    } catch (TypeExc2 e) {
        traiter_TypeExc2();
    }
} while (! reussi);
// Les instructions ont été exécutées sans erreurs !
```

Exercice 40 Adapter cet algorithme pour limiter le nombre de réessais à 5.

Exceptions : la clause finally

Un bloc **finally** peut être mis après le dernier **catch** d'un bloc **try**. Les instructions du bloc **finally** seront toujours exécutées qu'il y ait ou non une exception levée, qu'elle soit récupérée ou non.

```
try {
    instructions_qui_peuvent_échouer();
} catch (TypeExc1 e) {
    traiter_TypeExc1();
} catch (TypeExc2 e) {
    traiter_TypeExc2();
} finally {
    instructions_toujours_exécutées();
}
```

Intérêt : Être sûr de libérer une ressource (faire un `dispose` sur un élément graphique, fermer un fichier – à condition que l'ouverture ait réussi...).

Spécification des exceptions

Une exception correspond à un résultat transmis par une méthode. À ce titre, Java impose de spécifier les exceptions levées ou propagées dans la spécification d'une méthode.

Remarque : Java n'impose (et le compilateur ne vérifie) que la spécification des exceptions sous contrôle.

Syntaxe : Toutes les exceptions sous contrôle qui sont levées ou propagées par une méthode doivent être déclarées en utilisant le mot-clé **throws**.

```
<modifieurs> Type maMéthode(Type1 a) throws TypeExc1, TypeExc2 { ... }
```

```
public static double racineCarrée(double x) throws MathException { ... }
```

Remarque : Le compilateur vérifie que toutes les exceptions (sous-contrôle) produites par les instructions du code de la méthode sont :

- soit récupérées et traitées par la méthode ;
- soit déclarées comme étant propagées (clause **throws**).

Exceptions, héritage et redéfinition de méthode

Une méthode peut propager (donc lever) toute exception qui est :

- une descendante de `RuntimeException` ou `Error` (**throws** implicite) ;
- une descendante d'une des exceptions listées dans la clause **throws**.

Remarque : L'héritage entre exceptions permet (principe de substitution) :

- de limiter le nombre d'exceptions à déclarer (**throws**) ;
- de récupérer dans un même **catch** plusieurs exceptions.

Règle concernant la redéfinition de méthode : Une méthode redéfinie ne peut lever que des exceptions qui ont été spécifiées par sa déclaration dans la classe parente. Elle ne peut donc pas lever de nouvelles exceptions.

Définition d'une exception : exceptions utilisateur

Exception : Une exception est tout objet instance d'une classe qui hérite de Throwable. Cependant, une exception utilisateur hérite généralement de Exception ou de l'une de ses descendantes.

Conseil : Choisir soigneusement la classe parente de son exception (en particulier, l'exception doit-elle être sous contrôle ou hors contrôle ?).

Définition type d'une exception :

```
public class MathException extends Exception {
    public MathException(String s) {
        super(s);
    }
    public MathException() {    // utile ?
    }
}
```

Remarque : Cette classe, comme toute autre, peut avoir des attributs et des méthodes.

Exemple d'exception : la racine carrée

```
public class RacineCarree {
    public static double RC(double x) throws MathException {
        if (x < 0) {
            throw new MathException("Paramètre_de_RC_strictelement_négatif:_:" + x);
        }
        return Math.sqrt(x);
    }

    public static void main (String args []) {
        try {
            double d = Console.readDouble("Donnez_un_réel:_:");
            System.out.println("RC(" + d + ")_=_:" + RC(d));
            for (int i = 0; i < 10; i++) {
                System.out.println("RC(" + i*i + ")_=_:" + RC(i*i));
            }
        } catch (MathException e) {
            System.out.println("Anomalie:_:" + e);
        }
    }
}
```

Exemples d'exceptions sur la classe Fraction

```
public class DivisionParZeroException extends Exception {
    public DivisionParZeroException(String message) {
        super(message);
    }
}

public class Fraction {
    private int numérateur;
    private int dénominateur;
    public Fraction(int num, int dén) throws DivisionParZeroException {
        set(num, dén);
    }
    public void set(int n, int d) throws DivisionParZeroException {
        if (d == 0) {
            throw new DivisionParZeroException("Dénominateur_nul");
        }
        ...
    }
    public Fraction inverse() throws DivisionParZeroException {
        return new Fraction(dénominateur, numérateur);
    }
    ...
}
```

Utilisation des exceptions de la classe Fraction

```
public class TestFractionExceptions {
    public static void main (String args []) {
        Fraction f;
        try {
            int n = Console.readInt("Numérateur_:");
            int d = Console.readInt("Dénominateur_:");
            f = new Fraction(n, d);
            System.out.println("f_=");
            f.afficher();
            System.out.println("inverse_de_f_=");
            f.inverse().afficher();
        }
        catch (DivisionParZeroException e) {
            System.out.println("Le_dénominateur_d'une_fraction_"
                + "ne_doit_pas_être_nul");
            // Est-ce un message correct ?
        }
    }
}
```

Exercice 41 Peut-on connaître, dans le gestionnaire d'exception, la ligne du bloc `try` qui est à l'origine de l'exception (cf commentaire) ?

Une exception est une classe

Exercice 42 : Somme d'entiers

L'objectif est de calculer la somme des entiers donnés en argument de la ligne de commande en signalant les arguments incorrects. On indiquera le caractère incorrect (qui n'est donc pas un chiffre) et sa position dans l'argument comme dans les exemples suivants :

```
java Somme 10 15 20  
Somme : 45
```

```
java Somme 10 15.0 20.0  
Caractère interdit : >.< à la position 3 de 15.0
```

- 42.1** Écrire le programme qui réalise cette somme en signalant les erreurs éventuelles.
- 42.2** Comment faire pour indiquer le numéro de l'argument incorrect ?
- 42.3** Comment faire pour indiquer tous les arguments incorrects ?

L'exception `FormatException`

```
/** Exception indiquant une représentation erronée d'un entier en base 10 */
public class FormatException extends Exception {
    private String entier; // l'entier sous forme de chaîne de caractères
    private int indice; // indice de l'erreur dans entier

    public FormatException(String entier_, int ind_) {
        super("Caractère_invalide");
        entier = entier_;
        indice = ind_;
    }

    /** Représentation de l'entier en base 10 */
    public String getEntier() { return entier; }

    /** Position du premier caractère interdit */
    public int getPosition() { return indice; }

    public String toString() {
        return "FormatException:_Erreur_dans_" + entier
            + "_à_la_position_" + (indice+1);
    }
}
```

La méthode convertissant une chaîne en entier

