Chapitre 4

Programmation par objets

La programmation par objets est une méthodologie qui fonde la structure des programmes autour des objets. Dans ce chapitre, nous présenterons les éléments de base d'un langage de classe : objets, classes, instances et méthodes.

4.1 OBJETS ET CLASSES

Nous avons vu qu'un objet élémentaire n'est accessible que dans sa totalité. Un objet structuré est, par opposition, construit à partir d'un ensemble fini de composants, accessibles indépendamment les uns des autres.

Dans le monde de la programmation par objets, un programme est un système d'interaction d'une collection d'*objets dynamiques*. Selon B. MEYER [Mey88, Mey97], chaque objet peut être considéré comme un *fournisseur* de services utilisés par d'autres objets, les *clients*. Notez que chaque objet peut être à la fois fournisseur et client. Un programme est ainsi vu comme un ensemble de relations contractuelles entre fournisseurs de services et clients.

Les services offerts par les objets sont, d'une part, des données, de type élémentaire ou structuré, que nous appellerons des *attributs*, et d'autre part, des actions que nous appellerons *méthodes*. Par exemple, un rectangle est un objet caractérisé par deux attributs, sa largeur et sa longueur, et des méthodes de calcul de sa surface ou de son périmètre.

Les langages de programmation par objets offrent des moyens de description des objets manipulés par le programme. Plutôt que de décrire individuellement chaque objet, ils fournissent une construction, la *classe*, qui décrit un ensemble d'objets possédant les mêmes propriétés. Une classe comportera en particulier la déclaration des données et des méthodes. Les langages de programmation à objets qui possèdent le concept de classe sont appelés *langages de classes*.

La déclaration d'une classe correspond à la déclaration d'un nouveau type. L'ensemble des rectangles pourra être décrit par une déclaration de classe comprenant deux attributs de type réel :

```
classe Rectangle
     largeur, longueur type réel
finclasse Rectangle
```

La classe `Rectangle` fournira comme service à ses clients, l'accès à ses attributs : sa longueur et sa largeur. Nous verrons dans la section 7.2 comment déclarer les méthodes. La déclaration d'une variable `r` de type `Rectangle` s'écrit de la façon habituelle :

```
variable r type Rectangle
```

4.1.1 Création des objets

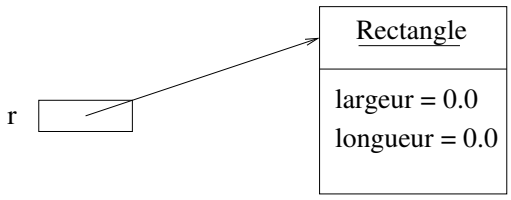
Il est important de bien comprendre la différence entre les notions de classe et d'objet[1]. Pour nous, les classes sont des descriptions purement *statiques* d'ensembles possibles d'objets. Leur rôle est de définir de nouveaux types. En revanche, les objets sont des *instances* particulières d'une classe. Les classes sont un peu comme des « moules » de fabrication dont sont issus les objets. En cours d'exécution d'un programme, seuls les objets existent.

La déclaration d'une variable d'une classe donnée *ne crée pas* l'objet. L'objet, instance de la classe, doit être explicitement créé grâce à l'opérateur **`créer`**. *Chaque* objet créé possède tous les attributs de la classe dont il est issu. Chaque objet possède donc ses propres attributs, distincts des attributs d'un autre objet : deux rectangles ont chacun une largeur et une longueur qui leur est propre.

