Algorithmique & Programmation
Semestre 2 ST
Structuration des programmes en sous-algorithmes

Rappels du semestre 1

Types agrégés	Structuration en sous-algorithmes	Tableaux de variables	Algorithmes de recherche et de tri	Précision, rapidité et complexité
Matrices de variables	Récursivité	Travaux dirigés	Travaux pratiques	Informations
Sujets de projet	Evaluation intermédiaire	Evaluation finale	Evaluation complémentaire	Archives

Cours TD TP

Problématiques	Programmation modulaire	Sous-algorithme	Syntaxe Algorithmique
Syntaxe Java	Paramètres résultats agrégés en langage Java	Paramètres agrégés en données et en résultats en langage algorithmique	Recommandation

Version PDF

Clavier.class - Ecran.class - Documentation

Problématique n°1

- Comment développer de grosses applications quand on ne dispose que du "for", du "while", du "if" et du "case" pour structurer l'algorithme principal?

Solution

- Utiliser une méthode de développement permettant de diviser un gros problème complexe en un ensemble de sous-problèmes
  - plus petits,

- individuellement moins complexes,
- aussi indépendants les uns les autres que possible

- Résoudre ces sous-problèmes
- Assembler les solutions

Exemple

- Concevoir une voiture est un problème d'une grande complexité.
- Décomposition en un ensemble de problèmes consistant à concevoir:
  - Une carrosserie
  - Un habitacle
  - Un moteur
  - Une transmission
  - Une direction
  - Un système de freinage
  - Un système d'éclairage
  - ...

Avantages

- Méthode de conception guidant l'analyse
- Conception globale simplifiée
- Solution de chaque problème élémentaire validée individuellement
- Validation globale simplifiée si certitude de la validation de chaque solution élémentaire
- Résolution indépendante possible de chaque problème élémentaire:
  - Travail simultané de plusieurs développeurs -> Développement accéléré
  - Affectation des problèmes à des spécialistes du domaine -> Meilleurs développements

Inconvénients

- L'ensemble des solutions élémentaires concourt à la résolution du problème global
  -> Solutions élémentaires non totalement indépendantes les unes des autres
  -> Conception de l'interaction entre les solutions élémentaires: Problème en soi
- Décomposition en sous-problèmes pas toujours possible

Problématique n°2

- Comment éviter de résoudre plusieurs fois un même problème quand celui-ci apparait plusieurs fois dans la même application ou dans des applications différentes?

- Comment faciliter la réutilisation de codes informatiques que l'on a développés?
- Comment faciliter l'utilisation de codes informatiques par d'autres personnes que celles qui les ont conçus?

Solution

- Utiliser une technique de développement permettant d'"englober" la résolution algorithmique d'un problème de manière qu'il soit possible de faire appel à l'algorithme sans le réécrire

Exemples

- En langage Java:
  - Math.sin(x) -> Calcul de sinus(x) et "retour" du résultat de ce calcul
  - "Fonction" standard du langage Java

- En langage Java:
  - Clavier.saisirInt() -> Lecture au clavier d'une valeur de type java int et "retour" du résultat de cette opération de lecture
  - "Fonction" non standard de Java (développée ad hoc pour les séances de TP -> Utilisation du fichier Clavier.class)

Avantages

- Technique algorithmique guidant l'analyse
- Réutilisation d'algorithmes développés et validés par soi-même ou par d'autres developpeurs
- Conception de vastes bibliothèques d'algorithmes (librairies, API (Application Programming Interface))
  - Exemples: Ecran.class, Clavier.class, l'API Java
- Correction des bugs plus facile
  - Trouver l'algorithme où est présent le bug
  - Le corriger dans cet algorithme
  - Correction automatiquement déployée partout où l'algorithme est utilisé

Inconvénients

- Aucun

Programmation modulaire à base de sous-algorithmes

- Sous-algorithme: Algorithme informatique développé dans le but d'être "utilisé dans" (appelé depuis) d'autres algorithmes

