Haute École de Bruxelles
École Supérieure d'Informatique

Rue Royale, 67 – 1000 Bruxelles
02/219.15.46 – esi@heb.be

Bachelor en Informatique

Algorithmique II

DEV2 – 2014

Activité d'apprentissage enseignée par :
L. Beeckmans, M. Codutti, G. Cuvelier, E. Levy et F. Servais.

Ce syllabus a été écrit à l'origine par M. Monbaliu à une époque où le cours s'appelait « Logique et techniques de programmation ». Il a ensuite été adapté par Mme Leruste, M. Beeckmans et M. Codutti. Qu'ils en soient tous remerciés. Nous remercions également tous ceux qui ont contribué à son amélioration grâce à leur lecture attentive et leurs remarques.

Document produit avec LaTeX.
Version du 20 janvier 2015.

Table des matières

Chapitre 1

L'orienté objet

Dans ce chapitre, nous présentons les bases de la programmation orientée objet. Nous commençons par expliquer les motivations qui ont amené ce type de programmation avant d'entrer dans le vif du sujet en explicitant le concept d'*encapsulation*. Les autres piliers de l'orienté objet (*héritage* et *polymorphisme*) ne seront pas vus cette année.

1.1 Motivation

Depuis son apparition, la puissance de l'ordinateur n'a cessé de croitre exponentiellement. Les tâches qui lui sont confiées ont fait de même. Ainsi les programmes à écrire sont de plus en plus gros et de plus en plus complexes.

Face à la complexité, la démarche est toujours la même : découper le problème en sous-problèmes (qui peuvent à leur tour être découpés) ce qui permet

- ▷ d'attaquer chaque problème séparément en évitant la surcharge cognitive ;
- ▷ de répartir le travail entre plusieurs personnes ;
- ▷ de pouvoir réutiliser du travail déjà produit si un sous-problème est déjà apparu dans le cadre d'un autre problème ;
- ▷ de produire un code plus lisible car s'exprimant avec des termes de plus haut niveau, plus proches du problème à résoudre. Ainsi, là où un tri devra être fait, on trouvera le mot « trier » qui fera référence à la partie de code qui s'occupe du tri. Cela va dans le sens d'une plus grande « abstraction » du code : un code qui s'éloigne du langage simpliste compris par le processeur pour s'approcher de la pensée humaine et des termes du problème à résoudre.

Les langages de programmation ont suivi cette approche en permettant toujours plus d'abstraction. Dans le cours d'algorithmique de DEV1, on vous a présenté la notion de module qui permet de découper la tâche à réaliser en sous-tâches ainsi que la notion de structure qui permet de regrouper des données. Il s'agit là de deux approches dissociées.

C'est cette lacune que se propose de combler l'orienté objet : permettre de définir des **objets** (composés de **données** et **d'instructions**) qui sont proches du problème à résoudre. Cela va permettre une meilleure lisibilité et une plus grande concision du code. Ainsi on pourra définir les notions de date, d'employé, de fournisseur, de plateau de jeu, de pion, de livre, d'emprunteur, de carte à jouer, de chambre, de réservation, de vol, de produit, de stock, de ristourne, de facture, de panier d'achats, de compte en banque, de banque, de client, de portefeuille d'actions...

1.2 La notion d'objet

1.2.1 Définition

Un **objet** [1] est une entité logicielle qui :

- ▷ a une **identité** ; c'est-à-dire que nous pouvons identifier un objet par un nom (tout comme une variable possède un nom).
- ▷ est capable de sauvegarder un **état**, c'est-à-dire un ensemble d'informations dans des variables internes ;
- ▷ répond à des **messages** précis en déclenchant des activations internes appropriées qui peuvent changer l'état de l'objet. Ces opérations sont appelées des **méthodes**. Ce sont des fonctions liées à des objets et qui précisent le **comportement** de ces objets.

1.2.2 État

Un objet contient de l'information, des données qui définissent son état.

Exemples

- ▷ Pour un produit, l'état peut être : l'intitulé du produit, son code barre, son prix...
- ▷ Pour un employé, on peut avoir : son nom, son prénom, son adresse, sa date d'embauche, son salaire mensuel, sa fonction, son téléphone...
- ▷ Une carte à jouer a une couleur et une valeur.
- ▷ L'état d'une date est le jour du calendrier qu'elle représente.
- ▷ L'état d'une heure est le moment de la journée qu'elle représente.

L'état d'un objet est mémorisé via des variables qu'on appelle des *attributs*.

1.2.3 Attributs

Les **attributs** d'un objet sont l'ensemble des informations se présentant sous forme de variables et permettant de représenter l'état d'un objet.

Nous verrons plus loin la syntaxe précise pour définir les attributs d'un objet.

Exemples

- ▷ L'intitulé d'un produit peut être représenté par une chaine. C'est également le cas des nom(s) et prénom(s) d'un employé.
- ▷ La date d'embauche peut être représentée par un « objet date » (une date est rarement un type primitif du langage utilisé). Un attribut d'un objet peut être lui même un objet.
- ▷ Un moment de la journée peut aussi être un objet représenté par trois entiers [2] : les heures, les minutes et les secondes (en supposant qu'on désire une précision de l'ordre de la seconde).
- ▷ L'adresse d'un employé peut être représentée par une seule chaine mais également par un « objet adresse » (qui contiendrait : une rue, un numéro, un code postal...).

Remarque : Certaines parties de l'état peuvent évoluer au fil du temps. D'autres parties sont immuables. Ainsi l'adresse d'une personne peut changer mais pas sa date de naissance.

1. Les définitions sont tirées du livre de Cardon et Dabancourt (cf. bibliographie)
2. Toutefois, on verra que ce n'est peut-être pas la meilleure solution.

Exercices - attributs

1. Quel(s) attribut(s) prendriez-vous pour représenter (l'état d') une date ?
2. Et pour un dé à 6 faces ?
3. Et pour un produit de magasin ?
4. Et pour une télévision ? (on peut en trouver vraiment beaucoup !)

1.2.4 Comportement

Le **comportement** d'un objet est défini par l'ensemble des messages ou requêtes auxquels il peut répondre.

Pour ce faire, il exécute un module qui pourra éventuellement retourner une information à l'émetteur du message.

Les messages peuvent interroger l'objet, le modifier, lui demander d'agir sur son environnement (afficher du texte, modifier un fichier...).

Exemples

- Quels « messages » peut-on envoyer à une date ? On peut lui demander (entre autres) :
  - des informations sur le jour du mois, le mois, l'année, le jour de la semaine ;
  - si elle est antérieure ou non à une autre date ;
  - si elle fait partie d'une année bissextile ;
  - le nombre de jours qui la sépare de la fin de l'année ;
  - de passer au jour suivant, à la semaine suivante...
- Et pour un stock de produits ? On peut
  - lui demander la quantité disponible d'un produit donné ;
  - lui annoncer l'arrivée d'une quantité donnée d'un produit donné ;
  - lui indiquer qu'un produit n'existe plus (à retirer du stock) ;
  - lui demander d'enlever une certaine quantité d'un produit du stock.
- Et pour un employé ? On peut
  - lui demander son adresse, son salaire ou sa fonction...
  - augmenter son salaire ;
  - le changer de fonction ;
  - le licencier (penser à prévoir une date de départ dans l'état !).
- Pour un moment de la journée on peut demander s'il se situe le matin ou pas...

Exercices - comportement

1. Quel comportement voyez-vous pour un téléviseur ?
2. Et pour un produit de magasin ?

1.2.5 Méthode

Un message lance l'exécution d'un module appelé **méthode** dans le jargon de l'orienté objet.

Exemples

- ▷ Pour permettre à une date de passer au jour suivant, nous allons définir une méthode qui incrémente le jour du mois en tenant compte d'un possible basculement au mois suivant ou à l'année suivante.
- ▷ Pour calculer le bénéfice d'un produit, nous allons définir une méthode qui, à partir du prix d'achat et du prix de vente, calcule le bénéfice.
- ▷ Pour permettre à un moment d'indiquer s'il est le matin ou pas, nous allons définir une méthode comme celle-ci (nous verrons plus tard comment l'associer aux objets)

```
// On suppose que 'heure' est un des attributs utilisés
// pour représenter l'état (le moment dans la journée)
méthode estMatin() → booléen
    retourner heure < 12                    // on considère que midi est situé l'après-midi
fin méthode
```

Cet exemple devrait vous sembler familier à deux exceptions près

- ▷ on utilise le mot « méthode » en lieu et place de « module » ;
- ▷ les attributs (l'heure ici) ne sont pas passés en paramètre. Une méthode appartient à un objet et connait les attributs de cet objet. Nous verrons plus loin la syntaxe précise.

Exercices - méthodes

1. Dans le comportement d'un téléviseur, on retrouve « éteindre » et « allumer ». À quoi ressemblerait le code de ces méthodes ?
2. Écrivez la méthode qui permet de passer au jour suivant.
3. Écrivez la méthode qui calcule le bénéfice réalisé lors de la vente d'un produit.

1.2.6 Activer un comportement

Pour activer un comportement d'un objet, il faut lui envoyer un message (ou dit autrement, appeler une de ses méthodes). La syntaxe que nous allons utiliser (c'est la plus courante) est la notation pointée.

```
nomObjet.nomMéthode()
```

Exemple : Supposons que le nom « maintenant » désigne un objet contenant un moment de la journée (on verra comment réaliser cela). Si on veut savoir si on est le matin, on peut écrire

```
si maintenant.estMatin() alors
    ...
fin si
```

Exercice – activer un comportement

Écrire la portion de code qui allume une télévision (désignée par « maTélévision ») et puis l'éteint aussitôt après.

1.2.7 Les paramètres d'un comportement

Activer un comportement revient à appeler une méthode de l'objet. Souvent il est nécessaire d'envoyer à l'objet des informations complémentaires pour préciser notre demande ce qui se fait via l'utilisation des paramètres.

Exemple : Si on veut modifier le salaire d'un employé, il faut que notre message contienne le nouveau salaire. Autrement dit, il faut communiquer ce nouveau salaire à la méthode de changement du salaire. Ce qui donne la méthode suivante :

```
méthode modifierSalaire(nouveauSalaire : entier)
    salaire ← nouveauSalaire
fin méthode
```

Exercices – paramètres du comportement

1. Prenons un objet représentant un produit de magasin. Nous supposerons qu'un produit a un *numéro*, un *libellé*, un *prixAchat*, un *prix de vente* et une *quantitéEnStock*. Donnez les **entêtes** des méthodes suivantes qui permettent de :
   - ▷ obtenir le prix de vente
   - ▷ calculer le bénéfice
   - ▷ donner la quantité restant en stock
   - ▷ dire si le produit est en rupture de stock.
2. Prenons un objet représentant une date du calendrier grégorien. Donnez les entêtes des méthodes suivantes qui permettent de :
   - ▷ demander le nom du jour correspondant (par exemple "lundi", "mardi"...)
   - ▷ savoir si une date est antérieure à une autre
   - ▷ connaitre le nombre de jours (absolu) séparant deux dates.
3. Utilisation. Soit deux dates *date*1 et *date*2 ; écrivez la portion de code qui utilise les méthodes ci-dessus pour
   - ▷ vérifier quelle date précède l'autre ;
   - ▷ calculer le nombre de jours d'écart entre ces deux dates.

1.3 L'encapsulation

Un objet possède un état qui est représenté par des attributs. Les bonnes pratiques de la programmation orientée objet préconisent fortement que les attributs d'un objet soient invisibles en dehors de l'objet. Ils ne pourront être accédés qu'au travers du comportement de l'objet, c'est-à-dire via ses méthodes.

Lorsque les détails de l'implémentation d'un objet sont masqués aux autres objets, on dit qu'il y a encapsulation des données et du comportement des objets.

Pourquoi une telle recommandation ? Le but est de garantir la cohérence de l'état de l'objet. Si on pouvait accéder directement à un attribut (et donc le modifier), on pourrait y mettre une valeur incohérente. Par exemple, on pourrait dire que les minutes d'un moment valent -3 ou 75 ou encore que le jour d'une date est 32 !

Dès lors, il nous faudra préciser pour chaque **membre** (attributs et méthodes) d'un objet s'il est **privé** (inconnu de l'extérieur) ou **public** (connu de l'extérieur).

Le bon usage impose que tous les attributs soient rendus privés et que les méthodes restent publiques. Toutefois, on pourra trouver également des méthodes privées. Ce sera notamment le cas si plusieurs méthodes d'un objet ont une partie commune ; il sera intéressant de la *factoriser*, c-à-d en faire une méthode privée (ex : un calcul de maximum).

Puisqu'un attribut est privé, il est courant pour chacun des attributs de rencontrer une méthode destinée à connaitre la valeur de cet attribut et une autre qui permet de la modifier.