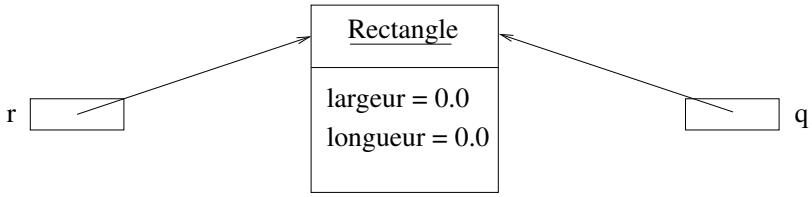
Les attributs de l'objet prennent des valeurs initiales données par un *constructeur*. Pour chaque classe, il existe un constructeur par défaut qui initialise les attributs à des valeurs initiales par défaut. Ainsi la déclaration suivante :

```
variable r type Rectangle créer Rectangle()
```

définit une variable `r` qui désigne un objet créé de type `Rectangle`, et dont les attributs sont initialisés à la valeur réelle 0 par le constructeur `Rectangle()`. La figure 7.1 montre une instance de la classe `Rectangle` avec ses deux attributs initialisés à 0. La variable `r` qui désigne l'objet créé est une *référence* à l'objet, et non pas l'objet lui-même.



Le fonctionnement d'une affectation d'objets de type classe n'est plus tout à fait le même que pour les objets élémentaires. Ainsi, l'affectation `q ←r`, affecte à `q` la référence à l'objet désigné par `r`. Après l'affectation, les références `r` et `q` désignent le même objet (voir la figure 7.2).



4.1.2 Destruction des objets

Que faire des objets qui ne sont plus utilisés ? Ces objets occupent inutilement de la place en mémoire, et il convient de la récupérer. Selon les langages de programmation, cette tâche est laissée au programmeur ou traitée automatiquement par le support d'exécution du langage. La première approche est dangereuse dans la mesure où elle laisse au programmeur la responsabilité d'une tâche complexe qui pose des problèmes de sécurité. Le programmeur est-il bien sûr que l'objet qu'il détruit n'est plus utilisé ? Au contraire, la seconde approche simplifiera l'écriture des programmes et offrira bien évidemment plus de sécurité.

Notre notation algorithmique ne tiendra pas compte de ces problèmes de gestion de la mémoire et aucune primitive ne sera définie.

4.1.3 Accès aux attributs

L'accès à un attribut d'un objet est valide si ce dernier existe, c'est-à-dire s'il a été créé au préalable. Cet accès se fait simplement en le *nommant*. Dans une classe, toute référence aux attributs définis dans la classe elle-même s'applique à l'instance courante de l'objet, que nous appellerons l'*objet courant*.

En revanche, l'accès aux attributs depuis des clients, *i.e.* d'autres objets, se fait par une notation pointée de la forme `n.a`, où `n` est un nom qui désigne l'objet, et `a` un attribut particulier.

La liste des services fournis aux clients est contrôlée explicitement par chaque objet. Par défaut, nous considérerons que tous les attributs d'une classe sont *privés*, et ne sont pas directement accessibles. Les attributs accessibles par les clients seront explicitement nommés dans une clause **`public`**. Pour rendre les deux attributs de la classe `Rectangle` publics, nous écrirons :

```
classe Rectangle
  public  largeur, longueur
  largeur, longueur type réel
finclasse Rectangle
```

Ainsi, dans cette classe, la notation `largeur` désigne l'attribut de même nom de l'objet courant. La notation `r.largeur` désigne l'attribut `largeur` de l'objet désigné par la variable `r` déclarée précédemment.

4.1.4 Attributs de classe partagés

Nous avons vu que chaque objet possède ses propres attributs. Toutefois, il peut être intéressant de partager un attribut de classe par toutes les instances d'une classe. Nous appellerons attribut *partagé*, un attribut qui possède cette propriété. Un tel attribut possédera une information représentative de classe tout entière. Le mot-clé **`partagé`** devra explicitement précéder la déclaration des attributs partagés.

4.1.5 Les classes en Java

La syntaxe des classes en JAVA suit celle des classes données ci-dessus dans notre pseudo-langage. La classe `Rectangle` s'écrira :

```
class Rectangle {
  public double largeur, longueur;
}
```

La création d'un objet se fait grâce à l'opérateur **`new`** et son initialisation grâce à un constructeur dont le nom est celui de la classe. La variable `r` précédente est déclarée et initialisée en JAVA de la façon suivante :

```
Rectangle r = new Rectangle();
```

La gestion de la destruction des objets est laissée à la charge de l'interprète du langage JAVA. Les objets qui deviennent inutiles, parce qu'ils ne sont plus utilisés, sont automatiquement détruits par le support d'exécution. Toutefois, il est possible de définir dans chaque classe la méthode `finalize`, appelée immédiatement avant la destruction de l'objet, afin d'exécuter des actions d'achèvement particulières.