```
/** Opérations sur les nombres */
public class Nombres {

    /** Conversion de chaîne de caractères en entier naturel.
     * @param s représentation d'un entier naturel en base 10
     * @return l'entier correspondant à s
     * @exception FormatEntierException la chaîne est mal formée
     */
    public static int atoi(String s) throws FormatEntierException {
        int resultat = 0;
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (c >= '0' && c <= '9') {
                resultat = resultat * 10 + (c - '0');
            } else {
                throw new FormatEntierException(s, i);
            }
        }
        return resultat;
    }
}
```

Le programme calculant la somme des arguments

```
/** Programme sommant les entiers en argument de la ligne de commande. */
class Somme {

    public static void main (String args []) {
        try {
            int somme = 0;
            for (int i = 0; i < args.length; i++) {
                somme += Nombres.atoi(args[i]);
            }
            System.out.println("Somme_:_" + somme);
        }
        catch (FormatException e) {
            System.out.println("Caractère_ interdit_:_" +
                + e.getEntier().charAt(e.getPosition())
                + "<_à_la_position_" + (e.getPosition()+1)
                + "_de_" + e.getEntier());
        }
    }
}
```

Exceptions : quelques conseils

- La gestion des exceptions n'est pas supposée remplacer un test simple.
- Ne pas faire une gestion ultrafine des exceptions : multiplier les blocs `try` pénalise le programme en terme de performance.
- Ne pas museler les exceptions.

```
try { beaucoup de code }  
catch (Exception e) {}
```

Si le compilateur vous signale des exceptions sous contrôle non gérées (ni récupérées, ni déclarées comme propagées), c'est pour vous aider !

L'ignorer en muselant les exceptions n'enlève pas le problème.

- Éviter d'imbriquer les blocs `try` (faire des méthodes auxiliaires).
- Ne pas avoir honte de propager une exception.

Si vous n'êtes pas capable de la traiter complètement, il est nécessaire de la propager vers l'appelant.

Programmation par contrat : analogie avec le contrat

Dans le **monde des affaires**, un contrat est une spécification précise (légalement non ambiguë) qui définit les obligations et les bénéfices des (deux) parties prenantes.

	obligations	bénéfices
client	payer un mois à l'avance son voyage sans annulation possible	obtenir un tarif préférentiel pour le voyage
fournisseur	sélectionner et réserver les hôtels, avions, dans le budget défini	faire des bénéfices, même si le client ne part pas

Exemple très simplifié : un client doit payer un certain montant (son obligation) et son fournisseur lui rend un service (réalise un projet).

Programmation : Une méthode est le fournisseur, l'appelant est le client.

Programmation par contrat : Mise en œuvre

Pour chaque méthode :

- *préconditions* : obligation du programme appelant et bénéfice de la méthode. C'est le programme appelant qui doit les vérifier.
- *postconditions* : bénéfice pour l'appelant et obligation pour la méthode. Elles doivent être remplies à la fin de l'exécution de la méthode (si ses préconditions étaient satisfaites).

Pour chaque classe :

- *invariants* : définissent les propriétés qui doivent toujours être vérifiées par un objet de la classe (depuis sa construction jusqu'à sa destruction) en particulier, (avant et) après l'appel de chaque méthode.

Non respect d'un contrat : C'est une erreur de programmation (du client dans le cas d'une précondition, du fournisseur sinon) provoquant l'arrêt du programme ou, mieux, la levée d'une exception (à ne pas récupérer).

Programmation par contrat et racine carrée

```
1 public class RacineCarree {
2     /** Racine carrée. */
3     //@ requires x >= 0;           // x positif
4     //@ ensures \result >= 0;    // la racine carrée est positive
5     //@ ensures \result * \result == x; // son carré est x
6     public static double RC(double x) {
7         return Math.sqrt(x);
8     }
9
10    public static void main (String args []) {
11        double d = Console.readDouble("Donnez un réel:");
12        if (d >= 0) { // test explicite !!!
13            System.out.println("RC(" + d + ") = " + RC(d));
14        }
15        for (int i = 0; i < 10; i++) {
16            System.out.println("RC(" + i*i + ") = " + RC(i*i));
17        }
18    }
19 }
```

Remarque : Utiliser l'égalité sur des réels est une erreur !

Programmation par contrat : bénéfiques

Intérêts :

- *définition des responsabilités* : chacun sait qui fait quoi ;
- *documentation* : les classes et méthodes sont documentées formellement donc sans ambiguïtés ;
- *aide à la mise au point* (instrumentation du code avec les assertions) :
 - vérification dynamique des assertions ;
 - détection des erreurs au plus tôt (près de leur origine) ;
- *code final optimisé* (non vérification des assertions).

Inconvénients : Les préconditions et surtout les postconditions sont souvent difficiles à identifier (complétude) et à exprimer (référence à l'état précédent, utilisation de quantificateurs existentiels et universels, etc.).

Exercice 43 : Pgcd

Donner les contrats de la méthode pgcd.

Programmation par contrat en pratique

La programmation par contrat existe :

- Eiffel : complètement intégrée au langage ;
- UML : des stéréotypes sont définis «invariant», «precondition», «postcondition» ainsi qu'un langage d'expression de contraintes OCL.
- Java : envisagée par les auteurs du langage mais non implantée.
 - Java 1.4 introduit une clause assert (voir T. 273).
 - Elle est cependant accessible au travers d'extensions telles que *iContract*, *jContractor*, *Jass...* et surtout *JML*.

Mise en œuvre avec *JML*

- spécifier le comportement dans des commentaires `/*@ ... */` et `//@` ;
- compiler en utilisant `jmlc` pour instrumenter les assertions ;
- exécuter avec `jmlrac` ;
- engendrer la documentation avec `jmldoc`.

JML : Java Modeling Language

JML : un langage de spécification du comportement introduisant préconditions (*requires*), postconditions (*ensures*) et invariants (*invariant*).

Un **contrat** est exprimé avec la syntaxe Java enrichie :

- de l'implication \Rightarrow (et \Leftarrow) et de l'équivalence \Leftrightarrow (et $\Leftarrow \Rightarrow$).

`a \Rightarrow b` est équivalent à `(! a) || b`

`a \Leftrightarrow b` (si et seulement si) est équivalent à `a == b`

- des quantificateurs universel (`\forall`) et existentiel (`\exists`).

`(\forall` déclarations; contraintes; expression booléenne)

`(\forall` **int** `i`; `i >= 0 && i < tab.length; tab[i] > 0`);

// tous les éléments de tab sont strictement positifs

`(\exists` **int** `i`; `i >= 0 && i < tab.length; tab[i] > 0`);

// Il existe un élément de tab strictement positif

- d'autres quantificateurs `\min`, `\max`, `\product`, `\sum`.

`(\sum` **int** `i`; `i >= -2 && i <= 1; i`); *// -2, -2 + -1 + 0 + 1*

`(\max` **int** `i`; `i > -2 && i < 3; i*i`); *// 4, max(-1*-1, 0*0, 1*1, 2*2)*

`(\min` **int** `i`; `i > -2 && i < 3; i*i`); *// 0, min(-1*-1, 0*0, 1*1, 2*2)*

`(\product` **int** `i`; `i > 0 && i < 5; i`); *// 24, 1 * 2 * 3 * 4 (= 4!)*

JML : Extension pour les postconditions

Dans la **postcondition** (ensures) d'une méthode *m*, on peut utiliser :

- `\result` pour faire référence au résultat de cette méthode *m* ;

