NFP121 — Programmation Avancée

—

Abstraction et Modularité : Classes

Xavier Crégut
<Prénom.Nom@enseeiht.fr>

ENSEEIHT
Télécommunications & Réseaux

Objectifs et structure de ce support

Les **objectifs de ce support** sont de présenter :

- les concepts objets limités à l'encapsulation et le masquage d'information (module)
- en les illustrant avec le langage Java
- et la notation UML (Unified Modelling Language)

La **structure du document** est :

- Exemple introductif : passage de l'impératif à une approche objet
- Encapsulation : classe et ses membres, objets
- Cycle de vie des objets : constructeurs et destructeur
- Masquage d'information : droits d'accès
- Algorithmique : une nécessité mais pas l'objet du cours (proche Langage C)
  - types
  - structures de contrôle
  - tableau
  - types énumérés
- Des aspects méthodologiques, en particulier :
  - Documenter : il faut se comprendre !
  - Tester : avoir et donner confiance dans le code écrit !
  - Spécifier (formellement) : programmation par contrat
- Des compléments : aspects utiles à connaître mais non fondamentaux.
  - Fabriques statiques, Ellipse, Module Java vs module en C, ...

Références

[1] B. Eckel, *Thinking in Java*.
Prentice-Hall, 3 ed., 2002.
http://www.mindviewinc.com/Books/.

[2] C. S. Horstmann and G. Cornell, *Au cœur de Java 2*, vol. 1 Notions fondamentales.
Campus Press, 8 ed., 2008.

[3] Oracle, "The Source for Java Technology." http://java.oracle.com.

[4] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*.
Addison-Wesley, 3 ed., Mar. 2005.
http://java.sun.com/docs/books/jls/.

[5] B. Meyer, *Object-oriented software construction*.
Prentice Hall, 2^{nd} ed., 1997.

[6] M. Fowler, *UML 2.0*.
CampusPress Référence, 2004.

[7] OMG, "UML Resource Page." http ://www.omg.org/uml/.

[8] http://stackoverflow.com.

[9] P.-A. Muller and N. Gaertner, *Modélisation objet avec UML*.
Eyrolles, 2è ed., 2003.

Sommaire

1. Exemple introductif
2. Modularité : Classe
3. Constructeurs et destructeur
4. Masquage d'information
5. Membres de classe
6. Programmation impérative
7. Aspects méthodologiques
8. Exemples de classe dans l'API Java
9. Compléments

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Les robots
- Les équations

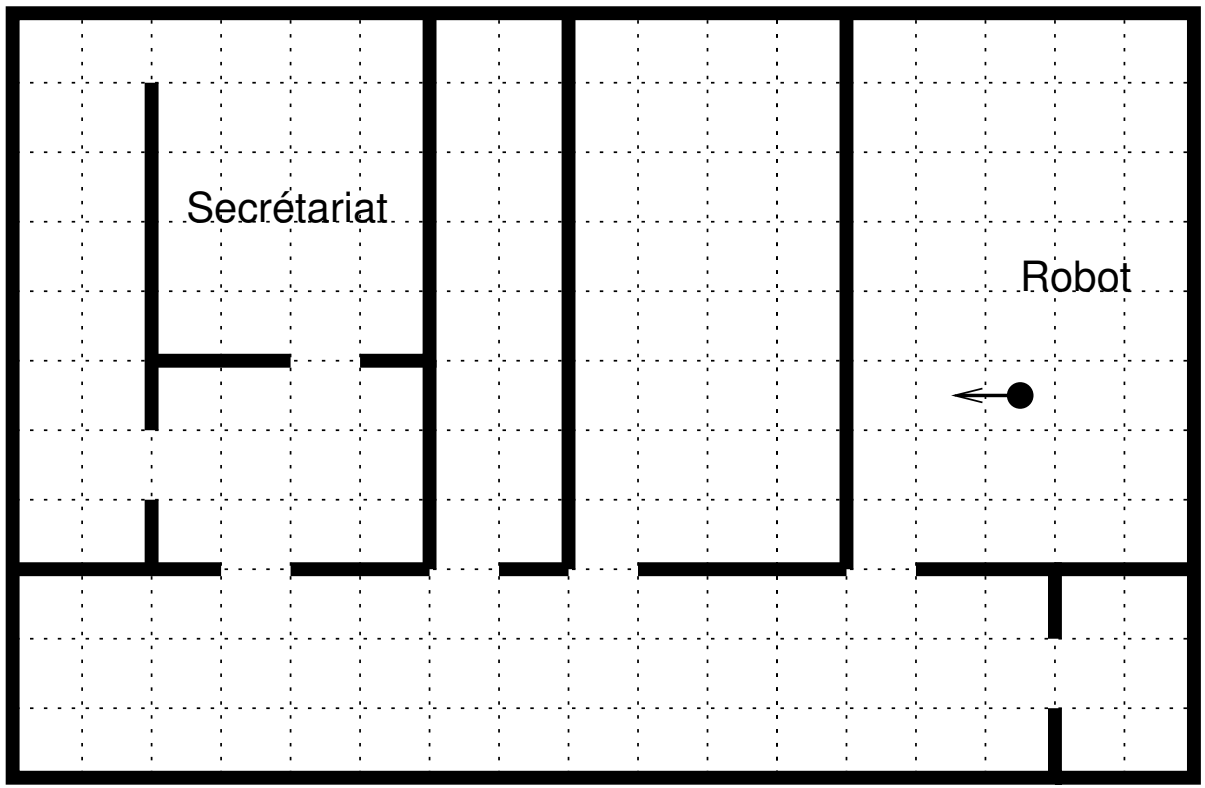
Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Les robots
- Les équations

Modéliser un robot

Exercice 1 Modéliser un robot capable d'avancer d'une case et de pivoter de 90° vers la droite. On pourra alors le guider de la salle de cours (position initiale du robot) jusqu'au secrétariat.



Types et sous-programmes associés

En pseudo-code

1. On sait définir un type Robot :

```
RobotType1 =
    Enregistrement
        x: Entier;     -- abscisse
        y: Entier;     -- ordonnée
        direction: Direction
    FinEnregistrement


Direction = (NORD, EST, SUD, OUEST)
```

2. On sait modéliser ses opérations :

```
Procédure avancer(r: in out RobotType1)
    -- faire avancer le robot r
  Début
    ...
  Fin

Procédure pivoter(r: in out RobotType1)
    -- faire pivoter le robot r
    -- de 90° à droite
  Début
    ...
  Fin

Procédure initialiser(r: out RobotType1
        x, y: in Entier, d: in Direction)
    -- initialiser le robot r...
   Début
     r.x <- x
     r.y <- y
     r.direction <- d
   Fin
```

3. On sait utiliser des robots :

```
  Variable
      r1, r2: RobotType1;
  Début
      initialiser(r1, 4, 10, EST)
      initialiser(r2, 15, 7, SUD)
      avancer(r1)
      pivoter(r2)
  Fin
```

Module : Encapsulation des données (types) et traitements (SP)

- **Principe :** Un type (prédéfini ou utilisateur) n'a que peu d'intérêt s'il n'est pas équipé d'opérations !
- **Justifications :**
  - éviter que chaque utilisateur du type réinvente les opérations
  - favoriser la réutilisation
- **Module :** Construction syntaxique qui :
  - regroupe des types et les opérations associées et
  - permet le masquage d'information (évolutivité), voir T. 63
- **Intérêt des modules :**
  - structurer l'application à un niveau supplémentaire par rapport aux sous-programmes (conception globale du modèle en V) ;
  - factoriser/réutiliser le code (type et SP) entre applications ;
  - améliorer la maintenance : une évolution dans l'implantation d'un module n'a pas d'impact sur les autres modules d'une application, voir T. 63 ;
  - améliorer les temps de compilation : compilation séparée de chaque module.
- **Question :** Pourquoi séparer données et traitements ?
- **Approche objet :** regrouper données et SP dans une entité appelée classe

Version objet

1. Modéliser les robots :

Classe = Données + Traitements

RobotType1
x : Entier y : Entier direction : Direction
avancer pivoter initialiser(**in** x, y : int, **in** d : Direction)

Notation UML :

Nom de la classe
attributs *(état)*
opérations *(comportement)*

2. Utiliser les robots (pseudo-code)

```
Variable
    r1, r2: RobotType1
Début
    r1.initialiser(4, 10, EST)
    r2.initialiser(15, 7, SUD)
    r1.pivoter
    r2.avancer
Fin
```

r1 : RobotType1
x = 4 y = 10 direction = ~~EST~~ SUD
avancer pivoter initialiser(**in** x, y : int, **in** d : Direction)

(objet r1)

r2 : RobotType1
x = 15 y = ~~7~~ 6 direction = SUD
avancer pivoter initialiser(**in** x, y : int, **in** d : Direction)

(objet r2)

Classes et objets

Classe : matrice (moule) pour créer des données appelées objets

- spécifie l'état et le comportement des objets (voir Objet)
- équivalent à un type équipé des opérations pour en manipuler les données

Objet : donnée créée à partir d'une classe (instance d'une classe)

- une identité : distinguer deux objets différents
  - r1 et r2 sont deux objets différents, instances de la même classe Robot
  - en général, l'identité est l'adresse de l'objet en mémoire
- un état : informations propres à un objet, décrites par les attributs/champs
  - exemple : l'abscisse, l'ordonnée et la direction d'un robot
  - propre à chaque objet : r1 et r2 ont un état différent
- un comportement : décrit les évolutions possibles de l'état d'un objet sous la forme d'opérations (sous-programmes) qui lui sont appliquables
  - exemples : avancer, pivoter, etc.
  - une opération manipule l'état de l'objet
  - tous les objets d'une même classe proposent les mêmes opérations

Remarque : On ne sait pas dans quel ordre les opérations seront appelées.

Constatations

- Notation pointée pour les attributs (`r1.x`) et les opérations (`r1.avancer`)
- Un paramètre privilégié :
  - `initialiser(r1, 4, 10, EST)` (version classique)
  - `r1.initialiser(4, 10, EST)` (version objet)
  - même nombre de paramètres : 4
  - r1 joue un rôle particulier dans l'approche objet
  - c'est « son » opération qui est exécutée
  - il est appelé *récepteur*
- Ce paramètre privilégié est implicite :
  - n'apparaît pas dans la signature

```
Procédure initialiser(x, y: in int, d: in Direction)
```

  - pour y faire référence dans l'implantation, il doit avoir un nom, **this** en Java :

```
void initialiser(int x, int y, Direction d) {
    this.x = x;
    this.y = y;
    this.direction = d;
}
```

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Les robots
- Les équations

Équation du second degré

Problème posé

Motivation : Exemple fil rouge utilisé tout au long de ce support.

Exercice 2 : Équation du second degré
Étant donnée une équation du second degré $ax^2 + bx + c = 0$, résoudre cette équation et afficher ses solutions. On se limitera au cas général en supposant que l'équation considérée admet effectivement deux solutions réelles.

Modélisation UML

Une équation possède :

- un état :
  - les coefficients
  - les racines
  - (le discriminant)
- un comportement :
  - résoudre
  - ...

Équation
coeffA : double coeffB : double coeffC : double x1 : double x2 : double
résoudre ...

Version initiale de la classe Equation

Équation.java

```
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 !

Exercice : En 2ème lecture, pourquoi est-elle simpliste ?

Le programme principal

RésolutionÉquation.java

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

Remarque : En Java, tout doit être défini dans une classe !

Exercice 3 Exécuter le programme.

Commentaires

- Variables locales :
  - allouées dans la pile d'exécution
  - allocation et libération complètement gérées par le compilateur
  - mémoire forcément contiguë (pile d'exécution : dernière allouée, première libérée)
- `Équation eq` :
  - Déclaration d'une variable (locale) `eq`
  - Son type est `Équation`
  - Il s'agit d'une poignée : contient l'adresse d'un objet Équation
  - *pointeur* et *référence* sont des synonymes pour poignée
- Opérateur **`new`** :
  - allocation dynamique de mémoire (à la demande du programmeur)
  - mémoire allouée dans le tas et non dans la pile d'exécution
  - équivalent d'une variable anonyme : une donnée en mémoire sans nom
  - la donnée n'est accessible que par son adresse

- **Exercice 4** Comparer la notion de module (type + SP) et la notion de classe.

Les outils du JDK (Java Development Kit)

Fichiers Java

- Les programmes Java sont écrits dans des fichiers .java
- Le nom du fichier correspond généralement au nom de la classe : A.java pour la classe A

Java est un langage compilé... vers du code intermédiaire (Byte code)

- javac RésolutionÉquation.java
  $\Longrightarrow$ produit RésolutionÉquation.class et Équation.class
- les .class contiennent le code intermédiaire (pas directement exécutable)
- javac compile automatiquement tous les fichiers requis

Le code intermédiaire est interprété par une machine virtuelle

- java RésolutionÉquation
  - la classe RésolutionÉquation est exécutée
- Attention, on donne le nom d'une classe et non un fichier. La casse est importante !

D'autres outils :

- javadoc : engendrer la documentation des classes.
  - javadoc -d doc *.java
    - $\Longrightarrow$ produit de nombreux fichiers HTML
- javap : désassembler les .class.
- ...

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Classe, objet et poignée
- Attributs
- Méthodes

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Classe, objet et poignée
  - Classe
  - Objet
  - Poignée
  - Représentation mémoire
- Attributs
- Méthodes

Classe

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
}
```

  - *attributs* = stockage d'information (état de l'objet).
  - *méthodes* = unités de calcul (sous-programmes : fonctions ou procédures).

  $\Longrightarrow$ La classe est donc une unité d'encapsulation
  C'est aussi un espace de noms :
  - deux classes différentes peuvent avoir des membres de même nom
- un **TYPE** qui permet de :
  - créer des objets (la classe est un moule, une matrice) ;
  - déclarer des poignées auxquelles seront attachés ces objets.

Les méthodes et attributs définis sur la classe comme MODULE pourront être appliqués à ses objets (par l'intermédiaire des poignées).

Objet

- Objet : donnée en mémoire qui est le représentant d'une classe.
  - l'objet est dit instance de la classe
- 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 : identifie de manière unique un objet (p.ex. son adresse en mémoire).
- Un objet est créé à partir d'une classe avec l'opérateur **`new`**

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

  - retourne l'identité de l'objet créé
  - Un objet n'existe que lors de l'exécution du programme.
- En Java, les objets sont soumis à la sémantique de la référence :
  - la mémoire qu'ils occupent est allouée dans le tas (pas dans la pile d'exécution)
  - un objet est l'équivalent d'une variable anonyme.
  - seule leur identité permet de les désigner
- Les identités des objets sont conservées dans des poignées.
  - L'objet est dit *attaché* à la poignée.

Poignée

- **Poignée :** moyen pour conserver un accès sur un objet
  - c'est une variable dont le type est une classe

```
Équation eq;        // déclaration d'une poignée
```

  - Certains auteurs parlent de *variable d'objet*.
- La **valeur d'une poignée** est :
  - l'identité d'un objet (l'objet est dit attaché à la poignée)

```
eq = new Équation();    // nouvel objet créé et attaché à eq
Équation eq2 = eq;      // eq et eq2 donnent accès au même objet
```

  - `null` pour indiquer qu'aucun objet ne lui est associé

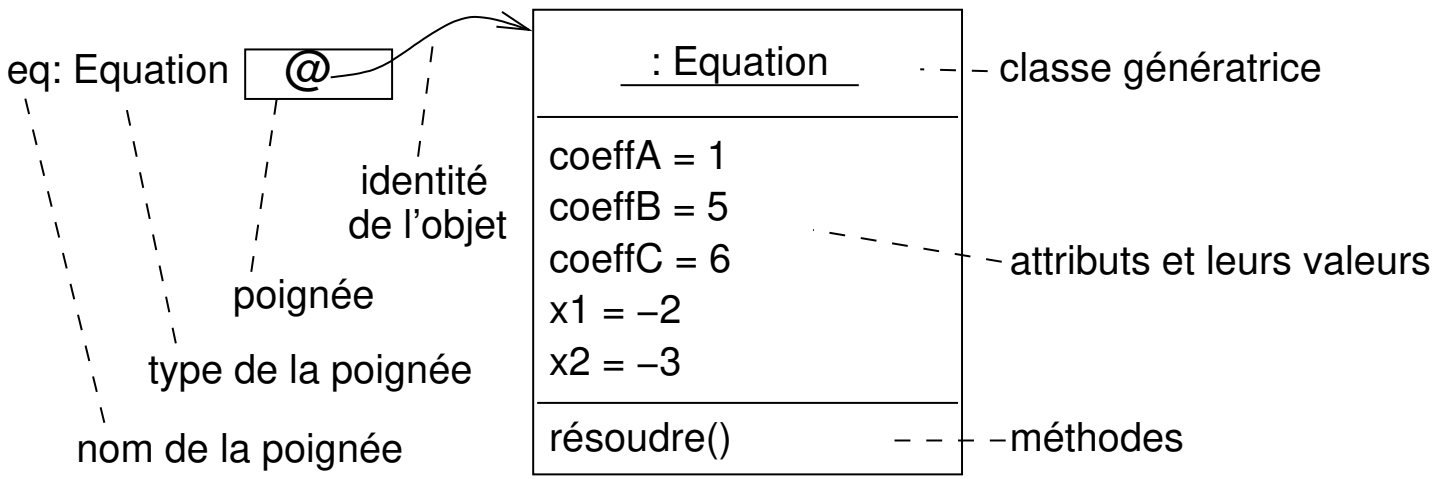
```
Équation eq3 = null;    // pas d'objet attaché à eq
```

  - indéterminée si la poignée est une variable locale non initialisée (vérifié par le compilateur)

```
Équation eq4;   // indéterminée (si variable locale) ou null (si attribut)
```

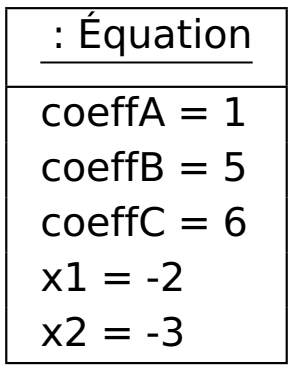
Représentation mémoire

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



Voir aussi T. 238.

Représentation en UML



- Les opérations ne sont pas représentées car ce sont les mêmes pour tous les objets instances d'une même classe.

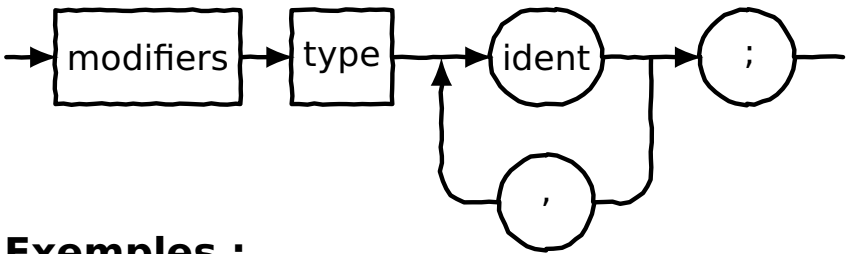
Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Classe, objet et poignée
- Attributs
- Méthodes

Attributs

- Les attributs stockent les informations spécifiques à un objet.
- **Déclaration :** (notation de Conway)



- **Exemples :**

```
double coeffA;
double x1, x2;
```

  - exemple de modifieur : **`final`** (attribut constant) voir T. 97

- **Utilisation :** notation pointée de la forme `identité.attribut`
  - identité : identité de l'objet (généralement une poignée), non nulle !
  - attribut : nom de l'attribut qui doit être défini sur le type de identité

```
Équation eq = new Équation();
double sol1 = eq.x1;    // accès en lecture
eq.coeffA = 1;          // accès en modification
double a = new Equation().coeffA;       // objet à usage unique !
```

Exemple d'erreur : utilisation d'une poignée nulle

```
1  class TestEquationErreur3 {
2      public static void main(String[] args) {
3          Equation eq = null;
4          eq.coeffA = 1;
5      }
6  }
```

L'exécution donne le résultat suivant :

```
Exception in thread "main" java.lang.NullPointerException
        at TestEquationErreur3.main(TestEquationErreur3.java:4)
```

Attention : Si la poignée est **`null`**, l'erreur n'est signalée qu'à l'exécution.

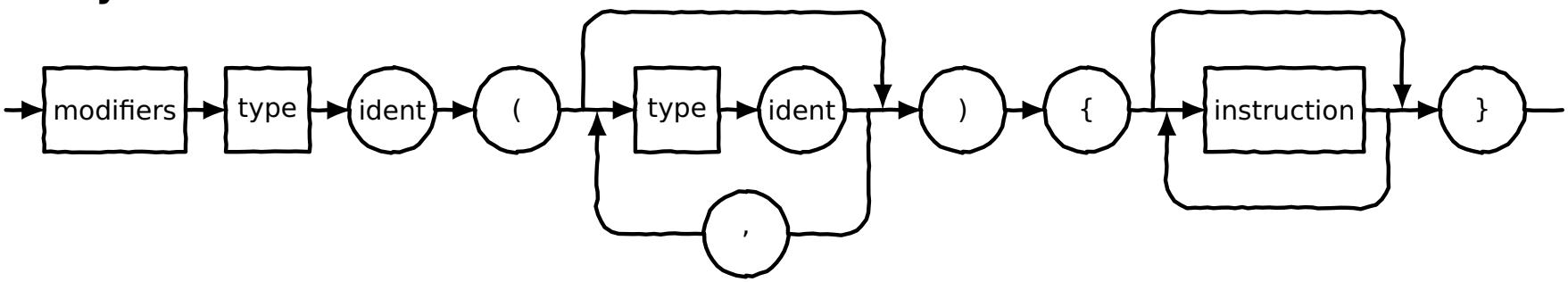
Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Classe, objet et poignée
- Attributs
- Méthodes
  - Définition
  - Utilisation
  - Surcharge
  - Espace de nom

Méthodes

- **Définition :** Une méthode est un sous-programme qui exploite l'état d'un objet (en accès et/ou en modification).
- **Syntaxe :**



- Un commentaire de documentation décrit son objectif, ses conditions d'utilisation, son effet, ses paramètres, la valeur retournée... (voir T. 135)
- Les instructions sont décrites T. 97
- L'instruction « **`return`** `expression` » arrête l'exécution du sous-programme.
  La valeur retournée est `expression`. (voir T. 103)
- Si la méthode ne retourne pas de valeur, le type est **`void`** : c'est une procédure.