L'accès aux attributs d'un objet, appelés *membres* dans la terminologie JAVA, se fait par la notation pointée vue précédemment. L'autorisation d'accès à un membre par des clients est explicitée grâce au mot-clé **`public`** ou **`private`**[2] placé devant sa déclaration.

Par convention, l'objet courant est désigné par le nom **`this`**. Ainsi, les deux notations `largeur` et **`this`**`.largeur` désignent le membre `largeur` de l'objet courant. On peut considérer que chaque objet possède un attribut **`this`** qui est une référence sur lui-même.

Un membre précédé du mot-clé **`static`** le rend partagé par tous les objets de la classe. On peut accéder à un membre **`static`** même si *aucun* objet n'a été créé, en utilisant dans la notation pointée le nom de classe. Ainsi, dans l'instruction `System.out.println()`, `System` est le nom d'une classe dans laquelle est déclarée la variable statique `out`.

4.2 LES MÉTHODES

Le second type de service offert par un objet à ses clients est un ensemble d'opérations sur les attributs. Le rôle de ces opérations, appelées *méthodes*, est de donner le modèle opérationnel de l'objet.

Nous distinguerons deux types de méthodes, les procédures et les fonctions. Leurs déclarations et les règles de transmission des paramètres suivent les mêmes règles que celles énoncées au chapitre 6. Toutefois, nous considérerons, d'une part, que les procédures réalisent une action sur l'état de l'objet, en modifiant les valeurs des attributs de l'objet, et d'autre part, que les fonctions se limitent à renvoyer un état de l'objet. Les procédures modifieront directement les attributs de l'objet sans utiliser de paramètres résultats.

Notez que ce modèle ne pourra pas toujours être suivi à la lettre. Certaines fonctions pourront être amenées à modifier des attributs, et des procédures à ne pas modifier l'état de l'objet.

Pour toutes les instances d'une même classe, il n'y a qu'un exemplaire de chaque méthode de la classe. Contrairement aux attributs d'instance qui sont propres à chaque objet, les objets d'une classe partagent la même méthode d'instance.

Nous pouvons maintenant compléter notre classe `Rectangle` avec, par exemple, deux fonctions qui renvoient le périmètre et la surface du rectangle, et deux procédures qui modifient respectivement la largeur et la longueur du rectangle courant. Notez que la définition de ces deux dernières méthodes n'est utile que si les attributs sont privés.

```
classe Rectangle
  public périmètre, surface, changerLargeur, changerLongueur
  {les attributs}
  largeur, longueur type réel

  {les méthodes}
  fonction périmètre() : réel
  {Rôle: renvoie le périmètre du rectangle}
      rendre 2×(largeur+longueur)
  finfonc {périmètre}

  fonction surface() : réel
  {Rôle: renvoie la surface du rectangle}
      rendre largeur×longueur
  finfonc {surface}

  procédure changerLargeur(donnée lg : réel)
  {Rôle: met la largeur du rectangle à lg}
      largeur ← lg
  finproc

  procédure changerLongueur(donnée lg : réel)
  {Rôle: met la longueur du rectangle à lg}
      longueur ← lg
  finproc
finclasse Rectangle
```

Si les méthodes possèdent des en-têtes différents, bien qu'elles aient le *même* nom, elles sont considérées comme distinctes et dites *surchargées*. La notion de surcharge est normalement utilisée lorsqu'il s'agit de définir plusieurs mises en œuvre d'une même opération. La plupart des langages de programmation la propose implicitement pour certains opérateurs ; traditionnellement, l'opérateur + désigne l'addition entière et l'addition réelle, ou encore la concaténation de chaînes de caractères, comme en JAVA. En revanche, peu de langages proposent aux programmeurs la surcharge des fonctions ou des procédures. Le langage EIFFEL l'interdit même, arguant que donner aux programmeurs la possibilité du choix d'un même nom pour deux opérations différentes est une source de confusion. Dans une classe, chaque propriété doit posséder un nom unique[3].