```
//@ requires x != 0;
//@ ensures Math.abs(x * \result - 1) <= EPSILON; // prendre l'un
//@ ensures Math.abs(inverse(\result) - x) <= EPSILON; // ou l'autre
public static /*@ pure @*/ double inverse(double x) {
    return 1.0 / x;
}
```

Remarque : On peut avoir un contrat récursif.

- `\old(expr)` : valeur de *expr* avant l'exécution de cette méthode *m* ;

```
class Compteur {
    private int valeur;
    public /*@ pure @*/ int getValeur() {
        return this.valeur;
    }
    //@ ensures getValeur() == \old(getValeur()) + 1;
    public void incrementer() {
        this.valeur++;
    }
}
```

JML : Contrats et droits d'accès

- Un contrat ne peut utiliser que des méthodes pures (sans effet de bord).

Justification : L'exécution du programme ne doit pas dépendre de l'effet d'un contrat car les contrats peuvent être (et seront) désactivés.

Conséquence : Utiliser `/*@ pure @*/` avant le type de retour de la méthode (car Java ne permet pas d'exprimer une telle propriété).

- Les invariants peuvent être **public** ou **private** (choix de réalisation).

Remarque : Un invariant privé n'a pas de conséquence sur l'utilisateur de la classe mais seulement sur l'implémenteur.

- Le contrat d'une méthode ne peut utiliser que des méthodes ou attributs de droits d'accès au moins égaux à ceux de cette méthode.

Justification : L'appelant doit pouvoir évaluer le contrat.

Programmation par contrat : la classe Fraction (1/4)

```
/** La classe Fraction représente les fractions rationnelles.
 * @author      Xavier Crégut */
public class Fraction {
    //@public invariant Entier.pgcd(getNumerateur(),
    //@                                     getDenominateur()) == 1;
    //@public invariant getNumerateur() == 0 ==> getDenominateur() == 1;

    private int num;    // le numérateur
    private int den;    // le dénominateur
    final public static Fraction UN = new Fraction(1); // fraction unité

    /** Initialiser une fraction.
     * @param n le numérateur
     * @param d le dénominateur (non nul !) */
    //@ requires d != 0;
    //@ ensures n * getDenominateur() == d * getNumerateur();
    public Fraction(int n, int d)    { set(n, d); }

    /** initialiser une fraction à partir d'un entier (numérateur).
     * @param n le numérateur */
    //@ ensures n == getNumerateur();
    //@ ensures 1 == getDenominateur();
    public Fraction(int n)            { this(n, 1); }
}
```

Programmation par contrat : Fraction (2/4)

```
/** @return le numérateur de la fraction */
public /*@ pure @*/ int getNumérateur()    { return num; }

/** @return le dénominateur de la fraction. */
//@ ensures getDenominateur() > 0;
public /*@ pure @*/ int getDenominateur()  { return den; }

/** Modifier une fraction.
    * @param n le nouveau numérateur
    * @param d le nouveau dénominateur (non nul !)
    */
//@ requires d != 0;
//@ ensures n * getDenominateur() == d * getNumérateur();
public void set(int n, int d) {
    num = n;
    den = d;
    normaliser();
}

/** normaliser la fraction. */
private void normaliser()                { ... }
```

Programmation par contrat : Fraction (3/4)

```
/** Changer le signe de la fraction. */
//@ ensures getNumerateur() == - \old(getNumerateur());
//@ ensures getDenominateur() == \old(getDenominateur());
public void opposer()          { num = - num; }

/** @return la fraction opposée */
//@ ensures \result.getNumerateur() == - getNumerateur();
//@ ensures \result.getDenominateur() == getDenominateur();
public Fraction opposee() {
    return new Fraction(- getNumerateur(), getDenominateur());
}

/** Inverser la fraction. */
//@ requires getNumerateur() != 0;
//@ ensures getNumerateur() * \old(getDenominateur())
//@          == \old(getNumerateur()) * getDenominateur();
public void inverser()        { set(den, num); }

/** Inverse de la fraction. */
//@ requires getNumerateur() != 0;
//@ ensures \result.produit(this).equals(UN);
public Fraction inverse()     { return new Fraction(den, num); }
```

Programmation par contrat : Fraction (4/4)

```
/** le produit avec une autre fraction. */  
//@ ensures \result.equals(new Fraction(num * autre.num, den * autre.den));  
public /*@ pure @*/ Fraction produit(Fraction autre) {  
    return new Fraction(num * autre.num, den * autre.den);  
}  
  
/** Égalité logique entre fractions. */  
//@ ensures \result == ((getNumerateur() == autre.getNumerateur()  
//@      && getDenominateur() == autre.getDenominateur()));  
public /*@ pure @*/ boolean equals(Fraction autre) {  
    return num == autre.num && den == autre.den;  
}  
  
/** Afficher la fraction. */  
public void afficher()  
{  
    System.out.print(getNumerateur());  
    if (getDenominateur() > 1) {  
        System.out.print("/") + getDenominateur());  
    }  
}  
}
```

Programmation par contrat : utilisation de Fraction

```
public class TestFractionDBC {
    public static void main (String args []) throws java.io.IOException {
        int n, d;          // numérateur et dénominateur d'une fraction
        do {
            n = Console.readInt("Numérateur_:");
            d = Console.readInt("Dénominateur_:");
            if (d == 0)
                System.out.println("Le_dénominateur_doit_être_>_0_!");
        } while (d == 0);
        Fraction f = new Fraction(n, d);
        System.out.print("La_fraction_est_:");
        f.afficher();
        System.out.println();
        if (f.getNumerateur() != 0) {
            System.out.print("Son_inverse_est_:");
            f.inverse().afficher();
            System.out.println();
        }
    }
}
```

Programmation par contrat et constructeurs

Un constructeur a pour rôle d'établir l'invariant de l'objet en cours de création.

Justification : Les invariants doivent toujours être vrais, donc dès la création d'un objet.

Remarque : Ce rôle des constructeurs permet de justifier leur présence et le fait qu'un constructeur est toujours appliqué lors de la création d'un objet.

Cependant le rôle du constructeur est, *à mon sens*, plus important : il sert non seulement à établir l'invariant mais aussi à initialiser l'objet dans l'état souhaité par l'utilisateur.

Programmation par contrat et héritage

Analogie avec la sous-traitance : Le fournisseur peut sous-traiter le travail pour un tarif inférieur ou égal au prix négocié avec le client en exigeant une qualité au moins égale.

Sous-traitance = redéfinition d'une méthode dans une sous-classe.

Règles à respecter :

- affaiblir les préconditions (**require else** en Eiffel) ;
- renforcer les postconditions (**ensure then** en Eiffel) ;
- les invariants sont hérités et peuvent être renforcés.

Attention : Ni le langage, ni le compilateur ne peuvent vérifier ces règles.

C'est donc au programmeur d'être vigilant !

Attention : L'instrumentation ne permet généralement pas de vérifier cette règle.

Utilisation de la clause assert