Exemples de méthodes dans la classe Équation

- Méthodes pour **initialiser** une équation (procédure)

```
/** Initialiser une équation à partir de la valeur de ses coefficients.
* @param a coefficient de x<sup>2</sup>
* @param b coefficient de x
* @param c coefficient constant
*/
void initialiser(double a, double b, double c) {
    this.coeffA = a;
    this.coeffB = b;
    this.coeffC = c;
}
```

- Méthodes pour obtenir le **discriminant** d'un équation (fonction)

```
/** Obtenir le discriminant de l'équation.
* @return le discriminant de l'équation
*/
double delta() {
    return this.coeffB * this.coeffB - 4 * this.coeffA * this.coeffC;
}
```

- `/** ... */` : commentaire de documentation (voir T. 135)

Utilisation d'une méthode

- Syntaxe pour l'appel de méthode

```
identité.nomMéthode(p1, ..., pn)        // forme générale
```

- **Règles :** identité est non **`null`** et la liaison statique est respectée.
- **Exemple :**

```
Équation eq = new Équation();   // Créer un objet Équation attaché à eq
eq.initialiser(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 ici ?
               // On peut toujours ignorer la valeur retournée par une méthode

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

Liaison statique : Le compilateur accepte l'appel p.m(a_1, ...,a_n) **ssi** il existe, dans le type de la poignée p, une méthode m d'arité n telle que les types de a_1, ..., a_n sont *compatibles* avec la signature de cette méthode.

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**.

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

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

- **this** nécessaire si un paramètre ou une variable a le même nom qu'un attribut.
  - Résolution de nom : identifiant cherché comme variable locale, paramètre puis attribut.
- **Conseil :** Mettre **this**.

Afficher une équation

- **Première idée :** on veut afficher une équation, la classe Equation doit donc fournir une méthode « afficher » (et on ferra `eq.afficher()`).

```
1  /** Afficher cette équation.  */
2  void afficher() {
3      System.out.print(this.coeffA + "*x2 + "
4              + this.coeffB + "*x + " + this.coeffC + " = 0");
5  }
```

- **Deuxième idée :** faire directement : `System.out.println(eq);`
  Compile, s'exécute mais affiche : `Équation@8814e9`
  Java utilise la méthode **`toString()`** pour convertir l'objet en chaîne de caractères.
  En définissant cette méthode :

```
1  /* Obtenir la représentation de cette équation sous forme
2  * d'une chaîne de caractères. */
3  public String toString() {
4      return this.coeffA + "*x2 + "
5              + this.coeffB + "*x + "
6              + this.coeffC + " = 0";
7  }
```

  on obtient alors l'affichage : `1.0*x2 + 5.0*x + 6.0 = 0`
  Remarque : La méthode `toString` doit être déclarée **`public`**.

Surcharge

Définition : 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 :

```
1  class A {
2    void afficher()                // afficher sans paramètre
3    void afficher(int i)           // afficher un entier
4    void afficher(long i)          // afficher un entier long
5    void afficher(String str)      // afficher une chaîne
6    void afficher(String s, int largeur); // sur une certaine largeur
7    void afficher(String s, int largeur, char mode); // mode == 'c', 'g', 'd'
8    void afficher(int nbFois, String str);        // contre-exemple !
9  }
```

Surcharge : résolution

- Le même nom peut être utilisé pour nommer des méthodes différentes.
  $\Longrightarrow$ Pour résoudre un appel de méthode, le compilateur s'appuie également sur le nombre et le type des paramètres effectifs (voir liaison statique, T. 33) :

```
1  A a = new A();
2  a.afficher("Bonjour", 20, 'c');   // afficher(String, int, char)
3  a.afficher("Bonjour");            // afficher(String)
4  a.afficher(true);                 // ERREUR à la compilation
5  a.afficher(10L);                  // afficher(long)
6  a.afficher(10);                   // afficher(int) (plus adapté que afficher(long))
7  a.afficher();                     // afficher()
8  a.afficher(20, "Bonjour");        // afficher(int, String)
```

- **Intérêt :** Éviter de multiplier les noms : `afficherInt`, `afficherLong`...
- **Conseil :** Respecter le sens sous-entendu par le nom de la méthode !

Exemples de surcharge

Exercice 5 On considère la classe suivante. Indiquer la méthode appelée pour chaque instruction de la méthode main.

```
1  class ExempleSurcharge {
2    static void m0(int a, int b)          { System.out.println("m0(i,i)"); }
3
4    static void m1(double a, double b)    { System.out.println("m1(d,d)"); }
5
6    static void m2(double a, double b)    { System.out.println("m2(d,d)"); }
7    static void m2(double a, int b)       { System.out.println("m2(d,i)"); }
8    static void m2(int a, double b)       { System.out.println("m2(i,d)"); }
9    static void m2(int a, int b)          { System.out.println("m2(i,i)"); }
10
11   static void m3(double d, int i)       { System.out.println("m3(d,i)"); }
12   static void m3(int i, double d)       { System.out.println("m3(i,d)"); }
13
14   public static void main(String[] args) {
15       m0(1, 1);          m1(1, 1);       m2(1, 1);       m3(1, 1);
16       m0(2, 2.0);        m1(2, 2.0);     m2(2, 2.0);     m3(2, 2.0);
17       m0(3.0, 3);        m1(3.0, 3);     m2(3.0, 3);     m3(3.0, 3);
18       m0(4.0, 4.0);      m1(4.0, 4.0);   m2(4.0, 4.0);   m3(4.0, 4.0);
19   }
20 }
```

Espace de noms

- Il est possible de définir la même méthode (même nom, même nombre et type de paramètres) dans deux classes différentes. (Idem pour les attributs.)

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

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

- Le compilateur s'appuie sur le type du récepteur pour sélectionner l'opération à exécuter (voir T. 33).
  $\Longrightarrow$ La classe définit un **espace de noms**.

```
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) spécifiée dans la classe A
x2.afficher(10);        // afficher(int) spécifiée dans la classe B
```

- **Remarque :** Pour certains auteurs, il s'agit de surcharge (le type du récepteur change).

Exercices

Exercice 6 : Version simplifiée d'une fraction

On s'intéresse à la notion de fraction.
On souhaite pouvoir afficher une fraction. Voici quelques exemples : 2/5, 1/2, -8/3, 7, 0.
On souhaite pouvoir changer le signe de la fraction. Par exemple, 2/5 devient -2/5.
On souhaite pouvoir additionner deux fractions mais aussi une fraction et un entier. Par exemple, 1/2 + 2/3 donne 7/6 ou 1/2 + 2 donne 5/2.

6.1 Modéliser avec UML la classe Fraction.

6.2 Écrire en Java la classe Fraction.

6.3 Indiquer quelles opérations pourraient être ajoutées.

Exercice 7 : Version simplifiée d'une date

Nous souhaitons pouvoir disposer d'un type date permettant d'obtenir le jour, le mois et l'année d'une date. Nous souhaitons pouvoir afficher une date suivant le format jj/mm/année, le jour et le mois étant toujours affichés sur deux positions (par exemple 05/10/2010). On veut pouvoir savoir si une date est antérieure (strictement) à une autre ou (strictement) postérieure à une autre. On veut aussi pouvoir incrémenter une date (c'est-à-dire passer au lendemain) ou ajouter une durée (exprimée en nombre de jours) à une date.

7.1 Modéliser avec UML la classe Date.

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Constructeurs
- Destructeur
- Cycle de vie d'un objet

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Constructeurs
- Destructeur
- Cycle de vie d'un objet

Motivation

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.

Risque : initialisation incorrecte ou non conforme aux souhaits de l'utilisateur

```
Equation eq = new Equation();   // réservation de la mémoire
eq.initialiser(1, 5, 6);        // initialiser la mémoire
```

Les **constructeurs** permettent alors :

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

Exemple : Une équation peut être initialisée à partir de la donnée de ses 3 coefficients.

Les constructeurs en Java

En Java, un constructeur ressemble à une méthode mais :

1. il a nécessairement le même nom que la classe ;
2. il ne doit pas avoir de type de retour ;

Exemple : Un constructeur pour les équations

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

Attention : Mettre un type de retour supprime le caractère « constructeur ». On a alors une simple méthode !

Exercice 8 Peut-on définir plusieurs constructeurs sur une même classe ? Combien ?

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 initialiser une équation à partir de la somme et du produit de ses racines.

```
/** Initialiser une équation à partir de la somme
* et du produit de ses racines
* @param somme  somme des racines
* @param produit produit des racines
*/
Équation(double somme, double produit) {
    this.coeffA = 1;
    this.coeffB = - somme;
    this.coeffC = produit;
}
```

Exercice 9 Peut-on définir un constructeur qui initialise une équation à partir de ses deux solutions ? Pourquoi ?

Appel d'un autre constructeur de la classe

Problème : Plutôt que d'avoir les mêmes trois affectations que dans le premier constructeur, pourquoi ne pas utiliser le premier constructeur ?
On peut le faire en utilisant **`this`**`(...)` :

```
/** Initialiser une équation à partir de la somme
* et du produit de ses racines
* @param somme  somme des racines
* @param produit produit des racines
*/
Équation(double somme, double produit) {
    this(1, -somme, produit);
        // Appel au constructeur  Équation(double, double, double)
        // Cet appel est nécessairement la première instruction !
}
```

Règle : L'appel à l'autre constructeur est *nécessairement* la première instruction du constructeur.
Bien sûr, les paramètres effectifs de **`this`**`(...)` permettent de sélectionner l'autre constructeur.

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

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

Le constructeur par défaut

- **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.

- **Exercice 10** Le programmeur peut toujours définir un constructeur par défaut... mais y a-t-il intérêt ?

  On considère le code suivant :

  ```
  Date d = new Date();
  ```

  Indiquer la valeur du jour, du mois et de l'année de cette date.

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) ;

Et en deuxième lecture :

- il ne peut pas être redéfini dans une sous-classe (cf héritage).
- le caractère de « constructeur » ne s'hérite pas (cf héritage).

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Constructeurs
- Destructeur
- Cycle de vie d'un objet

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 destructeur par classe.
- En Java, le « destructeur » est :

```
void finalize()
```

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

$\Longrightarrow$ Définir une méthode explicite (dispose, close, destroy, etc.)...
Et dire aux utilisateurs de penser à l'appeler (documentation) !

Voir aussi exceptions et `AutoCloseable` (Java1.7).

Exercice

Exercice 11 Complétons l'exercice 6.

11.1 Indiquer quels constructeurs définir sur la classe Fraction.

11.2 Doit-on définir un destructeur ?

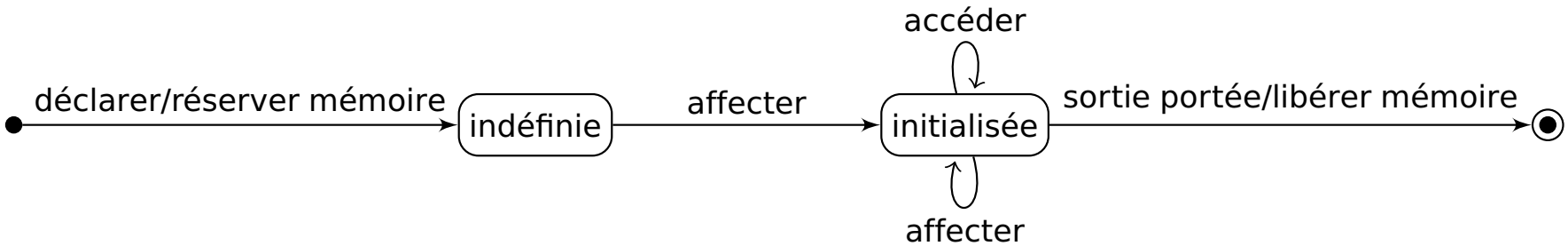
Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

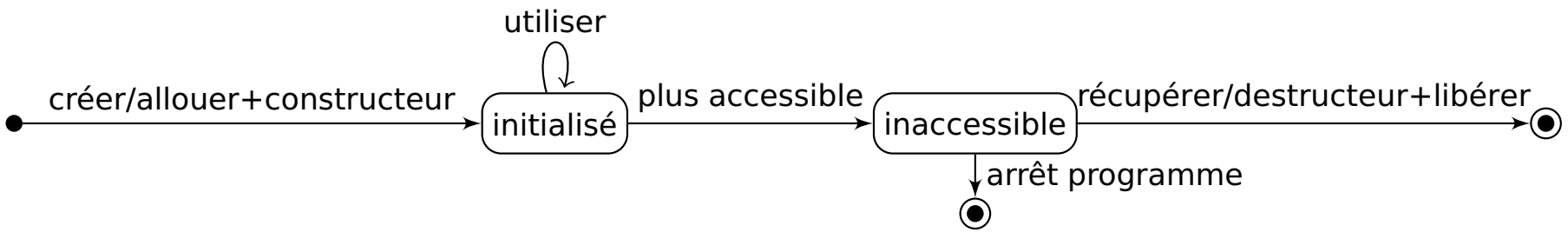
- Constructeurs
- Destructeur
- Cycle de vie d'un objet

Cycle de vie d'un objet

- Diagramme de machine à états d'une variable locale.



  - **Rq :** Le compilateur vérifie qu'une variable est initialisée avant d'être utilisée.
- Diagramme de machine à états d'un objet :



  - constructeur : réservation de la mémoire et initialisation atomiques
  - destructeur : libération des ressources avant libération de la mémoire

Exemple d'erreur : variable locale non initialisée

```
1  class TestEquationErreur2 {
2      public static void main(String[] args) {
3          Equation eq;
4          eq.coeffA = 1;
5      }
6  }
```

La compilation donne le résultat suivant :

```
TestEquationErreur2.java:4: error: variable eq might not have been initialized
        eq.coeffA = 1;
        ^
1 error
```

Remarque : Le compilateur Java vérifie qu'une variable locale est initialisée avant d'être utilisée.

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Paquetages
- Droits d'accès
- La classe Equation améliorée

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Paquetages
- Droits d'accès
- La classe Equation améliorée

Paquetages

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

```
// fichier A.java
package nom.du.paquetage;        // paquetage d'appartenance
class A { ...  }                 // texte Java de la classe
```

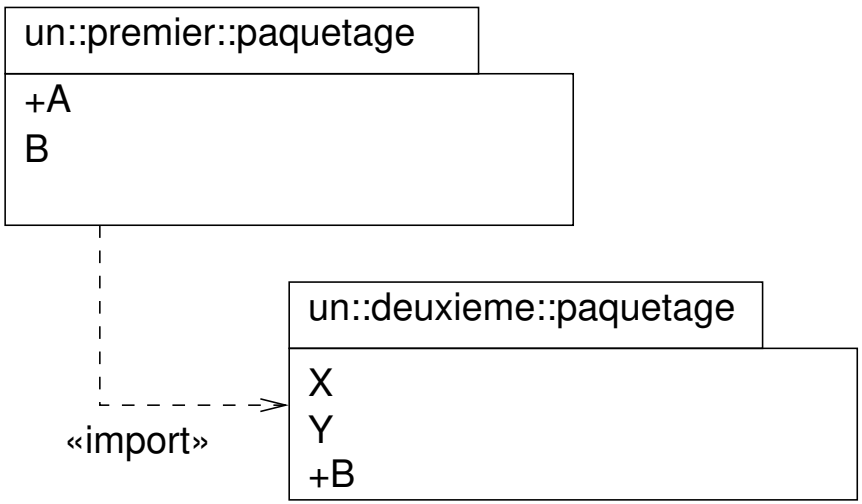
- La directive **package** doit être la première ligne du fichier.
- Une classe ne peut appartenir qu'à un seul paquetage.
- La structure des paquetages s'appuie sur le système de fichiers :
  - les paquetages sont des répertoires et
  - les classes sont des fichiers.
- Si le paquetage d'appartenance 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... (qui correspondent à java/lang/, java/util/...).

Droits d'accès

```
// fichier A.java
package un.premier.paquetage;
public class A { ... }
// fichier B.java
package un.premier.paquetage;
class B { ... }

// fichier B.java
package un.deuxieme.paquetage;
class X { ... }
class Y { ... }
public class B { ... }
```



- Une classe peut être publique (**public**) ou locale au paquetage.
- Une classe déclarée publique (**public**) peut être utilisée depuis d'autres paquetages
- Une classe locale (sans **public**) ne peut pas être utilisée à l'extérieur du paquetage.
- Un même fichier Java peut contenir plusieurs classes (déconseillé) mais une seule publique (elle impose son nom au fichier).
- Deux paquetages différents peuvent utiliser le même nom de classe.

Utiliser une classe d'un autre paquetage

- utiliser le nom complet de 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.

Bilan

- **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 noms).
- **Conseil :** Pour éviter toute ambiguïté, il est recommandé de ne pas utiliser $*$
- **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.

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Paquetages
- Droits d'accès
- La classe Equation améliorée

Droit d'accès (visibilité) des membres

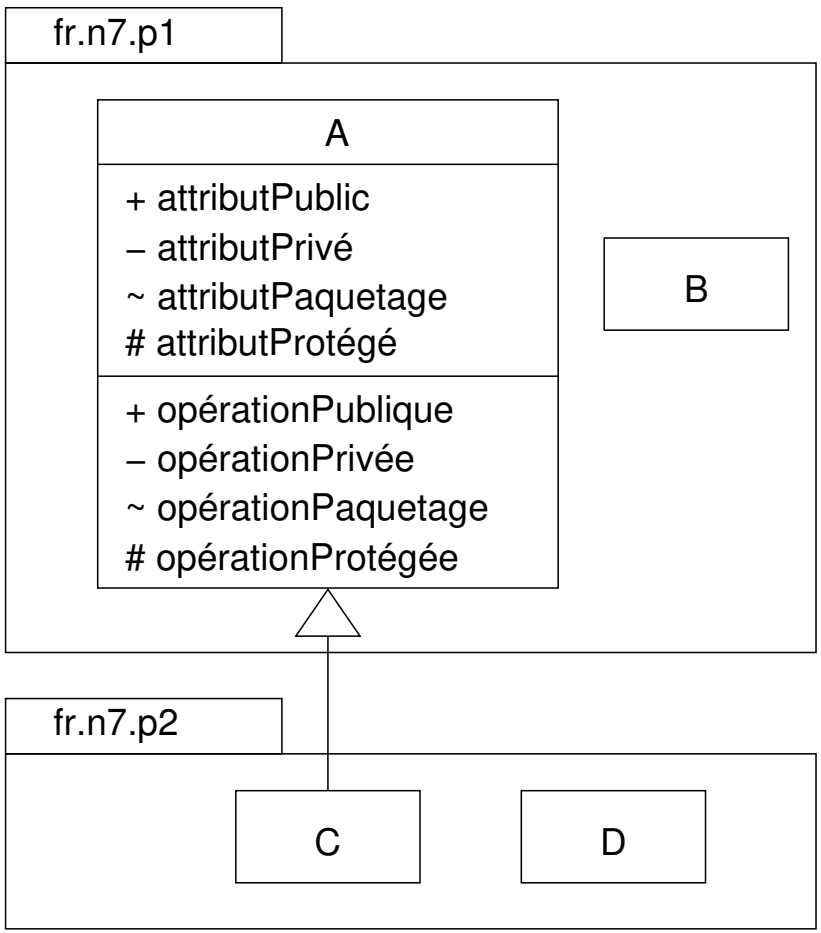
Chaque membre a un droit d'accès parmi les 4 niveaux suivants :

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

Accessible depuis une méthode définie dans	**Droit d'accès/Visibilité**			
	public	**protected**	**paquetage**	**private**
La même classe (A)	**oui**	**oui**	**oui**	**oui**
Une classe du même paquetage (B)	**oui**	**oui**	**oui**	**non**
Une sous-classe d'un autre paquetage (C)	**oui**	**oui**	**non**	**non**
Une autre classe d'un autre paquetage (D)	**oui**	**non**	**non**	**non**

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

Droits d'accès en UML et Java




```
package fr.n7.p1;

public class A {
  public    int attributPublic
  private   int attributPrivate
            int attributPaquetage
  protected int attributProtégé

  public    int opérationPublique  {...}
  private   int opérationPrivée    {...}
            int opérationPaquetage {...}
  protected int opérationProtégée  {...}
}
```

Exercice 12 Quels membres de A peuvent être utilisés depuis D ? Depuis C ? Depuis B ? Depuis A ?

Exemple d'erreur : droit d'accès insuffisant

```
1  class TestEquationErreur1 {
2      public static void main(String[] args) {
3          Equation eq = new Equation();
4          eq.coeffA = 1;
5      }
6  }
7  class Equation {
8      private double coeffA;
9      // ...
10 }
```

La compilation donne le résultat suivant :