4.2.1 Accès aux méthodes

L'accès aux méthodes suit les mêmes règles que celles aux attributs. Dans la classe, toute référence à une méthode s'applique à l'instance courante de l'objet, et depuis un client il faudra utiliser la notation pointée. Pour rendre les méthodes accessibles aux clients, il faudra, comme pour les attributs, les désigner publiques.

```
r.changerLargeur(2.5)
r.changerLongueur(7)
{r désigne un rectangle de largeur 2.5 et de longueur 7}
```

4.2.2 Constructeurs

Nous avons déjà vu que lors de la création d'un objet, les attributs étaient initialisés à des valeurs par défaut, grâce à un constructeur par défaut. Mais, il peut être utile et nécessaire pour une classe de proposer son propre constructeur. Il est ainsi possible de redéfinir le constructeur par défaut de la classe `Rectangle` pour initialiser les attributs à des valeurs autres que zéro.

```
classe Rectangle
  public
        périmètre, surface, changerLargeur, changerLongueur
  {les attributs}
  largeur, longueur type réel
  constructeur Rectangle()
     largeur ← 1
     longueur ← 1
  fincons
  {les méthodes}
   ...
finclasse Rectangle
```

Bien souvent, il est souhaitable de proposer plusieurs constructeurs aux utilisateurs d'une classe. Malgré les remarques précédentes, nous considérerons que toutes les classes peuvent définir un ou plusieurs constructeurs, dont les noms sont celui de la classe, selon le mécanisme de surcharge. La classe `Rectangle` pourra définir, par exemple, un second constructeur

pour permettre aux clients de choisir leurs propres valeurs initiales. Les paramètres d'un constructeur sont implicitement des paramètres « données ».

```
  constructeur Rectangle(données lar, lon : réel)
     largeur ← lar
     longueur ← lon
  fincons
```

Les créations de rectangles suivantes utilisent les deux constructeurs définis précédemment :

```
variables
       r type Rectangle créer Rectangle()
       {r désigne un rectangle (1,1)}
       s type Rectangle créer Rectangle(3,2)
       {s désigne un rectangle (3,2)}
```

4.2.3 Les méthodes en Java

La déclaration de la classe `Rectangle` s'écrit en JAVA comme suit :

```
class Rectangle {
    public double largeur, longueur;
    // les constructeurs
    public Rectangle() {
        largeur = longueur = 1;
    }
    public Rectangle(double lar, double lon) {
        largeur = lar;  longueur = lon;
    }
    // les méthodes
    public double périmètre() {
    // Rôle : renvoie le périmètre du rectangle
      return 2*(largeur+longueur);
    }
    public double surface() {
    // Rôle : renvoie la surface du rectangle
      return largeur*longueur;
    }
    public void changerLargeur(double lg) {
    // Rôle : met la largeur du rectangle à lg
      largeur = lg;
    }
    public void changerLongueur(double lg) {
    // Rôle : met la longueur du rectangle à lg
      longueur = lg;
    }
} // fin classe Rectangle
```

Les déclarations des méthodes débutent par le type du résultat renvoyé pour les fonctions, ou par **void** pour les procédures. Suit le nom de la méthode et ses paramètres formels pla-

cés entre parenthèses. Les noms des paramètres sont précédés de leur type, comme pour les attributs, et séparés par des virgules.

La transmission des paramètres se fait *toujours par valeur*. Mais, précisons que dans le cas d'un objet non élémentaire (*i.e.* défini par une classe), la valeur transmise est la *référence* à l'objet, et *non pas* l'objet lui-même.

La méthode `changerLongueur` est une procédure à un paramètre transmis par valeur, et la méthode `périmètre` est une fonction sans paramètre qui renvoie un réel double précision.