```
1  public class RacineCarree {
2      /** Racine carrée. */
3      public static double RC(double x) {
4          assert x >= 0 : "x_négatif_" + x;
5          double result = Math.sqrt(x);
6          assert result >= 0 : result;
7          assert result * result == x : result;
8          return result;
9      }
10
11     public static void main (String args []) {
12         double d = Console.readDouble("Donnez_un_réel:_");
13         if (d >= 0) { // test explicite !!!
14             System.out.println("RC(" + d + ")_=_ " + RC(d));
15         }
16         for (int i = 0; i < 10; i++) {
17             System.out.println("RC(" + i*i + ")_=_ " + RC(i*i));
18         }
19         // Attention, sans test !!!
20         System.out.println("RC(" + (-d) + ")_=_ " + RC(-d));
21     }
22 }
```

Utilisation de la clause assert

La clause assert a été ajoutée à la version 1.4 du langage Java.

Elle modifie le langage et, en conséquence, il faut dire explicitement au compilateur d'utiliser la version 1.4 du langage.

```
licorne> javac -source 1.4 RacineCarree.java
```

Par défaut, les assertions ne sont pas vérifiées. Elles sont activées avec l'option `-ea` et désactivées avec `-da` au niveau de la classe ou du paquetage (`-esa` et `-dsa` pour les classes système).

```
licorne> java -ea RacineCarree # toutes les classes sauf système
Donnez un réel : 4
RC(4.0) = 2.0
RC(0) = 0.0
...
RC(81) = 9.0
Exception in thread "main"java.lang.AssertionError: x négatif -4.0
    at RacineCarree.RC(RC_assert.java:4)
    at RacineCarree.main(RC_assert.java:20)
```

Exceptions vs programmation par contrat

Remarque : Ce sont deux techniques non exclusives !

Cas où préférer les exceptions :

- l'évaluation de la précondition est coûteuse et redondante avec le traitement (exemples : `parseInt`, recherche dans un arbre, etc.) ;
- impossibilité de donner une précondition (exemple : appel de procédure à distance, écriture dans un fichier, etc.) ;
- cas anormal mais que l'on considère comme pouvant se produire (\neq erreur de programmation) : saisies utilisateur, etc.

Cas où préférer la programmation par contrat :

- les parties liées par contrats sont maîtrisées ;

Remarque : La JVM charge dynamiquement les classes

\implies SUN préfère les exceptions pour les méthodes publiques !

- erreur manifeste de programmation (non récupérable) ;
- efficacité du programme (éviter des tests dans les méthodes appelées) ;
- mécanisme supporté par l'environnement de développement !

Classes internes...

Exercice 44 : Améliorer `OutilsListe.somme`

Dans l'exercice 20, nous avons défini une interface `Liste` et deux réalisations `ListeTab` (les éléments sont stockés dans un tableau) et `ListeChaînée` (les éléments sont chaînés). Nous avons également écrit une classe `OutilsListe` qui définit la méthode `somme`.

```
/** Quelques méthodes utiles sur les listes. */
public class OutilsListe {
    /* Calculer la somme des réels d'une liste.
     * @param l la liste dont on veut sommer les valeurs
     * @return la somme des valeurs de l */
    static public double somme(Liste l) {
        double resultat = 0;
        for (int i = 0; i < l.taille(); i++) {
            resultat += l.item(i);
        }
        return resultat;
    }
}
```

Sommer les éléments d'une liste chaînée est inefficace. Proposer un mécanisme plus efficace... et général.

Solution

Problème : Utiliser le méthode `item(int)` n'est pas efficace sur `ListeChaînée` (parcours des cellules depuis la première).

But : Fournir le moyen de faire un parcours de la liste (donc général) qui puisse être efficace pour `ListeTab`, `ListeChaînée`...

Solution : Définir une interface qui abstrait le parcours d'une liste.

```
1  /** Un itérateur permet de parcourir tous les éléments d'une liste.  
2    * @version 1.1 */  
3  public interface Iterateur<T> {  
4  
5    /** Est-ce qu'il y a encore des éléments à parcourir ? */  
6    boolean encore();  
7  
8    /** Récupérer l'élément suivant. */  
9    T suivant();  
10 }
```

et définir une réalisation pour `ListeTab`, `ListeChaînée`...

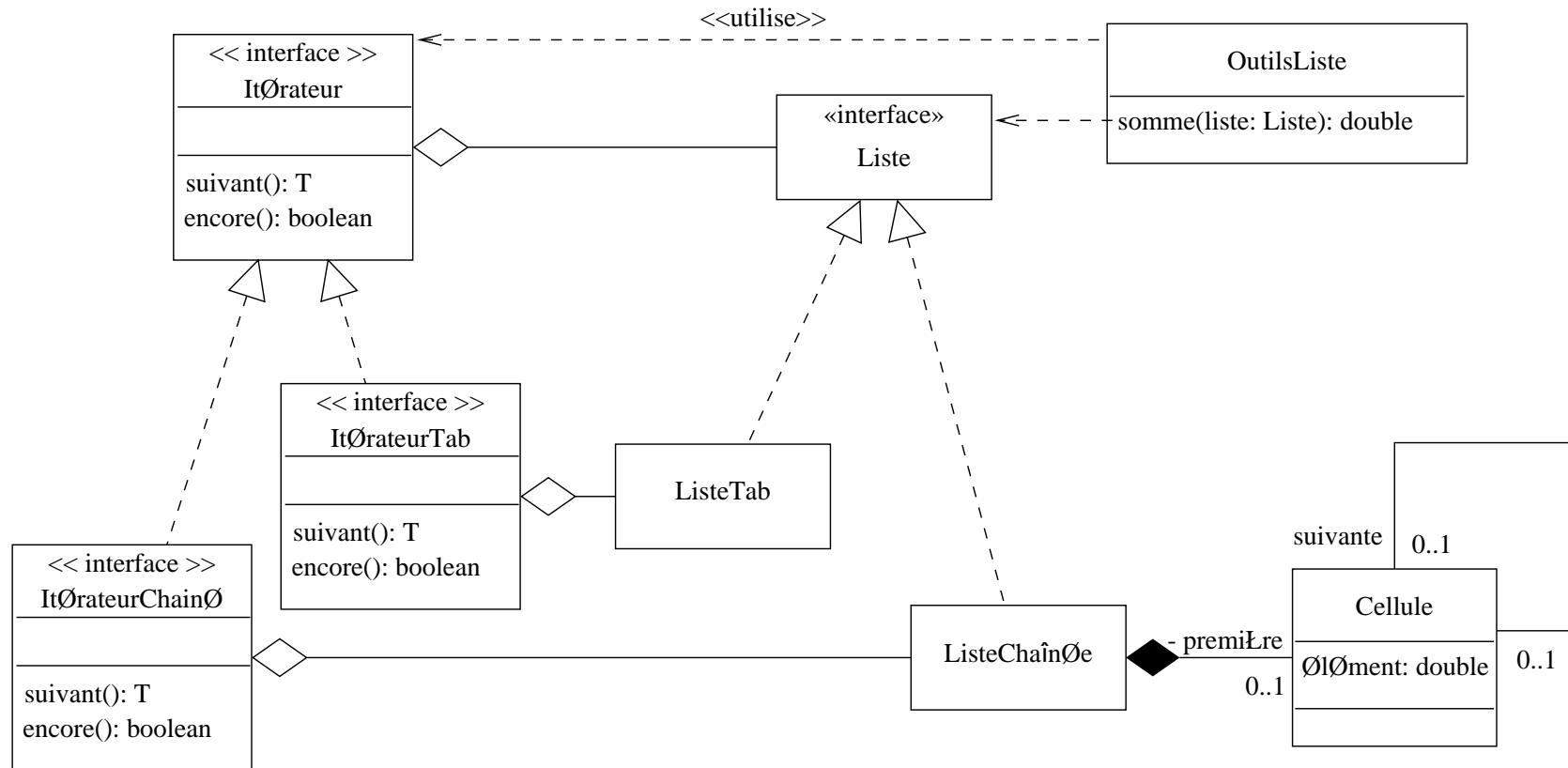
Ajouter un itérateur sur Liste