- Diverses dénominations pour désigner le concept de "sous-algorithme" suivant divers contextes de développement:
  - Fonction (terme usuel):
    - Programmation structurée (C, Pascal, Basic, Fortran, ADA, Java, Javascript, PHP, ...)
    - Programmation fonctionnelle (Lisp, Caml, ...)
    - Algorithmique PDL
  - Procédure:
    - Programmation structurée (Pascal, ...)
  - Méthode:
    - Programmation orientée objet (Java, C++, C#, ...)
  - Sous-programme:
    - Programmation structurée
  - Sous-algorithme:
    - Algorithmique PDL
  - Action:
    - Algorithmique PDL

Exemple n°1: Calcul de la valeur d'un polynôme

- **Problème**: Ecrire un algorithme de calcul de $f(x,m,n) = x^m+x^n$ où x est réel (différent de 0.0) et m et n sont des entiers positifs ou nuls. x, m et n sont acquis au clavier. Le résultat du calcul est affiché à l'écran.

Analyse

- Paramètres en entrée:
  - x: nommé x et défini de type réel
  - m: nommé m et défini de type entier
  - n: nommé n et défini de type entier
- Un paramètre résultat nommé res et défini de type réel
- Calcul de la première puissance: Si m = 0, x^m = 1.0 sinon x^m = x * x * x* ... *x (m-1 multiplications réalisées au moyen d'une boucle pour)
- Calcul de la seconde puissance: Si n = 0, x^n = 1.0 sinon x^n = x * x * x * ... * x (n-1 multiplications réalisées au moyen d'une boucle pour)

Conception en langage algorithmique:

```
action principale()
   Locales
   x : réel
   m,n : entier
```

```
   res : réel
   i : entier
   res1,res2 : réel
 x <- Clavier.saisirReel()
 m <- Clavier.saisirEntier()
 n <- Clavier.saisirEntier()
  si m = 0 alors                  // \
   res1 <- 1.0                    // |
    sinon                          // |
   res1 <- x                      // -- (1)
    pour i de 1 à m-1 faire       // |
     res1 <- res1 * x             // |
    fait                           // |
  fsi                             // /
  si n = 0 alors                  // \
 res2 <- 1.0                      // |
    sinon                          // |
   res2 <- x                      // -- (2)
    pour i de 1 à n-1 faire       // |
     res2 <- res2 * x             // |
    fait                           // |
  fsi                             // /
 res <- res1 + res2
 Ecran.afficher("f(x,m,n) = ",res)
fin action
```

- Blocs de code (1) et (2) très semblables (m pour n, res1 pour res2)
- Normal car ces blocs réalisent tous deux le calcul d'une puissance entière de x (une première fois avec n comme puissance, une seconde fois avec m)
- Exploitation de cette ressemblance pour simplifier l'écriture de notre action principale:
  -> Ecriture d'un sous-algorithme (une fonction) de calcul de la puissance entière d'un nombre réel
  -> Utilisation de ce sous-algorithme dans l'action principale une première fois à la place du bloc (1) et une seconde fois à la place du bloc (2)

Syntaxe souhaitée après réécriture:

```
action principale()
   locales
    x : réel
    m,n : entier
    res : réel
    res1,res2 : réel
  x <- Clavier.saisirReel()
```

```
  m <- Clavier.saisirEntier()
  n <- Clavier.saisirEntier()
  res1 <- puissance(x,m)        // (1)
  res2 <- puissance(x,n)        // (2)
  res <- res1 + res2
  Ecran.afficher("f(x,m,n) = ",res)
fin action
```

- Deux paramètres pour le sous-algorithme développé:
  - Dans l'ordre:
    - Un réel (nommons le v)
    - Un entier (nommons le n)
- Calcul de la puissance n de v
- "Retour" de cette valeur