1.3.1 Accesseur et mutateur

Accesseur [3] : méthode dont le but est de fournir la valeur d'un attribut.

Mutateur [4] : méthode dont le but est de modifier la valeur d'un attribut.

Par convention, ces méthodes sont nommées getNom et setNom où « nom » est le nom de l'attribut [5] Par facilité, on utilisera parfois le terme « accesseur » pour désigner à la fois les « accesseurs » et les « mutateurs ».

Exemple : Écrivons l'accesseur et le mutateur pour l'attribut « heure » d'un moment de la journée.

```
méthode getHeure() → entier
    retourner heure
fin méthode
```

```
méthode setHeure(uneHeure : entier)
    heure ← uneHeure
fin méthode
```

1.3.2 Que faire si le paramètre est invalide ?

Dans l'exemple précédent, que se passerait-il si le paramètre uneHeure vaut 25 ? Une valeur aberrante serait affectée à l'attribut heure.

Dans le cas de paramètres invalides, la plus mauvaise solution est de ne rien faire. Le programme continuerait en croyant que tout s'est bien passé et il court à la catastrophe. Il est préférable qu'un programme s'interrompe plutôt que de fournir une mauvaise réponse.

Dans certains langages (comme le C), l'usage est que chaque module retourne un entier indiquant s'il y a eu une erreur (et laquelle). L'inconvénient est que le module appelant n'est pas obligé de tenir compte de l'erreur.

Les **exceptions** sont un mécanisme du même genre mais qui oblige à fournir un code de traitement de l'erreur. Il ne sera pas étudié en première année [6].

3. On utilise aussi souvent le mot anglais « getter ».
4. On utilise aussi souvent le mot anglais « setter »
5. Pour un attribut booléen, on pourra préférer estNom ou isNom au lieu de getNom.
6. En tout cas pas au cours d'algorithmique mais vous étudierez cette notion au cours de Java.

Cette année, nous nous contenterons d'indiquer clairement dans nos codes qu'il s'agit d'une situation anormale via la primitive erreur qui arrête le déroulement du programme avec une courte explication du problème.

La syntaxe que nous allons retenir est

```
erreur "explication de l'erreur"
```

Ce qui donne :

```
méthode setHeure(uneHeure : entier)
    si uneHeure < 0 OU uneHeure > 23 alors
        erreur "heure invalide"
    fin si
    heure ← uneHeure
fin méthode
```

1.4 La notion de classe et d'instance

Pour pouvoir utiliser des objets nous allons devoir les définir (expliciter leur état et leur comportement). Cette définition est commune à tous les objets similaires. Par exemple tous les moments ont un même comportement et un même type d'état (des heures, des minutes et des secondes).

Une classe est un ensemble d'objets qui ont en commun les mêmes méthodes et qui partagent les mêmes types d'attributs.

Une **instance**[7] d'une classe est un objet particulier d'une classe qui peut activer les méthodes de la classe et qui a des valeurs particulières pour ses attributs.

On peut établir le parallélisme avec les types de base que vous avez déjà vus. Définir une classe revient à définir un nouveau type de données. En gros, on peut dire qu'un **objet est à une classe ce qu'une variable est à un type**.

Comprenons bien que les objets d'une même classe ont le même « type » d'état mais pas le même état proprement dit. Deux objets « moment » représentent tous deux un moment (heures, minutes, secondes) de la journée mais pas (forcement) le même ! Ils auront donc les mêmes attributs mais avec des valeurs différentes !

1.4.1 Définition d'une classe

Nous devons d'abord définir une classe avant de pouvoir en instancier les objets que nous voulons utiliser. Précisons la syntaxe utilisée pour définir une classe

```
classe NomDeLaClasse
 privé:
    // liste des attributs (donc privés par convention)
 public:
    // liste des méthodes publiques
 privé:
    // liste des méthodes privées
fin classe
// Par souci de lisibilité, on pourra indiquer uniquement les entêtes des
// méthodes et donner le code complet des méthodes à la suite de la classe.
```

Exemple : la classe Moment qui représente un moment de la journée.

7. Vous pouvez considérer les termes « instance de classe » et « objet » comme synonymes.

```
classe Moment
 privé:
    heure : entier
    minute : entier
    seconde : entier
 public:
    méthode getHeure() → entier
    méthode getMinute() → entier
    méthode getSeconde() → entier
    méthode setHeure(uneHeure : entier)
    méthode setMinute(uneMinute : entier)
    méthode setSeconde(uneSeconde : entier)
    méthode estMatin() → booléen
fin classe

méthode estMatin() → booléen
    retourner heure < 12
fin méthode
// + les accesseurs et les mutateurs
```

1.4.2 Instanciation d'une classe

« Instancier » signifie créer un objet d'une classe. Cela s'écrit avec l'instruction nouveau. Pour lui donner un nom, on l'assigne à une variable déclarée du type de la classe.

```
nomObjet : nomClasse                          // déclaration de l'objet
nomObjet ← nouveau nomClasse()                // instanciation de l'objet
```

Dans ce cours de Logique, nous adopterons le fait que les noms des paramètres soient différents de ceux des attributs (on préconisera d'imaginer des noms variés tels que uneDate, maDate, laListe, autreObjet...), ce qui évitera toute ambiguïté (entre minuscule et majuscule par exemple).

Exemple : pour créer un moment de la journée.

```
module test()
    midi : Moment                                  // déclaration
    midi ← nouveau Moment()                        // instanciation
    midi.setHeure( 12 )                            // mutateur
    midi.setMinute( 0 )                            // " "
    midi.setSeconde( 0 )                           // " "
    si midi.estMatin() alors
        afficher "Midi est considéré comme étant encore le matin"
    sinon
        afficher "Midi est considéré comme étant l'après-midi"
    fin si
fin module
```

Exercices – classe et instance

1. Pour les produits, vous avez déjà écrit les attributs et les en-têtes des méthodes. Regroupez le tout en une classe Produit **en respectant les notations que vous venez de voir.**
2. Écrivez un module qui affiche le prix d'achat d'un produit, son prix de vente hors TVA et son prix de vente TVA comprise.

1.5 Les constructeurs

L'encapsulation nous permet de contrôler l'état de l'objet et de l'empêcher de tomber dans un état invalide. Mais qu'en est-il de l'état de départ ? Est-il valide ?

Il serait bon, lorsqu'on crée un objet (via nouveau) de pouvoir indiquer l'état initial de l'objet et que cet état puisse être validé. C'est le rôle précis des constructeurs.

Un **constructeur** est une méthode particulière permettant d'initialiser les attributs d'un objet lors de sa création effective. Elle porte le même nom que sa classe et ne retourne pas de valeur.

Il peut y avoir plusieurs constructeurs ce qui permet d'offrir plusieurs possibilités d'indiquer l'état initial de l'objet. Toutefois, nous limiterons au maximum le nombre de constructeurs dans une classe.

Remarquez que cela demande de définir plusieurs méthodes qui portent le même nom.

Surcharge : le fait de définir plusieurs méthodes portant le même nom. On doit pouvoir les différencier via leurs paramètres.

Exemple : Écrivons des constructeurs pour un moment de la journée :

```
classe Moment
 privé:
    // pas de changement
    heure : entier
    minute : entier
    seconde : entier
 public:
    constructeur Moment(uneHeure, uneMinute, uneSeconde : entiers)
    constructeur Moment(uneHeure, uneMinute : entiers)          // 0 seconde par défaut
    constructeur Moment(uneHeure : entier)                  // initialiser à une heure pile

    // pas de changement au niveau des méthodes :
    méthode getHeure() → entier
    méthode getMinute() → entier
    méthode getSeconde() → entier
    méthode setHeure(uneHeure : entier)
    méthode setMinute(uneMinute : entier)
    méthode setSeconde(uneSeconde : entier)
    méthode estMatin() → booléen
fin classe
```

```
constructeur Moment(uneHeure, uneMinute, uneSeconde : entiers)
    setHeure(uneHeure)
    setMinute(uneMinute)
    setSeconde(uneSeconde)
fin constructeur

constructeur Moment(uneHeure, uneMinute : entiers)
    setHeure(uneHeure)
    setMinute(uneMinute)
    setSeconde(0)
fin constructeur

constructeur Moment(uneHeure : entier)
    setHeure(uneHeure)
    setMinute(0)
    setSeconde(0)
fin constructeur

// + les accesseurs, les mutateurs et les autres méthodes
```

Contrairement à ce qu'on peut trouver dans certains langages, comme Java par exemple, nous n'autorisons pas ici d'appel d'un constructeur d'une classe A dans un autre constructeur de cette même classe A.

Par contre, il est courant en logique qu'un constructeur appelle les mutateurs afin d'effectuer les tests sans avoir à les dupliquer. Mais c'est une démarche que vous éviterez de faire dans des langages comme Java par exemple (cela vous sera expliqué plus tard).

Lorsqu'on instancie un objet, les paramètres qu'on donne déterminent le constructeur qui est effectivement utilisé pour initialiser l'état de l'objet.

Exemple : Instancions quelques moments de la journée.

```
heureDépart ← nouveau Moment(14, 23, 56)
heureLever ← nouveau Moment(9, 30)
heureGouter ← nouveau Moment(17)
```

Le fait qu'un objet est instancié via la primitive nouveau et pas implicitement à la déclaration permet de postposer sa construction effective au moment où l'état initial qu'on veut lui donner sera connu (ce qui peut résulter d'un calcul). On est ainsi assuré que tous les objets manipulés sont valides ce qui permet d'éviter les situations où une méthode fait des dégâts suite à la manipulation d'un objet invalide.

Exercices - constructeur

1. Écrivez un ou des constructeur(s) pour un Produit
2. Adaptez le module écrit plus haut pour qu'il affiche le prix hors TVA puis le prix TVA comprise du produit numéro 105176 (Lego réveil figurine policier) au prix d'achat de 25€, au prix de vente de 30€et dont il y a 10 exemplaires en stock.

1.6 Du choix de la représentation de l'état

Lorsqu'on définit une classe, il faut choisir les attributs qui vont permettre de représenter l'état des objets. Cela peut paraitre immédiat mais il n'en est rien.

Exemple

Pour un moment de la journée, nous avons choisi d'utiliser trois attributs entiers (les heures, les minutes et les secondes). Nous aurions tout aussi bien pu choisir d'utiliser un seul entier représentant le nombre de secondes écoulées depuis minuit.

Ces deux représentations sont tout-à-fait équivalentes en terme de potentiel mais la grande différence est l'efficacité du code des méthodes.

Prenons deux méthodes symptomatiques : celle qui donne l'heure et celle qui compare deux moments de la journée. La première est beaucoup plus simple à écrire et plus rapide avec la première représentation alors que la seconde méthode est plus simple à écrire et plus rapide avec la seconde représentation.

Dès lors, quelle représentation choisir ? Il faut examiner, pour chaque représentation possible, le nombre de méthodes qui sont efficaces mais aussi imaginer la fréquence de leur utilisation (ce qui est difficile et changeant). Heureusement, ce choix n'est pas définitif. Si on change d'avis, on peut changer la représentation. Il faudra bien sûr réécrire les méthodes de la classe mais il ne faudra rien changer au reste du code, c-à-d les lignes du code utilisant la classe. C'est d'ailleurs là une des grandes forces de la programmation orientée objet.

Exercices – représentation de l'état

1. Compléter la classe Moment en écrivant la méthode « getHeure » et celle qui compare deux moments pour les deux représentations imaginées ci-dessus.
2. Écrire le module qui crée deux moments de la journée et vérifie si le premier est avant le second. Ce code dépend-il des attributs choisis pour définir la classe Moment ?

Remarque

Précédemment, nous avons défini un **accesseur** comme une méthode permettant d'accéder à la valeur d'un attribut. Mais c'est au développeur de définir quels sont les attributs ; c'est totalement caché à l'utilisateur de la classe. On voit donc bien que cette notion d'accesseur n'a pleinement de sens qu'en interne, pour le développeur de la classe. Pour l'utilisateur il s'agit d'une méthode comme les autres.

1.7 Quelques éléments de syntaxe

Clarifions certaines notations liées aux objets.

- ▷ On peut directement afficher un objet. Cela affiche son état, c'est-à-dire les valeurs de ses attributs.

```
rendezVous : Moment
rendezVous ← nouveau Moment(14, 23, 56)
afficher rendezVous          // affichera 14, 23 et 56 dans un format lisible quelconque
```

- ▷ De même, on peut directement lire un objet, ce qui a pour effet de créer un objet avec un état correspondant aux valeurs lues pour ses attributs.

```
rendezVous : Moment
lire rendezVous
```

- ▷ Le signe « = » peut être utilisé pour comparer deux objets. Ils seront considérés comme égaux s'ils sont dans le même état, c'est-à-dire que leurs attributs ont la même valeur.