Le corps d'une méthode est parenthésé par des accolades ouvrantes et fermantes. Il contient une suite de déclarations locales et d'instructions. Dans une fonction, l'instruction **`return`** `e;` termine l'exécution de la fonction, et renvoie le résultat de l'évaluation de l'expression `e`. Le type de l'expression `e` doit être compatible avec le type de valeur de retour déclaré dans l'en-tête de la fonction. Une procédure peut également exécuter l'instruction **`return`**, mais sans évaluer d'expression. Son effet sera simplement de terminer l'exécution de la procédure.

Les déclarations des constructeurs suivent les règles précédentes, mais le nom du constructeur n'est précédé d'aucun type, puisque c'est lui-même un nom de type (*i.e.* celui de la classe). Notez que le constructeur de l'objet courant est désigné par **`this`**`()`.

Les méthodes et les constructeurs peuvent être surchargés dans une même classe. Leur distinction est faite sur le nombre et le type des paramètres, mais, attention, pas sur celui du type du résultat.

JAVA permet de déclarer une méthode statique en faisant précéder sa déclaration par le mot-clé **`static`**. Comme pour les attributs, les méthodes statiques existent indépendamment de la création des objets, et sont accessibles en utilisant dans la notation pointée le nom de classe. La méthode `main` est un exemple de méthode statique. Elle doit être déclarée statique puisqu'aucun objet de la classe qui la contient n'est créé. Notez que la notion de méthode statique remet en cause le modèle objet, puisqu'il est alors possible d'écrire un programme JAVA sans création d'objet, et exclusivement structuré autour des actions.

4.3 ASSERTIONS SUR LES CLASSES

Nous avons déjà dit que la validité des programmes se démontre de façon formelle, à l'aide d'assertions. Les assertions sur les actions sont des affirmations sur l'état du programme avant et après leur exécution.

De même, il faudra donner des assertions pour décrire les propriétés des objets. B. MEYER[4] nomme ces assertions des *invariants de classe*. Un invariant de classe est un ensemble d'affirmations, mettant en jeu les attributs et les méthodes publiques de la classe, qui décrit les propriétés de l'objet. L'invariant de classe doit être vérifié :

– après l'appel d'un constructeur ;
– avant et après l'exécution d'une méthode publique.

Dans la mesure où les dimensions de rectangle doivent être positives, un invariant possible pour la classe `Rectangle` est :

{largeur ⩾ 0 et longueur ⩾ 0}

Le modèle de programmation contractuelle établit une relation entre une classe et ses clients, qui exprime les droits et les devoirs de chaque partie. Ainsi, une classe peut dire à ses clients :

- Mon invariant de classe est vérifié. Si vous me promettez de respecter l'antécédent de la méthode *m* que vous désirez appeler, je promets de fournir un état final conforme à l'invariant de classe *et* au conséquent de *m*.
- Si vous respectez l'antécédent du constructeur, je promets de créer un objet qui satisfait l'invariant de classe.

Si ce contrat passé entre les classes et les clients est respecté, nous pourrons garantir la validité du programme, c'est-à-dire qu'il est conforme à ses spécifications. Toutefois, deux questions se posent. Comment vérifier que le contrat est effectivement respecté ? Et que se passe-t-il si le contrat n'est pas respecté ?

La vérification doit se faire de façon automatique, le langage de programmation devant offrir des mécanismes pour exprimer les assertions et les vérifier. Il est à noter que très peu de langages de programmation offrent cette possibilité.

Si le contrat n'est pas respecté[5], l'exécution du programme provoquera une erreur. Toutefois, peut-on quand même poursuivre l'exécution du programme lorsque la rupture du contrat est avérée, tout en garantissant la validité du programme ? Nous verrons au chapitre 13, comment la notion d'exception apporte une solution à ce problème.

4.4 EXEMPLES

Dans le chapitre précédent, nous avons vu comment structurer en sous-programmes la résolution d'une équation du second degré et le calcul de la date du lendemain. Nous reprenons ces deux exemples en les réorganisant autour des objets.