Sous-algorithme

- Conçu pour réaliser un traitement
  - Bien défini
  - Bien délimité
  - Si possible de manière parfaitement indépendante au contexte particulier de l'algorithme appelant pour qu'il puisse être utilisé dans n'importe quel contexte

- Généralement muni de paramètres
  - 0, 1 ou plusieurs paramètres "passés" en entête et transmis au sous-algorithme par l'algorithme appelant en tant que données du traitement
    -> Paramêtre(s) en entrée, en donnée
  - 0, 1 ou plusieurs paramètres "passés" (retournés) en entête en tant que résultats du traitement réalisé par le sous-algorithme
    -> Paramètre(s) en sortie, en résultat

    Résolution du problème de la communication des données de l'algorithme appelant à destination du sous-algorithme appelé
    Résolution du problème de la communication des résultats du sous-algorithme appelé à destination de l'algorithme appelant

- Optionnellement muni de variables locales
  - Utilisation possible de variables temporaires au fonctionnement du sous-algorithme pour assurer le traitement interne du sous-algorithme

- Comment déterminer les paramètres d'entrée et de sortie?
  Analyser l'énoncé du problème car celui-ci est généralement précis sur cette

question
En cas d'imprécision, responsabilité du choix laissée à la personne qui réalise l'analyse

Syntaxe en langage algorithmique

- Un entête (extrêmement important car il définit comment le sous-algorithme pourra être appelé par l'algorithme appelant):
  - Le nom du sous-algorithme (par l'intermédiaire duquel il sera désigné lors d'un appel)
  - Les paramètres passés en donnée
  - Les paramètres passés en résultat
  - Les paramètres passés tout à la fois en donnée et en résultat (on pourrait dire en "modification")
- Une zone de définition pour d'éventuelles variables locales
- Un corps de réalisation de tous les traitements de calcul des résultats à partir des données (en utilisant les variables locales)
- Syntaxe:

```
action nom_action(liste exhaustive des noms des
parametres)
  Données
    déclarations de parametres
  Résultats
    déclarations de parametres
  Données/résultats
    déclarations de parametres
  Locales
    déclarations de variables
  corps du sous-algorithme
fin action
```

- Liste exhaustive des noms des paramètres (séparateur: virgule)
- Zone Données:
  - Déclaration de tous les paramètres passés en donnée
  - Utilisation de la syntaxe *nomParametre : type* pour chacun d'eux
- Zone Résultats:
  - Déclaration de tous les paramètres passés en résultat
  - Utilisation de la syntaxe *nomParametre : type* pour chacun d'eux
- Zone Données/Résultats:
  - Déclaration de tous les paramètres gérés en donnée/résultat (modification)

  - Utilisation de la syntaxe *nomParametre : type* pour chacun d'eux
- Zone Locales
  - Déclaration de toutes les variables locales au sous-algorithme
  - Utilisation de la syntaxe *nomVariable : type* pour chacun d'eux
- Présence de tous les paramètres de la liste d'entête soit en donnée, soit en résultat, soit en donnée/résultat

- Utilisation permise du mot réservé "fonction" au lieu de "action"

- Appel à ce sous-algorithme au moyen d'une instruction formatée sous la forme:
  `nom_action(liste de parametres)`
  où les paramètres de la liste d'appel sont donnés dans le bon ordre, avec le bon type et séparés par des virgules

Nom des actions

- Nom choisi de manière explicite vis à vis du traitement réalisé
- Lettres non accentuées, chiffres et _ autorisés
- Interdiction de débuter par un chiffre
- **Conventions**
  - Ecriture en minuscule
  - Si nom composé, à l'exception de la première, initiale de chaque item en majuscule
- **Exemples**: `testEgalite`, `saisirEntier`

Exemple

```
action puissance(v,n,p)
  Données
    v : réel
    n : entier
  Résultats
    p : réel
  Locales
    i : entier
  si n = 0 alors
    p <- 1.0
    sinon
    p <- v
    pour i de 1 à n-1 faire
      p <- p*v
    fait
  fsi
fin action
```