▷ **Un attribut privé n'est pas connu en dehors de la classe.**

Précisons : un attribut privé n'est connu que des instances de cette classe, ce qui signifie qu'il est également connu par tous les autres objets de la même classe.

Exemple : écrivons la méthode qui teste si un moment précède un autre (en supposant que l'état est représenté par un seul entier, totalSecondes, le nombre de secondes depuis minuit)

```
méthode estAntérieur(autre : Moment) → booléen
    retourner totalSecondes < autre.totalSecondes
    // c'est équivalent à retourner totalSecondes < autre.getTotalSecondes()
fin méthode
```

▷ Lorsqu'on déclare un objet, il n'est pas encore créé. On peut utiliser la valeur spéciale « rien » pour indiquer ou tester qu'un objet n'est pas encore créé.

```
moment : Moment                                          // moment = rien
moment ← nouveau Moment( 14, 23, 56 )                    // moment ≠ rien
si moment ≠ rien alors
    moment ← rien                                        // moment = rien
fin si
```

1.8 Représentation modélisée d'une classe

Un dessin étant souvent plus lisible qu'un texte, on peut représenter graphiquement une classe. Une notation courante est celle utilisée en UML [8]. Pour faire simple, une classe est représentée par un rectangle composé de 3 zones : la première pour le nom de la classe, la deuxième pour les attributs et la troisième pour les méthodes. On indique par un signe « + » (resp. « - ») que le membre est public (resp. privé)

Exemple

Moment
- heure : entier - minute : entier - seconde : entier
+ getHeure() → entier + setHeure(uneHeure : entier) + avancer1Heure() ...

Remarquons qu'on indique l'entête des méthodes mais pas le code associé. En fonction du niveau de détail désiré, on pourrait aussi omettre les paramètres et types de retour.

1.9 Un exemple complet : une durée

Examinons un exemple complet pour fixer les notions introduites par ce chapitre. Lors de l'apprentissage du pseudo-code, vous avez écrit quelques modules manipulant des heures (conversion du format HMS en nombre de secondes depuis minuit, conversion inverse, différence entre 2 heures, ...). Il est souvent utile, lorsqu'on développe un algorithme, d'avoir à sa disposition un tel type de données au même titre que les types prédéfinis. Faisons-le !

8. Unified Modeling Langage. On vous en parlera plus en détail au cours d'Analyse.

1.9.1 Ce que l'on veut vraiment

Avant tout, il faut bien préciser ce que l'on veut décrire. L'« heure » est un concept multi-facettes. Parle-t-on de l'heure comme moment dans la journée ou de l'heure comme représentant une durée ? Dans le premier cas, elle ne peut dépasser 24h et la différence entre 2 heures n'a pas de sens (ou plus précisément n'est pas une heure, mais une durée !). Dans le deuxième cas, on n'a pas ces contraintes. Nous allons ici adopter la deuxième approche et pour bien la distinguer, nous allons plutôt appeler cela une **durée**.

1.9.2 Le comportement (les méthodes)

La première question à se poser est celle des services qu'on veut fournir, c'est-à-dire des méthodes publiques de la classe. On doit pouvoir *construire* une durée. On doit pouvoir connaitre le nombre de jours, d'heures, minutes ou secondes correspondant à une durée. On doit pouvoir effectuer des calculs avec des durées (addition, soustraction). Enfin, on doit pouvoir comparer des durées. Arrêtons-nous là, mais en pratique, on pourrait trouver encore bon nombre d'autres méthodes qu'il serait intéressant de fournir. Ce qui nous donne jusqu'à présent

```
classe Durée
 privé:
    // rien encore
 public:
    constructeur Durée(secondes : entier)
    constructeur Durée(heure, minute, seconde : entiers)

    méthode getJour() → entier                          // nb de jours dans une durée
    méthode getHeure() → entier                          // entier entre 0 et 23 inclus
    méthode getMinute() → entier                         // entier entre 0 et 59 inclus
    méthode getSeconde() → entier                        // entier entre 0 et 59 inclus

    méthode getTotalJours() → entier                      // Le nombre total de jours
    méthode getTotalHeures() → entier                     // Le nombre total d'heures
    méthode getTotalMinutes() → entier                   // Le nombre total de minutes
    méthode getTotalSecondes() → entier                  // Le nombre total de secondes

    méthode ajouter(autreDurée : Durée)
    méthode différence(autreDurée : Durée) → Durée
    méthode égale(autreDurée : Durée) → booléen
    méthode plusPetit(autreDurée : Durée) → booléen
fin classe
```

Quelques remarques

- On a deux constructeurs, ce qui offre plus de souplesse pour initialiser un objet. Ceci est un exemple supplémentaire du concept de « **surcharge** ».
- Faisons bien la distinction entre les méthodes getXXX() et getTotalXXX(). Par exemple, la méthode getMinute() retourne la valeur de la composante « minutes » dans une représentation HMS tandis que la méthode getTotalMinutes() retourne le nombre total de minutes entières pour cette durée. Ex : pour 1h23'12", getMinute() retourne 23 et getTotalMinutes() retourne 83. Idem avec les jours, les heures et les secondes.
- Les méthodes getTotalXXX() retournent le nombre (toujours entier) de XXX contenus dans la durée. Exemple, avec la durée 0h23'52", getTotalMinutes() retourne 23 et pas 24 (autrement dit, il n'y a pas d'arrondi vers le haut).
- Il n'y a pas de *mutateur* (setXXX()). Ce qui signifie qu'on ne peut pas changer directement la valeur de l'objet après son initialisation. On aurait pu en définir mais nous n'avons pas jugé utile de le faire dans ce cas précis.

- ▷ La méthode ajouter() ne retourne rien. En effet, elle ajoute la durée à l'objet sur lequel est appelée la méthode. C'est un choix ; on aurait aussi pu dire que la méthode ne modifie pas l'objet mais en retourne un autre qui représente la somme. Dans ce cas, on l'aurait plutôt appelée « plus( ) ».
- ▷ La méthode différence(), elle, renvoie toujours une durée (positive).

1.9.3 La représentation de l'état (les attributs)

La question suivante est : « Comment représenter une durée en interne ? ». Plusieurs possibilités existent. Par exemple :

- ▷ via le nombre d'heures, de minutes et de secondes
- ▷ via le nombre total de secondes
- ▷ via une chaine, par exemple au format « HH :MM :SS » où HH pourrait éventuellement excéder 23.

Le premier choix semble le plus évident mais réfléchissons-y de plus près. D'une part, pourquoi se limiter aux heures. On pourrait introduire un champ 'jour' (après tout on a bien une méthode getJour()).

Quel critère doit vraiment nous permettre de décider ? Il faut une représentation qui soit suffisante (tout est représenté) et qui permette d'écrire des méthodes lisibles et si possible efficaces (c'est-à-dire où le calcul est rapide). Selon ces critères, la deuxième représentation est de loin la meilleure. Ce qui nous donne

```
classe Durée
 privé:
    totalSecondes : entier
 public:
    // idem
fin classe
```

1.9.4 L'implémentation

On est à présent prêt pour écrire le code des méthodes. Ce qui nous donne pour la classe dans son entièreté :

```
classe Durée
 privé:
    totalSecondes : entier
 public:
    constructeur Durée(secondes : entier)
    constructeur Durée(heure, minute, seconde : entiers)

    méthode getJour() → entier                    // nb de jours dans une durée
    méthode getHeure() → entier                   // entier entre 0 et 23 inclus
    méthode getMinute() → entier                  // entier entre 0 et 59 inclus
    méthode getSeconde() → entier                 // entier entre 0 et 59 inclus

    méthode getTotalHeures() → entier             // Le nombre total d'heures
    méthode getTotalMinutes() → entier            // Le nombre total de minutes
    méthode getTotalSecondes() → entier           // Le nombre total de secondes

    méthode ajouter(autreDurée : Durée)
    méthode différence(autreDurée : Durée) → Durée
    méthode égale(autreDurée : Durée) → booléen
    méthode plusPetit(autreDurée : Durée) → booléen
fin classe
```

```
constructeur Durée(secondes : entier)
    si secondes < 0 alors
        erreur "paramètre négatif"
    fin si
    totalSecondes ← secondes
fin constructeur

constructeur Durée(heure, minute, seconde : entiers)
    si heure < 0 OU minute < 0 OU seconde < 0 alors
        erreur "un des paramètres est négatif"
    fin si
    totalSecondes ← 3600*heure + 60*minute + seconde
fin constructeur
```

```
// Retourne le nombre de jours dans une représentation JJ/HH :MM :SS
méthode getJour() → entier
    retourner totalSecondes DIV (3600*24)
fin méthode

// Retourne le nombre d'heures dans une représentation JJ/HH :MM :SS
méthode getHeure() → entier
    // On doit enlever les jours éventuels
    retourner (totalSecondes DIV 3600) MOD 24
fin méthode

// Retourne le nombre de minutes dans une représentation JJ/HH :MM :SS
méthode getMinute() → entier
    // On doit enlever les heures éventuelles
    retourner (totalSecondes DIV 60) MOD 60
fin méthode

// Retourne le nombre de secondes dans une représentation JJ/HH :MM :SS
méthode getSeconde() → entier
    // On doit enlever les minutes éventuelles
    retourner totalSecondes MOD 60
fin méthode
```

```
// Retourne le nombre entier d'heures complètes
méthode getTotalHeures() → entier
    retourner totalSecondes DIV 3600
fin méthode

// Retourne le nombre entier de minutes complètes
méthode getTotalMinutes() → entier
    retourner totalSecondes DIV 60
fin méthode

// Retourne le nombre entier de secondes complètes
méthode getTotalSecondes() → entier
    retourner totalSecondes
fin méthode
```

```
méthode ajouter(autreDurée : Durée)
    totalSecondes ← totalSecondes + autreDurée.totalSecondes
fin méthode

méthode différence(autreDurée : Durée) → Durée
    retourner nouvelle Durée(valeurAbsolue(totalSecondes - autreDurée.totalSecondes))
fin méthode

méthode égale(autreDurée : Durée) → booléen
    retourner totalSecondes = autreDurée.totalSecondes
fin méthode

méthode plusPetit(autreDurée : Durée) → booléen
    retourner totalSecondes < autreDurée.totalSecondes
fin méthode
```

Et c'est tout ! Chaque méthode est très petite. C'est une constante en orienté objet : écrire de petites méthodes qui font chacune une et une seule chose bien précise.

1.10 Exercices

Vous trouverez en annexe, quelques classes que vous pouvez utiliser sans les définir : Date, Moment et Durée

1 Une personne

Créer une classe Personne, une personne étant constituée d'un nom, d'un prénom et d'une date de naissance. Cette classe utilisera la classe Date.

On doit pouvoir construire une personne :

- ▷ avec 3 arguments : le nom (chaine), le prénom (chaine) et la date de naissance de la personne (Date)
- ▷ avec 2 arguments de type chaine : le nom et le prénom de la personne ; la date de naissance est alors initialisée à « rien »

Écrire aussi tous les accesseurs et mutateurs que vous jugez pertinents. Dans un module principal, créer une personne :

a) avec comme arguments "Durant" et "Zébulon"

b) avec comme arguments "Durant", "Zébulon" et la date de naissance du 1^{er} février 1989

Pour réaliser les constructeurs recevant la date de naissance en paramètre, il faudra tester si cette date n'est pas antérieure à la date du jour.

2 Anniversaire des personnes

À l'aide de la classe Personne écrite plus haut, écrire un module qui lit des Personne (au clavier) et affiche les noms et le nombre de celles nées ce mois-ci. On suppose que la lecture de « rien » indique la fin des données.

Chapitre 2

Les tableaux à 2 dimensions

2.1 Définition

La **dimension** d'un tableau est le nombre d'indices qu'on utilise pour faire référence à un de ses éléments. Attention de ne pas confondre avec la taille !

En DEV_1, nous avons introduit les tableaux à une dimension. Un seul indice suffisait à localiser un de ses éléments. Pour le dire autrement, chaque case possédait **un** numéro. De nombreuses situations nécessitent cependant l'usage de tableaux à deux dimensions. Ils vous sont déjà familiers par leur présence dans beaucoup de situations courantes : calendrier, grille horaire, grille de mots croisés, sudoku, jeux se déroulant sur un quadrillage (damier, échiquier, scrabble ...). Dans ces situations, chaque case est désignée par **deux** numéros.

2.2 Notations

2.2.1 Déclarer

Pour **déclarer** un tableau à 2 dimensions, on écrira :

```
nomTableau : tableau [ligMin à ligMax, colMin à colMax] de TypeElément
```

où ligneMin, ligneMax, colMin et colMax sont des expressions entières quelconques.

Exemple :

```
tab : tableau [1 à 5, 1 à 10] d'entiers
```

déclare un tableau de 5 lignes par 10 colonnes dont chaque case contient un entier.