4.4.1 Équation du second degré

L'objet central de ce problème est l'équation. On considère que chaque équation porte ses racines. Ainsi, chaque objet de type `Éq2Degré` possède comme attributs les solutions de l'équation, qui sont calculées par le constructeur. Ce constructeur possède trois paramètres qui sont les trois coefficients de l'équation. La classe fournit une méthode pour afficher les solutions.

```
classe Éq2Degré
{Invariant de classe:  ax²+bx+c=0 avec a≠0}
public  affichersol

r1, r2, i1, i2 type réel

constructeur Éq2Degré(données a, b, c : réel)
```

```
{Antécédent :  a≠0}
{Conséquent : (x - (r1+i×i1)) (x - (r2 +i×i2)) = 0}
     résoudre(a,b,c)
fincons

procédure résoudre(données a, b, c : réel)
{Antécédent: a, b, c coefficients réels de l'équation
             ax²+bx+c=0 et a≠0}
{Conséquent: (x-(r1+i×i1)) (x-(r2+i×i2)) = 0}
 constante
       ε = ? {dépend de la précision des réels sur la machine}
 variable Δ type réel {le discriminant}

     Δ ←carré(b)-4×a×c
     si Δ⩽0 alors {calcul des racines réelles}
        si b>0 alors r1 ← -(b+√Δ)/(2×a)
           sinon r1 ← (√Δ-b)/(2×a)
        finsi
        {r1 est la racine la plus grande en valeur absolue}
        si |r1|< ε alors r2 ←0
                  sinon r2 ← c/(a×r1)
        finsi
        i1 ← 0 i2 ← 0
        {(x-r1)(x-r2)=0}
     sinon  {calcul des racines complexes}
        r1 ← r2 ← -b/(2×a)
        i1 ← √-Δ/(2×a)
        i2 ← -i1
     finsi
     {(x-(r1+ii1)) (x-(r2+ii2)) = 0}
 finproc {résoudre}

 procédure affichersol()
  {Antécédent:  (x-(r1+i×i1)) (x-(r2+i×i2)) = 0}
  {Conséquent : les solutions (r1,i1) et (r2,i2) sont
                affichées sur la sortie standard}
      écrire(r1,i1)
      écrire(r2,i2)
  finproc
finclasse Éq2Degré
```

La traduction en JAVA de cet algorithme est immédiate et ne pose aucune difficulté :

```
class Éq2Degré {
    // Invariant de classe : ax²+bx+c=0 avec a≠0
    private double r1, r2, i1, i2;

    public Éq2Degré(double a, double b, double c)
    // Antécédent : a≠0
    // Conséquent : (x - (r1+i×i1)) (x - (r2+i×i2)) = 0
    { résoudre(a, b, c); }
```

```
    public void résoudre(double a, double b, double c)
    // Antécédent : a, b, c coefficients réels de l'équation
    // ax²+bx+c=0 et avec a≠0
    // Conséquent : (x-(r1+i×i1)) (x-(r2+i×i2)) = 0
    {
        final double ε = 1E-100;
        final double Δ = (b*b)-4*a*c;

        if (Δ>=0) { // calcul des racines réelles
            if (b>0) r1 = -(b+Math.sqrt(Δ))/(2*a);
            else r1 = (Math.sqrt(Δ)-b)/(2*a);
            // r1 est la racine la plus grande en valeur absolue
            r2 = Math.abs(r1) < ε ? 0 : c/(a*r1);
            i1=i2=0;
            // (x-r1)(x-r2)=0
        }
        else { // calcul des racines complexes
            r1 = r2 = -b/(2*a);
            i1=Math.sqrt(-Δ)/(2*a); i2=-i1;
        }
        // (x-(r1+ii×i1)) (x-(r2+i×i2)) = 0
    }
    public String toString()
    // Conséquent : renvoie une chaîne de caractères
    // qui contient les deux racines
    {
        return "r1=␣(" + r1 + "," + i1 + ")\n" +
               "r2=␣(" + r2 + "," + i2 + ")";
    }
} // fin classe Éq2Degré
```

Vous notez la présence de la méthode `toString` qui permet la conversion d'un objet `Éq2Degré` en une chaîne de caractères[6]. Celle-ci peut être utilisée implicitement par certaines méthodes, comme par exemple, la procédure `System.out.println` qui n'accepte en paramètre qu'une chaîne de caractères. Si le paramètre n'est pas de type `String`, deux cas se présentent : soit il est d'un type de base, et alors il est converti implicitement ; soit le paramètre est un objet, et alors la méthode `toString` de l'objet est appelée afin d'obtenir une représentation sous forme de chaîne de caractères de l'objet courant.