Cas particulier: Sous-algorithme sans donnée et/ou sans résultat

- Exemple: Ecran.sautDeLigne()

- Possiblement aucun paramètre d'entête en donnée
- Possiblement aucun paramètre d'entête en résultat

Cas particulier: Sous-algorithme avec acquisition interne des données et/ou exploitation interne des résultats

- Si l'énoncé du problème le justifie: Acquisition des données du traitement (définies au moment de l'analyse) directement au sein du sous-algorithme (clavier, fichier, réseau, ...) et non en entête

- Si l'énoncé du problème le justifie: Exploitation des résultats du traitement (définis au moment de l'analyse) directement au sein du sous-algorithme (affichage à l'écran, écriture dans un fichier, ...) sans transmission en résultat en entête

- Conséquences:
  - Possiblement aucun paramètre d'entête en donnée
  - Possiblement aucun paramètre d'entête en résultat

Exemples n°2

- Enoncé (1)
  - Ecrire un sous-algorithme qui réalise les traitements suivants:
    - Lecture au clavier d'un paramètre de type réel
    - Calcul du carré de ce paramètre
    - Affichage de la valeur obtenue
  - Mini-analyse:
    Une seule donnée de traitement acquise au clavier au sein du sous-algorithme donc non passée en entête
    Un seul résultat de traitement directement affiché et donc non passé en résultat de sous-algorithme

- Enoncé (2)
  - Ecrire un sous-algorithme de calcul du carré d'un réel
  - Mini-analyse:
    Une seule donnée de traitement acquise en entête
    Un seul résultat de traitement passé en résultat de sous-algorithme

Implantation de l'énoncé 1 de exemple n°2

```
action carreEnnonce1()
   Locales
   x : réel
   res : réel
  Ecran.afficher("SVP, valeur? ")
 x <- Clavier.saisirReel()
 res <- x*x
 Ecran.afficherln(x," au carre = ",res)
fin action
```

Implantation de l'énoncé 2 de exemple n°2

```
action carreEnnonce2(x,res)
   Données
   x : réel
   Résultats
   res : réel
 res <- x*x
fin action
```

Syntaxe alternative très fréquemment employée

- Conception d'un sous-algorithme de manière à "retourner" un résultat de manière plus explicite que les autres paramètres résultats
- Utilisation systématisée de cette syntaxe lors de la conception d'un sous-algorithme à résultat unique (cf exemple n°1)

- Syntaxe générale:

```
type action nom_action(liste des parametres)
  Données
    déclarations de parametres
  Résultats
    déclarations de parametres
  Données/résultats
    déclarations de parametres
  Locales
    déclarations de variables dont la variable résultat
  corps du sous-algorithme dont affectation de résultat
  retourner résultat
fin action
```

- Type de sous-algorithme fréquemment appelé "fonction" par analogie aux fonctions mathématiques dont le but est de calculer et de rendre **une** valeur

- Type de donnée indiqué avant le mot réservé action: Type de la valeur "retournée" (rendue) par l'action
- Présence spécifique dans l'action d'une variable dédiée de ce type parmi les variables locales
- Affectation obligatoire de cette variable avec la valeur que l'on souhaite rendre
- Utilisation de l'instruction **`retourner variable`** juste avant le **`fin action`** pour indiquer explicitement la variable et donc la valeur retournée

- Remarque: Première ligne d'entête possiblement décrite sous la forme

```
action nom_action(liste des parametres): type
```

- Remarque: Pour ce type de sous-algorithme, utilisation recommandée du mot réservé "fonction" à la place du mot réservé "action"

- Exemple d'appel à ce sous-algorithme: Affectation du résultat d'exécution à une variable
  `variable <- nom_action(liste de parametres)`
  où les paramètres d'appel sont séparés par des virgules et apparaissent dans l'ordre de définition
- Appel direct autorisé au sein d'une expression:
  `Ecran.afficherln(nom_action(liste de parametres))`