2.2.2 Utiliser

Pour **accéder** à une case du tableau on donnera les deux indices entre crochets.

Exemple :

```
afficher tab[2,4]
```

affiche le 4^e élément de la 2^e ligne du tableau nommé tab.

2.2.3 Visualiser

Notez que la vue sous forme de tableau avec des lignes et des colonnes est une vision humaine. Il n'y a pas de lignes ni de colonnes en mémoire. Pour être précis, on devrait juste parler de première dimension et de deuxième dimension mais la notion de ligne et de colonne est un abus de langage qui simplifie le discours.

On pourrait aussi visualiser un tableau à deux dimensions comme un tableau à une dimension dont chacun des éléments est lui-même un tableau à une dimension.

Exemple : Soit le tableau déclaré ainsi :

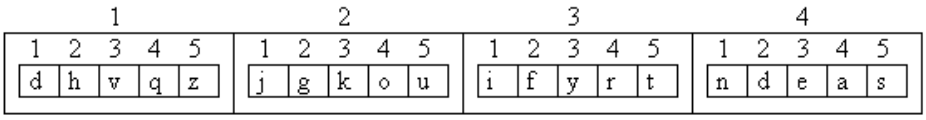
```
ntabLettres : tableau[1 à 4, 1 à 5] de caractères
```

On peut le visualiser à l'aide d'une grille à 4 lignes et 5 colonnes.

	1	2	3	4	5
1	d	h	v	q	z
2	j	g	k	o	u
3	i	f	y	r	t
4	n	d	e	a	s

Ainsi, la valeur de tabLettres[3,4] est le caractère 'r'.

La vision « tableau de tableau » (ou décomposition en niveaux) donnerait :



Dans cette représentation, le tableau tabLettres est d'abord décomposé à un premier niveau en quatre éléments auxquels on accède par le premier indice. Ensuite, chaque élément de premier niveau est décomposé en cinq éléments de deuxième niveau accessibles par le deuxième indice.

2.2.4 Exemples

Exemple 1 – Remplir les coins. Dans ce petit exemple, on a un tableau de chaines et on donne des valeurs aux coins.

"NO"				"NE"
"SO"				"SE"

```
// Déclare un tableau et donne des valeurs aux coins.
module remplirCoins()
    grille : tableau [1 à 3, 1 à 5] d'entiers
    grille[1,1] ← "NO"
    grille[1,5] ← "NE"
    grille[3,1] ← "SO"
    grille[3,5] ← "SE"
fin module
```

Exemple 2 – Gestion des stocks. Reprenons l'exemple du stock de 10 produits qui a servi d'introduction au chapitre sur les tableaux mais, cette fois, pour chaque jour de la semaine.

	article1	article2	article3	...	article8	article9	article10
lundi	cpt[1,1]	cpt[1,2]	cpt[1,3]	...	cpt[1,8]	cpt[1,9]	cpt[1,10]
mardi	cpt[2,1]	cpt[2,2]	cpt[2,3]	...	cpt[2,8]	cpt[2,9]	cpt[2,10]
mercredi	cpt[3,1]	cpt[3,2]	cpt[3,3]	...	cpt[3,8]	cpt[3,9]	cpt[3,10]
jeudi	cpt[4,1]	cpt[4,2]	cpt[4,3]	...	cpt[4,8]	cpt[4,9]	cpt[4,10]
vendredi	cpt[5,1]	cpt[5,2]	cpt[5,3]	...	cpt[5,8]	cpt[5,9]	cpt[5,10]
samedi	cpt[6,1]	cpt[6,2]	cpt[6,3]	...	cpt[6,8]	cpt[6,9]	cpt[6,10]
dimanche	cpt[7,1]	cpt[7,2]	cpt[7,3]	...	cpt[7,8]	cpt[7,9]	cpt[7,10]

```
// Calcule et affiche la quantité vendue de 10 produits
// pour chaque jour de la semaine (de 1 : lundi à 7 : dimanche).
module statistiquesVentesSemaine()

    cpt : tableau [1 à 7, 1 à 10] d'entiers
    produit, jour : entiers

    initialiser(cpt)

    // Pour chaque jour de la semaine
    pour jour de 1 à 7 faire
        traiterStock1Jour(cpt, jour)
        pour produit de 1 à 10 faire
            afficher "quantité vendue de produit ", produit, " ce jour ", jour, " : ", cpt[jour][i]
        fin pour
    fin pour
fin module
```

```
// Ce module initialise le tableau d'entiers à 0
module initialiser(entiers↓↑ : tableau [1 à 7, 1 à 10] d'entiers)
    i, j : entiers
    pour i de 1 à 7 faire
        pour j de 1 à 10 faire
            cpt[i,j] ← 0
        fin pour
    fin pour
fin module
```

```
// Ce module effectue le traitement du stock pour une journée.
module traiterStock1Jour(cpt ↓↑ : tableau [1 à 7, 1 à 10] d'entiers, jour : entier)
    numéroProduit, quantité : entiers
    afficher "Introduisez le numéro du produit :"
    lire numéroProduit

    tant que numéroProduit > 0 faire

        afficher "Introduisez la quantité vendue :"
        lire quantité

        cpt[jour,numéroProduit] ← cpt[jour,numéroProduit] + quantité

        afficher "Introduisez le numéro du produit :"
        lire numéroProduit

    fin tant que
fin module
```

Pour plus d'exemples, allez faire un tour à la section 2.4 page suivante.

2.3 La troisième dimension (et au-delà)

Certaines situations complexes nécessitent l'usage de tableaux à 3 voire plus de dimensions.

Pour déclarer un tableau statique à k dimensions, on écrira :

```
nomTableau : tableau [ bMin_1 à bMax_1, ..., bMin_k à bMax_k] de TypeElément
```

où chaque paire de bornes bMin_i et bMax_i limite l'indice correspondant à la $i^{ème}$ dimension du tableau.

2.4 Parcours d'un tableau à deux dimensions

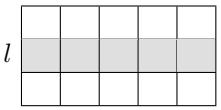
Comme nous l'avons fait pour les tableaux à une dimension, envisageons le parcours des tableaux à deux dimensions (n lignes et m colonnes). Nos algorithmes sont valables quel que soit le type des éléments. Utilisons T pour désigner un type quelconque.

```
tab : tableau [1 à n, 1 à m] de T
```

Commençons par des cas plus simples où on ne parcourt qu'une seule des dimensions puis attaquons le cas général.

2.4.1 Parcours d'une dimension

On peut vouloir ne parcourir qu'une seule ligne du tableau. Si on parcourt la ligne l, on visite les cases $(l, 1)$, $(l, 2)$, ..., (l, m). L'indice de ligne est constant et c'est l'indice de colonne qui varie.



Ce qui donne l'algorithme :

```
// Parcours de la ligne l d'un tableau à deux dimensions
pour c de 1 à m faire
    traiter tab[l,c]
fin pour
```

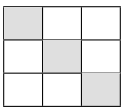
Retenons : pour parcourir une ligne, on utilise une boucle sur les colonnes.

Symétriquement, on pourrait considérer le parcours de la colonne c avec l'algorithme suivant.

```
// Parcours de la colonne c d'un tableau à deux dimensions
pour l de 1 à n faire
    traiter tab[l,c]
fin pour
```

Si le tableau est carré ($n = m$) on peut aussi envisager le parcours des deux diagonales.

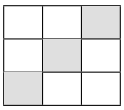
Pour la diagonale descendante, les éléments à visiter sont $(1, 1)$, $(2, 2)$, ..., (n, n).



Une seule boucle suffit comme le montre l'algorithme suivant.

```
// Parcours de la diagonale descendante d'un tableau carré
pour i de 1 à n faire
    traiter tab[i,i]
fin pour
```

Pour la diagonale montante, on peut envisager deux solutions, avec deux indices ou un seul en se basant sur le fait que $i + j = n + 1 \Rightarrow j = n + 1 - i$.



```
// Parcours de la diagonale montante d'un tableau carré - 2 indices
j ← n
pour i de 1 à n faire
    traiter tab[i,j]
    j ← j - 1
fin pour
```

```
// Parcours de la diagonale montante d'un tableau carré - 1 indice
pour i de 1 à n faire
    traiter tab[i, n + 1 - i]
fin pour
```

2.4.2 Parcours des deux dimensions

Parcours par lignes et par colonnes

Les deux parcours les plus courants sont les parcours ligne par ligne et colonne par colonne. Les tableaux suivants montrent dans quel ordre chaque case est visitée dans ces deux parcours.

Parcours ligne par ligne

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

Parcours colonne par colonne

1	4	7	10	13
2	5	8	11	14
3	6	9	12	15

Le plus simple est d'utiliser deux boucles imbriquées

```
// Parcours d'un tableau à 2 dimensions, ligne par ligne
pour lg de 1 à n faire
    pour col de 1 à m faire
        traiter tab[lg,col]
    fin pour
fin pour
```

```
// Parcours d'un tableau à 2 dimensions, colonne par colonne
pour col de 1 à m faire
    pour lg de 1 à n faire
        traiter tab[lg,col]
    fin pour
fin pour
```

Mais on peut obtenir le même résultat avec une seule boucle si l'indice sert juste à compter le nombre de passages et que les indices de lignes et de colonnes sont gérés manuellement.

L'algorithme suivant montre ce que ça donne pour un parcours ligne par ligne. La solution pour un parcours colonne par colonne est similaire et laissée en exercice.

```
// Parcours d'un tableau à 2 dimensions via une seule boucle
lg ← 1
col ← 1
pour i de 1 à n*m faire
    traiter tab[lg,col]
    col ← col + 1                                          // Passer à la case suivante
    si col > m alors                  // On déborde sur la droite, passer à la ligne suivante
        col ← 1
        lg ← lg + 1
    fin si
fin pour
```

L'avantage de cette solution apparaitra quand on verra des situations plus difficiles.

Interrompre le parcours

Comme avec les tableaux à une dimension, envisageons l'arrêt prématuré lors de la rencontre d'une certaine condition. Et, comme avec les tableaux à une dimension, transformons d'abord nos **pour** en **tant que**.

Par exemple, montrons les deux parcours ligne par ligne, avec une et deux boucle(s).

```
// Parcours d'un tableau à 2 dimensions, ligne par ligne, via un tant que
lg ← 1
tant que lg ≤ n faire
    col ← 1
    tant que col ≤ m faire
        traiter tab[lg, col]
        col ← col + 1
    fin tant que
    lg ← lg + 1
fin tant que
```

```
// Parcours d'un tableau à 2 dimensions via une seule boucle et un tant que
lg ← 1
col ← 1
i ← 1
tant que i ≤ n*m faire                                              // ou "lg ≤ n"
    traiter tab[lg,col]
    col ← col + 1                                          // Passer à la case suivante
    si col > m alors                  // On déborde sur la droite, passer à la ligne suivante
        col ← 1
        lg ← lg + 1
    fin si
    i ← i + 1
fin tant que
```

On peut à présent introduire le test comme on l'a fait dans les algorithmes de parcours des tableaux à une dimension.

Illustrons-le au travers de deux exemples. Le premier introduit un test en utilisant un booléen alors que le second introduit un test sans utiliser de booléen.

```
// Parcours avec test d'arrêt - deux boucles et un booléen
trouvé ← faux
lg ← 1
tant que lg ≤ n ET NON trouvé faire
    col ← 1
    tant que col ≤ m ET NON trouvé faire
        si tab[lg, col] impose l'arrêt du parcours alors
            trouvé ← vrai
        sinon                                   // Ne pas modifier les indices si arrêt demandé
            col ← col + 1
        fin si
    fin tant que
    si NON trouvé alors                         // Ne pas modifier les indices si arrêt demandé
        lg ← lg + 1
    fin si
fin tant que
```

```
// Parcours avec test d'arrêt - une boucle et pas de booléen
lg ← 1
col ← 1
i ← 1
tant que i ≤ n*m ET tab[lg, col] n'impose pas l'arrêt faire
    col ← col + 1                                          // Passer à la case suivante
    si col > m alors                  // On déborde sur la droite, passer à la ligne suivante
        col ← 1
        lg ← lg + 1
    fin si
    i ← i + 1
fin tant que
// Arrêt prématuré si i ≤ n*m.
```

Parcours plus compliqué - le serpent

Envisageons un parcours plus difficile illustré par le tableau suivant.