```
1 public interface Liste {
2     /** L'itérateur qui permet de réaliser un parcours de la liste. */
3     Iterateur iterateur();
4
5 }
```

Nouvelle écriture de somme :

```
1 /** Quelques méthodes utiles sur les listes. */
2 public class OutilsListe {
3     /** Calculer la somme des réels d'une liste.
4      * @param l la liste dont on veut sommer les valeurs
5      * @return la somme des valeurs de l */
6     static public double somme(Liste l) {
7         double resultat = 0;
8         Iterateur it = l.iterateur();
9         while (it.encore()) {
10             resultat += it.suivant();
11         }
12         return resultat;
13     }
14 }
```

Diagramme de classes des itérateurs de Liste



```

1  /** Itérateur sur un tableau (par exemple de ListeTab)
2  * @author      Xavier Crégut
3  * @version     1.1 */
4  class IterateurTab implements Iterateur {
5      private double[] elements; // les éléments à parcourir
6      private int nb; // le nb d'éléments
7      private int curseur; // élément à parcourir
8
9      public IterateurTab(double[] elts, int taille) {
10         this.elements = elts;
11         this.nb = taille;
12         this.curseur = 0;
13     }
14
15     public boolean encore() {
16         return this.curseur < this.nb;
17     }
18
19     public double suivant() {
20         return this.elements[this.curseur++];
21     }
22 }

```

Discussion

De manière générale, l'itérateur (ici `IterateurTab`) devrait avoir accès à la structure de données (ici `ListeTab`). Voir le diagramme de classe !

- `IterateurTab` est une classe extérieure à `ListeTab`
- `IterateurTab` doit accéder à la représentation interne de `ListeTab`

⇒ deux solutions :

1. `ListeTab` ajoute des méthodes de manipulation de sa représentation interne MAIS violation du principe d'encapsulation !
2. `ListeTab` fournit en paramètre du constructeur de `IterateurTab` ses données internes (solution choisie).

Comment faire pour garder une solution plus proche du diagramme de classe : l'itérateur a accès à la structure de données ?

⇒ On utilise une classe interne !

```

1  public class ListeTab<T> implements Liste<T> {
2      private T[] elements;          // les éléments de la liste
3      private int nb;              // la taille de la liste
4
5      public Iterateur<T> iterateur() {
6          return new IterateurTab<T>(this);
7      }
8
9      static private class IterateurTab<T> implements Iterateur<T> {
10         private ListeTab<T> liste; // liste à parcourir
11         private int curseur;      // élément à parcourir
12
13         public IterateurTab(ListeTab<T> l) {
14             this.liste = l;
15             this.curseur = 0;
16         }
17
18         public boolean encore() {
19             return this.curseur < this.liste.nb;
20         }
21
22         public T suivant() {
23             return this.liste.elements[this.curseur++];
24         }
25     }
26 }

```

Discussion sur la classe interne « statique »

Ici, nous avons défini une classe interne « statique » et privée.

Remarque : Une classe interne a un droit d'accès.

Les intérêts de la classe interne « statique » sont :

- Une classe interne a accès à toutes les caractéristiques (y compris **private**) de la classe qui la contient.
- Ici, ceci permet donc :
 - de conserver l'encapsulation de `ListeTab` : aucune méthode ajoutée ;
 - de conserver le sens du diagramme UML : `IterateurTab` utilise `ListeTab` ;
 - **MAIS** ceci est assez lourd. Ne pourrait-on pas directement avoir accès aux caractéristiques de l'objet de la classe englobante ?
 - ⇒ C'est la notion de classe interne propre (non « statique ») :

Attention : Une classe interne ne peut pas définir d'information statique sauf si elle est déclarée **final**.

```

1  public class ListeTab<T> implements Liste<T> {
2      private T[] elements;           // les éléments de la liste
3      private int nb;                // la taille de la liste
4
5      public Iterateur<T> iterateur() {
6          return new IterateurTab();
7      }
8
9      private class IterateurTab implements Iterateur<T> {
10         private int curseur;       // élément à parcourir
11
12         public IterateurTab() {
13             this.curseur = 0;
14         }
15
16         public boolean encore() {
17             return this.curseur < nb;
18         }
19
20         public T suivant() {
21             return elements[this.curseur++];
22         }
23     }
24 }

```

Classe interne

Défi nition : Une classe interne est une classe définie à l'intérieur d'une autre classe.

Avantages : La classe interne est dans l'espace de nom de l'autre classe et a donc accès à toutes ses informations (y compris celles déclarées **private**).

Classe interne membre : (T. 285) Elle contient un accès sur l'objet qui a permis sa création (et récupère les paramètres de généricité).