Nous pouvons écrire la classe `Test`, contenant la méthode `main`, pour tester la classe `Éq2Degré`.

```
import java.io.*;
class Test {
    public static void main (String[] args) throws IOException
    {
```

```
        Éq2Degré e = new Éq2Degré(StdInput.readDouble(),
                                  StdInput.readDouble(),
                                  StdInput.readlnDouble());
        // écrire les racines solutions sur la sortie standard
        System.out.println(e);
    }
} // fin classe Test
```

La variable `e` désigne un objet de type `Éq2Degré`. Les trois paramètres de l'équation sont lus sur l'entrée standard et passés au constructeur à la création de l'objet. L'appel de la méthode `println` a pour effet d'écrire sur la sortie standard les deux racines de l'équation `e`.

4.4.2 Date du lendemain

L'objet central du problème est bien évidemment la date. Nous définissons une classe `Date`, dont les principaux attributs sont trois entiers qui correspondent au jour, au mois et à l'année. Cette classe offre à ses clients les services de calcul de la date du lendemain et d'affichage (en toutes lettres) de la date. Notez que la liste de services offerte par cette classe n'est pas figée ; si nécessaire, elle pourra être complétée ultérieurement.

Le calcul de la date du lendemain modifie l'objet courant en ajoutant un jour à la date courante. La vérification de la validité de la date sera faite par le constructeur au moment de la création de l'objet. Ainsi, l'invariant de classe affirme que la date courante, représentée par les trois attributs `jour`, `mois` et `année`, constitue une date valide supérieure ou égale à une année minima. L'attribut qui définit cette année minima est une constante publique.

```
classe Date
 {Invariant de classe: les attributs jour, mois et année
                       représentent une date valide ⩾ annéeMin}
 public demain, afficher, annéeMin

 jour, mois, année type entier
 constante annéeMin = 1582

 constructeur Date(données j, m, a : entier)
 {Antécédent:}
 {Conséquent: jour = j, mois = m et année = a
              représentent une date valide}
 ...
 fincons

 procédure demain()
 {Antécédent : jour, mois, année représentent une date valide}
 {Conséquent : jour, mois, année représentent la date du lendemain}
 ...
 finproc
 procédure afficher()
 {Conséquent : la date courante est affichée sur la sortie standard}
  ...
 finproc
```