1	2	3	4	5
10	9	8	7	6
11	12	13	14	15

Le plus simple est d'adapter l'algorithme de parcours avec une seule boucle en introduisant un sens de déplacement, ce qui donne l'algorithme :

```
// Parcours du serpent dans un tableau à deux dimensions
lg ← 1
col ← 1
depl ← 1                                        // 1 pour avancer, -1 pour reculer
pour i de 1 à n*m faire
    traiter tab[lg, col]
    si 1 ≤ col + depl ET col + depl ≤ m alors
        col ← col + depl                          // On se déplace dans la ligne
    sinon
        lg ← lg + 1                               // On passe à la ligne suivante
        depl ← -depl                                  // et on change de sens
    fin si
fin pour
```

2.5 Exercices

1 Affichage

Écrire un module qui affiche tous les éléments d'un tableau à n lignes et m colonnes
a) ligne par ligne b) colonne par colonne

2 Les nuls

Écrire un module qui reçoit un tableau (n x m) d'entiers et qui affiche la proportion d'éléments nuls dans ce tableau.

3 Tous positifs

Écrire un module qui reçoit un tableau (n x m) d'entiers et qui vérifie si tous les nombres qu'il contient sont strictement positifs. Bien sûr, on veillera à éviter tout travail inutile ; la rencontre d'un nombre négatif doit arrêter le module.

4 Le tableau de cotes

Soit un tableau à n lignes et m colonnes d'entiers où une ligne représente les notes sur 20 d'un étudiant et les colonnes toutes les notes d'un cours.

Écrire un algorithme recevant ce tableau en paramètre et affichant le pourcentage d'étudiants ayant obtenu une moyenne supérieure à 50%.

5 Le carré magique

Un carré magique est un tableau d'entiers carré (c'est-à-dire possédant autant de lignes que de colonnes) ayant la propriété suivante : si on additionne les éléments d'une quelconque de ses lignes, de ses colonnes ou de ses deux diagonales, on obtient à chaque fois le même résultat.

Écrire un module recevant en paramètres le tableau [1 à n, 1 à n] d'entiers représentant le carré et renvoyant une valeur booléenne indiquant si c'est un carré magique ou pas.

6 Le triangle de Pascal

Le triangle de Pascal est construit de la façon suivante :

- ▷ la ligne initiale contient un seul élément de valeur 1 ;
- ▷ chaque ligne possède un élément de plus que la précédente ;
- ▷ chaque ligne commence et se termine par 1 ;
- ▷ pour calculer un nombre d'une autre case du tableau, on additionne le nombre situé dans la case située juste au-dessus avec celui dans la case à la gauche de la précédente.

Écrire un module qui reçoit en paramètre un entier n, et qui renvoie un tableau contenant les $n + 1$ premières lignes du triangle de Pascal (indicées de 0 à n).

N.B. : le « triangle » sera bien entendu renvoyé dans un tableau carré. Quid des cases non occupées ?

Par exemple, pour n qui vaut 5, on aura le tableau suivant :

1					
1	1				
1	2	1			
1	3	3	1		
1	4	6	4	1	
1	5	10	10	5	1

7 Lignes et colonnes

Écrire un module qui reçoit un tableau d'entiers à 2 dimensions en paramètre et qui retourne un booléen indiquant si ce tableau possède 2 lignes ou 2 colonnes identiques.

Dans l'affirmative, ce module renverra également en paramètres les informations suivantes :

- ▷ les indices des lignes ou colonnes identiques
- ▷ un caractère valant 'L' ou 'C' selon qu'il s'agit de lignes ou de colonnes

Dans la négative, les valeurs de ces paramètres seront indéterminées ou quelconques, elles ne seront de toute façon pas utilisées par le module appelant.

8 Le contour du tableau

On donne un tableau d'entiers tabEnt à n lignes et m colonnes. Écrire un module retournant la somme de tous les éléments *impairs* situés sur le bord du tableau.

Exemple : pour le tableau suivant, le module doit renvoyer 32

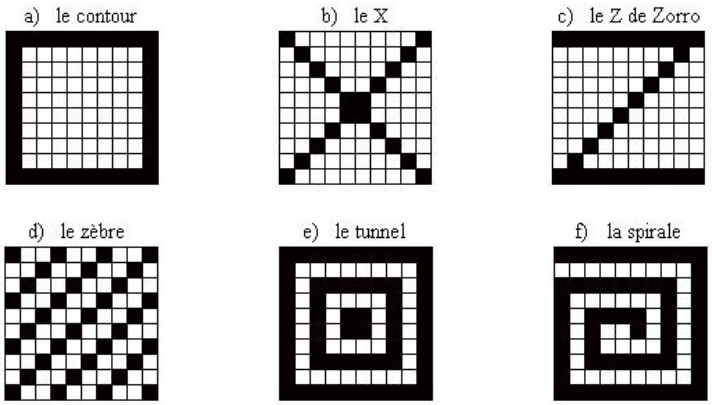
3	4	6	11
2	21	7	9
1	5	12	3

Et pour le suivant, le module doit renvoyer 6

4	1	2	8	5

9 À vos pinceaux !

On possède un tableau à n lignes et n colonnes dont les éléments de type Couleur valent NOIR ou BLANC. On suppose que le tableau est initialisé à BLANC au départ. Écrire un module qui *noircit* les cases de ce tableau comme le suggèrent les dessins suivants (les exemples sont donnés pour un tableau 10 x 10 mais les algorithmes doivent fonctionner quelle que soit la taille du tableau).



10 Exercices sur la complexité

Quelle est la complexité

a) d'un algorithme de parcours d'un tableau n x n ?

b) des algorithmes que vous avez écrits pour les exercices : "Les nuls", "Tous positifs", "Le carré magique" et "Le contour d'un tableau" ?

c) des algorithmes que vous avez écrits pour résoudre les exercices du pinceau ?

Chapitre 3

La liste

Imaginons qu'on désire manipuler par programme une liste de contacts ou encore une liste de rendez-vous. Cette liste va varier ; sa taille n'est donc pas fixée. Utiliser un tableau à cet effet n'est pas l'idéal. En effet, la taille d'un tableau ne peut plus changer une fois le tableau créé. Il faudrait le sur-dimensionner, ce qui n'est pas économe.

Il serait intéressant de disposer d'une structure qui offre toutes les facilités d'un tableau tout en pouvant « grandir » si nécessaire. Construisons une telle structure de données et appelons-la « Liste » pour rester en phase avec son appellation commune en Java.

3.1 La classe Liste

Nous verrons plus loin comment la réaliser en pratique mais nous pouvons déjà définir le comportement qu'on en attend (les méthodes qu'elle doit fournir)

```
classe Liste <T>                                                    // T est un type quelconque
 privé:
    // sera complété plus tard
 public:
    constructeur Liste <T>()                                        // construit une liste vide
    méthode get(pos : entier) → T                                   // donne un élément en position pos
    méthode set(pos : entier, valeur : T)                           // modifie un élément en position pos
    méthode taille() → entier                                       // donne le nombre d'éléments
    méthode ajouter(valeur : T)                                     // ajoute un élément en fin de liste
    méthode insérer(pos : entier, valeur : T)                       // insère un élément en position pos
    méthode supprimer()                                             // supprime le dernier élément
    méthode supprimerPos(pos : entier)                              // supprime l'élément en position pos
    méthode supprimer(valeur : T) → booléen                         // supprime l'élément de valeur donnée
    méthode vider()                                                 // vide la liste
    méthode estVide() → booléen                                     // la liste est-elle vide ?
    méthode existe(valeur ↓ : T, pos ↑ : entier) → booléen          // recherche un élément
fin classe
```

Quelques précisions s'imposent :

- ▷ Comme les tableaux, les listes peuvent contenir des éléments de n'importe quel type tout en restant uniforme au sein d'une même liste (on pourra manipuler une liste d'entiers, une liste de contacts... mais pas mélanger). Il serait rédhibitoire de devoir définir une Liste pour chaque type d'éléments. On utilise dès lors la possibilité en OO

d'écrire un code **générique**[1]. Le « T » dans la définition de la classe indique le type des éléments qui sera spécifié lors de l'utilisation de la classe. On écrira par exemple « Liste d'entiers » pour utiliser une liste d'entiers.

- ▷ Les méthodes « get » et « set » permettent de connaitre ou modifier un élément de la liste. On considère, au cours de logique, que le premier élément de la liste est en position 1.
- ▷ « ajouter » ajoute un élément en fin de liste (elle grandit donc d'une unité)
- ▷ « insérer » insère un élément à une position donnée (entre 1 et taille+1). L'élément qui s'y trouvait est décalé d'une position ainsi que tous les éléments suivants.
- ▷ La méthode « supprimerPos » supprime un élément d'une position donnée en décalant les éléments suivants. On pourrait imaginer une technique plus rapide consistant à placer le dernier élément à la place de l'élément supprimé mais ce faisant on changerait l'ordre relatif des éléments ce qui va à l'encontre de l'idée intuitive qu'on se fait d'une liste. Cette amélioration pourrait plutôt s'envisager dans une structure de type **ensemble** pour lequel il n'y a pas d'ordre relatif entre les éléments.
- ▷ La version de « supprimer » avec une valeur en paramètre enlève un élément de valeur donnée. Elle retourne un booléen indiquant si la suppression a pu se faire ou pas (ce qui sera le cas si la valeur n'est pas présente dans la liste). Si la valeur existe en plusieurs exemplaires, on prendra la convention arbitraire que la méthode n'en supprime que la première occurrence.
- ▷ La méthode « existe » permet de savoir si un élément donné existe dans la liste.
  - ▷ si c'est le cas, elle précise aussi sa position dans le paramètre sortant pos
  - ▷ si l'élément n'existe pas, ce paramètre est indéterminé
  - ▷ si l'élément est présent en plusieurs exemplaires, la méthode donne la position de la première occurrence.
- ▷ En pratique, il serait intéressant de chercher un élément à partir d'une partie de l'information qu'elle contient mais c'est difficile à exprimer de façon générique c'est-à-dire lorsque le type n'est pas connu à priori.

Exemple : recherche du minimum

Dans le chapitre sur les tableaux, vous avez fait un exercice consistant à afficher tous les indices où se trouve le minimum d'un tableau. Reprenons-le et modifions-le afin qu'il retourne la liste des indices où se trouvent les différentes occurrences du minimum. On pourrait l'écrire ainsi :

```
module indicesMinimum(tab : tableau [1 à n] d'entiers) → Liste <entier>
    i, min : entier
    indicesMin : Liste <entier>
    min ← tab[1]
    indicesMin ← nouvelle Liste <entier>()
    indicesMin.ajouter( 1 )
    pour i de 2 à n faire
        selon que
         tab[i] = min:
            indicesMin.ajouter( i )
         tab[i] < min:
            indicesMin.vider()
            indicesMin.ajouter( i )
            min ← tab[i]
         tab[i] > min:
            // rien à faire dans ce cas
        fin selon que
    fin pour
    retourner indicesMin
fin module
```

1. on parle aussi de « template ».

3.2 Comment implémenter l'état

Cette liste est bien utile mais comment la réaliser en pratique ? Comment représenter une liste variable d'éléments ? Pour l'instant, la seule structure qui peut accueillir plusieurs éléments de même type est le tableau. Nous allons donc prendre comme attribut principal de la liste, un tableau que nous appellerons éléments. Comment, dès lors, contourner le problème de la limitation de la taille de ce tableau ?

Repartons donc de la notion de tableau et tentons de comprendre sa limitation. Lors de sa création, un tableau se voit attribuer un espace bien précis et contigu en mémoire. Il se peut très bien que l'espace « juste après » soit occupé par une autre variable ce qui l'empêche de grandir. La parade est claire : si un tableau s'avère trop petit lors de son utilisation, il suffit d'en créer un autre plus grand ailleurs en mémoire et d'y recopier tous les éléments du premier. Évidemment, cette opération est coûteuse en temps et on cherchera à l'effectuer le moins souvent possible.

Quelle taille donner au nouveau tableau ? L'idée qui vient immédiatement est d'augmenter la taille d'une unité afin d'accueillir le nouvel élément mais cette approche implique de fréquents agrandissements. Il est plus efficace d'augmenter la taille proportionnellement, par exemple en la multipliant par un facteur 2.

1	5	7	⇒	1	5	7	.	.	.

Taille logique et taille physique. À tout moment, le tableau aura une et une seule taille même si celle-ci pourra changer au cours du temps. Puisqu'on multipliera la taille du tableau par 2 pour des raisons d'efficacité, il y aura toutefois une différence entre la **taille physique** d'un tableau et sa **taille logique**. La taille physique est le nombre de cases réservées pour le tableau alors que la taille logique est le nombre de cases effectivement occupées. Dans ce qui suit, on s'arrangera pour que les cases occupées soient groupées à gauche du tableau (il n'y a pas de trou). Pour l'utilisateur, seule la taille logique a un sens (on lui cache les détails d'implémentation).

Exemple : pour le tableau suivant, la taille logique est de 6 (c'est cette taille qui a du sens pour l'utilisateur de la liste) et la taille physique est de 8.

2	5	4	8	3	12	.	.