```
TestEquationErreur1.java:4: error: coeffA has private access in Equation
        eq.coeffA = 1;
          ^
1 error
```

Remarque : Le compilateur Java sait que l'attribut `coeffA` est défini dans la classe `Equation` mais le droit d'accès n'est pas suffisant pour être utilisé dans `TestEquationErreur1`.

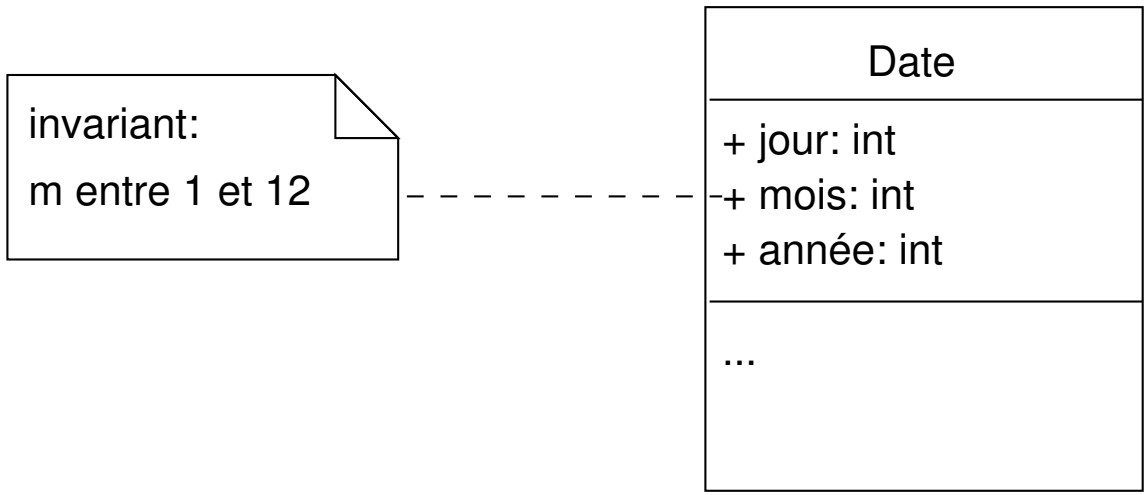
Règle sur le droit d'accès d'un attribut

Règle : Un attribut devrait toujours être déclaré **`private`** pour respecter :

- le principe de la protection en écriture et permettre ainsi à l'auteur de la classe de garantir l'intégrité des objets de la classe
- le principe d'accès uniforme : l'utilisateur n'a pas à savoir s'il accède à une information calculée ou stockée

Exercice 13 Définir une date avec jour, mois et année.
Indiquer comment faire pour changer la valeur du mois.
Indiquer comment faire pour obtenir la valeur du mois.

Modélisation initiale d'une date



- Est-il utile de mettre l'annotation (mois compris entre 1 et 12) ?
- **Principe :** Si un objet est dans un état incohérent, c'est une erreur qui incombe à l'auteur de la classe de cet objet.

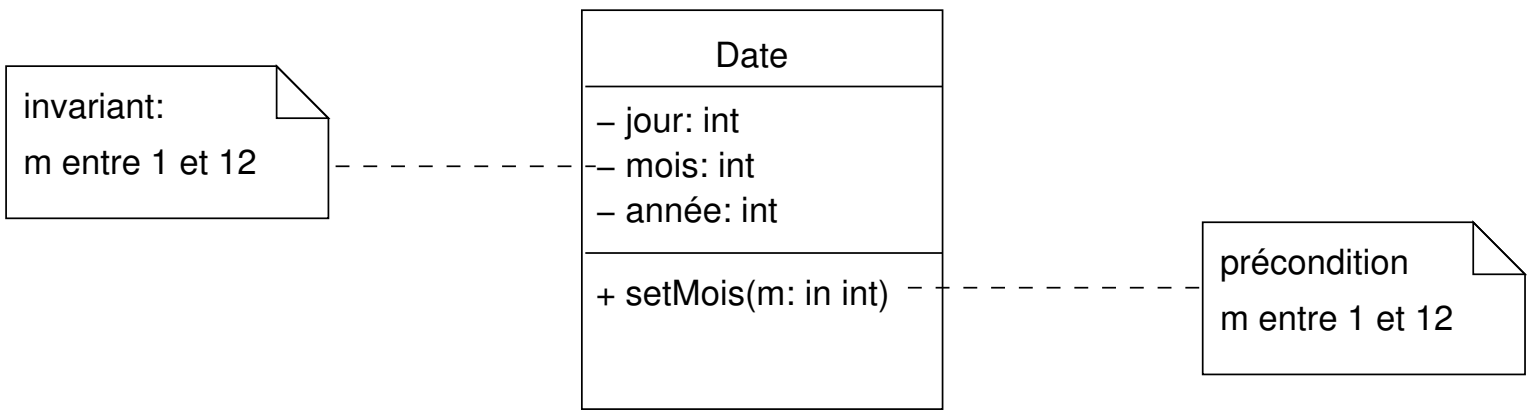
Principe de protection en écriture

```
class TestDateModifierMois {
    public static void main(String[] args) {
        Date d = new Date();
        ...
        d.mois = 15;    // 15 invalide !
        d.afficher();
    }
}
```

1. Que penser de ce programme ?
2. Comment faire pour éviter ceci ?
3. Que faut-il ajouter dans la classe Date ?

Solution

1. Le programme conduit à avoir un objet Date dans un état interdit (mois = 15) !
2. Il ne faut pas que l'utilisateur puisse directement modifier l'attribut.
   $\Longrightarrow$ L'attribut doit être privé (**`private`**)
3. L'intention du programmeur reste la même :
   - Il souhaite modifier le mois.
   - Une méthode doit exister. Elle s'assurera de la cohérence du nouveau mois.
   - Par convention, on l'appelle setMois.
   - C'est un modifieur.



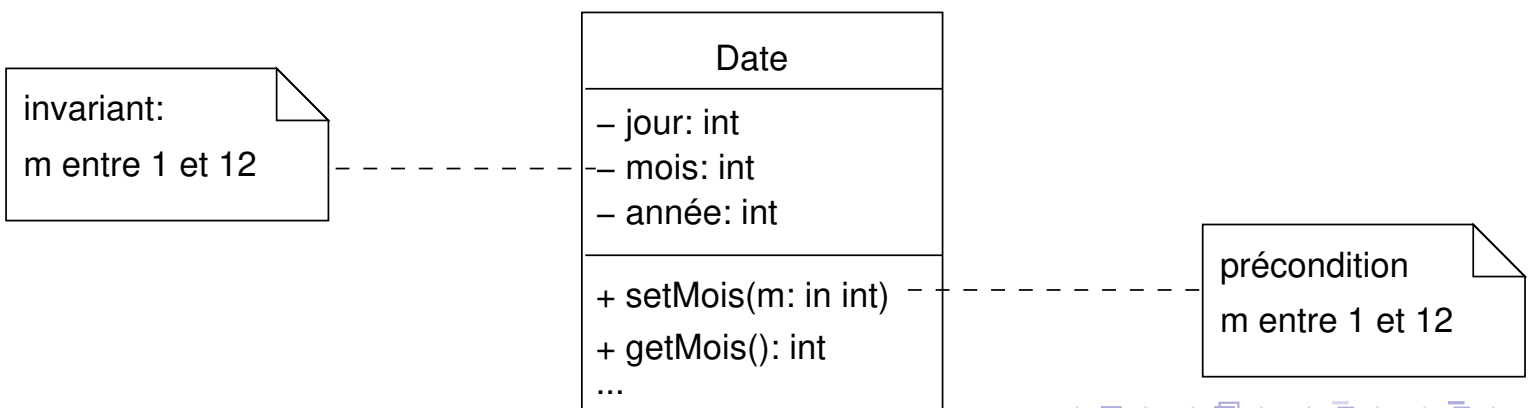
Principe de l'accès uniforme

```
class TestDateModifierMois {
    public static void main(String[] args) {
        Date d = ...;
        int leMois = d.mois;    // obtenir la valeur de mois
    }
}
```

1. On se contente de consulter la valeur de l'attribut
   $\Longrightarrow$ pas de risque d'incohérence.
2. Les opérations suivantes seraient utiles (nécessaires) sur la classe Date :
   - opérations de comparaison ($<, >, \leq, \geq$)
   - distance entre deux dates (en nombre de jour)
   - incrémenter une date
   - ajouter un nombre de jour à une date
   - ...
3. Comment faire pour les implanter de manière efficace ?
4. Quelles incidences sur le programme d'utilisation et la classe Date ?

Solution

- Si on conserve les trois attributs, les opérations sont :
  - peu pratiques à écrire
  - peu efficaces
- **Solution plus efficace :**
  - n'avoir qu'un seul attribut : le nombre de jours depuis une date de référence (nbJours)
- **Conséquences :**
  - la classe TestDateUtiliserMois ne compile plus : l'attribut n'existe pas !
  - il faut donc que la classe fournisse un moyen d'obtenir le mois
  - par une méthode appelée getMois() par convention. C'est un accesseur
  - l'accesseur aurait pu être défini sur la première version de la classe
    - la méthode retourne alors la valeur de l'attribut
  - il faut obliger l'utilisateur de Date à passer par l'accesseur et non l'attribut
  - l'attribut doit donc être déclaré privé (**`private`**)



Modularité

Une application modulaire favorise la **réutilisation** et l'**évolution** :

- un module peut être réutilisé dans l'application actuelle et les futures
- **principe de continuité** (B. Meyer) : une petite évolution du cahier des charges devrait se traduire par une évolution d'un module (ou d'un petit nombre de modules).

La modularité s'appuie sur deux concepts :

- **Encapsulation :** regrouper des informations logiquement liées.
  Exemple : ici les caractéristiques d'une date.
- **Masquage d'information :** certaines informations sont cachées (**`private`**) à l'utilisateur du module.
  Intérêt : Elles peuvent être changées sans impact sur les utilisateurs du module (évolution)
  B. Meyer : Tout devrait être privé par défaut.
  Définir quelque chose comme public doit être un acte volontaire et réfléchi !

Ces concepts ne sont pas spécifiquement objet et se retrouvent (devraient se retrouver) dans tous les langages de programmation.

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Paquetages
- Droits d'accès
- La classe Equation améliorée

La classe Equation (avec constructeurs et droits d'accès)

```
1  /** Equation du second degré avec deux solutions réelles.
2    * @author Xavier Crégut
3    */
4  public class Équation {
5
6      private double coeffA, coeffB, coeffC;  // coefficient de x2, x et x0
7      private double x1, x2;           // solution de l'équation
8
9      /** Initialiser une équation à partir de ses coefficients ... */
10     public Équation(double a, double b, double c) {
11         this.coeffA = a;
12         this.coeffB = b;
13         this.coeffC = c;
14     }
15
16     /** Initialiser une équation à partir de la somme
17     * et du produit de ses racines
18     * @param somme  somme des racines
19     * @param produit produit des racines
20     */
21     public Équation(double somme, double produit) {
22         this(1, -somme, produit);
23     }
```

La classe Equation (avec constructeurs et droits d'accès) (2)

```
25      /** Initialiser une équation à partir de la valeur de ses coefficients.
26      * @param a coefficient de x<sup>2</sup>
27      * @param b coefficient de x
28      * @param c coefficient constant
29      */
30      public void initialiser(double a, double b, double c) {
31          this.coeffA = a;
32          this.coeffB = b;
33          this.coeffC = c;
34      }
35
36      /** Obtenir le discriminant de l'équation.
37      * @return le discriminant de l'équation
38      */
39      private double delta() {
40          return this.coeffB * this.coeffB - 4 * this.coeffA * this.coeffC;
41      }
42
43      /** Déterminer les racines de l'équation du second degré.  */
44      public void résoudre() {
45          double delta = this.delta();
46          this.x1 = (- this.coeffB + Math.sqrt(delta)) / 2 / this.coeffA;
47          this.x2 = (- this.coeffB - Math.sqrt(delta)) / 2 / this.coeffA;
48      }
```

La classe Equation (avec constructeurs et droits d'accès) (3)

```
50      /** Afficher cette équation.  */
51      public void afficher() {
52          System.out.println(this);
53      }
54
55      /* Obtenir la représentation de cette équation sous forme
56      * d'une chaîne de caractères. */
57      public String toString() {
58          return this.coeffA + "*x2 + "
59                  + this.coeffB + "*x + "
60                  + this.coeffC + " = 0";
61      }
62
63  }
```

Remarque : Java n'impose aucun ordre sur les éléments d'une classe.
Les conventions de programmation préconisent l'ordre suivant : attributs, puis constructeurs, puis méthodes. Voir `http://www.oracle.com/technetwork/java/codeconvtoc-136057.html` et `https://www.securecoding.cert.org/confluence/display/java/Java+Coding+Guidelines`

Exercice 14 Indiquer les faiblesses de cette classe

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

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

Attributs et méthodes de classe

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 : le modifieur **static** indique qu'un attribut ou une méthode est de classe

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(...)`
NomClasse peut être omis quand on est dans cette classe.

Question : Que penser de la méthode principale ?

En UML : On souligne le membre de classe (ou on le préfixe par $).

Attribut de classe

Définition : Un attribut de classe est un attribut non spécifique à un objet donné mais 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 chaque 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++;
    ...                  // inutile de modifier les autres constructeurs
  }                      // s'ils s'appuient sur celui-ci : this(...)
}
```

Attention : Éviter les attributs de classe ! Comment compter deux sortes d'équations ?
Remarque : On peut créer des objets sans passer par un constructeur !

Initialisation des attributs de classe

- Les attributs de classe sont initialisés au chargement de la classe.
- Ils sont initialisés avec :
  - d'abord la valeur par défaut de leur type,
  - puis la valeur par défaut fournie par le programmeur,
  - et enfin grâce aux initialiseurs statiques (blocs d'instructions précédés de **`static`**).
- Exemple

```
class UneClasse {
  static int i;      // initialisé à 0 (valeur par défaut des int)
  static int j = 10; // valeur par défaut du programmeur
  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`**.
- **Attention :** Les attributs de classe limitent l'extensibilité (variable globale)

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.

```
public class Fraction {
  private int num;
  private int dén;
  private static int pgcd(int a, int b) {
    // ne manipule que a et b
    // et aucun membre d'instance
    // NE PEUT PAS UTILISER this !
  }
  ...
}
```

Méthode de classe et membres d'instance

```
1  class A {
2      private int i;       // attribut d'instance
3      static private int k;   // attribut de classe
4
5      public void m1() {  // méthode d'instance
6          this.i++;   // ou i++;
7          A.k++;      // ou k++; ou this.k++;
8      }
9
10     public static void m2() {   // méthode de classe
11         k++;
12         this.i++;
13         i++;
14     }
15
16     public static void m3(A a) {    // méthode de classe
17         m1();
18         this.m1();
19         m2();
20         a.k++;
21         a.i++;
22         new A().i++;
23     }
24 }
```

Le résultat de la compilation :

```
1  MethodeClasseAccesMembres.java:12: error: non-static variable this cannot be referenced
       from a static context
2                  this.i++;
3                  ^
4  MethodeClasseAccesMembres.java:13: error: non-static variable i cannot be referenced
       from a static context
5                  i++;
6                  ^
7  MethodeClasseAccesMembres.java:17: error: non-static method m1() cannot be referenced
       from a static context
8                  m1();
9                  ^
10 MethodeClasseAccesMembres.java:18: error: non-static variable this cannot be referenced
       from a static context
11                 this.m1();
12                 ^
13 4 errors
```

Bilan

Un attribut de classe :

- est accessible de n'importe où (variable globale),
- **MAIS** est souvent un frein pour l'évolution de l'application (par exemple s'il faut différencier l'attribut suivant les contextes).

$\Longrightarrow$ Éviter les attributs de classe !... Ou être conscient des limites induites !

Une méthode de classe :

- est accessible de partout (à partir de sa classe),
- permet de conserver le caractère symétrique de certaines méthodes (somme, égalité...),
- **MAIS** pas d'accès aux membres d'instance,
- **NI** redéfinition possible **NI** liaison dynamique (Voir interface et héritage)

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          out.println(sqrt(PI));
7          System.out.println(Math.sqrt(Math.PI));
8      }
9  }
```

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 des classes `Math` ou `java.awt.Color`.

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Quizz
- Types primitifs
- Algorithmique
- Tableaux
- Types énumérés

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Quizz
- Types primitifs
- Algorithmique
- Tableaux
- Types énumérés

Quizz

Remarque : Ce questionnaire a pour but de vous motiver à lire les transparents qui suivent...

1. Pourquoi l'instruction suivante est refusée par le compilateur ?

```
float x = 4.21;
```

2. Comment est évaluée l'instruction suivante (ajouter les parenthèses) :

```
char c = ...;
boolean estChiffre = c >= '0' && c <= '9';
```

3. Indiquer les 5 erreurs du code suivant :

```
/** Retourne la position de x dans tab, -1 si non présent... */
boolean indiceDe(int tab[], int x) {
    int indice = 1;
    while  tab[indice] != x && indice <= tab.length {
        indice++;
    }
    return tab[indice] == x ? indice : -1;
}
```

Quizz (2)

4. Est-ce que les entiers sont compatibles avec les booléens ?
5. Quelle est la valeur de `"A" + 3` ? Et `'A' + 3` ?
6. Comment convertir un réel en un entier ?
7. Comment convertir une chaîne (par exemple "421") en l'entier correspondant (421) ?
8. Comment définir une constante ?
9. Indiquer les valeurs successives de n :

```
int n = 5;
n += 3;
n += 2.5;
```

10. Comment représenter un **SinonSi** ?
11. Comment représenter un **Répéter ... JusquÀ** ?
12. Avec un **for**, connait-on *a priori* le nombre d'itérations ?

Quizz (3)

13. Qu'affichent les instructions suivantes :

```
int[] nombres = { 3, 5, 7 };
for (int n : nombres) {
    System.out.println(n);
}
```

14. Définir un type qui représente les états possibles d'une lampe : éteinte, allumée, clignotante.
15. Que signifient les notions d'« égalité physique » et « égalité logique » ?
16. Est-ce que les objets de la classe Equation sont immuables ?

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Quizz
- Types primitifs
- Algorithmique
- Tableaux
- Types énumérés

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.
- Entiers et caractères sont compatibles (attention à 3 + 'a').
- entiers et booléens ne sont pas compatibles

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 : conversion (coercition)
float x2 = 4.21;       // Erreur de compilation : 4.21 est un double !
float x2 = 4.21f;      // OK : 4.21f un float
int n = (int) 4.21;    // conversion explicite
```

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Quizz
- Types primitifs
- Algorithmique
  - Méthode principale
  - Instructions
  - Opérateurs
  - Structures de contrôle
- Tableaux
- Types énumérés

La méthode principale


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

- En Java, tout doit être défini à l'intérieur d'une classe.
- Pour l'instant, nous nous limitons à la *méthode principale : `main`*.
- Tous les mots sont importants : seul l'identificateur (args) peut être changé.

Variables et constantes

- Exemples :

```
final int MAX = 10;                  // une constante entière !
double rayon = 0;                    // le rayon du cercle
double perimetre = 2 * PI * rayon;       // le périmètre du cercle
int a, b;                            // valeurs saisies au clavier
int x = 5, y = x * x, z;             // dès que la variable est déclarée,
                                     // on peut l'utiliser
```

- Ne déclarer qu'une variable par ligne pour pouvoir expliquer son rôle.
  La dernière déclaration est un exemple à ne pas suivre !
- Aligner les commentaires pour une meilleure lisibilité
- **Conseil :** Ne déclarer une variable que quand on sait l'initialiser.
- Le mot-clé **`final`** signifie que la variable ne pourra être réinitialisée
  $\Longrightarrow$ c'est donc une constante !

Affectation

- L'affectation est considérée comme un opérateur.
  Le résultat est la valeur affectée
  $\Longrightarrow$ on peut considérer l'affectation comme une expression. **À éviter !**
- Exemple :

```
a = b = 0;  // équivalent à a = (b = 0);    et b = 0; a = b;
```

Bloc

- Exemple :

```
{
    int memoire = a;
    a = b;
    b = memoire;
}
```

- Un bloc permet de grouper plusieurs instructions entre accolades pour les considérer comme une seule.
  $\Longrightarrow$ Utile pour les structures de contrôle (voir T. 113) !
- Un bloc délimite la portée d'une variable locale (voir T. 100).

Portée et durée de vie d'une variable

Portée d'une variable locale

- Portion du texte du programme où est accessible une variable locale.
- Commence avec la déclaration de la variable.
- Se termine avec l'accolade fermante du bloc de la déclaration de la variable.

Durée de vie d'une variable

- La durée de vie d'une variable locale coïncide avec sa portée.

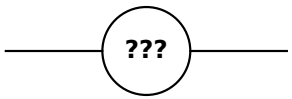
Règles :

- Une variable locale ne peut pas porter le même nom qu'une variable locale d'un bloc englobant.

Illustration de la portée

```
1   {
2       int i = 1;
3       int k = 10
4       double k = 20;      // interdit !  Comment distinguer les deux k ?
5       {
6           int j = 3;
7
8           // on a : i == 1 && j == 3;
9
10          {
11              int i = 4;  // interdit ! i utilisé dans un bloc englobant
12          }
13      }
14      // ici, pas de j accessible (j == 3)
15
16      int j = 5;  // autorisé : pas de j dans un bloc englobant
17
18      // on a : i == 1 && j == 5;
19  }
```

L'instruction Écrire




```
System.out.print("Bonjour␣");
System.out.println(nom);
System.out.println("prix␣:␣" + quantite * prixUnitaire);
System.out.println("somme␣:␣" + (a + b));   // Attention aux ( )
System.out.printf("Le␣capitaine␣%s␣a␣%d␣ans.\n", nom, age);
```

- Écrire l'expression sur la sortie standard (`out`) ou la sortie en erreur (`err`).
- `println` se comporte comme `print` mais ajoute en plus un retour à la ligne
- On peut écrire toute expression (elle est convertie en chaîne de caractères).
- L'opérateur de concaténation (+) facilite l'affichage de plusieurs expressions
- Il existe aussi printf qui fonctionne avec un format (comme en C)

L'instruction return


```
int somme(int a, int b) {
    return a+b;
}
```

- interrompt l'exécution d'une méthode
- l'expression est la valeur retournée par la méthode
- règles qu'il est conseillé de respecter :
  return devrait toujours être la dernière instruction d'une méthode non **void**.
- voir aussi T. 200

L'instruction assert


```
assert a > 0;
assert b > 0 : "b non positif: " + b;
```

- permet de vérifier une propriété lors de l'exécution du programme
- à utiliser pour traduire les commentaires algorithmiques `{ ... }`
- après les « `:` », on peut donner une explication
- les `assert` ne sont pas évalués par défaut !
  Il faut activer l'option `-ea` (*enable assertions*)

Opérateurs

Opérateurs relationnels

```
==   // égalité                          3 == 3
!=   // différence (non égalité)         4 != 5
>    // plus grand que                   4 > 3
>=   // plus grand que ou égal           4 >= 3
<    // plus petit que                   3 < 4
<=   // plus petit que ou égal           3 <= 4
```

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

Opérateurs arithmétiques

```
+ -  // addition et soustraction (op. binaires)   10 + 5 == 15
* /  // multiplication et division               10 / 3 == 3
%    // modulo : le reste de la division entière  10 % 3 == 1
+    // opérateur unaire                          +10 == 10
-    // opérateur unaire (opposé de l'opérande)   -10
```

Opérateurs (2)

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
```

Intérêt : comment s'évalue `(n != 0) && (s / n >= 10)` ?