Implantation alternative de l'énoncé 2 de exemple n°2

```
réel fonction carreEnnonce2(x)
   Données
   x : réel
   Locales
   res : réel
 res <- x*x
  retourner res
fin fonction
```

Implantation de l'exemple n°1

```
réel fonction puissance(v,n)
   Données
    v : réel
    n : entier
   Locales
    p : réel
    i : entier
   si n = 0 alors
```

```
    p <- 1.0
     sinon
    p <- v
     pour i de 1 à n-1 faire
      p <- p * v
     fait
   fsi
   retourner p
fin fonction

action principale()
   locales
    x : réel
    m,n : entier
    res : réel
    res1,res2 : réel
  x <- Clavier.saisirReel()
  m <- Clavier.saisirEntier()
  n <- Clavier.saisirEntier()
  res1 <- puissance(x,m)
  res2 <- puissance(x,n)
  res <- res1 + res2
  Ecran.afficher("f(x,m,n) = ",res)
fin action
```

Seconde implantation de l'exemple n°1

```
réel fonction puissance(v,n)
   Données
    v : réel
    n : entier
   Locales
    p : réel
    i : entier
   si n = 0 alors
    p <- 1.0
     sinon
    p <- v
     pour i de 1 à n-1 faire
      p <- p * v
     fait
   fsi
   retourner p
fin fonction

réel fonction polynome(x,m,n)
   Données
    x : réel
    m : entier
```

```
    n : entier
  Locales
    res : réel
  res <- puissance(x,m)+puissance(x,n)
   retourner res
fin fonction

action principale()
   locales
    x : réel
    m,n : entier
    res : réel
  x <- Clavier.saisirReel()
  m <- Clavier.saisirEntier()
  n <- Clavier.saisirEntier()
  res <- polynome(x,m,n)
  Ecran.afficher("f(x,m,n) = ",res)
fin action
```

Implantation de sous-algorithmes en langage Java

- Dénomination usuelle en langage Java: "méthode" (ou plus usuellement "fonction")
- Application Java:
  - Obligatoirement une méthode main d'entête "`public static void main (Strings [] args)`"
  - Méthode main implicitement exécutée au lancement de l'application
  - Présence possible d'autres méthodes que la méthode main:
    - Méthodes incluses dans la classe application au dessus ou en dessous de la méthode main (pas à l'intérieur)

Entête d'une méthode Java

Syntaxe de l'entête de toute méthode Java ayant des paramètres:

```
static type nomMethode(type param1, type param2, ..., type paramN)
```

- Paramètres d'entête indiqués sous la forme d'une liste avec le caractère virgule comme séparateur
- Type de chacun d'eux spécifié dans la liste avant son nom -> placé dans l'entête
- Nom de la méthode choisi selon les mêmes conventions que les noms de variable

Si méthode sans paramètre, syntaxe de l'entête:

```
static type nomMethode()
```

- Nom de type juste après le mot réservé static: Type de l'**unique** valeur rendue par la méthode
- Utilisation du type "void" (vide en anglais) si pas de valeur retournée
  Emploi de void comme type de valeur retournée **obligatoire** pour de telles méthodes
  Méthodes nommées "méthodes void"

Corps d'une méthode Java

- Une zone de déclaration de variables locales
- Une zone de réalisation des traitements

Si méthode avec valeur rendue (méthode "non void"), dernière instruction de traitement:

```
return valeur;
```

- valeur: Une variable du type rendu spécifié en entête

Si méthode sans valeur rendue (méthode "void"): Aucune instruction return (rendre quoi?)