```
Liste<Double> ll = ...;  
Iterateur<Double> it = ll.iterateur();  
    // it est construit à partir de ll
```

Classe interne « statique » : (T. 283) Elle n'est associée à aucun objet de la classe englobante.

Classe anonyme

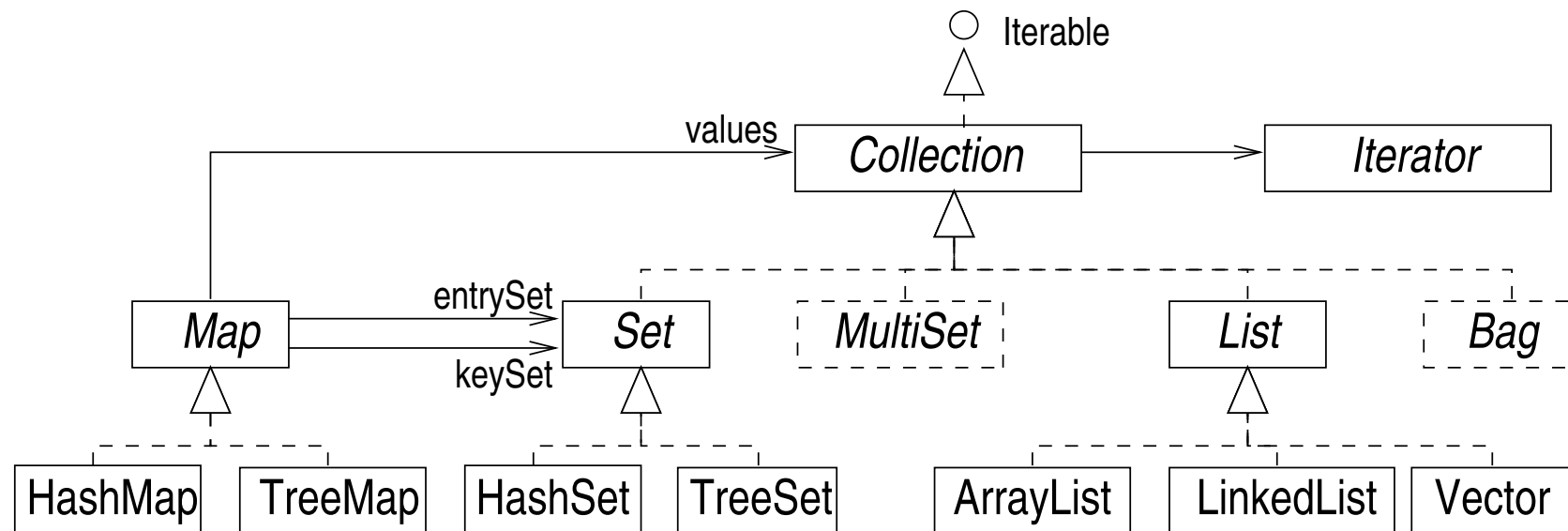
```
1 public class ListeTab<T> implements Liste<T> {
2     private T[] elements;           // les éléments de la liste
3     private int nb;                 // la taille de la liste
4
5     public Iterateur<T> iterateur() {
6         return new Iterateur<T>() {
7             private int curseur = 0;    // élément à parcourir
8
9             public boolean encore() {
10                return this.curseur < nb;
11            }
12
13            public T suivant() {
14                return elements[this.curseur++];
15            }
16        };    // Ne pas oublier le point-virgule
17    }
18 }
```

- Une classe anonyme est une classe qui n'a qu'une seule instance.
- Peut avoir accès aux variables locales du sous-programme à condition qu'elles soient déclarées **final**;

Principales API du langage Java

Hiérarchie des structures de données en Java

Voici un extrait du diagramme de classes concernant les structures de données de Java.



Remarque : *Vector* est une « ancienne » classe qui a été modifiée pour faire partie de la hiérarchie *Collection*. C'est la seule classe « synchronisée ».

L'interface *java.util.Collection*

L'interface *Collection* est la racine de l'arbre d'héritage des structures de données de Java.

Définition : Une collection représente un groupe d'*éléments*, de type E (généricité).

Propriétés : Voici quelques propriétés des collections :

- Avant Java 1.5, les éléments d'une collection étaient Object (T. 206).
- Une collection peut autoriser ou non plusieurs occurrences d'un même élément.
- Les éléments peuvent être ordonnés ou non.
- Toute classe dérivée de *Collection* devrait définir un constructeur par défaut (qui crée une collection vide) et un constructeur qui prend en paramètre une collection.
- Certaines méthodes peuvent ne pas être implémentées dans les sous-classes qui devraient alors lever *UnsupportedOperationException*.

L'interface *java.util.Collection*<E>

C'est la racine de la hiérarchie définissant les collections (groupe d'éléments).

```
boolean add(E o)           // ajouter l'élément (false si doubles interdits)
boolean addAll(Collection<? extends E> c) // ajouter les éléments de c
void clear()               // supprimer tous les éléments de la collection
boolean contains(T o)      // est-ce que la collection contient o ?
boolean containsAll(Collection<?> c)    // ... tous les éléments de c ?
boolean isEmpty()         // aucun élément ?
Iterator<E> iterator()     // un itérateur sur la collection
boolean remove(Object o) // supprimer l'objet o (collection changée ?)
boolean removeAll(Collection<?> c)      // supprimer tous les élts de c
boolean retainAll(Collection<?> c)      // conserver les éléments de c
int size()                 // le nombre d'éléments dans la collection
Object[] toArray()         // un tableau contenant les éléments de la collection
```

Remarque : Certaines opérations sont optionnelles ou imposent des restrictions. Dans ce cas, elles doivent lever une exception !

La classe `java.util.Vector<E>`

Objectif : Disposer de tableaux automatiquement redimensionnables.

Remarque : La classe `Vector` est définie dans le paquetage `java.util`.

Constructeurs : Sauf contre-indication, ils créent des vecteurs vides.