Formulations équivalentes : (la seconde est préférable)

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

Opérateurs (3)

Opérateur de concaténation

```
+   concaténation des chaînes de caractères
          "x" + 3   <==>   "x3"
          3 + "x"   <==>   "3x"
          3 + '0'   <==>   51     un entier !
     "" + 3 + '0'   <==>   "30"   une chaîne !
```

Toute expression Java peut être considérée comme une chaîne de caractères !

Opérateur conditionnel (si arithmétique)

```
condition ? valeur_vrai : valeur_faux
```

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

```
System.out.println(age >= 18 ? "majeur" : "mineur")
```

Attention : Peut être difficile à lire !

Opérateurs (4)

Opérateurs sur les bits

```
expr1 & expr2 // ET bit à bit                  14 ...00001110
              //      14 & 7 == 6               7 ...00000111
              //                                & ...00000110
expr1 | expr2 // OU bit à bit
              //      14 | 7 == 15              | ...00001111
expr1 ^ expr2 // XOR bit à bit
              //      14 ^ 7 == 9               ^ ...00001001
~ expr1       // inverse les bits de l'opérande
              //      ~7 == -8                  ~ 1..11111000
expr << nb    // décalage de expr à gauche de nb bits (* 2^nb)
              //      3 << 2 == 12               -3 << 2 == -12
expr >> nb    // décalage de expr à droite de nb bits (/ 2^nb)
              //      12 >> 2 == 3               -12 >> 2 == -3
expr >>> nb   // idem >> (sans préservation du signe)
              //      12 >>> 2 == 3              -12 >>> 2 == 1073741821
```

Affectation contractée

Affectation contractée

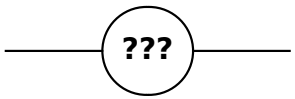
L'opérateur d'affectation peut être combiné avec la plupart des opérateurs :

```
x += y    /*  x = x + y  */      |    double x = 1.5;
x -= y    /*  x = x - y  */      |    int n = 0;
x %= y    /*  x = x % y  */      |    n += x;
x |= y    /*  x = x | y  */      |    // possible ?
...                              |    // valeur de n ?
```

Attention : Si x est une variable déclarée du type T et # un opérateur, on a :

x #= y est équivalent à (T)((x) # (y))

Opérateurs ++ et -- (pré/post)(in/dé)crémentation



- **Évaluation :** augmenter (++) ou diminuer (--) de 1 la valeur d'une variable.
  - **placé avant la variable :** il est évalué juste avant l'instruction
  - **placé après la variable :** il est évalué juste après l'instruction
- **Exemples :**

```
1       int x = 8, y = 15;
2       x++;          // incrémenter x
3       ++x;          // idem
4       // x == 10 && y == 15
5       int z = x++ + ++y;  // Lisible ?  Attention aux espaces !
6       // x == 11 && y == 16 && z == 26
7       int t = ++x + x--;  // intérêt ?
8       // x == 11 && t == 24
```

- **À utiliser avec modération !**

Exemple à ne pas suivre

```
public class PreIncrementation {
        public static void main(String[] args) {
                int x = 2;
                int a, b;
                int y =  (a = ++x) + (b = ++x);
                // valeurs de a ? b ? x ? y ?
        }
}
```

- Ne pas abuser des ++ et - -
- Ne pas utiliser l'affectation comme expression
- Écrire du code lisible !

Priorité et associativité des opérateurs

1G	[]	accès à un tableau
	()	
	.	p.x ou p.m()
2D	++ --	post-incrémentation et post-décrémentation
3D	++ --	pré-incrémentation et pré-décrémentation
	+ - ! ~	unaires
	(type)	transtypage
	new	
4G	* / %	
5G	+ -	
6G	<< >> >>>	
7G	< <= > >=	instanceof
8G	== !=	
9G	&	
10G	^	
11G	|	
12G	&&	
13G	||	
14D	?:	
15D	= += -= *= *= /= %= &= ^= |= <<= >>= >>>=	

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

Évaluation de : `n != 0 && s / n >= 10`

Conditionnelles **Si** : if

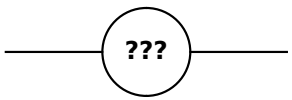
- `expression` est nécessairement une expression booléenne
- Mettre des { } si on veut plusieurs instructions.
- **Règle :** Toujours mettre les accolades !
- Utiliser **`else if`** pour faire un **SinonSi**

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

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

```
if (n1 == n2) {
    resultat = "égaux";
} else if (n1 > n2) {
    resultat = "plus grand";
} else {
    resultat = "plus petit";
}
```

Choix multiples




```
switch (<expression>) {
case <expr_cste_1>:
    <instructions_1>;
    break;
...
case <expr_cste_n>:
    <instructions_n>;
    break;

default:
    <instruction>;
}
```

```
switch (c) { // c caractère
case 'o':
case 'O':
    res =  "Affirmatif";
    break;

case 'n':
case 'N':
    res =  "Négatif";
    break;
default:
    res =  "!?!?!?!?";
}
```

- **Principe :** L'expression est évaluée et l'exécution continue à la première instruction qui suit la 1reexpression 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.

Choix multiples (2)

- **Conseil :** Mettre un **`break`** après chaque groupe d'instructions d'un **`case`**.
- **Java7 :** On peut faire un **`switch`** sur le type String.

Répétitions **TantQue** : 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.

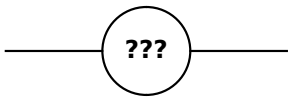
Répétitions **Répéter** : do ... while


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

```
java.util.Random generateur
        = new java.util.Random();
int nb;
do {
    nb = generateur.nextInt(100);
    System.out.println("nb = " + nb);
} while (nb <= 90);
```

- **Sémantique :** `<instructions>` est exécutée puis, tant que `<condition>` est vraie, `<instructions>` est exécutée.
- **Remarque :** `<instructions>` est exécutée au moins une fois.
- **Attention :** dans le **`while`**, on ne peut pas utiliser les variables déclarées dans les accolades du **`while`**

Répétitions for (Pour ?)




```
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> (incrémentation) sont exécutées.

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

- **Conséquence :** Une variable déclarée dans <init> n'est visible que dans le bloc du **for** (cas du **int** i = 1, par exemple).
- **Attention :** Ce n'est donc pas un **Pour** algorithmique ! Voir **foreach** T. 127

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.

```
1  première: for (int i = 0; i < 7; i++) { // for étiquetté « première »
2      System.out.println("i = " + i);
3      for (int j = 0; j < 4; j++) {
4          System.out.println("    j = " + j);
5          if (j == i) {
6              continue première;          // ==> passer au i suivant !
7          }
8      }
9  }
10 System.out.println("Fin !");
```

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Quizz
- Types primitifs
- Algorithmique
- Tableaux
- Types énumérés

Déclaration et création d'un tableau

- Java n'a pas la sémantique de la valeur mais la **sémantique de la référence** !
  - le tableau représente les données en mémoire
  - il est accessible à travers une variable appelée *poignée*
- **Création d'un tableau :**

```
new Type[capacité]
```

- **Déclaration d'une poignée sur tableau :**

```
Type[] tab;     // forme à préférer
Type tab[];     // autre forme (héritée du langage C)
```

- En général, on fait les deux ensembles :

```
Type[] tab = new Type[capacité];
```

- Exemple :

```
int[] nb = new int[10];
final int MAX = 10;
double[] valeurs = new double[MAX];
```

Caractéristiques d'un tableau

- 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 sont entre 0 (premier élément) et `length-1` (dernier élément)
  indice invalide $\Longrightarrow$ erreur signalée à l'exécution : `ArrayIndexOutOfBoundsException`

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

- La valeur **`null`** indique qu'aucun tableau n'est associé à une poignée.
  Attention : interdit d'utiliser les [] sur une poignée **`null`** (`NullPointerException`)

  - ```
    int[] tab1 = new int[10];
    int[] tab2 = new int[20];
    tab1 = tab2;    // Quel sens ?  Que vaut tab1.length ?
    ```

- Pour copier les éléments, voir `System.arraycopy(...)`.
    ```

Autres caractéristiques d'un tableau

- Quand un tableau est créé, chaque case est initialisée avec la valeur par défaut de son type.
- Il est possible d'initialiser explicitement le tableau :

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

- Les tableaux ne peuvent pas être redimensionnés : `length` est une constante
  $\Longrightarrow$ Voir structures de données (collections)
- À l'exception des tableaux de caractères, `System.out.println` n'est pas très lisible.

```
System.out.println("nomJours = " + nomJours);
--> nomJours = [Ljava.lang.String;@bd0108
```

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          }
6  }
```

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;
```

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  }
```

Tableaux à plusieurs dimensions : utilisation

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

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

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

```
1       assert 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               if (mat[i].length > 0) {
7                   System.out.print(mat[i][0]);
8                   for (int j = 1; j < mat[i].length; j++) {
9                       System.out.print(",␣" + mat[i][j]);
10                  }
11              }
12              System.out.println();
13          }
14      }
```

Pour Chaque (**foreach**)

```
for (<type> <var> : <tab>) {
    <instructions>;
}
```

```
public static void main(String[] args) {
  for (String nom : args) {
    System.out.println("Bonjour␣" + nom);
  }
}
```

- Correspond à « **Pour Chaque** <var> **Dans** <tab> » (foreach ... in ...).
- **Sémantique :**
  - <tab> est un *tableau*, une *collection*... (en fait un « Iterable »).
  - Les instructions sont exécutées pour <var> prenant chaque valeur de <tab>.
- **Avantage :** Écriture simple conservant la sémantique du **Pour** algorithmique.
- **Conseil :** Déclarer la variable de boucle avec **final**.
  En effet, changer sa valeur n'aurait pas d'incidence sur la boucle !

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
- Quizz
- Types primitifs
- Algorithmique
- Tableaux
- Types énumérés

7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

Les énumérations

```
1  public enum Fruit { POMME, POIRE, ORANGE, PRUNE };
```

```
1  public enum Couleur { JAUNE, VIOLET, ORANGE };
```

```
1  class FruitMain {
2      public static void main(String[] args) {
3          Fruit[] fruits = { Fruit.POMME, Fruit.POIRE, Fruit.ORANGE, Fruit.PRUNE };
4                  // voir Fruit.values()
5          for (Fruit f : fruits) {
6              // Afficher une couleur possible pour f
7              // Déterminer la couleur de f
8              Couleur couleur = null;
9              switch (f) {
10                 case POMME: couleur = Couleur.JAUNE; break;
11                 case ORANGE: couleur = Couleur.ORANGE; break;
12                 case PRUNE: couleur = Couleur.VIOLET; break;
13             }
14
15             // Afficher le fruit et sa couleur
16             System.out.println(f + "␣est␣" + couleur);
17         }
18     }
19 }
```

Quelques commentaires

- Un type énuméré (comme une classe) est défini dans un fichier qui porte son nom (`Fruit.java`, `Couleur.java`...)
- **Convention :** Les valeurs d'un type énuméré sont notées en majuscules.
- Le même nom (`ORANGE`) peut être utilisé dans deux énumérations.
- Utiliser le tableau `fruits` est maladroit ! Voir `Fruit.values()`
- On doit toujours préfixer la valeur d'un type énuméré par son type (sauf dans les **`case`** d'un **`switch`**).
- Résultat de l'exécution :

```
POMME est JAUNE
POIRE est null
ORANGE est ORANGE
PRUNE est VIOLET
```

- On peut afficher une valeur d'un type énuméré.
- Un type énuméré est une classe.
  $\Longrightarrow$ La suite sera plus facile à comprendre plus tard !

Principales caractéristiques

- Le type énuméré est une classe qui propose les méthodes de classe :
  - values() : le tableau des valeurs du type (dans l'ordre de déclaration)
  - valueOf(String nom) : obtenir la valeur du type à partir de son nom

```
Couleur d = Couleur.valueOf("VIOLET");
```

mais aussi des méthodes d'instance :

  - ordinal() : le numéro d'ordre d'une valeur du type

```
Couleur.JAUNE.ordinal()          // 0
Couleur.VIOLET.ordinal()         // 2
```

  - name() : le nom de la valeur dans le programme Java

```
Couleur.JAUNE.name()    // "JAUNE"
```

  - compareTo(autre) : relation d'ordre (ordre de déclaration dans la classe). Retourne un entier dont le signe caractérise this - autre.

```
Couleur.ORANGE.compareTo(Couleur.VIOLET)        // 1
        // ORANGE - VIOLET > 0 donc ORANGE > VIOLET
d.compareTo(Couleur.VIOLET)     // 0
```

- Chaque valeur du type énuméré est l'équivalent d'une constante de classe.

Intérêt

- Évite d'avoir à définir des constantes (entières, caractères...) et expliciter une convention de codage.

```
final int JAUNE = 0;
final int VIOLET = 1;
final int ORANGE = 2;
```

- Il est impossible de définir de nouvelles valeurs (le type est fermé).
- Contrôle de type fort (à la compilation !)
  - Deux types énumérés différents ne sont pas compatibles !
  - Les constantes ne sont pas compilées dans le code client
- Ce sont des classes : possibilité d'ajouter attributs et méthodes.

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Documentation
- Requêtes et commandes
- Définir une classe
- Attribuer un SP à une classe
- Test unitaire avec JUnit
- Programmation par contrat
- Unified Modeling Language (UML)

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Documentation
- Requêtes et commandes
- Définir une classe
- Attribuer un SP à une classe
- Test unitaire avec JUnit
- Programmation par contrat
- Unified Modeling Language (UML)

Documentation avec javadoc

- **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.
- **Justification :** Limiter les risques d'incohérence.
- **Moyen :**
  - des commentaires spécifiques `/** */` placés *devant* l'élément à documenter
  - C'est bien entendu au programmeur de fournir les éléments de la documentation !
- **Production de la documentation :** l'outil javadoc du JDK
  - `javadoc` engendre la documentation des classes 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 favorise (mais ne garantit pas !) sa cohérence.
  - javadoc permet une présentation standardisée de la documentation.
- **Attention :** Par défaut, seules les informations publiques sont documentées.

Commentaires de documentation

Les commentaires de documentation `/** ... */` peuvent contenir :

- des étiquettes spécifiques à `javadoc` :
  - elles commencent par `@`
  - exemples : `@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, un constructeur, etc.
 */
```

- **Attention aux pléonasmes !**
  - Inutile de dire qu'un attribut est un attribut ! Il faut donner son rôle.
  - Inutile de dire qu'une méthode est une méthode, il faut donner son objectif, ses conditions d'utilisation, son effet, le rôle des paramètres (@param), la signification de la valeur retournée (@return)...
- Voir « How to Write Doc Comments for the Javadoc Tool »

  `http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html`

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

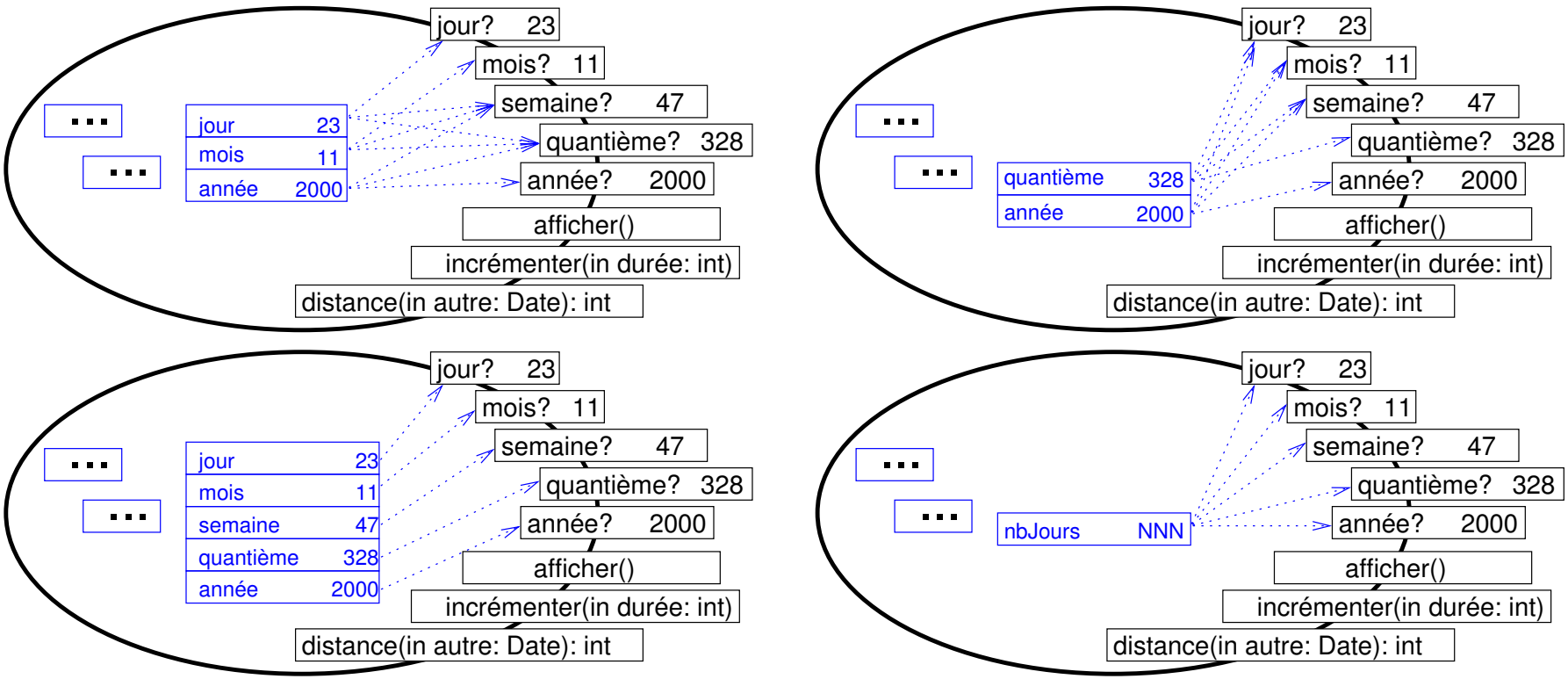
- Documentation
- Requêtes et commandes
- Définir une classe
- Attribuer un SP à une classe
- Test unitaire avec JUnit
- Programmation par contrat
- Unified Modeling Language (UML)

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 l'implanter.
- 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 (conservent 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.
- requêtes et commandes se traduisent en méthodes
- une requête correspond à une information stockée (attribut) ou calculée.
- toute méthode publique, non **`void`** et sans effet de bord correspond à une requête.

Exemple : la classe Date



- La vue utilisateur est toujours la même.
- La vue programmeur (interne) change !

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Documentation
- Requêtes et commandes
- Définir une classe
- Attribuer un SP à une classe
- Test unitaire avec JUnit
- Programmation par contrat
- Unified Modeling Language (UML)

La classe Compteur

Exercice 15 : 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.

15.1 Modéliser en utilisant la notation UML la classe Compteur.

15.2 Écrire un programme de test de la classe Compteur.

15.3 Écrire en Java la classe Compteur.

15.4 Comment être sûr que la valeur du compteur est toujours positive ?

1. Définir la vue utilisateur

Compteur
requêtes
valeur : int
commandes
raz
incrémenter
set(valeur : int)

```
public class Compteur {

    public int getValeur()      { ... }

    public void raz()           { ... }

    public void incrémenter()   { ... }

    public void set(int valeur) { ... }
}
```

- Chaque requête et chaque commande devient une méthode
- La requête `valeur` devient soit `getValeur()`, soit `valeur()`
- On sait donc comment utiliser un compteur !

2. Définir le(s) constructeur(s)

Compteur
requêtes
valeur : int
commandes
raz incrémenter set(valeur : int)
constructeurs
Compteur(valeur : int)

```
public class Compteur {

    public Compteur(int valeur) { ... }

    public int getValeur()      { ... }

    public void raz()           { ... }

    public void incrémenter()   { ... }

    public void set(int valeur) { ... }

}
```

- On sait donc aussi créer un compteur.

3. Programme de test

```
1   class TestCompteur {
2       public static void main(String[] args) {
3           Compteur c = new Compteur(10);
4           assert c.getValeur() == 10;
5           c.incrémenter();
6           assert c.getValeur() == 11;
7           c.raz();
8           assert c.getValeur() == 0;
9           c.set(5);
10          assert c.getValeur() == 5;
11      }
12  }
```

- Compiler : `javac ExempleCompteur.java`
- Exécuter : `java -ea ExempleCompteur` (*ea* comme *enable assertions*)
- On exécute avant d'écrire l'implantation de la classe Compteur !
- Meilleure technique : utiliser JUnit !

4. Définir la vue programmeur : choisir les attributs

Compteur
– valeur : int
+ getValeur() : int + raz + incrémenter + set(valeur : int)
+ Compteur(valeur : int)

```
public class Compteur {
    private int valeur;

    public Compteur(int valeur) { ... }

    public int getValeur()      { ... }

    public void raz()           { ... }

    public void incrémenter()   { ... }

    public void set(int valeur) { ... }
}
```

- Il ne reste plus (!) qu'à écrire le code des méthodes.