Quand il faut insérer un élément (en position valide) ou en ajouter un en fin de liste, deux cas se présentent :

- ▷ si la taille logique est plus petite que la taille physique, il suffit d'ajouter l'élément dans le tableau et d'adapter la taille logique.
- ▷ si la taille logique est égale à la taille physique, il faut procéder à un agrandissement du tableau.

Les tableaux dynamiques. En DEV_1, nous n'avons vu que des tableaux qu'on appellera *statiques*, qui sont créés lors de leur déclaration. Ici, nous avons besoin de tableaux qu'on appellera *dynamiques*, créés dans le code (comme le sont les tableaux en Java).

Introduisons une notation. Un tableau dynamique sera déclaré puis créé ainsi :

```
tab : tableau de T                                  // où T est un type quelconque
// Autres déclarations et instructions
tab ← nouveau tableau [1 à n] de T                  // n doit avoir une valeur
```

Implémentation. Présentons les attributs nécessaires et l'algorithme d'agrandissement du tableau.

```
classe Liste <T>
 privé:
    éléments : tableau de T
    tailleLogique : entier
    taillePhysique : entier
 privé:
    méthode agrandir()
        i : entier
        nouveauTab : tableau de T
        taillePhysique ← taillePhysique * 2
        nouveauTab ← nouveau tableau [ 1 à taillePhysique ] de T
        pour i de 1 à tailleLogique faire
            nouveauTab[ i ] ← éléments[ i ]
        fin pour
        éléments ← nouveauTab
    fin méthode
fin classe
```

Réduction du tableau. Tout comme on agrandit le tableau si nécessaire, on pourrait le réduire lorsque des suppressions d'éléments le rendent sous-utilisé (par exemple lorsque la taille logique devient inférieure au tiers de la taille physique). Nous n'aborderons pas cette problématique cette année.

3.3 Implémentation du comportement

Nous avons à présent toutes les cartes en main pour écrire les méthodes publiques de la classe.

```
constructeur Liste <T>()
    tailleLogique ← 0                                    // la liste est vide au départ
    taillePhysique ← 32                                  // une bonne valeur pour commencer
    éléments ← nouveau tableau [ 1 à taillePhysique ] de T
fin constructeur
```

```
méthode get(pos : entier) → T
    si  pos < 1 OU pos > tailleLogique alors
        erreur "position invalide"
    fin si
    retourner éléments[ pos ]
fin méthode
```

```
méthode set(pos : entier, valeur : T)
    si  pos < 1 OU pos > tailleLogique alors
        erreur "position invalide"
    fin si
    éléments[ pos ] ← valeur
fin méthode
```

```
méthode taille() → entier
    retourner tailleLogique                              // et pas la taille physique !
fin méthode
```

```
méthode ajouter(valeur : T)
    si tailleLogique = taillePhysique alors
        agrandir()                                       // méthode privée détaillée supra
    fin si
    tailleLogique ← tailleLogique + 1
    éléments[ tailleLogique ] ← valeur
fin méthode
```

```
méthode insérer(pos : entier, valeur : T)
    si  pos < 1 OU pos > tailleLogique+1 alors
        erreur "position invalide"
    fin si
    si tailleLogique = taillePhysique alors
        agrandir()
    fin si
    décalerDroite( pos )                                    // voir ci-dessous
    tailleLogique ← tailleLogique + 1
    éléments[ pos ] ← valeur
fin méthode
```

```
méthode supprimer()
    // supprime le dernier élément
    si tailleLogique = 0 alors
        erreur "liste vide"
    fin si
    tailleLogique ← tailleLogique - 1
fin méthode
```

```
méthode supprimerPos(pos : entier)
    si  pos < 1 OU pos > tailleLogique alors
        erreur "position invalide"
    fin si
    décalerGauche( pos + 1 )                                // voir méthode ci-dessous
    tailleLogique← tailleLogique - 1
fin méthode
```

```
méthode supprimer(valeur : T) → booléen
    estPrésent : booléen
    pos : entier
    estPrésent ← existe(valeur, pos)
    si estPrésent alors
        supprimer( pos )
    fin si
    retourner estPrésent
fin méthode
```

```
méthode vider()
    tailleLogique ← 0              // Les éléments ne sont pas effacés mais sont ignorés
fin méthode
```

```
méthode estVide() → booléen
    retourner tailleLogique = 0
fin méthode
```

```
méthode existe(valeur↓ : T, pos↑ : entier) → booléen
    pos ← 1
    // Rq : le ET ci-dessous est une évaluation court-circuitée (cf. le cours d'Algo en DEV1)
    tant que  pos ≤ tailleLogique ET éléments[ pos ] ≠ valeur faire
        pos ← pos + 1
    fin tant que
    retourner pos ≤ tailleLogique
fin méthode
```

```
// Ces méthodes-ci sont privées

méthode décalerDroite(début : entier)
    // Décale tous les éléments d'une position vers la droite à partir de début
    i : entier
    pour i de tailleLogique à début par –1 faire
        éléments[ i + 1 ] ← éléments[ i ]
    fin pour
fin méthode

méthode décalerGauche(début : entier)
    // Décale toutes les éléments d'une position vers la gauche à partir de début ;
    // ce paramètre vaut toujours au moins 2.
    i : entier
    pour i de début à tailleLogique faire
        éléments[ i - 1 ] ← éléments[ i ]
    fin pour
fin méthode
```

La recherche se fait sur un élément complet.

Prenons comme exemple une liste de contacts. Lors d'une recherche, on doit fournir **tout** le contact à rechercher. Il s'agit juste de savoir s'il est présent et où. Une autre méthode intéressante serait de retrouver un contact à partir d'une partie de l'information, par exemple son nom. Cette méthode est fort proche de notre méthode de recherche mais il serait très difficile de l'écrire génériquement. On vous demandera d'écrire explicitement une telle méthode de recherche en cas de besoin.

3.4 Et sans tableau dynamique ?

Certains langages (c'est le cas de Cobol) ne permettent pas de créer dynamiquement un nouveau tableau. Il vous faudra travailler avec un tableau classique en le créant suffisamment grand.

Les algorithmes d'ajout/suppression/recherche vus pour la liste peuvent être appliqués tels quels à un tableau statique à une modification près : lors d'un ajout dans un tableau plein, on ne peut pas l'agrandir ; il faut générer une erreur.

3.5 Exercices

1 Liste des premiers entiers

Écrire un module qui reçoit un entier n en paramètre et retourne la liste contenant les entiers de 1 à n dans l'ordre décroissant. On peut supposer que n est positif.

2 Somme d'une liste

Écrire un module qui calcule la somme des éléments d'une liste d'entiers.

3 Les extrêmes

Écrire un module qui supprime le minimum et le maximum des éléments d'une liste d'entiers. On peut supposer que le maximum et le minimum sont uniques.

4 Anniversaires

Écrire un module qui reçoit une liste de Personne (nom + prénom + date de naissance ; cf. exercice dans le chapitre OO) et retourne la liste de ceux qui sont nés durant un mois passé en paramètre (donné sous la forme d'un entier entre 1 et 12).

5 Concaténation de deux listes

Écrire un module qui reçoit 2 listes et ajoute à la suite de la première les éléments de la seconde ; la seconde liste n'est pas modifiée par cette opération.

6 Fusion de deux listes

Soit deux listes **ordonnées** d'entiers (redondances possibles). Écrire un module qui les fusionne. Le résultat est une liste encore ordonnée contenant tous les entiers des deux listes de départ (qu'on laisse inchangées).

Exemple : Si les 2 listes sont (1, 3, 7, 7) et (3, 9), le résultat est (1, 3, 3, 7, 7, 9).

7 Le nettoyage

Écrire un module qui reçoit une liste de chaines en paramètre et supprime de cette liste tous les éléments de valeur donnée en paramètre. L'algorithme retournera le nombre de suppressions effectuées.

8 Éliminer les doublons d'une liste

Soit une liste **ordonnée** d'entiers avec de possibles redondances. Écrire un module qui enlève les redondances de la liste.

Exemple : Si la liste est (1, 3, 3, 7, 8, 8, 8), le résultat est (1, 3, 7, 8).

a) Faites l'exercice en créant une **nouvelle liste** (la liste de départ reste inchangée)
b) Refaites l'exercice en **modifiant** la liste de départ (pas de nouvelle liste)

9 Rendez-vous

Soit la structure « RendezVous » composée d'une date et d'un motif de rencontre. Écrire un module qui reçoit une liste de rendez-vous et la met à jour en supprimant tous ceux qui sont désormais passés.

Chapitre 4

Représentation des données

Nous voici arrivés au terme de ce cours d'algorithmique. Ce chapitre apporte une synthèse des différentes notions vues tout au long de vos cours d'algorithmiques de 1re année et propose quelques pistes de réflexion quant au choix d'une bonne représentation des données qui se pose lors de la résolution de problèmes de programmation avancés.

Pour la plupart de ces exercices, la difficulté tient en partie dans le bon choix d'une représentation des données et de la démarche algorithmique la plus adéquate à mettre en œuvre pour agir sur ces données en vue d'obtenir le résultat escompté. Noter que l'efficacité d'un algorithme est lié étroitement au choix de la représentation.

4.1 Se poser les bonnes questions

Revenons à la case départ : nous avons commencé ce cours en situant les notions de **problème** et de **résolution**. Nous avons vu qu'un problème bien spécifié s'inscrit dans le schéma :

étant donné [les données] on demande [l'objectif]

Une fois le problème correctement posé, on peut partir à la recherche d'une **méthode de résolution**, c'est-à-dire d'un algorithme en ce qui concerne les problèmes à résoudre par les moyens informatiques.

Tout au long de l'année, nous avons vu divers modèles et techniques algorithmiques adaptées à des structures particulières (les nombres, les chaines, les tableaux, les variables structurées, les objets, les listes...). La plupart des exercices portaient directement sur ces structures (par ex. calculer la somme des nombres d'un tableau, extraire une sous-liste à partir d'une liste donnée). Ces exercices d'entrainement et de formation quelque peu théoriques constituent en fait des démarches algorithmiques de base qui trouvent toutes une place dans des problèmes plus complexes.

Mais la plupart des problèmes issus des situations de la vie courante auxquels se confronte le programmeur s'expriment généralement de manière plus floue : par ex. dresser la comptabilité des dépenses mensuelle d'une firme, faire un tableau récapitulatif du résultat des élections par cantons électoraux, faire une version informatique d'un jeu télévisé... Les exemples sont infinis !

C'est dans le cadre de ce genre de problème plus complexe que se pose le problème de la **représentation de données**. Une fois le problème bien spécifié (par les données et l'objectif) apparaissent naturellement les questions suivantes : quelles données du problème sont

réellement utiles à sa résolution ? (Il est fréquent que l'énoncé d'un problème contienne des données superflues ou inutiles). Y a-t-il des données plus importantes que d'autres ? (données principales ou secondaire). Les données doivent-elles être consultées plusieurs fois ? Quelles données faut-il conserver en mémoire ? Sous quelle forme ? Faut-il utiliser un tableau ? Une liste ? Faut-il créer une nouvelle classe ? Les données doivent-elles être classées suivant un critère précis ? Ou la présentation brute des données suffit-elle pour solutionner le problème posé ?

Les réponses ne sont pas directes, et les différents outils qui sont à notre disposition peuvent être ou ne pas être utilisés. Il n'y a pas de règles précises pour répondre à ces questions, c'est le flair et le savoir-faire développés patiemment par le programmeur au fil de ses expériences et de son apprentissage qui le guideront vers la solution la plus efficace. Parfois plusieurs solutions peuvent fonctionner sans pouvoir départager la meilleure d'entre-elles.

Ce type de questionnement est peut-être l'aspect le plus délicat et le plus difficile de l'activité de programmation, car d'une réponse appropriée dépendra toute l'efficacité du code développé. Un mauvais choix de représentation des données peut mener à un code lourd et maladroit. En vous accompagnant dans la résolution des exercices qui suivent, nous vous donnerons quelques indices et pistes de réflexion, qui seront consolidées par l'expérience acquise lors des laboratoires de langages informatiques ainsi que par les techniques de modélisation vues au cours d'analyse.

4.2 Les structures de données

Rappelons brièvement les différentes structures étudiées dans ce cours :

- ▷ les **données « simples »** (variables isolées : entiers, réels, chaines, caractères, booléens)
- ▷ les **variables structurées**, qui regroupent en une seule entité une collection de variables simples
- ▷ le **tableau**, qui contient un nombre déterminé de variables de même types, accessibles via un indice ou plusieurs pour les tableaux multidimensionnels
- ▷ les **objets**, qui combinent en un tout une série d'attributs et des méthodes agissant sur ces attributs
- ▷ la **liste**, qui peut contenir un nombre indéfini d'éléments de même type

D'autres structures particulières s'ajouteront dans le cours de 2^{e} année : les listes chainées, les piles, les files, les arbres et les graphes.