```
Vector(int capacitéInitiale, int incrémentCapacité)
Vector(int capacitéInitiale) // Vector(capacitéInitiale, 4)
Vector() // Vector(10);
Vector(Collection<? extends E> c) // vecteur contenant les éléments de c
```

Méthodes élémentaires : (équivalentes à celles des tableaux)

```
int size() // le nombre d'éléments du vecteur (taille effective)
int capacity() // capacité du vecteur
```

```
E get(int index) // vecteur[index] 0 <= index < size()
E elementAt(int index) // get(index)
```

```
boolean add(E o) // ajouter o comme dernier élément du vecteur avec
// redimensionnement si nécessaire (renvoie true)
```

```
boolean addElement(E o) // idem add(Object)
add(int index, E o) // vecteur[index] = o et 0 <= index <= size()
set(int index, E o) // vecteur[index] = o et 0 <= index < size()
```

La classe `java.util.Vector` (suite)

```
E firstElement()           // get(0)
E lastElement()            // get(size()-1)

boolean isEmpty()         // aucun élément ?

void clear()              // supprimer tous les éléments du vecteur
void removeAllElements()  // idem clear()
boolean remove(Object o)  // supprime la première occurrence
E remove(int index)        // supprimer vecteur[index] (changé ?)

boolean contains(Object o) // vecteur contient-il o ?

int indexOf(Object o)      // premier index de o dans le vecteur
int indexOf(Object o, int i) // ... à partir de l'indice i
int lastIndexOf(Object o)  // dernier index de o dans le vecteur
int lastIndexOf(Object o, int i) // ... à partir de l'indice i

void setSize(int ns)      // changer la taille effective du vecteur

Object[] toArray()         // un tableau contenant les éléments du vecteur
```

Remarque : Il existe des méthodes qui manipulent une collection.

L'interface `java.lang.Iterator<E>`

Objectif : Un itérateur est un objet qui permet de parcourir tous les éléments d'une collection.

```
boolean hasNext() // reste-t-il des éléments dans la collection ?  
E next()           // l'élément suivant (et avancer)  
void remove()    // supprimer l'élément (peut ne pas être définie)
```

Exemple d'utilisation :

```
Collection<Double> c = ...  
Iterator<Double> it = c.iterator();  
while (it.hasNext()) {  
    Double élément = it.next(); // retourne l'élément courant et avance  
    // manipuler élément...  
}
```

Inspiration : Analogie avec le parcours d'un tableau.

L'interface `java.util.Iterable<E>`

```
public interface Iterable<E> {  
    Iterator<E> iterator();  
}
```

Intérêt : Il est possible d'utiliser `foreach` sur toute classe qui réalise `Iterable`.

```
Iterable<E> collection = ...  
for (E o : collection) {  
    faire(o);  
}
```

Ceci est équivalent à :

```
Iterable<E> collection = ...  
Iterator<E> it = collection.iterator();  
while (it.hasNext()) {  
    E o = it.next();  
    faire(o);  
}
```

Les entrées/sorties

L'API d'entrée/sortie sont définies dans le paquetage `java.io`. Elle :

- fournit une interface standard pour gérer les *flux* d'entrée/sortie ;
- et libère le programmeur des détails d'implantation pour une plateforme particulière.

Flux : Séquence ordonnée de données qui a une *source* (*input stream*) ou une *destination* (*output stream*).

Classes de base : Java propose 4 classes principales (abstraites) pour les entrées/sorties qui seront ensuite spécialisées en fonction de la nature de la source ou de la destination.

	Caractères	Octets
Entrée	Reader	InputStream
Sortie	Writer	OutputStream

La classe `InputStream`

But : Lire des octets (bytes) depuis un flux d'entrée.

```
public abstract class InputStream {  
    public abstract int read() throws IOException;  
        // Lire un octet renvoyé sous la forme d'un entier entre 0 et 255.  
        // Retourne -1 si le flux d'entrée est terminé. Bloquante.  
    public int read(byte[] buf, int off, int len) throws IOException;  
        // Lire au plus len octets et les stocker dans buf à partir de off.  
        // Retourne le nombre d'octets effectivement lus (-1 si fin de flux).  
        // @throws IndexOutOfBoundsException, NullPointerException...  
    public int read(byte[] b) throws IOException;  
        // Idem read(b, 0, b.length)  
    public long skip(long n) throws IOException  
        // Sauter (et supprimer) n octets du flux.  
        // Retourne le nombre d'octets effectivement sautés.  
    public void close() throws IOException;  
        // Fermer le flux et libérer les ressources système associées.  
    public int available() throws IOException;  
        // nombre d'octets disponibles (sans bloquer)  
}
```

La classe OutputStream

But : Écrire des octets (bytes) dans un flux de sortie.

```
public abstract class OutputStream {  
  
    public abstract void write(int b) throws IOException;  
    // Écrire b dans ce flux (seuls les 8 bits de poids faible).  
  
    public void write(byte[] buf, int off, int len) throws IOException;  
    // Écrire len octets de buf[off] à buf[off+len-1] dans ce flux.  
    // @throws IndexOutOfBoundsException, NullPointerException...  
  
    public void write(byte[] b) throws IOException;  
    // Idem write(b, 0, b.length)  
  
    public void flush() throws IOException  
    // Vider ce flux.  
    // Si des octets ont été bufférisés, ils sont effectivement écrits.  
  
    public void close() throws IOException;  
    // Fermer le flux et libérer les ressources système associées.  
}
```

Les classe Reader et Writer

Principe : Équivalentes à InputStream et OutputStream mais avec des caractères et non des octets.

Remarque : int pour stocker 16 bis d'un caractères ou -1 pour fin du flux.

```
public abstract class Reader {
    public int read() throws IOException;
    public abstract int read(char[] buf, int off, int len) throws IOException;
    public int read(char[] b) throws IOException;
    public long skip(long n) throws IOException
    public abstract void close() throws IOException;
    public boolean ready() throws IOException;    // prêt à être lu ?
}
public abstract class Writer {
    public void write(int b) throws IOException;
    public abstract void write(char[] buf, int off, int len) throws IOException;
    public void write(char[] b) throws IOException;
    public void write(String str) throws IOException;
    public void write(String str, int off, int len) throws IOException;
    public void flush() throws IOException
    public abstract void close() throws IOException;
}
```

Spécialisation de ces classes abstraites

Flux bufferisés : BufferedXXX. Les écritures/lectures ne se font pas directement dans le flux mais sont « bufferisées ». On gagne en performance, par exemple pour des accès fichiers.

Remarque : Pour forcer l'écriture, utiliser flush().

Flux sur fichiers : FileXXX. Le constructeur prend en paramètre le nom du fichier (String), un objet de type File ou FileDescriptor. Ils peuvent lever les exceptions :

```
FileNotFoundException // le fichier n'existe pas
SecurityException    // pas de droit en lecture/écriture
```

Flux vers chaînes de caractères : StringXXX (sauf StringOutputStream).

Flux vers tube : PipedXXX.

Flux Print : PrintStream et PrintWriter surcharge print (et println) pour les types primitifs.

Les classes `InputStreamReader` et `OutputStreamWriter`

But : Ces deux classes permettent de faire le lien entre `InputStream/OutputStream` et `Reader/Writer`.

Elles sont des spécialisations `Reader/Writer`.