5. Écrire le code

```java
/** Définition d'un compteur avec incrémentation.
 * @author      Xavier Crégut
 * @version     Revision : 1.4 */
public class Compteur {
    private int valeur;         // valeur du compteur

    /** Initialiser un compteur à partir de sa valeur initiale.
     * @param valeurInitiale valeur initiale du compteur
     */
    public Compteur(int valeurInitiale) { this.valeur = valeurInitiale; }

    /** Augmenter d'une unité le compteur */
    public void incrementer()           { 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; }
}
```

Comment définir une classe

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

1. Définir la spécification d'un point de vue utilisateur :
   - Spécifier les requêtes
   - Spécifier les commandes
2. Préparer les tests
3. Spécifier les constructeurs
4. Choisir une représentation (choix des attributs)
   Les attributs sont définis **`private`**.
5. 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.
6. Tester au fur et à mesure !

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Documentation
- Requêtes et commandes
- Définir une classe
- Attribuer un SP à une classe
- Test unitaire avec JUnit
- Programmation par contrat
- Unified Modeling Language (UML)

Raffinage et sous-programmes, classes et méthodes

- Conserver la méthode des raffinages est possible
- Chaque étape peut être un sous-programme, avec sa signature
- Le type des paramètres de la signature peuvent devenir des classes
- L'étape (le sous-programme) peut devenir une méthode de toute classe qui
  - est type d'un de ses paramètres
  - ce paramètre étant en `in` ou `in out`
  - ce paramètre devient implicite (**`this`**)
- Si plusieurs paramètres sont de type classe :
  - possibilité d'avoir plusieurs méthodes, une par classe
  - souvent l'une appelle l'autre (attention à la récursivité mutuelle)
  - voir définition des responsabilités.
- D'autres méthodes peuvent être ajoutées sur les classes identifiées pour favoriser la réutilisation

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Documentation
- Requêtes et commandes
- Définir une classe
- Attribuer un SP à une classe
- Test unitaire avec JUnit
- Programmation par contrat
- Unified Modeling Language (UML)

Pourquoi écrire un programme de test ?

Écrire des programmes de test permet de :

- spécifier le comportement attendu
  - d'une méthode,
  - d'une classe,
  - d'une application...
- trouver des erreurs dans le code écrit
- donner confiance dans le programme écrit : pour soi, pour le client !
- rejouer les tests quand le code évolue (maintenance évolutive ou corrective)
- ...

Principe du test

- Principe des trois A :
  - acteur : objet sur lequel le test porte
  - action : appeler la méthode à tester sur l'acteur
  - assertions : vérifier que l'action a eu l'effet escompté

```
@Test public void testIncrementer() {
    Compteur c = new Compteur(15);          // acteur
    c.incrementer();                        // action
    assertEquals(16, c.getValeur());        // assertion
}

@Test public void testRaz() {
    Compteur c = new Compteur(15);          // acteur
    c.raz();                                // action
    assertEquals(0, c.getValeur());         // assertion
}
```

  - Chaque méthode décrit un test

Factoriser les acteurs

- Plusieurs tests peuvent utiliser les mêmes acteurs
  $\Longrightarrow$ factoriser leur création pour les différents tests
- Les acteurs deviennent des attributs
- Une méthode permet de décrire la création des acteurs
  - cette méthode est annotée (`@Before`) et souvent appelée `setUp`
- Cette méthode est exécutée avant chaque test
- On peut bien sûr écrire un test qui n'utilise pas d'acteur

```
class CompteurTest {
    protected Compteur c;

    @Before public void setUp() {
        this.c = new Compteur(15);
    }

    ...
}
```

Classe de test JUnit4 pour la classe Compteur

```
1  import org.junit.*;
2  import static org.junit.Assert.*;
3
4  public class CompteurTest {
5
6     protected Compteur c;
7
8     @Before public void setUp() {
9        this.c = new Compteur(15);
10    }
11
12    @Test public void testIncrementer() {
13       c.incrementer();
14       assertEquals(16, c.getValeur());
15    }
16
17    @Test public void testRaz() {
18       c.raz();
19       assertEquals(0, c.getValeur());
20    }
21
22    @Test public void testSet() {
23       c.set(5);
24       assertEquals(5, c.getValeur());
25    }
26
27    @Test public void testPlus() {
28       Compteur cptr = new Compteur(0);
29       cptr.incrementer();
30       assertEquals(1, cptr.getValeur());
31       cptr.raz();
32       assertEquals(1, cptr.getValeur());
33    }
34
35 }
```

- JUnit 4 utilise les annotations (`@Test`, `@Before`)
- JUnit 3 s'appuie sur des conventions de nommage (test*, setUp, etc.).

Exécution d'une classe de test JUnit

Une classe de test n'est pas directement exécutable par la machine virtuelle (pas de main).
Pour exécuter une classe de test JUnit4, on fait :

```
java org.junit.runner.JUnitCore CompteurTest
```

On obtient alors :

```
JUnit version 4.12
...E.
Time: 0,004
There was 1 failure:
1) testPlus(CompteurTest)
java.lang.AssertionError: expected:<1> but was:<0>
        at org.junit.Assert.fail(Assert.java:88)
        at org.junit.Assert.failNotEquals(Assert.java:834)
        at org.junit.Assert.assertEquals(Assert.java:645)
        at org.junit.Assert.assertEquals(Assert.java:631)
        at CompteurTest.testPlus(CompteurTest.java:32)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at org.junit.runner.JUnitCore.main(JUnitCore.java:36)

FAILURES!!!
Tests run: 4,  Failures: 1
```

Question : Comment interpréter la trace d'exécution d'un test en erreur ?
Remarque : JUnit n'est pas une bibliothèque standard de Java, il faut l'ajouter au CLASSPATH...

Pourquoi utiliser un environnement de test de type JUnit

- **Écrire relativement facilement les tests :**
  - attributs pour les acteurs
  - méthode d'initialisation (`@Before`)
  - méthodes pour les tests (`@Test`)
- **Automatiser l'oracle** (décider si le test réussit ou échoue)
  - Ce sont les assertions qui jouent le rôle d'oracle.
- **Formaliser et capitaliser les tests** (sous forme de sous-programmes)
- **Exécuter automatiquement les tests :**
  - non régression,
  - développement en continu
- **Obtenir des statistiques** sur les tests lancés
- ...

Comment écrire un programme de test ?

- **Réfléchir aux données de test** (et factoriser leur initialisation : `@Before`)
- **Définir un ou plusieurs de tests pour chaque méthode.**
  - chaque méthode de test doit tester un aspect (une exigence).
    $\Longrightarrow$ Être capable de donner l'objectif du test (nom de la méthode, commentaire)
  - Vérifier que chaque exigence du cahier des charges est traduite par au moins un test
    - construire une matrice de traçabilité (exigences/tests)
  - Ajouter des tests (structurels) pour atteindre le critère de taux de couverture souhaité
- **Ne pas oublier la précision pour l'égalité des réels**
  - Trois paramètres pour `assertEquals` sur les réels :

```
assertEquals(double attendue, double réelle, double précision);
```

  - L'instruction suivante détectera toujours une erreur :

```
assertEquals(10, 10.0); // forcément une erreur
```

  - Car elle est interprétée comme :

```
assertEquals(new Integer(10), new Double(10.0));
             // erreur car objets différents
```

Comment identifier les tests

- **Tests fonctionnels :**
  - issus de la compréhension du cahier des charges / du problème posé
  - matrice de traçabilité : exigence / tests correspondants
- **Tests structurels :**
  - utiliser des techniques de couverture de code pour compléter les tests fonctionnels
    - exécution de chaque instruction une fois au moins
    - exécution de chaque décision une fois au moins
    - exécution de chaque itération de 0 à p fois
    - ...
  - Utiliser les valeurs des conditions : cas limites
  - ...
- **Lors d'un plantage du programme**
  - le programme plante, il manquait un test pour l'identifier
  - il faut trouver l'erreur puis écrire un programme de test qui la reproduit
  - ce sera un bon test puisqu'il détecte une erreur réelle
  - utile pour les tests de non régression

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Documentation
- Requêtes et commandes
- Définir une classe
- Attribuer un SP à une classe
- Test unitaire avec JUnit
- Programmation par contrat
  - Motivation
  - Programmation par contrat
  - Java Modeling Language (JML)
  - La clause assert
- Unified Modeling Language (UML)

Motivation

Soit une classe Fraction définie par son numérateur et son dénominateur :

```
1  public class Fraction {
2      public int numérateur;
3      public int dénominateur;
4      public Fraction(int num, int dén)   { ... }
5      public void normaliser() { ... }
6      public Fraction inverse() { ... }
7      ...
8  }
```

- 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 ?

Responsabilités de la classe Fraction

Les responsabilités d'une classe décrivent ce à quoi elle s'engage au moyen :

- d'invariants : 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 sous forme de :
  - conditions d'utilisation (portent sur les paramètres en `in` et `in out`)
  - effets : décrivent le nouvel état des paramètres en `out` et `in out` en fonction des paramètres en `in` et `in out`

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 (condition d'utilisation).
  L'inverse d'une fraction multiplié par cette fraction est 1 (effet).
- etc.

Que faire pour traiter les conditions d'utilisation ?

1. **La programmation par contrat** :
   - Un contrat entre l'appelant et l'appelé définit qui doit faire quoi.
   - C'est l'objet de cette section !
2. **Programmation défensive :** l'appelé ne fait pas confiance à l'appelant et teste explicitement les cas anormaux.
   Que faire si un tel cas est détecté (ex : une fraction nulle dans inverse) ?
   - **Signaler une erreur et arrêter l'exécution**
     - on considère qu'il s'agit d'une erreur non récupérable
     - il faut corriger le programme et le relancer
   - **Retourner un code d'erreur**
     - C'est l'appelant direct qui doit traiter le problème.
     - Il faut pouvoir particulariser une valeur
     - Ne peut pas fonctionner pour les constructeurs !
     - **Remarque :** Explique pourquoi en Java et C on peut ignorer le résultat d'une méthode.
   - **Lever une exception :** mécanisme souple pour transférer le flot de contrôle de la partie du programme qui détecte un problème vers celle qui sait le traiter.
     - c'est la bonne façon de faire !
     - sera abordée dans un futur chapitre

Programmation par contrat : analogie avec la notion de 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 et son fournisseur lui rend un service.

	obligations	bénéfices
client	payer le prix du service	bénéficier du service
fournisseur	rendre le service	gagner le prix du service

Programmation : Une méthode est le fournisseur, l'appelant est le client.

Application à la programmation

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.

	obligations	bénéfices
appelant	satisfaire préconditions	postconditions (et invariants) satisfaits
appelé	satisfaire postconditions (et invariants)	préconditions satisfaites

Remarques :

- Le terme anglais est **Design by Contract (DBC)**
- On devrait en effet dire *Conception par contrat*
- Car intervient en amont de la phase de programmation

Exemples

Exercice 16 : Racine carrée
Donner les contrats `isqrt`, méthode qui calcule la racine carrée entière d'un entier.

Exercice 17 : Contrats de la classe Compteur
Donner les contrats de la classe Compteur.

Compteur
requêtes
valeur : int
commandes
raz
incrémenter
set(valeur : int)
constructeurs
Compteur(valeur : int)

Description du compteur : Un compteur a une valeur (entière et positive) qui peut être incrémentée d'une unité. Elle peut également être remise à 0. Il est possible d'initialiser le compteur à partir d'une valeur entière positive.

Non respect d'un contrat

Que se passe-t-il si on contrat n'est pas respecté ?

- C'est une erreur de programmation !
- Le programme doit être corrigé
- Ces erreurs devraient être détectées dans les phases de test.

Qui est responsable ? La partie du contrat non respectée permet de le dire :

- précondition $\Longrightarrow$ c'est l'appelant qui a commis l'erreur
- postconditions ou invariant $\Longrightarrow$ c'est l'auteur de la méthode appelée

Comment sait-on qu'un contrat n'est pas respecté ?

- par lecture de la documentation (javadoc) et du code
- par instrumentation du code avec les contrats
  - idée : ajouter des tests qui détectent les violations de contrats
  - comment faire pour instrumenter
    - les préconditions ?
    - les postconditions ?
    - les invariants ?
  - que doit faire l'instrumentation quand un contrat n'est pas vérifié ?
    - lever une exception !
  - L'instrumentation ne doit être effective que dans les phases de test

Exemple de programmation par contrat : ISqrt

```
1  public class ISqrt {
2
3      /** Obtenir la racine carrée entière d'un entier.
4        * @param n l'entier naturel
5        * @return la racine carrée entière de n
6        */
7      //@ requires n >= 0;
8      //@ ensures \result * \result <= n;
9      //@ ensures (\result + 1) * (\result + 1) > n;
10     static public int isqrt(int n) {
11         return (int) Math.sqrt(n);
12     }
13
14     public static void main(String[] args) {
15         System.out.println("isqrt(0) = " + isqrt(0));
16         System.out.println("isqrt(2) = " + isqrt(2));
17         System.out.println("isqrt(3) = " + isqrt(3));
18         System.out.println("isqrt(4) = " + isqrt(4));
19         System.out.println("isqrt(-4) = " + isqrt(-4));
20     }
21 }
```

Exécution avec ou sans instrumentation

Avec instrumentation (jml4c) :

```
> jml4 ISqrt
isqrt(0) = 0
isqrt(2) = 1
isqrt(3) = 1
isqrt(4) = 2
Exception in thread "main" org.jmlspecs.jml4.rac.runtime.JMLInternalPreconditionError:
By method ISqrt.isqrt
Regarding specifications at
  File "ISqrt.java", line 7, character 15
With values
  n: -4

        at ISqrt.main(ISqrt.java:16)
```

Sans instrumentation :

```
> java ISqrt
isqrt(0) = 0
isqrt(2) = 1
isqrt(3) = 1
isqrt(4) = 2
isqrt(-4) = 0
```

Intérêts de la programmation par contrat

- *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) ;
- *exploitable par outils d'analyse statique et générateurs de tests* ;
- *code final optimisé* (non vérification des assertions).

Difficultés

- Instrumenter les contrats nécessite des contrats formels.

  $\Longrightarrow$ s'appuyer sur les expressions du langage cible
- Les expressions du langage considéré (Java) sont
  - insuffisantes : résultat d'une méthode, état d'une variable avant l'exécution d'une méthode...
  - limitées : quantificateurs existentiels ou universels...
- La complétude des postconditions est souvent difficile à atteindre.
  - Mais les postconditions naturelles suffisent pour détecter les erreurs de prog. classiques.
  - Exemple : les contrats de isqrt ne sont pas complets, ni justes !

Exercice 18 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`.
  - Elle est cependant accessible au travers d'extensions telles que *iContract*, *jContractor*, *Jass*... et surtout *JML*.

Mise en œuvre en Java 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`.
- engendrer les programmes de test (`jmlunit`)

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 ==> (et <==) et de l'équivalence <==> (et <=!=>).

```
a ==> b  est équivalent à (! a) || b
a <==> 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++;
    }
}
```

Contrats, méthodes 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.
  Remarque : Seules les requêtes peuvent être utilisées dans les contrats.
  Conséquence : Utiliser `//@ pure` ou `/*@ 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 droit d'accès au moins égal à celui de cette méthode.**
  Justification : L'appelant doit pouvoir évaluer le contrat.

Exercice 19 Définir les contrats de Fraction.

Programmation par contrat : la classe Fraction (1/3)

```
1  public class Fraction {
2      //@ public invariant Entier.pgcd(getNumerateur(), getDenominateur()) == 1;
3      //@ public invariant getNumerateur() == 0 ==> getDenominateur() == 1;
4      //@ public invariant getDenominateur() > 0;
5      //@ private invariant num == getNumerateur();
6      //@ private invariant den == getDenominateur();
7
8      private int num;    // le numérateur
9      private int den;    // le dénominateur
10     final public static Fraction UN = new Fraction(1);  // fraction unité
11
12     /** Initialiser une fraction.
13      * @param  n le numérateur
14      * @param  d le dénominateur (non nul !) */
15     //@ requires d != 0;
16     //@ ensures n * getDenominateur() == d * getNumerateur();
17     public Fraction(int n, int d)   { den = 1; set(n, d); }
18
19     /** initialiser une fraction à partir d'un entier (numérateur).
20      * @param  n le numérateur */
21     //@ ensures n == getNumerateur();
22     //@ ensures 1 == getDenominateur();
23     public Fraction(int n)          { this(n, 1); }
```

Programmation par contrat : Fraction (2/3)

```
25      public String toString() {
26          return "" + num + (den == 1 ? "" : "/" + den);
27      }
28
29      public /*@ pure helper @*/ int getNumerateur()      { return num; }
30      //@ ensures getDenominateur() > 0;  // redondant avec l'invariant
31      public /*@ pure helper @*/ int getDenominateur()    { return den; }
32
33      /** Modifier une fraction.
34       * @param  n le nouveau numérateur
35       * @param  d le nouveau dénominateur (non nul !)
36       */
37      //@ requires d != 0;
38      //@ ensures n * getDenominateur() == d * getNumerateur();
39      public void set(int n, int d) {
40          num = n;
41          den = d;
42          normaliser();
43      }
44
45      /** Normaliser la fraction. */
46      //@ helper  // ==> pas de vérification des invariants.
47      private void normaliser()       { ... }
```

Programmation par contrat : Fraction (3/3)

```
49      /** Inverse de la fraction. */
50      //@ requires getNumerateur() != 0;
51      //@ ensures \result.produit(this).equals(UN);
52      //@ pure
53      public Fraction inverse() {
54          return new Fraction(den, num);
55      }
56
57      /** le produit avec une autre fraction. */
58      public /*@ pure @*/ Fraction produit(Fraction autre) {
59          return new Fraction(num * autre.num, den * autre.den);
60      }
61
62      /** Égalité logique entre fractions. */
63      /*@ ensures \result == ((getNumerateur() == autre.getNumerateur()
64              && getDenominateur() == autre.getDenominateur()));  @*/
65      public /*@ pure @*/ boolean equals(Fraction autre) {
66          return num == autre.num && den == autre.den;
67      }
68
69  }
```

Programmation par contrat : utilisation de Fraction

```
1  public class TestFractionDBC {
2      public static void main (String args []) throws java.io.IOException {
3          // Saisir une fraction
4          int n, d;   // numérateur et dénominateur d'une fraction
5          do {
6              n = Console.readInt("Numérateur : ");
7              d = Console.readInt("Dénominateur : ");
8              if (d == 0) {
9                  System.out.println("Le dénominateur doit être > 0 !");
10             }
11         } while (d == 0);
12         Fraction f = new Fraction(n, d);
13
14         // Afficher la fraction
15         System.out.println("La fraction est : " + f);
16
17         // Afficher son inverse
18         if (f.getNumerateur() != 0) {
19             System.out.println("Son inverse est : " + f.inverse());
20         } else {
21             System.out.println("Pas d'inverse !");
22         }
23     }
24 }
```

Quelques questions sur la classe Fraction

- Comment exprimer formellement les choix de réalisation faits ?
- Est-on sûr que la fraction UN correspondra toujours à la fraction unité (1/1) ?
- Est-ce que la méthode `equals` définie est suffisante pour définir l'égalité logique entre fractions ?
- Que serait-il préférable de définir au lieu de la méthode `afficher()` ?
- Pourquoi dans le premier constructeur de Fraction fait-on `den = 1` ?

Quelques remarques :

- l'indentation n'est pas bonne (pour économiser de la place)
- **`this`** a été omis (pas forcément à conseiller)
- les commentaires de documentation ne sont pas tous complets

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.

JML pour les variants et invariants de boucle

Pour les boucles :

- maintaining : exprimer un **invariant** de boucle
- decreasing : exprimer le **variant**

Exemple : somme des n premiers entiers

```
1       //@ requires n >= 0;
2       //@ ensures \result == (\sum int k; k >= 0 && k <= n; k);
3       static public int somme(int n) {
4           int somme = 0;
5           int i = 1;  // déclaration en dehors du for pour JML
6           //@ maintaining somme == (\sum int j; j >= 0 && j < i; j);
7           //@ decreasing n - i + 1;
8           for (i = 1; i <= n; i++) {
9               somme += i;
10          }
11          return somme;
12      }
```

Limites de l'implantation utilisée de JML

La version utilisée de JML (jml4c) n'est pas très robuste et son développement est arrêté.
Voici quelques conseils pour l'utiliser plus efficacement :

- Toujours préfixer un attribut ou méthode de classe par la classe
  (même dans la classe elle-même)
- Compiler régulièrement avec jmlc (dès qu'un nouveau contrat est ajouté)
  - Quand le seul message affiché est ERROR, c'est alors plus facile de localiser l'erreur !
- Compiler avec javac pour corriger les erreurs Java

OpenJML (`openjml.org`) est plus récent, en développement mais ne semble pas encore au point :-(

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- Documentation
- Requêtes et commandes
- Définir une classe
- Attribuer un SP à une classe
- Test unitaire avec JUnit
- Programmation par contrat
- Unified Modeling Language (UML)

Unified Modeling Language (UML)

Principales caractéristiques d'UML [9, 6]

- notation graphique pour décrire les éléments d'un développement (objet)
- normalisée par l'OMG [7] (Object Management Group)
- version 1.0 en janvier 1997. Version actuelle : 2.4 (beta), mars 2011.
- *s'abstraire du langage de programmation et des détails d'implantation*

Utilisations possibles d'UML [6]

- **esquisse (*sketch*) :** communiquer avec les autres sur certains aspects
  - simplification du modèle : seuls les aspects importants sont présentés
  - échanger des idées, évaluer des alternatives (avant de coder), expliquer (après)
  - n'a pas pour but d'être complet
- **plan (*blueprint*) :** le modèle sert de base pour le programmeur
  - le modèle a pour but d'être complet
  - les choix de conception sont explicités
  - seuls les détails manquent (codage)
- **langage de programmation :** *le modèle est le code !*
  - pousser à l'extrême l'approche « plan »
    $\Longrightarrow$ mettre tous les détails dans le modèle
  - engendrer automatiquement le programme à partir du modèle.