finclasse Date

Nous n'allons pas récrire les corps du constructeur et des deux méthodes dans la mesure où leurs algorithmes donnés page 61 restent les mêmes. Toutefois, le calcul de la date du lendemain nécessite de connaître le nombre de jours maximum dans le mois et de vérifier si l'année est bissextile. Le nombre de jours maximum est obtenu par la fonction publique `nbJoursDansMois`. Enfin, la fonction `bissextile` sera une méthode que l'on pourra aussi définir publique, puisque d'un usage général. Voici, l'écriture en JAVA de la classe `Date`.

```
/**
 * La classe Date représente des dates de la forme
 * jour mois année.
 * Invariant de classe: les attributs jour, mois et année
 * représentent une date valide >= annéeMin
 *
 */
class Date {
  private static final int AnnéeMin = 1582;
  private int jour, mois, année;
  /** Conséquent: jour = j, mois = m et année = a représentent
   *               une date valide
   */
  public Date(int j, int m, int  a) {
      if (a < annéeMin) {
          System.err.println("Année incorrecte");
          System.exit(1);
      }
      année = a;
      // l'année est bonne, tester le mois
      if (m<1 || m>12) {
          System.err.println("Mois incorrect");
          System.exit(1);
      }
      mois = m;
      // le mois est bon, tester le jour
      if (j<1 || j>nbJoursDansMois(mois)) {
          System.err.println("Jour incorrect");
          System.exit(1);
      }
      // le jour est valide
      jour = j;
  }
  /** Antécédent : 1 ⩽ m ⩽ 12
   *  Rôle : renvoie le nombre de jours du mois m
   */
  public int nbJoursDansMois(int m) {
      assert  1<=m && m<=12;
      switch (m) {
         case  1 : ;
```

```
        case  3 : ;
        case  5 : ;
        case  7 : ;
        case  8 : ;
        case 10 : ;
        case 12 : return 31;
        case  4 : ;
        case  6 : ;
        case  9 : ;
        case 11 : return 30;
        case  2 : return bissextile() ? 29 : 28;
     }
     return -1;
}
/** Antécédent : jour, mois, année représentent une date valide
 *  Conséquent : jour, mois, année représentent la date du lendemain
 */
public void demain() {
   assert jour<=nbJoursDansMois(mois) && 1<=mois &&
             mois<=12 && année>=annéMin;
   // jour ⩽ et mois dans [1,12] et année>=annéeMin
   if (jour<nbJoursDansMois(mois))
       // le mois et l'année ne changent pas
       jour++;
   else {
       // c'est le dernier jour du mois, il faut passer au
       // premier jour du mois suivant
       jour=1;
       if (mois<12) mois++;
       else {
            // c'est le dernier mois de l'année, il faut passer
            // au premier mois de l'année suivante
            mois=1; année++;
       }
     }
}
/** Rôle : teste si l'année courante est bissextile ou non */
public boolean bissextile() {
   return  année%4 == 0 && année%100 != 0
           || année%400 == 0;
}
/** Conséquent : renvoie la date courante sous forme
 *               d'une chaîne de caractères
 */
public String toString() {
   String smois="";
   switch (mois) {
     case 1  : smois = "janvier";   break;
     case 2  : smois = "février";   break;
     case 3  : smois = "mars";      break;
```

```
        case 4  : smois = "avril";     break;
        case 5  : smois = "mai";       break;
        case 6  : smois = "juin";      break;
        case 7  : smois = "juillet";   break;
        case 8  : smois = "août";      break;
        case 9  : smois = "septembre"; break;
        case 10 : smois = "octobre";   break;
        case 11 : smois = "novembre";  break;
        case 12 : smois = "décembre";
      }
      return jour + "␣" + smois + "␣" + année;
  }
} // fin classe Date
```

Le constructeur `Date` vérifie la validité de la date. Si la date est incorrecte, il se contente d'écrire un message sur la sortie d'erreur et d'arrêter brutalement le programme.

La classe de test donnée ci-dessous crée un objet de type `Date`. Le jour, le mois et l'année sont lus sur l'entrée standard. Elle calcule la date du lendemain et l'affiche.

```
import java.io.*;
class DateduLendemain {
  public static void main (String[] args) throws IOException
  {
      Date d = new Date(StdInput.readInt(),
                        StdInput.readInt(),
                        StdInput.readlnInt());
      d.demain();
      System.out.println(d);
  }
} // fin classe DateduLendemain
```

TD a rendre

Exercice 1. Ajoutez à la classe `Éq2Degré` les fonctions `premièreRacine` et `secondeRacine` qui renvoient, respectivement, la première et la seconde racine de l'équation.

Exercice 2. Définissez une classe `Complexe`, pour représenter les nombres de l'ensemble $\mathbb{C}$. Un objet complexe aura deux attributs, une partie réelle et une partie imaginaire. Vous définirez un constructeur par défaut qui initialisera les deux attributs à zéro, ainsi qu'un constructeur qui initialisera un nombre complexe à partir de deux paramètres réels. Vous écrirez la méthode `toString` qui donne une représentation d'un nombre complexe sous la forme `(r,i)`.

Exercice 3. Utilisez votre classe `Complexe` pour représenter les racines solutions de l'équation du second degré.