Chacune de ces structures possède ses spécificités propres quant à la façon d'accéder aux valeurs, de les parcourir, de les modifier, d'ajouter ou de supprimer des éléments à la collection.

4.3 Exercices

1 Le lièvre et la tortue

lu sur le net : http://mathemathieu.free.fr/2b/doc/pb_algo/problemes_et_algorithmique.pdf

Le lièvre est plus rapide que la tortue. Pour donner plus de chance à la tortue de gagner une course de 5 km, on adopte la règle de jeu suivante :

On lance un dé. Si le 6 sort, le lièvre est autorisé à démarrer et gagne la course en quelques secondes ; sinon on laisse la tortue avancer d'un kilomètre.

Recommencer le procédé jusqu'à la victoire du lièvre ou de la tortue

Écrire un module simulant cette course.

2 Un jeu de poursuite

Deux joueurs A et B se poursuivent sur un circuit de 50 cases. Chaque case contient une valeur vrai ou faux indiquant si le joueur pourra rejouer. Au départ, A se trouve sur la case 1 et B est placé sur la case 26. C'est A qui commence. Chaque joueur joue à son tour en lançant un dé dont la valeur donne le nombre de cases duquel il doit avancer sur le jeu. Si la case sur laquelle tombe le joueur contient la valeur vrai il avance encore une fois du même nombre de cases (et de même s'il tombe encore sur vrai). Lorsqu'un joueur arrive sur la case 50 et qu'il doit encore avancer, il continue son parcours à partir de la case 1. Le jeu se termine lorsqu'un joueur rattrape ou dépasse l'autre.

Écrire un algorithme de simulation de ce jeu qui se terminera par l'affichage du vainqueur ainsi que le nombre de tours complets parcourus par ce vainqueur. Le lancement du dé sera simulé par l'appel du module sans argument lancerDé( ) qui retourne une valeur aléatoire entre 1 et 6.

Aide : Définir la classe JeuPoursuite

Elle permet de représenter

- ▷ le circuit des 50 cases
- ▷ la position des 2 joueurs
- ▷ le nombre de tours effectués par chacun des joueurs
- ▷ qui est le joueur courant

Plusieurs possibilités existent ; faites votre choix !

- ▷ Le constructeur reçoit la configuration du circuit (pour savoir si les cases contiennent vrai ou faux)
- ▷ La méthode initialiser() initialise le jeu (placement des joueurs, ...).
- ▷ La méthode jouer() lance le jeu jusqu'à son terme et donne le vainqueur et le nombre de tours effectués.
- ▷ Vous êtes également fortement invités à définir d'autres méthodes en privé pour modulariser au mieux votre code. Par exemple, on pourrait définir
- ▷ la méthode « jouerCoup » qui joue pour un joueur et indique s'il a rattrapé l'autre joueur (sans répétition si on arrive sur une case vrai)
- ▷ la même méthode « jouerTour » effectue la même tâche mais avec répétition si on arrive sur une case vrai. On fera évidemment appel à la méthode ci-dessus.
- ▷ la méthode « joueurSuivant » qui permet de passer au joueur suivant.

Avec ces 3 méthodes, la méthode publique « jouer » devient triviale.

3 La course à la case 64

Une piste de 65 cases (numérotées de 0 à 64) doit être parcourue le plus rapidement possible par quatre joueurs. Un tableau joueurs de quatre chaines contient les noms et prénoms des joueurs. Au départ, tous les joueurs se trouvent sur la case de départ (la case numéro 0). Les joueurs jouent à tour de rôle, dans l'ordre où ils apparaissent dans le tableau Joueur. Le joueur qui gagne est celui qui arrive le premier sur la case 64.

La longueur des déplacements est déterminée à l'aide d'un dé à six faces, un joueur pouvant avancer d'autant de cases que le point du dé. Si la case sur laquelle s'arrête un joueur est déjà occupée par un autre, ce dernier est renvoyé à la case départ. D'autre part, chaque fois qu'un joueur obtient la face 6, il a le droit de rejouer avant le tour du joueur suivant.

Écrire un algorithme de simulation de ce jeu qui fournit le nom du vainqueur. Comme dans l'exercice précédent, le lancement du dé est simulé par le module lancerDé( ) qui retourne une valeur aléatoire entre 1 et 6.

Imaginer la classe Course64 qui va permettre de résoudre ce problème. Comment faire pour pouvoir accepter un nombre quelconque de joueurs ?

4 Mots croisés

Un tableau grille à 10 lignes et 10 colonnes contient les données relatives à un jeu de mots croisés simulé sur ordinateur. Chaque élément de ce tableau est une structure Case, contenant les deux champs :

- ▷ noir : variable booléenne affectée à vrai si la case correspondante de la grille est une case noire ;
- ▷ lettre : contient soit le caractère inscrit par le joueur dans une case, soit le caractère « espace » (‘ ‘) si la case est encore blanche ; lorsque noir est vrai, le contenu de lettre est indéterminé et ne peut donc être utilisé.

Écrire une classe Grille offrant les méthodes suivantes :

- ▷ placer une lettre à un endroit de la grille (une case non noire bien sûr)
- ▷ donner le nombre de cases noires sur la grille
- ▷ donner le nombre total de mots de la grille (donc y compris ceux que le joueur n'a pas encore complétés). Attention, les mots d'une seule lettre ne sont pas pris en compte.
- ▷ donner le nombre de mots déjà complétés par le joueur

Exemple : dans la grille ci-dessous, le nombre de cases noires est 14, le nombre total de mots de la grille est 37 (19 horizontaux et 18 verticaux) et le nombre de mots déjà complété par le joueur est 6.

		A				■			
		L		■					
L	O	G	I	Q	U	E	■		
		O	■				■		
■		R		■		■			
E	S	I	■	O		H		■	
	■	T	A	B	L	E	A	U	■
		H	■	J		B			
		M		E		■			
		E		T					

5 Mastermind

Dans le jeu du Mastermind, un joueur A doit trouver une combinaison de k pions de couleurs, choisie et tenue secrète par un autre joueur B. Cette combinaison peut contenir éventuellement des pions de même couleur. À chaque proposition du joueur A, le joueur B indique

le nombre de pions de la proposition qui sont corrects et bien placés et le nombre de pions corrects mais mal placés.

Supposons une énumération Couleur avec toutes les couleurs possibles de pion.

a) Écrire une classe « Combinaison » pour représenter une combinaison de k pions. Elle possède une méthode pour générer une combinaison aléatoire (que vous ne devez pas écrire) et une méthode pour comparer une combinaison à la combinaison secrète (que vous devez écrire)
b) Écrire ensuite une classe « MasterMind » qui représente le jeu et permet d'y jouer. La taille de la combinaison et le nombre d'essais permis seront des paramètres du constructeur.

6 Le Jeu du Millionnaire

Un questionnaire de quinze questions à choix multiples de difficulté croissante est soumis à un candidat. Quatre possibilités de réponses (dont une seule est correcte) sont proposées à chaque fois. Au plus le candidat avance dans les bonnes réponses, au plus son gain est grand. S'il répond correctement aux quinze questions, il empoche la somme rondelette de 500.000 €.

Par contre, si le candidat donne une mauvaise réponse, il risque de perdre une partie du gain déjà acquis. Cependant, certains montants intermédiaires constituent des paliers, c'est-à-dire une somme acquise que le candidat est sûr d'empocher, quoiqu'il arrive dans la suite du jeu.

À chaque question, le candidat a donc trois possibilités :

- ▷ il donne la réponse correcte : dans ce cas il augmente son gain, et peut passer à la question suivante
- ▷ il ne connait pas la réponse, et choisit de s'abstenir : dans ce cas, le jeu s'arrête et le candidat empoche le gain acquis à la question précédente
- ▷ il donne une réponse incorrecte : le jeu s'arrête également, mais le candidat ne recevra que le montant du dernier palier qu'il a atteint et réussi lors de son parcours. En particulier, si le candidat se trompe avant d'avoir atteint le premier palier, il ne gagne pas un seul euro !

Exemple : Le tableau ci-contre contient les gains associés à chaque question et une indication booléenne mise à vrai lorsque la question constitue un palier. Un concurrent qui se trompe à la question 3 ne gagnera rien ; un concurrent qui se trompe à la question 6 gagnera 500 € (palier de la question 5) et de même s'il se trompe à la question 10 ; un concurrent qui se trompe à la question 13 gagnera 12500 € (palier de la question 10) ; s'il décide de ne pas répondre à la question 13, il garde le montant acquis à la question 12, soit 50000 €.

1	25 €	faux
2	50 €	faux
3	125 €	faux
4	250 €	faux
5	500 €	vrai
6	1000 €	faux
7	2000 €	faux
8	3750 €	faux
9	7500 €	faux
10	12500 €	vrai
11	25000 €	faux
12	50000 €	faux
13	100000 €	vrai
14	250000 €	faux
15	500000 €	vrai

Il y aurait de nombreuses façons de coder ce problème ; en voici une :

La structure Question

Une question est composée du libellé de la question, des 4 libellés pour les réponses et d'une indication de la bonne réponse (un entier de 1 à 4). Par simplicité on en fait une structure mais on pourrait en faire une classe si on voulait par exemple vérifier que la « bonne réponse » possède une valeur correcte.

La structure Gain

Représente un niveau de gain. Elle contient les champs : montant (entier) et palier (un booléen à vrai si cette somme est assurée, faux sinon)

La classe Millionnaire

Cette classe code le moteur du jeu. On y retrouve

- questionnaire : un tableau de Question
- gains : un tableau de Gain
- autres attributs à déterminer (cf. méthodes)

ainsi que les méthodes pour

- initialiser le jeu à partir d'un questionnaire et du tableau de gains
- connaitre la question en cours
- donner la réponse du candidat à la question en cours
- savoir si le jeu est fini ou pas
- arrêter le jeu en repartant avec les gains
- les accesseurs nécessaires pour connaitre l'état du jeu.

Le jeu proprement dit

Le module jeuMillionaireConsole() reçoit le questionnaire et les gains et simule le jeu :

- Il propose les questions au candidat
- Il lit ses réponses (chiffre 1 à 4 ou 0 pour arrêter) et fait évoluer le jeu en fonction.
- lorsque le jeu est terminé, il indique au candidat le montant de ses gains.
- Attention ! Ce module devrait être le plus petit possible. Imaginez que vous devez également coder une version graphique. Tout code commun doit se trouver dans la classe Millionnaire !

7 Chambre avec vue

Un grand hôtel a décidé d'informatiser sa gestion administrative. Il a confié ce travail à la société ESI_INFO dans laquelle vous êtes un informaticien chevronné. On vous a confié la tâche particulière de la gestion des réservations pour ses 100 chambres. Pour ce faire, on vous demande d'écrire une classe Hôtel qui offre notamment une méthode qui permet d'enregistrer une réservation.

Pour représenter l'occupation des chambres un jour donné, nous allons utiliser un tableau de 100 entiers. Un 0 indique que la chambre est libre, une autre valeur (positive) indique le numéro du client qui occupe cette chambre ce jour-là.

Nous utiliserons une Liste de tels tableaux pour représenter l'occupation des chambres sur une longue période ; les éléments se suivant correspondant à des jours successifs.

Nous vous imposons les attributs de la classe, à savoir :

- occupations : une Liste de tableaux de 100 entiers comme expliqué ci-dessus.
- premierJour : donne le jour concerné par le premier élément de la liste. Ainsi s'il vaut 10/9/2014 cela signifie que le premier élément de la liste « occupations » renseigne sur l'occupation des chambres ce 10/9/2014 ; que le deuxième élément de la liste concerne le 11/9/2007 et ainsi de suite...

Écrire la méthode suivante

```
méthode effectuerRéservation(demande↓ : DemandeRéservation, chambre↑ : entier) → booléen
```

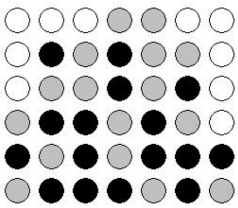
où la structure de demande de réservation est définie ainsi

```
structure DemandeRéservation
    numéroClient : entier
    débutRéservation : Date
    nbNuitées : entier
fin structure
```

- ▷ Le booléen retourné indique si la réservation a pu se faire ou pas
- ▷ Si elle a pu se faire, le paramètre de sortie chambre indique la chambre qui a été choisie
- ▷ Si plusieurs chambres sont libres, on choisit celle avec le plus petit numéro
- ▷ La demande de réservation peut couvrir une période qui n'est pas encore reprise dans la liste ; il faudra alors l'agrandir

8 Puissance 4

Le jeu de puissance 4 se déroule dans un tableau vertical comportant 6 rangées et 7 colonnes dans lequel deux joueurs introduisent tour à tour des jetons (rouges pour l'un, jaunes pour l'autre). Avec l'aide de la gravité, les jetons tombent toujours le plus bas possible dans les colonnes où on les place. Le jeu s'achève lorsqu'un des joueurs a réussi à aligner 4 de ses jetons horizontalement, verticalement ou en oblique, ou lorsque les deux joueurs ont disposé chacun leur 21 jetons sans réaliser d'alignement (match nul).