Représentation d'une classe

Date
− jour : int − mois : int − année : int
+ getJour() : int + getMois() : int + getAnnée() : int + afficher() + ajouter(nbJours : int) + distance(autre : Date) : int − incrémenter() + estBissextile(année : int) : boolean
+ Date(jour : int, mois : int, année : int)

- Ordre des rubriques :
  - nom de la classe
  - attributs
  - opérations
  - constructeurs
- Droits d'accès :
  - public : +
  - privé : −
  - protected : #
  - paquetage : (rien)
- membre de classe : souligné

- Les constructeurs peuvent être mis avec les opérations, précédés du stéréotype «create»
- Pour le premier diagramme (analyse), il est conseillé de faire requêtes/commandes plutôt que attributs/opérations.

Adaptation des rubriques

En UML, il est possible d'adapter les rubriques.
Par exemple, pour un *diagramme d'analyse*, on ne s'intéresse pas aux attributs.
On fait alors une distinction requêtes/commandes.

Date
requêtes
jour : int mois : int année : int estBissextile(année : int) : boolean distance(autre : Date) : int
commandes
afficher() ajouter(nbJours : int) incrémenter()
constructeurs
Date(jour : int, mois : int, année : int)

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- String
- Les classes enveloppe

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- String
- Les classes enveloppe

La classe String

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

```
String s0;      // Une poignée non initialisée (ou 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 + 1;          // Total = 51
String s7 = "Total␣=␣"+ (valeur + 1);        // Total = 6
```

Attention : Les parenthèses changent l'évaluation !

« String » est une classe

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

Égalité physique et égalité logique (appliquée aux String)

Deux types d'égalité :

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

```
String s1 = "INP-N7";
int iN7 = s1.indexOf("N7");
String s2 = s1.substring(iN7, iN7+2);
String s3 = "N7";
boolean estN7 = s2 == s3;        // faux !
```

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

```
boolean estN7 = s2.equals(s3); // vrai !
boolean estN7 = s1.substring(iN7, iN7+2).equals("N7"); // vrai !
```

Important : Ces notions d'égalité physique et logique s'appliquent à tout objet.

Ordre lexicographique : voir compareTo

Objet immuable et objet muable

- objet immuable (*immutable*) : objet dont l'état ne peut pas être modifié après sa création.

  Exemple : Toute opération de « modification » de la classe String retourne un nouvel objet.

- Objet muable (*mutable*) : objet dont l'état peut être changé.

  Exemple : StringBuilder, classe qui fournit les mêmes opérations que String plus :
  - `append` : ajouter à la fin de la chaîne ;
  - `insert` : ajouter à une position spécifiée de la chaîne.

```
1  StringBuilder sb = new StringBuilder("String");
2  sb.append("Builder");
3  sb.append(' ');
4  sb.append(10);
5  String chaine = sb.toString();
6  assert "StringBuilder 10".equals(chaine);
```

- **Conséquences liées au caractères immuable d'un objet :**
  - On peut partager l'objet, sans risque qu'il soit modifié,
    - entre plusieurs objets (manière de réaliser une relation de composition),
    - entre plusieurs fils d'exécution (threads).
      Voir StringBuffer (T. 228) pour une version « thread safe » de StringBuilder.
  - Mais chaque "modification" crée en fait un nouvel objet !

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- String
- Les classes enveloppe

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,
  - construire un objet à partir d'une chaîne de caractère,
  - obtenir la valeur du type primitif correspondant,
  - traduire vers d'autres types primitifs.
- Les instances des classes enveloppes sont des objets *immuables* !
- **Intérêt :** Les classes enveloppes n'ont d'intérêt que si l'on a besoin de considérer les types primitifs comme des objets (Généricité).
- Depuis Java 1.5, la conversion entre types primitifs et classes enveloppes est automatique (*auto boxing/unboxing*).

Exemple : la classe 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 < 1.5)

```
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 $\geq$ 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 }
```

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- L'instruction return
- Fabriques statiques
- Ellipse
- Valeurs par défaut et initialiseurs
- Passage de paramètre
- String ou StringBuilder ?
- Classe Java vs module en C
- Exécution de RésolutionÉquation

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- L'instruction return
- Fabriques statiques
- Ellipse
- Valeurs par défaut et initialiseurs
- Passage de paramètre
- String ou StringBuilder ?
- Classe Java vs module en C
- Exécution de RésolutionÉquation

Du bon usage de **return**

- **Règle :** Un SP ne doit avoir qu'un seul **return**, dernière instruction
- **Justification :** Meilleure lisibilité. Ne pas avoir à chercher les **return** !
  Un SP doit être une boîte noire avec un seul point d'entrée (la première instruction) et un seul point de sortie (la dernière instruction).
- **Tolérances :**

Approche « fonctionnelle »
ex : maximum de deux entiers

```
static int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

Remarque : le corps pourrait être :
`return a > b ? a : b;`

Utilisation d'un foreach
ex : recherche d'un élément dans un tableau

```
static boolean estPresent(int x, int[] tab) {
    for (int n : tab) {
        if (n == x) {
            return true;
        }
    }
    return false;
}
```

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- L'instruction return
- Fabriques statiques
- Ellipse
- Valeurs par défaut et initialiseurs
- Passage de paramètre
- String ou StringBuilder ?
- Classe Java vs module en C
- Exécution de RésolutionÉquation

Fabrique statique

Une utilisation recommandée des méthodes de classe est de s'en servir à la place des constructeurs.

```
public class Equation {
  ...
  protected Equation(double a, double b, double c) { ... }

  public static Equation depuisCoefficients(double a, double b, double c) {
      return new Equation(a, b, c);
  }

  public static Equation depuisSommeEtProduit(double s, double p) {
      return new Equation(1, -s, p);
  }

  public static Equation depuisRacines(double x1, double x2) {
      return Equation.depuisSommeEtProduit(x1 + x2, x1 * x2);
  }
}
```

Remarque : Le constructeur n'est pas public pour obliger à utiliser les fabriques statiques.

Fabrique statique : utilisation

```
class Exemple {

    public static void main(String[] arguments) {
        Equation eq1 = Equation.depuisCoefficients(1, 5, 6);
        Equation eq2 = Equation.depuisSommeEtProduit(2, 1);
        Equation eq3 = Equation.depuisRacines(2, 1);
        ...
    }

}
```

Avantages :

- les méthodes de classe ont un nom. Il peut donc être explicite !
- une méthode de classe n'est pas obligée de créer un nouvel objet à chaque appel. On peut retourner un objet déjà créé.
- la méthode peut retourner un objet d'un sous-type (à venir)

Inconvénients :

- pas de différence syntaxique entre fabrique statique et méthodes de classe
- déviation de la norme (les constructeurs !)

$\Longrightarrow$ difficile de retrouver les fabriques statiques dans la documentation
Quelques *conventions* pour nommer : `valueOf` (conversion de type), `getInstance`.

2ème lecture : pourquoi le constructeur est défini **`protected`** et non **`private`** ?

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- L'instruction return
- Fabriques statiques
- Ellipse
- Valeurs par défaut et initialiseurs
- Passage de paramètre
- String ou StringBuilder ?
- Classe Java vs module en C
- Exécution de RésolutionÉquation

Ellipse : nombre variable d'arguments

- Comment définir une méthode somme qui calcule la somme des entiers qui sont passés en paramètre. Elle doit pouvoir prendre en paramètre un nombre quelconque de paramètre.

```
somme 2 3        -> 5
somme 2 -3 4     -> 3
somme 1          -> 1
somme            -> 0
```

Solution 1 : Surcharge (Pas réaliste !)

```java
// commentaires de documentation volontairement omis.

public static int somme() {
    return 0;
}

public static int somme(int a) {
    return a;
}

public static int somme(int a, int b) {
    return a + b;
}

public static int somme(int a1, int a2, int a3) {
    return a1 + a2 + a3;
}

// ... réaliste ?

public static void testerSomme() {
    assert 5 == somme(2, 3);        // facile à utiliser
    assert 3 == somme(2, -3, 4);
    assert 1 == somme(1);
    assert 0 == somme();
}
```

Solution 2 : Tableau (Utilisation peu pratique !)

```
1       /** Obtenir la somme des éléments.
2         * @param elements les éléments à sommer
3         * @return la somme des éléments
4         */
5       public static int somme(int[] elements) {   // facile à écrire
6           int somme = 0;
7           for (int n : elements) {
8               somme += n;
9           }
10          return somme;
11      }
12
13      public static void testerSomme() {
14          assert 5 == somme(new int[] {2, 3});    // lourd à utiliser
15          assert 3 == somme(new int[] {2, -3, 4});
16          assert 1 == somme(new int[] {1});
17          assert 0 == somme(new int[] {});
18      }
```

Solution 3 : Ellipse (La bonne !)

```java
/** Obtenir la somme des éléments.
  * @param elements les éléments à sommer
  * @return la somme des éléments
  */
public static int somme(int... elements) {  // facile à écrire
    int somme = 0;
    for (int n : elements) {
        somme += n;
    }
    return somme;
}

public static void testerSomme() {
    assert 5 == somme(2, 3);                    // facile à utiliser
    assert 3 == somme(2, -3, 4);
    assert 1 == somme(1);
    assert 0 == somme();
    assert 3 == somme(new int[] {2, -3, 4});    // tableau possible !
}
```

Autre définition de somme : le paramètre est bien un tableau !

```java
public static int somme2(int... elements) {
    int somme = 0;
    for (int i = 0; i< elements.length; i++) {
        somme += elements[i];
    }
    return somme;
}
```

Ellipse : conclusion

- Utiliser la surcharge n'est pas réaliste car :
  - trop de méthodes à définir (potentiellement une infinité)
  - code peu pratique à écrire : chaque paramètre nommé explicitement
- Le tableau pour le paramètre de somme est la bonne solution mais :
  - son utilisation est plus difficile : construction explicite du tableau
- Ellipse (T...) allie les avantages des deux :
  - une seule méthode à écrire
  - utilisation simple
  - possibilité de fournir un tableau lors de l'appel
  - possibilité de particulariser, mais pas conseillé (la méthode spécifique est prioritaire) :

```
1       public static int somme(int a, int b) {
2           return a + b;
3       }
```

- Conclusion : **Ellipse = facilité syntaxique**

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- L'instruction return
- Fabriques statiques
- Ellipse
- Valeurs par défaut et initialiseurs
- Passage de paramètre
- String ou StringBuilder ?
- Classe Java vs module en C
- Exécution de RésolutionÉquation

Valeurs par défaut et initialiseurs

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.
- Références en avant interdites dans un initialiseur !

Exemple :

```
class C {
        private int a = 5;      // valeur par défaut du programmeur : 5
        private int b;          // valeur par défaut du type : 0
        {       // initialiseur 1
                a = 7;
        }
        {       // initialiseur 2
                a = 3;
        }
}
```

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).

Sommaire

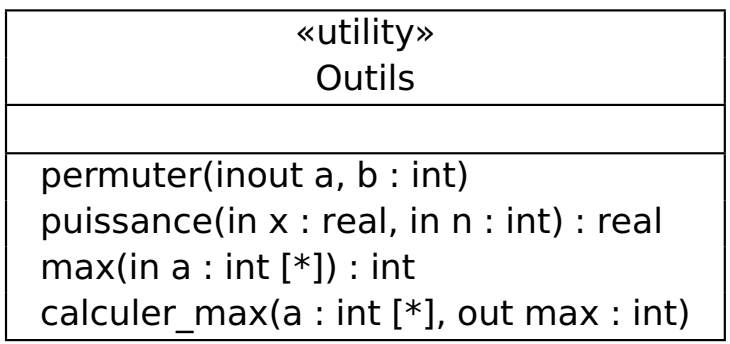
1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- L'instruction return
- Fabriques statiques
- Ellipse
- Valeurs par défaut et initialiseurs
- Passage de paramètre
- String ou StringBuilder ?
- Classe Java vs module en C
- Exécution de RésolutionÉquation

Modes de passage en UML

Le point de vue est celui du programme appelé :

- **in** : donnée en entrée
  - donnée fournie par l'appelant
  - ne pouvant pas être modifiée par l'appelé
  - l'appelant est sûr que sa donnée n'est pas modifiée
- **out** : donnée en sortie
  - la donnée est calculée par l'appelé
  - fournie à l'appelant par l'intermédiaire du paramètre
  - l'appelé ne peut pas utiliser la valeur du paramètre
- **in out** : donnée en entrée et en sortie
  - donnée fournie par l'appelant,
  - modifiée par l'appelé,
  - les modifications sont visibles de l'appelant



Passage de paramètres en Java

- En Java, un seul mode de passage : le passage par valeur.
- Mais deux types de données manipulées :
  - les données de types primitifs qui sont des valeurs
  - les objets qui sont toujours manipulés à travers leur adresse
- 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.
  - la méthode appelée peut modifier l'état de l'objet et
  - la modification sera visible de l'appelant
- L'appelant ne verra jamais les modifications apportées à la valeur du paramètre transmis (type primitif ou poignée).
- Une méthode qui retourne un objet, retourne une poignée sur l'objet.
  $\Longrightarrow$ risque de rupture de l'encapsulation.

Passage de paramètre de type primitif

Le programme

```
1  class TestTypePrimitif {
2
3      static void incrementer(int n) {
4          System.out.println("incr, debut : n = " + n);
5          n++;
6          System.out.println("incr, fin   : n = " + n);
7      }
8
9      /** Tester l'effet de incrémenter */
10     public static void main(String[] args) {
11         int a = 5;
12         System.out.println("main, avant incrementer, a = " + a);
13         incrementer(a);
14         System.out.println("main, apres incrementer, a = " + a);
15         incrementer(100);   // appel avec une constante
16         incrementer(4*a);   // appel avec une expression
17     }
18 }
```

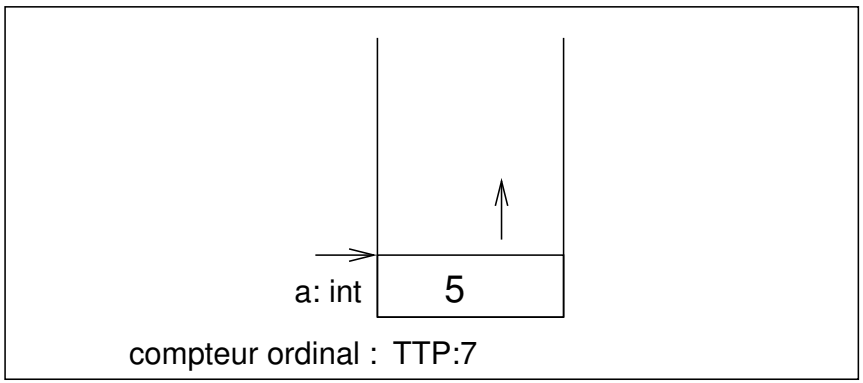
Passage de paramètre de type primitif

Résultat de l'exécution

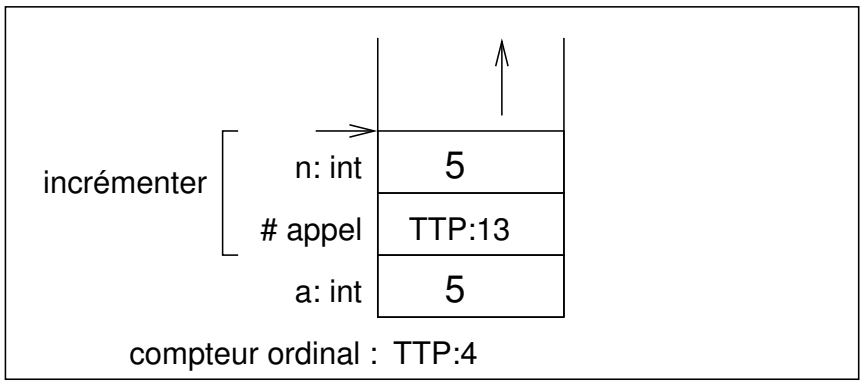
```
main, avant incrementer, a = 5
incr, debut : n = 5
incr, fin   : n = 6
main, apres incrementer, a = 5
incr, debut : n = 100
incr, fin   : n = 101
incr, debut : n = 20
incr, fin   : n = 21
```

Passage de paramètre de type primitif

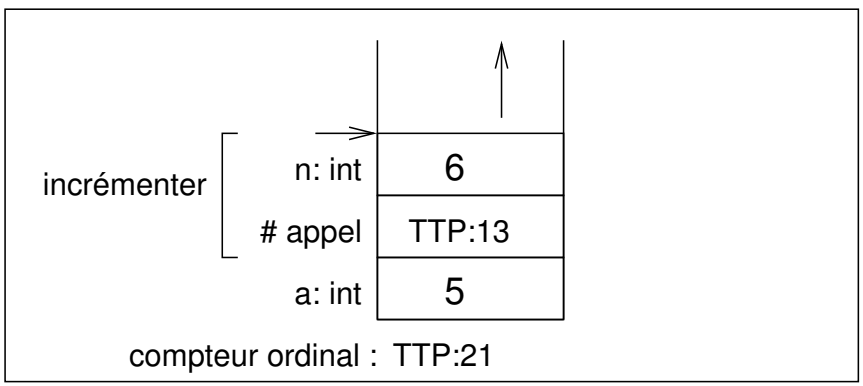
La mémoire



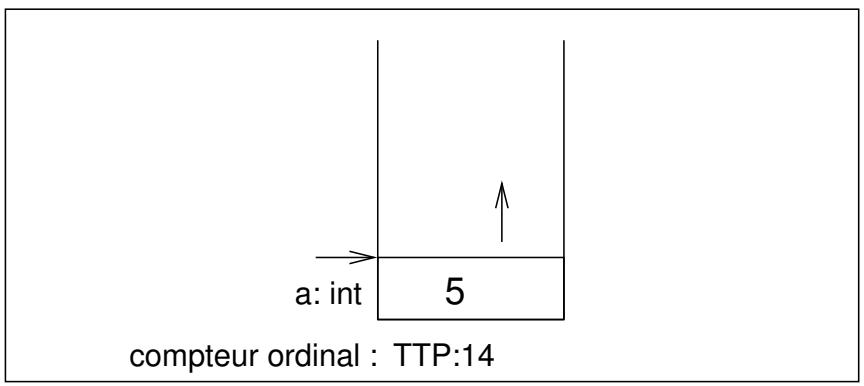
initialisation de a



début appel de incrementer



exécution de incrementer



fin de l'appel à incrementer(a)

Passage de paramètre de type classe

Le programme

```java
class Entier {
  int valeur;
  void ajouter(int val) {
    this.valeur += val;
  }
  void setValeur(int nv) {
    this.valeur = nv;
  }
  int getValeur() {
    return this.valeur;
  }
  public String toString() {
    return "" + this.valeur;
  }
}
```

```java
class TestTypeClasse {
  public static void incrementer(Entier n) {
    System.out.println("incr, debut, n = " + n);
    n.ajouter(1);
    System.out.println("incr, fin,   n = " + n);
  }

  /** Tester l'effet de incrémenter */
  public static void main(String[] args) {
    Entier a = new Entier();
    a.setValeur(5);
    System.out.println("main, avant, a = " + a);
    incrementer(a);
    System.out.println("main, apres, a = " + a);
  }
}
```

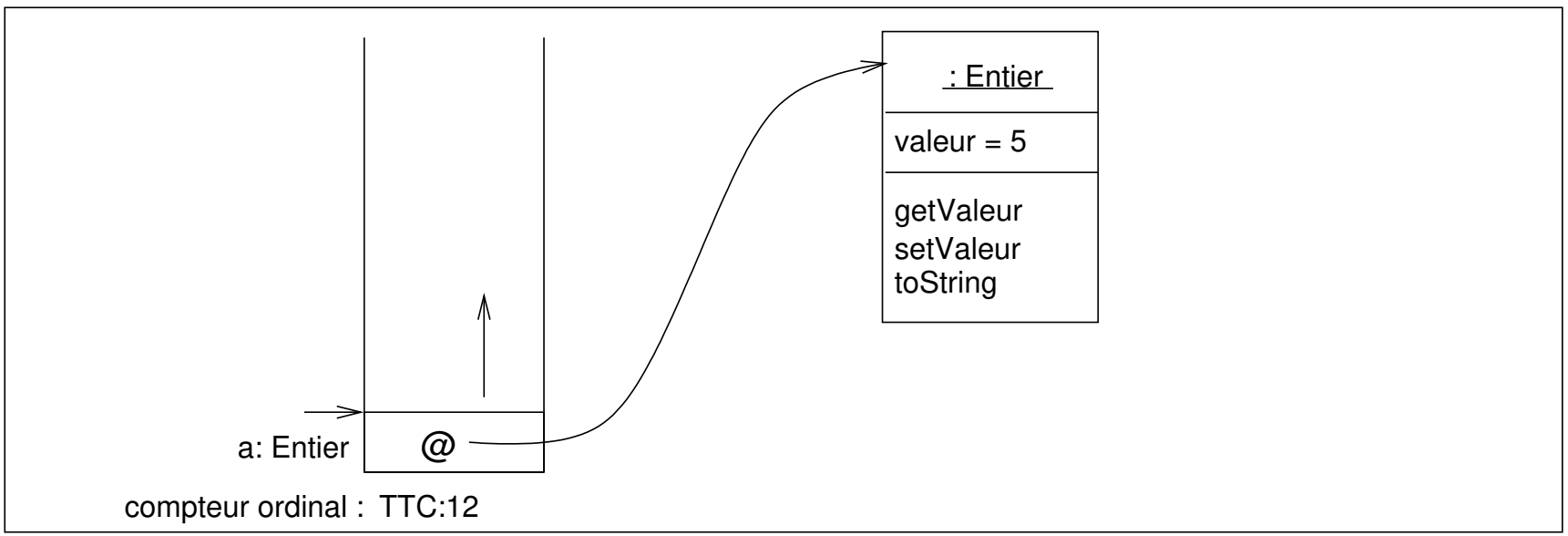
Passage de paramètre sur type classe

Résultat de l'exécution