```
public class InputStreamReader extends java.io.Reader {  
  
    public InputStreamReader(InputStream in, String charsetName)  
        throws UnsupportedOperationException  
        // Le flux d'entrée est in. charsetName est le nom du codage  
        // utilisé (US-ASCII, ISO-8859-1, UTF-8...)  
  
    public InputStreamReader(InputStream in);  
        // Le flux est in avec le jeu de caractères par défaut  
}
```

Signature similaire pour `OutputStream`.

Exemple :

```
BufferedReader in  
    = new BufferedReader(new InputStreamReader(System.in));
```

La classe Object

En Java, si une classe n'a pas de classe parente, elle hérite implicitement de la classe Object. C'est l'ancêtre commun à toutes les classes.

Elle contient en particulier les méthodes :

- **public boolean** equals(Object obj); Égalité de **this** et obj (par défaut égalité des adresses). Elle a cependant le sens d'égalité logique et doit donc être redéfinie (String, etc.)
- **public String** toString(); chaîne de caractères décrivant l'objet. Elle est utilisée dans print, println et l'opérateur de concaténation + par l'intermédiaire de String.valueOf(Object).
- **protected void** finalize(); Méthode appelée lorsque le ramasse-miettes récupère la mémoire d'un objet.
- **protected Object** clone(); Copie logique de l'objet (cf Cloneable)
- **public Class** getClass(); Pour l'introspection.
- ...

L'interface Cloneable

Problème : Comment faire pour obtenir une copie physique d'un objet ?

Constatations :

- Il est logique que ce soit la classe qui donne accès à la copie.
- La classe Object définit la méthode `Object clone()` mais en **protected** et qui ne fait qu'une copie binaire des attributs (\implies partage des objets).

Conséquence : Le concepteur d'une classe doit décider si ses objets peuvent ou non être clonés. Dans l'affirmative, il doit :

1. Implémenter l'interface Cloneable ;
2. Redéfinir la méthode clone en la déclarant **public** et en donnant un code qui réalise effectivement la copie.

Remarque : Cloneable est une *interface de marquage* (le concepteur montre qu'il a défini correctement clone()) \neq interface classique.

```
interface Cloneable { } // Cloneable ne déclare rien !!!
```

L'interface Cloneable : la classe Point

```
public class Point implements Cloneable {
    private double x, y;    // coordonnées cartésiennes
    ...
    public Object clone() {
        try {
            return super.clone();    // une copie superficielle suffit !
        } catch (CloneNotSupportedException e) {
            return null; // ne peut pas se produire car implements Cloneable
        }
    }
}
```

Remarque : Implémenter Cloneable autorise à faire `super.clone()` !

Attention : Lors d'un appel à `clone()`, aucun constructeur n'est appelé !

Remarque : `try/catch` correspond à la notion d'exception (cf Exceptions).

Ici, l'exception traduit la tentative de « cloner » un objet qui n'implante pas l'interface Cloneable.

L'interface Cloneable : la classe Segment

```
/** Un segment est <strong>composé</strong> de deux points extrémités */  
public class Segment implements Cloneable {  
    private Point extrémité1, extrémité2;  
  
    public Segment(Point ext1, Point ext2) {  
        extrémité1 = (Point) ext1.clone();    // car composition  
        extrémité2 = (Point) ext2.clone();  
    }  
  
    public Object clone() {  
        try {  
            Segment s = (Segment) super.clone();    // copie superficielle  
            s.extrémité1 = (Point) extrémité1.clone();    // copie de l'extrémité1  
            s.extrémité2 = (Point) extrémité2.clone();    // copie de l'extrémité2  
            return s;  
        }  
        catch (CloneNotSupportedException e) {  
            return null;  
        }  
    }  
}
```

Conclusions

Idées clés

- Techniques de décomposition et d'architecture.
- Architectures logicielles flexibles et décentralisées (ni centre, ni sommet).
- Construction de logiciels par combinaison ascendante (*bottom-up*) d'éléments logiciels réutilisables.
- Éléments partiellement affinés décrivant des comportements communs.
- Spécification précise de chaque composant (contrat logiciel).
- Ouverture sur le monde extérieur : routines externes, empaquetage d'outils existants.
- BIBLIOTHÈQUES

Qu'est ce que la technologie objet ?

- Un principe d'architecture :

MODULE = TYPE (CLASSE)

- Une discipline épistémologique :

L'ABSTRACTION

- Une règle de classification :

L'HÉRITAGE (le sous-typage !)

- Une exigence de validité :

LA CONCEPTION PAR CONTRAT

- Une obligation d'ingénieur :

RÉUTILISABILITÉ ET EXTENSIBILITÉ

Liste des exercices

Exercice 1 : Équation du second degré	5
Exercice 2 : Lister les erreurs de la classe Équation	9
Exercice 3 : Évolution de la mémoire	10
Exercice 4 : Comparaison de résoudre en C et Java	14
Exercice 5 : Caractéristiques des objets fractions	62
Exercice 6 : Obtenir la valeur d'attributs privés	70
Exercice 7 : Description UML de la classe Fraction	70
Exercice 8 : Accéder à un attribut privé	72
Exercice 9 : Définir une date avec jour, mois, année et quantième	72
Exercice 10 : Supérieur et inférieur	76
Exercice 11 : Compteur	77

Exercice 12 : Paramètre avec valeur par défaut	79
Exercice 13 : Surcharge des méthodes de Fraction	79
Exercice 14 : Surcharge et ambiguïté	80
Exercice 15 : Comprendre le passage de paramètres en Java	83
Exercice 16 : Constructeur de Équation	88
Exercice 17 : Incrémenter le jour d'une date	95
Exercice 18 : Méthode de classe ou d'instance ?	100
Exercice 19 : Site WEB en UML	132
Exercice 20 : Liste de réels	135
Exercice 21 : Sens de p.m()	148
Exercice 22 : Questions sur ListeTab	149
Exercice 23 : Formaliser le comportement de la liste	151
Exercice 24 : Généraliser les listes	156

Exercice 25 : Échanger deux élément d'un tableau	165
Exercice 26 : Plus grand élément d'un tableau	167
Exercice 27 : Définition d'une point nommé	171
Exercice 28 : Déplier la classe PointNommé	180
Exercice 29 : Schémas mathématiques : les classes abstraites	190
Exercice 30 : Méthode retardée vs méthode redéfinie	195
Exercice 31 : Définir un groupe d'objets géométriques	196
Exercice 32 : Définir un menu textuel	196
Exercice 33 : Pourquoi les interfaces	198
Exercice 34 : Modélisation d'une équipe de football	205
Exercice 35 : Généricité vs Héritage	206
Exercice 36 : Généricité et sous-typage	218

Exercice 37 : Afficher les éléments d'une liste	220
Exercice 38 : Copier une liste dans une autre	222
Exercice 39 : « Trajet » d'une exception	234
Exercice 40 : Exception avec nombre de réessais limité	245
Exercice 41 : Ambiguïté sur l'origine d'une exception	252
Exercice 42 : Somme d'entiers	253
Exercice 43 : Pgcd	261
Exercice 44 : Améliorer <code>OutilsListe.somme</code>	277