N.B. : sur ce dessin noir et blanc, les jetons rouges apparaissent en noir, les jetons jaunes en gris et les cases blanches désignent l'absence de jetons. Cet exemple montre une situation du jeu où le joueur « jaune » est gagnant. En introduisant un jeton dans la 4^{e} colonne, il a réalisé un alignement de 4 jetons en oblique.

On demande d'implémenter une classe Puissance4 qui permette de contrôler l'état des différentes phases du jeu. Déterminez les attributs de cette classe et décrivez-les brièvement de manière à justifier votre choix. Dotez ensuite la classe des méthodes permettant de :

- ▷ savoir si la grille est pleine
- ▷ mettre la grille à jour lorsque le joueur n (1 ou 2) joue dans la colonne j (entre 1 et 7). Cette méthode renverra la valeur booléenne faux si la colonne en question est déjà pleine
- ▷ vérifier si le joueur qui vient de jouer dans la colonne j a gagné la partie

N.B. : pour la structure qui contiendra le contenu du tableau de jetons, on adoptera la convention suivante : 0 pour l'absence de jeton, 1 représentera un jeton du 1^{er} joueur, et 2 un jeton du 2^{e} joueur (on peut donc faire abstraction de la couleur du jeton dans ce problème).

9 Les congés

Les périodes de congés des différents employés d'une firme sont reprises dans un tableau booléen **Congés** bidimensionnel à n lignes et 366 colonnes. Chaque ligne du tableau correspond à un employé et chaque colonne à un jour de l'année. Une case de ce tableau est mise à **vrai** si l'employé correspondant est en congé le jour correspondant. La firme en question

est opérationnelle 7 jours sur 7, on n'y fait donc pas de distinction entre jours ouvrables, week-end et jours fériés.

Ce tableau permet de visualiser l'ensemble des congés des travailleurs, et d'accorder ou non une demande de congé, suivant les règles suivantes :

1. une période de congé ne peut excéder 15 jours ;
2. un employé a droit à maximum 40 jours de congé par an ;
3. à tout moment, 50% des employés doivent être présents dans la firme.

Écrire un algorithme qui détermine si cette demande peut être accordée ou non à un employé dont on connait le nom, ainsi que les dates de début et de fin d'une demande de congé (objets de la classe Date). Dans l'affirmative, le tableau **Congés** sera mis à jour.

Pour établir la correspondance entre ce tableau et les noms des employés, vous avez à votre disposition un tableau **Personnel** de chaines. L'emplacement du nom d'un employé dans ce tableau correspond à l'indice ligne du tableau **Congés**.

Il est permis d'utiliser pour résoudre cet exercice la méthode suivante de la classe Date, sans devoir détailler son code :

```
méthode numéroJour() → entier        // la position du jour dans l'année (entre 1 et 366)
```

10 L'ensemble

La notion d'ensemble fini est une notion qui vous est déjà familière pour l'avoir rencontrée dans plusieurs cours. Nous rappelons certaines de ses propriétés et opérations.

Étant donnés deux ensembles finis **S** et **T** ainsi qu'un élément **x** :

- ▷ $\mathbf{x} \in \mathbf{S}$ signifie que l'élément **x** est un élément de l'ensemble **S**.
- ▷ L'ensemble vide, noté $\emptyset$ est l'ensemble qui n'a pas d'élément ($\mathbf{x} \in \emptyset$ est faux quel que soit **x**).
- ▷ L'ordre des éléments dans un ensemble n'a aucune signification, l'ensemble {1,2} est identique à {2,1}.
- ▷ Un élément **x** ne peut pas être plus d'une fois élément d'un même ensemble (pas de répétition).
- ▷ L'union $\mathbf{S} \cup \mathbf{T}$ est l'ensemble contenant les éléments qui sont dans **S** ou (non exclusif) dans **T**.
- ▷ L'intersection $\mathbf{S} \cap \mathbf{T}$ est l'ensemble des éléments qui sont à la fois dans **S** et dans **T**.
- ▷ La différence $\mathbf{S} \setminus \mathbf{T}$ est l'ensemble des éléments qui sont dans **S** mais pas dans **T**.

Créer la classe Ensemble décrite ci-dessous.

```
classe Ensemble <T>                                  // T est le type des éléments de l'ensemble
 public:
    constructeur Ensemble <T>()                          // construit un ensemble vide
    méthode ajouter(élt : T)                            // ajoute l'élément à l'ensemble
    méthode enlever(élt : T))                          // enlève un élément de l'ensemble
    méthode contient(élt : T) → booléen                  // dit si l'élément est présent
    méthode estVide() → booléen                           // dit si l'ensemble est vide
    méthode taille() → entier                           // donne la taille de l'ensemble
    méthode union(autreEnsemble : Ensemble <T>) → Ensemble <T>
    méthode intersection(autreEnsemble : Ensemble <T>) → Ensemble <T>
    méthode moins(autreEnsemble : Ensemble <T>) → Ensemble <T>
    méthode listeÉléments() → Liste <T>                          // conversion en liste
fin classe
```

Quelques remarques :

- ▷ La méthode d'ajout (resp. de suppression) n'a pas d'effet si l'élément est déjà (resp. n'est pas) dans l'ensemble.
- ▷ Les méthodes union(), intersection() et moins() retournent un troisième ensemble, résultat des 2 premiers sans toucher à ces 2 ensembles. On aurait pu envisager des méthodes modifiant l'ensemble sur lequel on les appelle.
- ▷ La méthode listeÉlément() est nécessaire si on veut parcourir les éléments de l'ensemble (par exemple pour les afficher).

Autres opérations ensemblistes : Nous avons défini des opérations ensemblistes ne touchant pas aux ensembles de départ. Que deviennent-elles si on considère qu'elles **modifient** l'ensemble sur lequel elles sont appliquées ?

11 Casino

Pour cet exercice, on vous demande un petit programme qui simule un jeu de roulette très simplifié dans un casino.

Dans ce jeu simplifié, vous pourrez miser une certaine somme et gagner ou perdre de l'argent (telle est la fortune, au casino !). Quand vous n'avez plus d'argent, vous avez perdu.

Notre règle du jeu

Bon, la roulette, c'est très sympathique comme jeu, mais un peu trop compliqué pour un exercice de première année. Alors, on va simplifier les règles et je vous présente tout de suite ce que l'on obtient :

- ▷ Le joueur mise sur un numéro compris entre 0 et 49 (50 numéros en tout). En choisissant son numéro, il y dépose la somme qu'il souhaite miser.
- ▷ La roulette est constituée de 50 cases allant naturellement de 0 à 49. Les numéros pairs sont de couleur noire, les numéros impairs sont de couleur rouge. Le croupier lance la roulette, lâche la bille et quand la roulette s'arrête, relève le numéro de la case dans laquelle la bille s'est arrêtée. Dans notre programme, nous ne reprendrons pas tous ces détails « matériels » mais ces explications sont aussi à l'intention de ceux qui ont eu la chance d'éviter les salles de casino jusqu'ici. Le numéro sur lequel s'est arrêtée la bille est, naturellement, le numéro gagnant.
- ▷ Si le numéro gagnant est celui sur lequel le joueur a misé (probabilité de 1/50, plutôt faible), le croupier lui remet 3 fois la somme misée.
- ▷ Sinon, le croupier regarde si le numéro misé par le joueur est de la même couleur que le numéro gagnant (s'ils sont tous les deux pairs ou tous les deux impairs). Si c'est le cas, le croupier lui remet 50% de la somme misée. Si ce n'est pas le cas, le joueur perd sa mise.

Dans les deux scénarios gagnants vus ci-dessus (le numéro misé et le numéro gagnant sont identiques ou ont la même couleur), le croupier remet au joueur la somme initialement misée avant d'y ajouter ses gains. Cela veut dire que, dans ces deux scénarios, le joueur récupère de l'argent. Il n'y a que dans le troisième cas qu'il perd la somme misée.

Annexe A

Aide-mémoire

Cet aide-mémoire peut vous accompagner lors d'une interrogation ou d'un examen. Il vous est permis d'utiliser ces classes et méthodes sans les développer. Si vous sentez le besoin d'utiliser un objet ou une méthode qui n'apparait pas ici, il faudra en écrire explicitement le contenu et le code.

A.1 Les caractères et les chaines

```
// Est-ce ?

estLettre(car : caractère)  → booléen                          // est-ce une lettre ?
estChiffre(car : caractère)  → booléen                         // est-ce un chiffre ?
estMajuscule(car : caractère)  → booléen                       // est-ce une majuscule ?
estMinuscule(car : caractère)  → booléen                       // est-ce une minuscule ?

// Conversions

majuscule(car : caractère)  → caractère      // convertit une minuscule en une majuscule.
minuscule(car : caractère)  → caractère      // convertit une majuscule en une minuscule.
numLettre(car : caractère)  → entier         // donne la position de la lettre dans l'alphabet.
lettreMaj(n : entier)  → caractère           // donne la lettre majuscule de position donnée.
lettreMin(n : entier)  → caractère           // donne la lettre minuscule de position donnée.
chaine(car : caractère)  → chaine                    // convertit le caractère en une chaine.
varChaine ←  varCaractère                                                          // idem
chaine(n : entier)  → chaine                         // convertit un entier en une chaine.
chaine(x : réelr)  → chaine                          // convertit un réel en une chaine.
nombre(ch : chaine)  → réel                          // convertit une chaine en un nombre.

// Manipulations

longueur(ch : chaine)  → entier                                  // donne la taille de la chaine.
car(ch : chaine, n : entier)  → caractère        // donne le caractère à une position donnée.
sousChaine(ch : chaine, pos : entier, long : entier)  → chaine      // extrait une sous-chaine
estDansChaine(ch : chaine, sous-chaine : chaine [ou caractère])  → entier
                // dit où commence une sous-chaine dans une chaine donnée (0 si pas trouvé)
concat(ch1, ch2, ..., chN : chaine)  → chaine                    // concatène des chaines
ch ←  ch1 + ch2 + ...+ chN                                                         // idem
```

A.2 La liste

```
classe Liste <T>
 public:
    constructeur Liste <T>()
    méthode get(pos : entier) → T                    // retourne l'élément en position pos
    méthode set(pos : entier, valeur : T)            // modifie l'élément en position pos
    méthode taille() → entier                        // retourne la taille de la liste
    méthode ajouter(valeur : T)                      // ajoute une valeur en fin de liste
    méthode insérer(pos : entier, valeur : T)        // insère un élément en position pos
    méthode supprimer()                              // supprime le dernier élément
    méthode supprimerPos(pos : entier)               // supprime l'élément en position pos
    méthode supprimer(valeur : T) → booléen          // supprime l'élément de valeur donnée
    méthode vider()                                  // vide la liste
    méthode estVide() → booléen                      // indique si la liste est vide
    méthode existe(valeur↓ : T, pos↑ : entier) → booléen   // recherche un élément
fin classe
```

A.3 Date, Moment, Durée

```
classe Date
 public:
    constructeur Date()                              // Crée la date du jour
    constructeur Date(j, m, a : entiers)
    méthode getJour() → entier
    méthode getMois() → entier
    méthode getAnnée() → entier
    méthode égale(autreDate : Date) → booléen
    méthode estAntérieure(autreDate : Date) → booléen
fin classe
```

```
classe Moment
 public:
    constructeur Moment()                            // Crée le moment courant
    constructeur Moment(h, m, s : entiers)
    méthode getHeure() → entier
    méthode getMinute() → entier
    méthode getSeconde() → entier
    méthode setHeure(h : entier)
    méthode setMinute(m : entier)
    méthode setSeconde(s : entier)
    méthode égal(autreMoment : Moment) → booléen
    méthode estAntérieur(autreMoment : Moment) → booléen
fin classe
```

```
classe Durée
 public:
    constructeur Durée(secondes : entier)
    constructeur Durée(h, m, s : entiers)
    méthode getJour() → entier
    méthode getHeure() → entier
    méthode getMinute() → entier
    méthode getSeconde() → entier
    méthode getTotalHeure() → entier
    méthode getTotalMinute() → entier
    méthode getTotalSeconde() → entier
    méthode ajouter(autreDurée : Durée)
    méthode soustraire(autreDurée : Durée)
    méthode égale(autreDurée : Durée) → booléen
    méthode plusPetit(autreDurée : Durée) → booléen
fin classe
```