```
main, avant, a = 5
incr, debut, n = 5
incr, fin,   n = 6
main, apres, a = 6
```

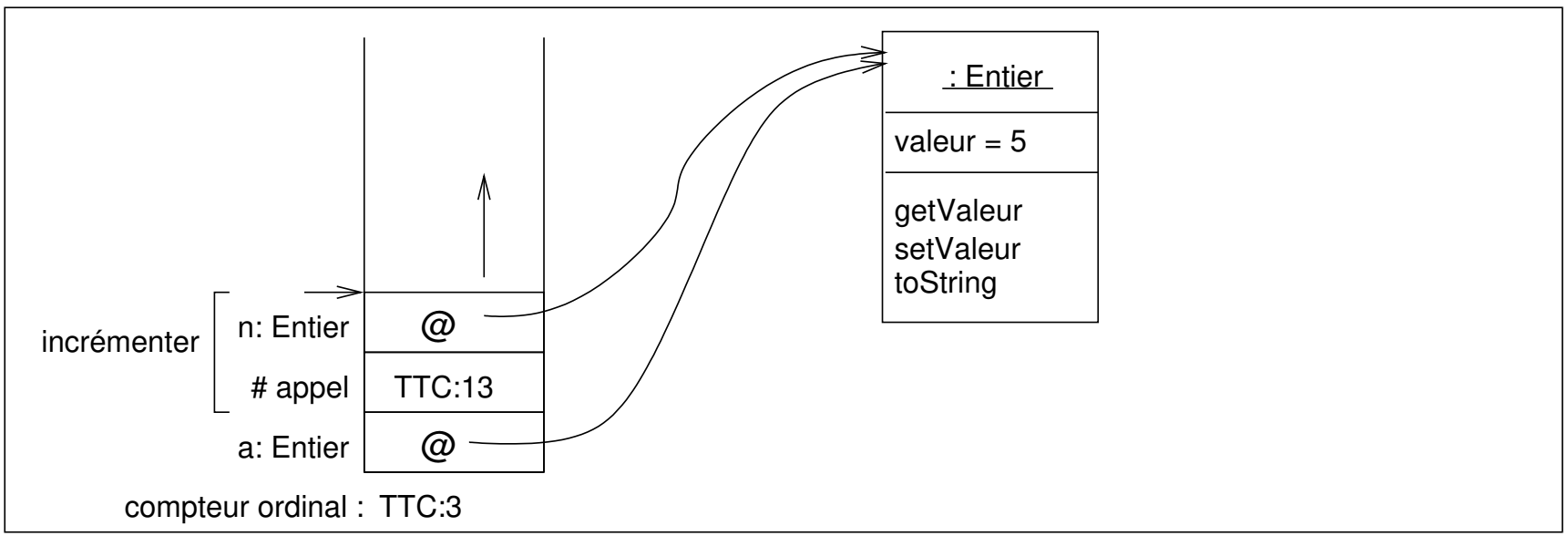
Passage de paramètre sur type classe

La mémoire



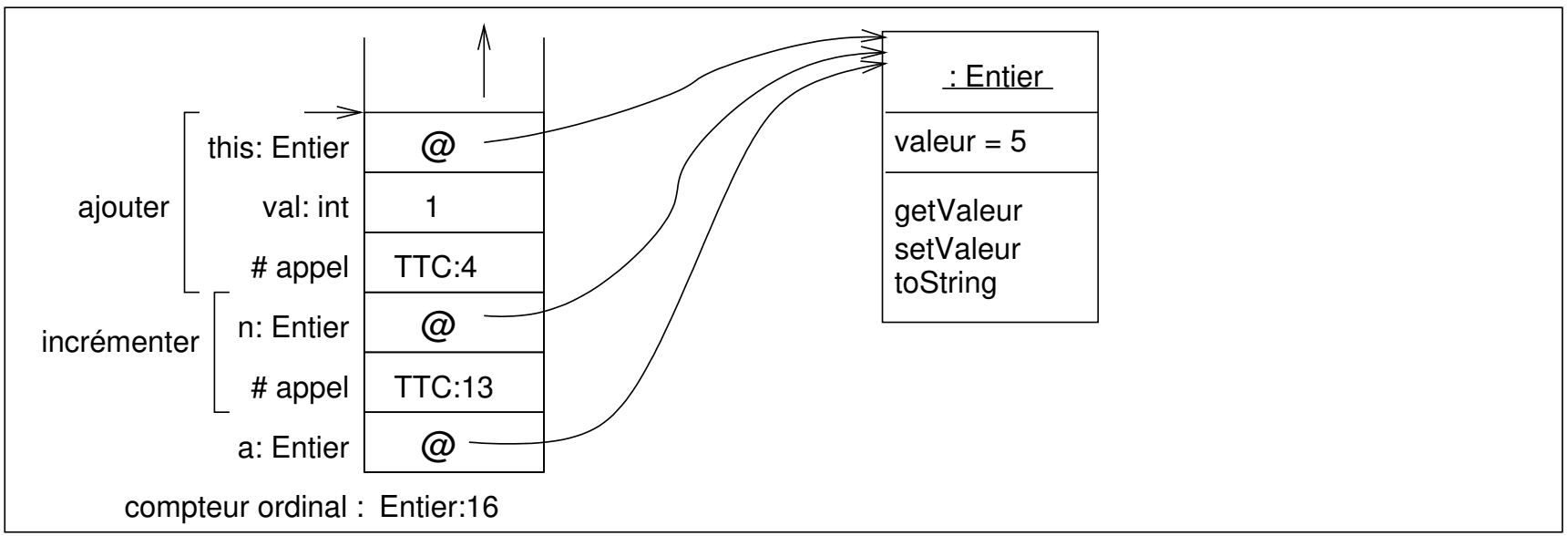
Passage de paramètre sur type classe

La mémoire



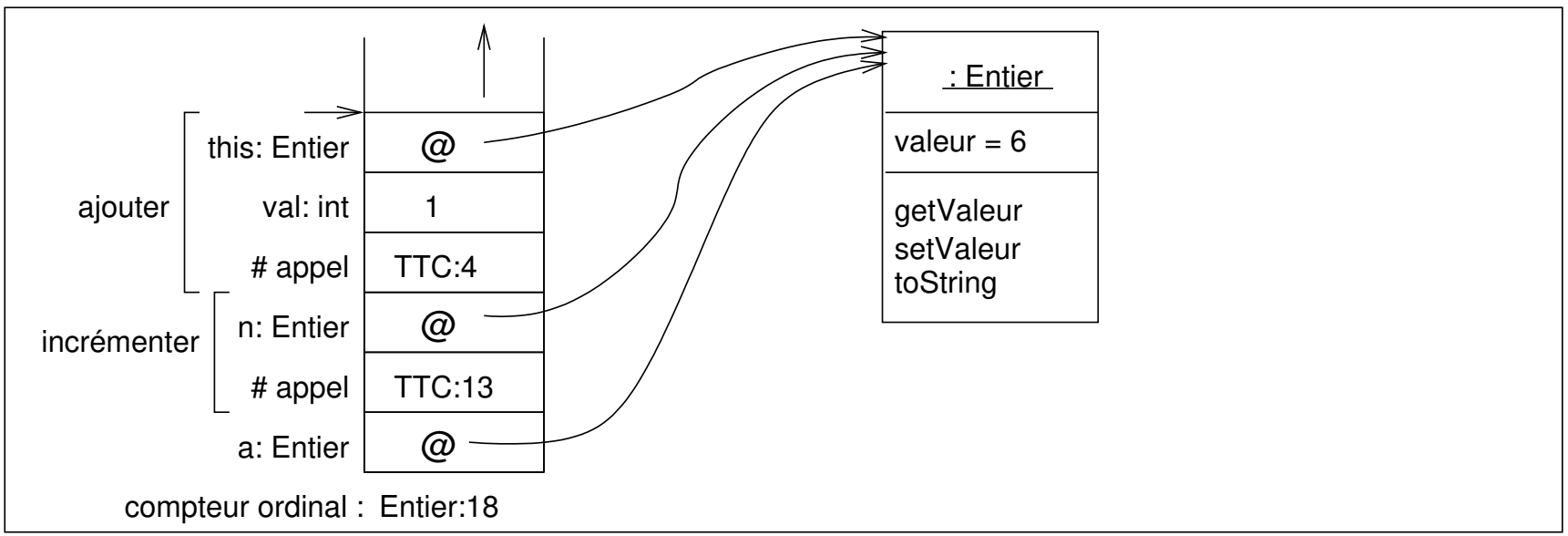
Passage de paramètre sur type classe

La mémoire



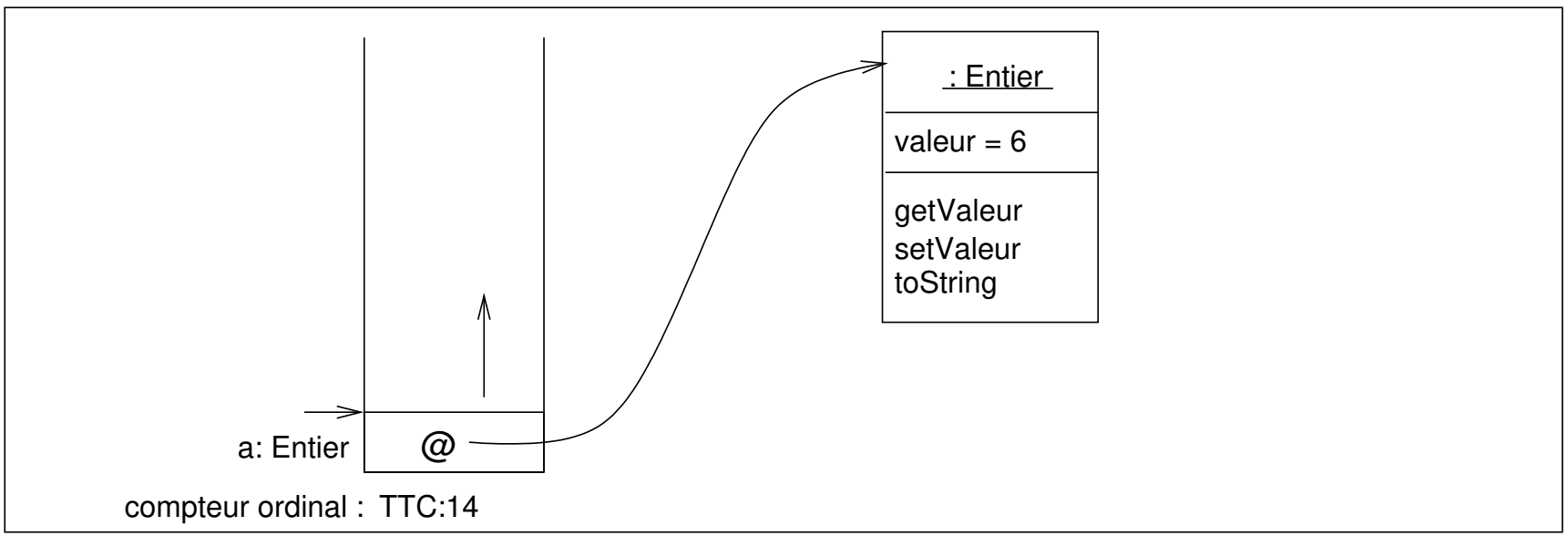
Passage de paramètre sur type classe

La mémoire



Passage de paramètre sur type classe

La mémoire



Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- L'instruction return
- Fabriques statiques
- Ellipse
- Valeurs par défaut et initialiseurs
- Passage de paramètre
- String ou StringBuilder ?
- Classe Java vs module en C
- Exécution de RésolutionÉquation

Pourquoi String et StringBuilder ?

Le caractère immuable d'un String entraîne que :

- on peut faire du partage sans risque de modification par les autres
  $\Longrightarrow$ gain de mémoire et de temps (aucune copie nécessaire).
- toute « modification » nécessite la création d'une nouvelle String
  $\Longrightarrow$ perte de mémoire et de temps (allocation + copie)

StringBuilder permet de faire des modifications sur la même zone mémoire et évite donc les réallocations MAIS attention au partage ! ! !
Remarque : StringBuilder 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 StringBuilder().append("a").append(4).append("c").toString()
```

String ou StringBuilder ? 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 ConcatenerStringBuilder {
    public static void main(String[] args) {
        StringBuilder tampon = new StringBuilder();
        for (int i = 0; i < 100000; i++) {
           tampon.append('x');
        }
        String chaîne = tampon.toString();
        System.out.println("Longueur de chaîne = " + chaîne.length());
    }
}
```

String ou StringBuilder ?

Temps d'exécution :

```
> time java ConcatenerString
Longueur de chaîne = 100000
real    0m6.926s
user    0m5.868s
sys     0m1.260s
```

```
> time java ConcatenerStringBuilder
Longueur de chaîne = 100000
real    0m0.163s
user    0m0.092s
sys     0m0.092s
```

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

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

StringBuffer, classe plus ancienne que StringBuilder, est « thread safe ».

Voir `http://java.sun.com/developer/technicalArticles/Interviews/community/kabutz_qa.html`

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

- L'instruction return
- Fabriques statiques
- Ellipse
- Valeurs par défaut et initialiseurs
- Passage de paramètre
- String ou StringBuilder ?
- Classe Java vs module en C
- Exécution de RésolutionÉquation

Autre version des Équations 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 20 Comparer la fonction `résoudre` en C et la méthode `résoudre` en Java (T. 16).

Le programme principal : test_equation.c

```c
#include <stdio.h>
#include <stdlib.h>

#include "equation.h"

int main()
{
    Equation uneEquation;       /* notre équation */

    /* Initialiser les coefficients */
    uneEquation.coeffA = 1;
    uneEquation.coeffB = 5;
    uneEquation.coeffC = 6;

    /* Calculer les racines de l'équation */
    resoudre(&uneEquation);

    /* Afficher les résultats */
    printf("␣Racine␣1␣:␣%f\n", uneEquation.x1);
    printf("␣Racine␣2␣:␣%f\n", uneEquation.x2);

    return EXIT_SUCCESS;
}
```

Constatations

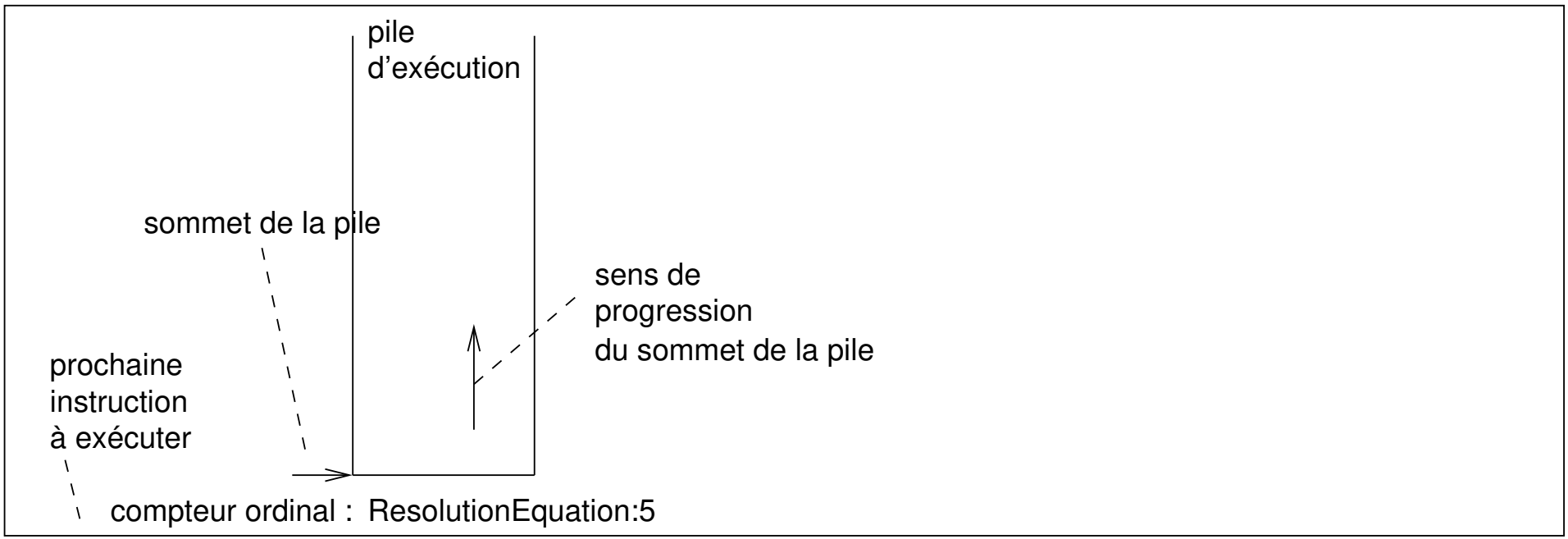
- On peut avoir une approche objet même avec un langage non objet !
  Attention : On n'aura pas tous les bénéfices d'une approche objet...
  sauf à faire des choses très (trop !) compliquées.
- Dans la version C, il y a séparation entre la spécification (interface) et l'implantation (corps) du module 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 `résoudre` 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 indique où se trouve l'élément utilisé : `Math.sqrt`.
- Le **`new`** de Java correspondrait à un `malloc` en C. En Java, la mémoire est libérée automatiquement (pas de `delete` ou `free`).

Sommaire

1 Exemple introductif
2 Modularité : Classe
3 Constructeurs et destructeur
4 Masquage d'information
5 Membres de classe
6 Programmation impérative
7 Aspects méthodologiques
8 Exemples de classe dans l'API Java
9 Compléments

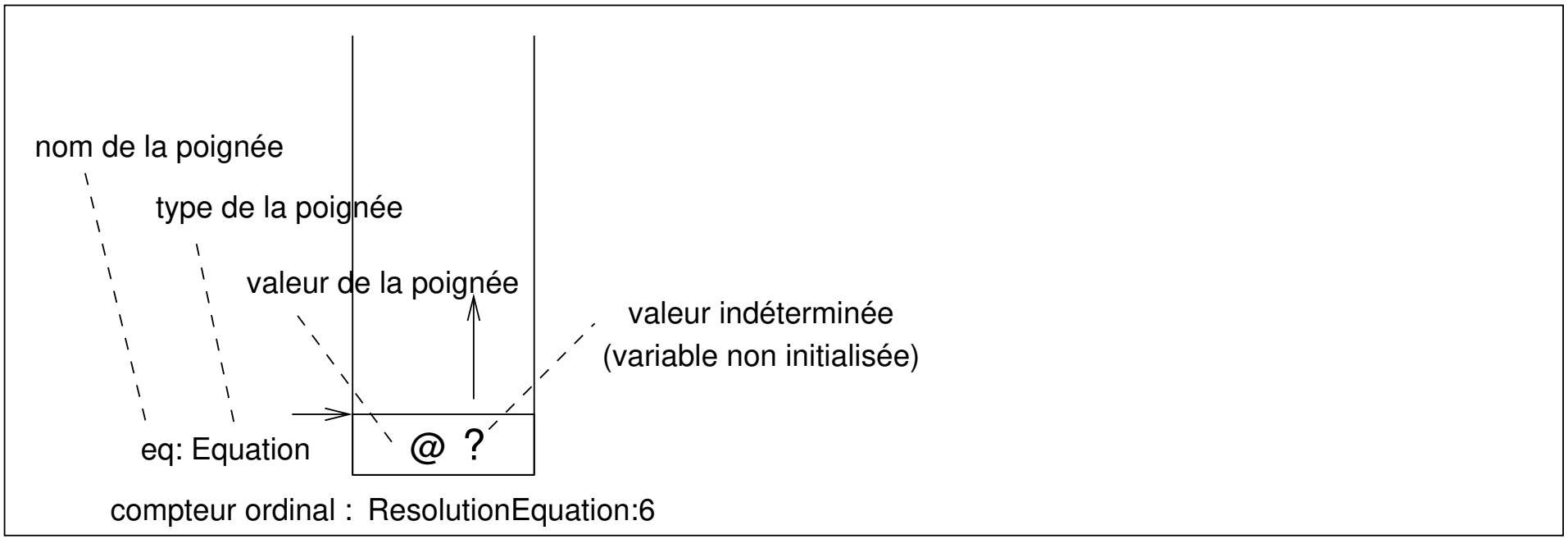
- L'instruction return
- Fabriques statiques
- Ellipse
- Valeurs par défaut et initialiseurs
- Passage de paramètre
- String ou StringBuilder ?
- Classe Java vs module en C
- Exécution de RésolutionÉquation

Exécution de RésolutionÉquation



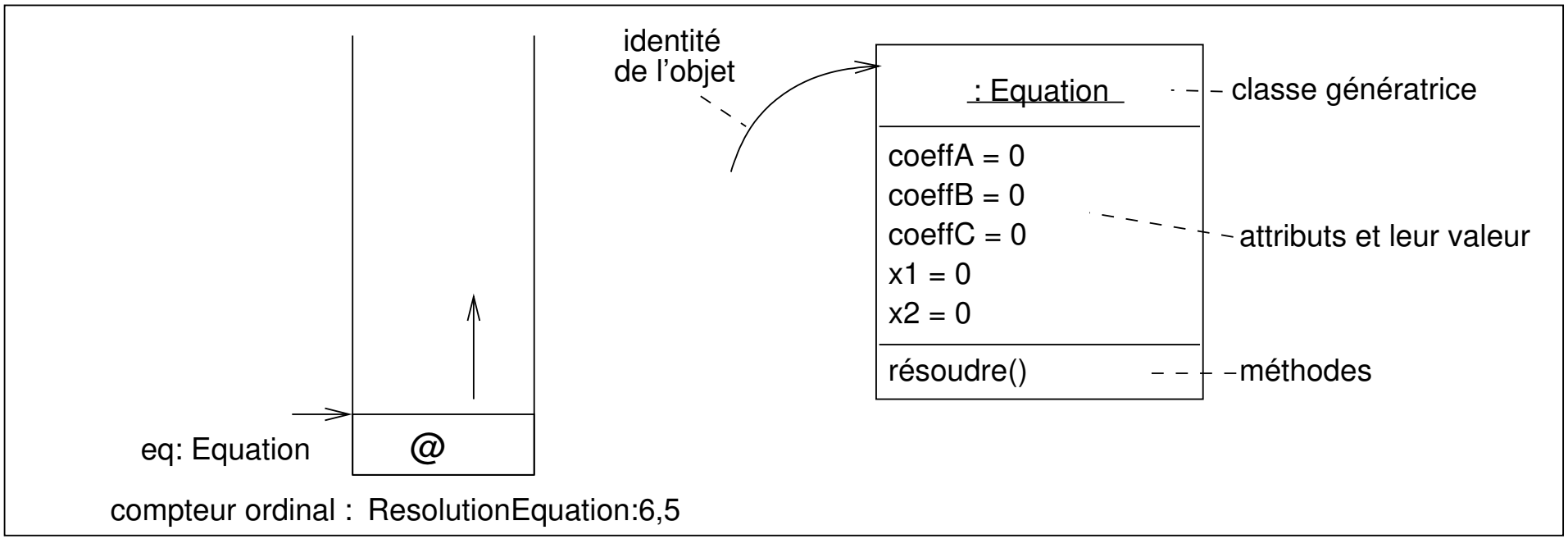
- Avoir une représentation de ce qu'est l'exécution d'un programme
- La pile d'exécution : mémoire automatiquement allouée
- Compteur ordinal : prochaine instruction à exécuter

Exécution de RésolutionÉquation (2)



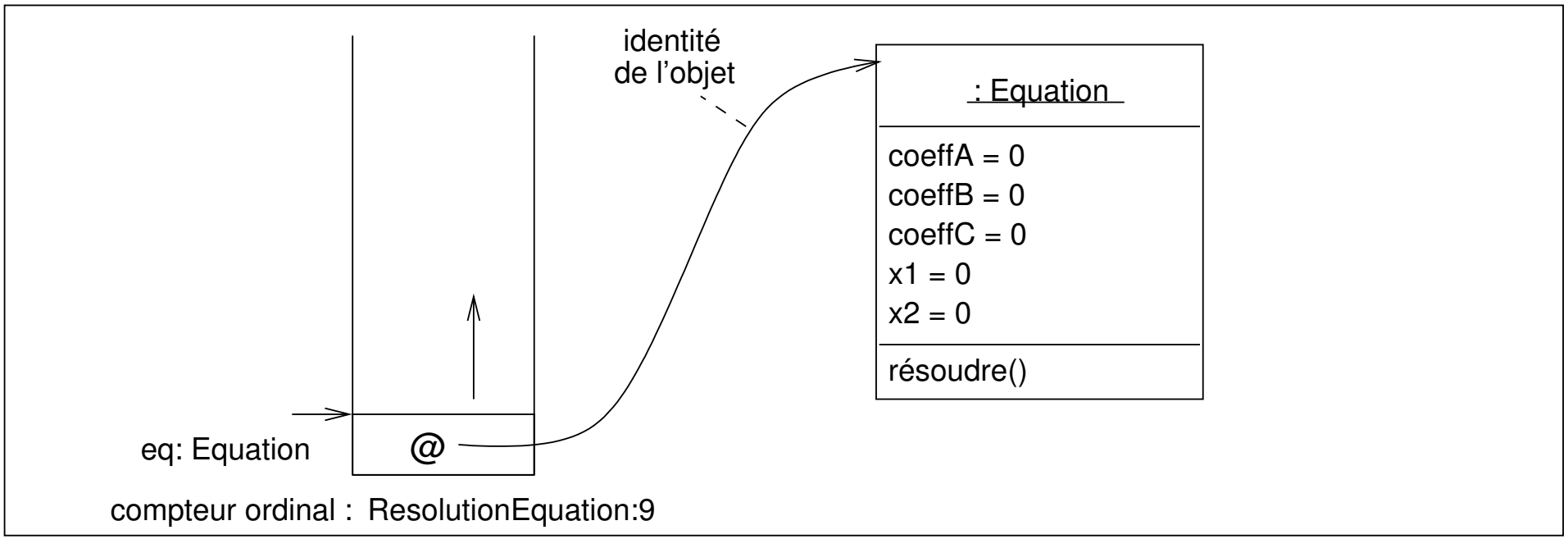
- Variables locales :
  - Allouées dans la pile d'exécution
  - Ne sont pas initialisées (valeur indéterminée, notée « ? »)
  - Une poignée contient une adresse (d'où le @)

Exécution de RésolutionÉquation (3)



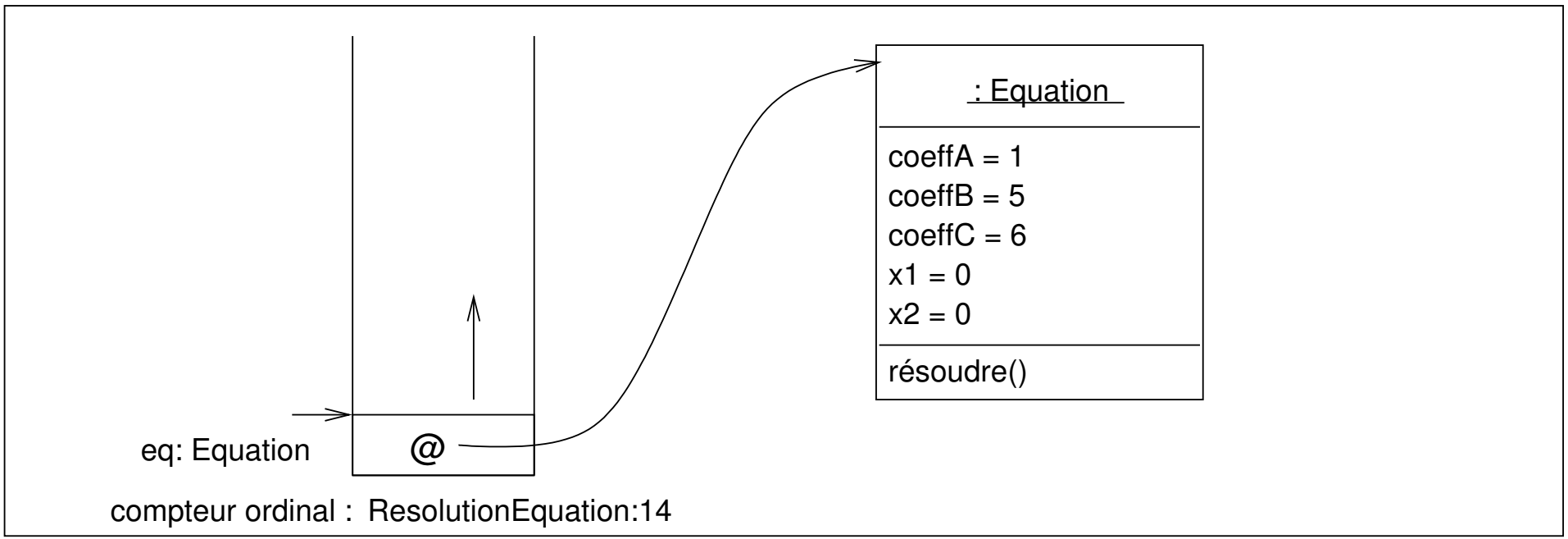
- Création d'un objet
  - réservation de l'espace mémoire
  - initialisation avec les valeurs par défaut du type
  - renvoie de l'identité de l'objet (adresse en mémoire)

Exécution de RésolutionÉquation (4)



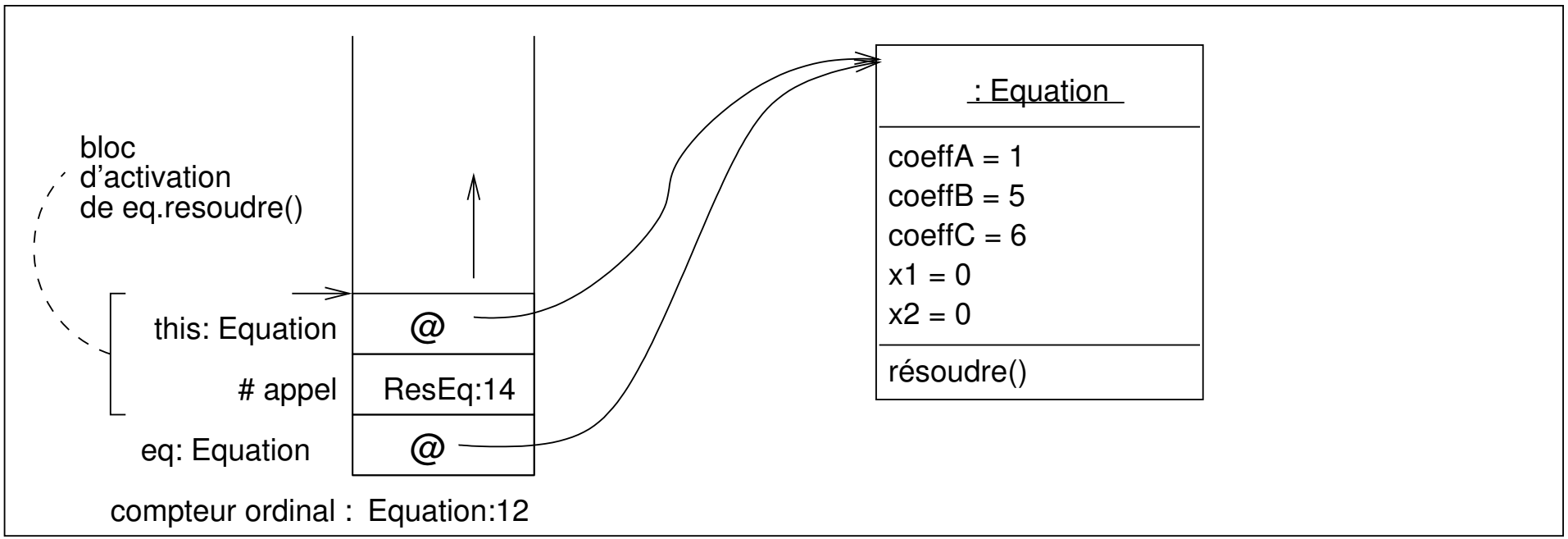
- Initialisation de la poignée
  - avec l'identité de l'objet créé

Exécution de RésolutionÉquation (5)



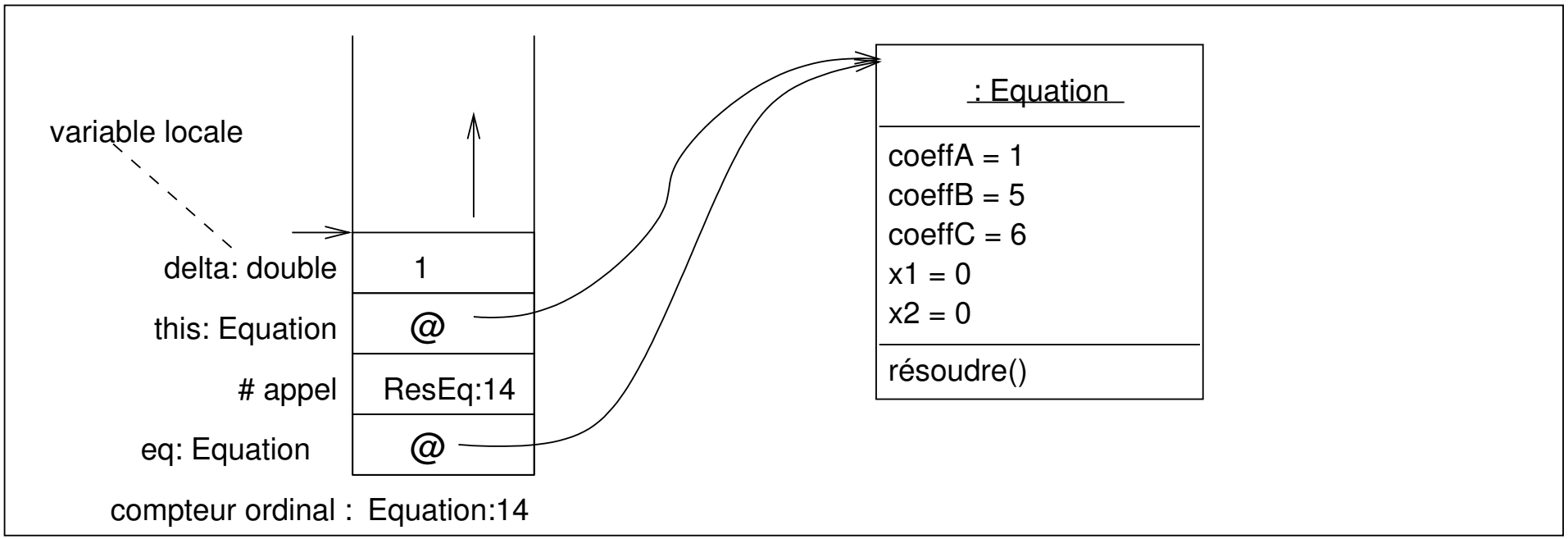
- exécution de « eq.coeffA = 1 »
  - « eq » est l'équation
  - « . » : on se place sur l'objet associé à la poignée
  - « coeffA » : on désigne l'attribut correspondant
  - et on l'initialise à 1
- idem pour coeffB et coeffC

Exécution de RésolutionÉquation (6)



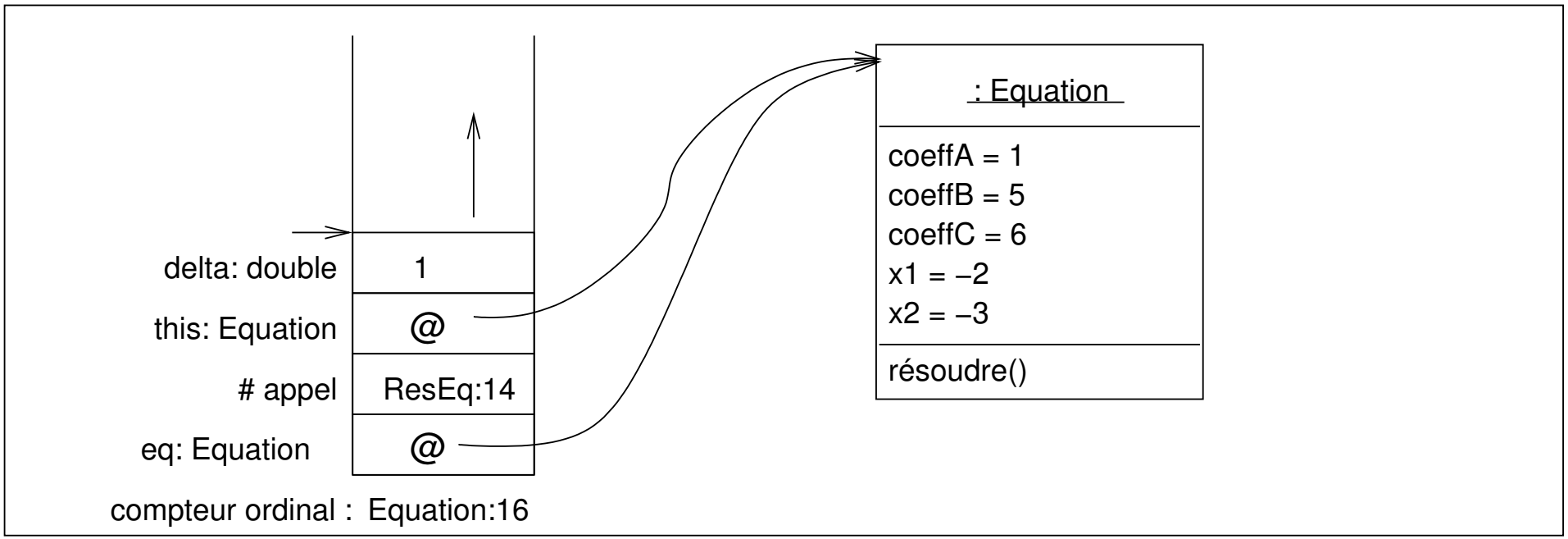
- Appel de résoudre : « eq.resoudre() » :
  - « eq » est l'équation
  - « . » : on se place sur l'objet associé à la poignée
  - « resoudre » est bien une méthode de l'objet
- Le bloc d'activation gère l'appel :
  - mémorisation de l'instruction de l'appel
  - réservation de la place pour les paramètres (ici, seulement **`this`**)
  - initialisation des paramètres avec le paramètre effectif correspondant

Exécution de RésolutionÉquation (7)



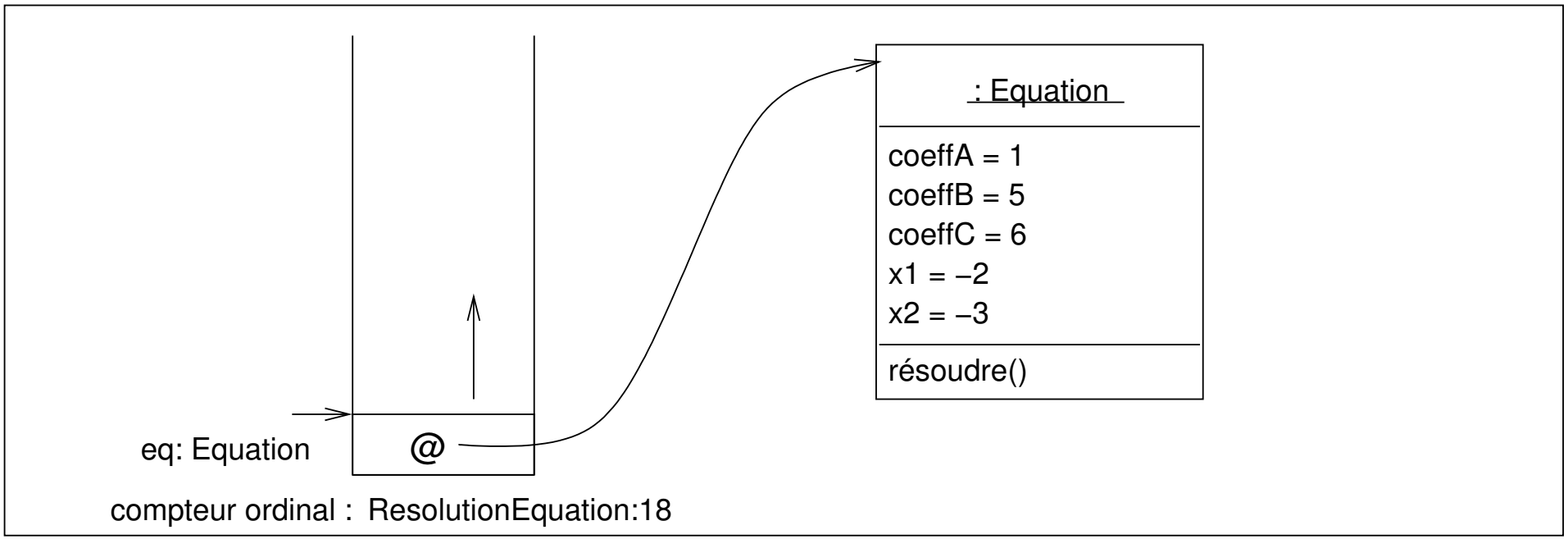
- Réservation de la place pour la variable locale « delta »
  - dans la pile d'exécution
  - initialisée à 1 (en utilisant this.coeffA, etc.)

Exécution de RésolutionÉquation (8)



- Initialisation des racinces (x1 et x2)

Exécution de RésolutionÉquation (9)



- libération de la place utilisée par delta (dans pile d'exécution)
- libération de la place utilisée par le bloc d'activation (dans pile d'exécution)
- continuation de l'exécution après l'appel

Index

égalité, 191
  logique, 191
  physique, 191
énumération, 132

accès uniforme, 66
attribut, 28
  accès uniforme, 66
  d'instance, 28
  de classe, 78, 80
  protection en écriture, 66
  valeur par défaut, 212

classe
  classe enveloppe, 194
  comment la définir, 147
  vue programmeur, 138
  vue utilisateur, 138
constructeur, 43
  par défaut, 48
  surcharge, 45
  this(, 46

dbc, *voir* programmation par contrat
destructeur, 51
do ... while, 117
droit d'accès, 63

enum, *voir* énumération

for, 118
foreach, 127

if, 113
import, 60
import static, 86
initialiseur, 212
Integer, 195
  parseInt, 196

JML, 172
  old, 173
  result, 173

liaison statique, 33

méthode, 31
  d'instance, 31
  de classe, 78, 82
  passage de paramètre, 216
méthode principale, 96
main, *voir* méthode principale

objet, 23
  création, 47
  initialisation, 213
  notation graphique, 25
old, 173
opérateur, 105–112
  priorité, 112

paquetage, 58
paramètre implicite, *voir* this
passage de paramètre, 216
poignée, 24
programmation par contrat, 163
  bénéfices, 169
  difficultés, 169
  invariant, 164
  JML, *voir* JML
  post-condition, 164
  pré-condition, 164
protection en écriture, 66

result, 173

static, 78
  fabrique statique, 202
  import static, 86
String, 189
StringBuffer, 230
StringBuilder, 228
surcharge, 36
  résolution, 37
switch, 114

tableau, 121
  plusieurs dimensions, 124
this, 34
toString, 35
types primitifs, 93

while, 116