Implantation Java de l'exemple n°1

```
public class Polynome {

/* Calcul de la puissance entiere d'un reel    */

   static double puissance(double v,int n) {
     double p;
     int i;
     if ( n == 0 ) {
      p = 1.0; }
       else {
      p = v;
       for ( i = 1 ; i <= n-1 ; i = i+1 ) {
        p = p*v; } }
     return p;
  }

/* Calcul d'un polynome                        */

   static double polynome(double x,int m,int n) {
     double res;
    res = puissance(x,m)+puissance(x,n);
     return res;
  }

/* Programme principal                         */
```

```
   public static void main(String [] args) {
     double x;
     int m;
     int n;
     double p;
    Ecran.afficher("SVP, x : ");
    x = Clavier.saisirDouble();
    Ecran.afficher("SVP, m : ");
    m = Clavier.saisirInt();
    Ecran.afficher("SVP, n : ");
    n = Clavier.saisirInt();
    p = polynome(x,m,n);
    Ecran.afficherln("Resultat = ",p);
  }
}
```

Polynome.java - Exemple d'exécution

Autres exemples

- **Exemple 1**
  - Une méthode surfaceCercle de calcul de la surface d'un cercle connu par son rayon (un réel en entête: le rayon, un réel retourné: la surface)
  - Une méthode main qui utilise la méthode surfaceCercle pour calculer et afficher la surface d'un cercle dont le rayon a été préalablement lu au clavier

```
public class SurfaceCercle {

/* Calcul de la surface d'un cercle              */
/* Le rayon r est donne                          */

   static double surfaceCercle(double r) {
     double s;
    s = Math.PI*r*r;
     return s;
  }

/* Programme principal                           */

   public static void main(String [] args) {
     double rayon;
     double surface;
    Ecran.afficher("SVP, rayon : ");
    rayon = Clavier.saisirDouble();
    surface = surfaceCercle(rayon);
    Ecran.afficherln("Surface = ",surface);
```

```
   }
}
```

SurfaceCercle.java - Exemple d'exécution

- **Exemple 2**
  - Une méthode lectureEntierPositif de lecture au clavier d'un entier positif (essais successifs) (pas de paramètre d'entête, un entier retourné: la valeur lue)
  - Une méthode main qui utilise la méthode lectureEntierPositif pour lire au clavier et afficher un entier (simple validation du fonctionnement)

```
public class LectureEntierPositif {

/* Lecture au clavier et retour d'un entier     */
/* Saisie par essais successifs                 */

   static int lectureEntierPositif() {
     int v;
    v = Clavier.saisirInt();
     while ( v < 0 ) {
      Ecran.afficher("Erreur, resaisissez : ");
      v = Clavier.saisirInt(); }
     return v;
  }

/* Programme principal                          */

   public static void main(String [] args) {
     int n;
    Ecran.afficher("SVP, n : ");
    n = lectureEntierPositif();
    Ecran.afficherln("Valeur lue = ",n);
  }
}
```

LectureEntierPositif.java - Exemple d'exécution

- **Exemple 3**
  - Une méthode afficherEntiersPairs d'affichage de tous les entiers pairs entre 0 et une valeur limite (un entier en entête: la valeur limite, pas de valeur rendue),
  - Une méthode main de validation

```
public class AffichageEntiersPairs {

/* Affichage de tous les entiers pairs entre 0 et n */

   static void afficherEntiersPairs(int n) {
     int i;
     for ( i = 0 ; i <= n ; i = i+2 ) {
      Ecran.afficher(i," "); }
  }

/* Programme principal                              */

   public static void main(String [] args) {
     int n;
    Ecran.afficher("SVP, n : ");
    n = Clavier.saisirInt();
    afficherEntiersPairs(n);
    Ecran.afficherln();
  }
}
```

AffichageEntiersPairs.java - Exemple d'exécution

Quels types de paramètre en entête de méthode Java?

- Tous!
  - Types pédéfinis
  - Types agrégés

Exemple: Calculs du produit scalaire et du produit vectoriel de deux directions 3D:

```
/* Type agrege de stockage d'une direction      */
/* dans un espace trois dimensions              */
/* a coordonnees reelles                        */

static class Direction3D {
   double x = 0.0;
   double y = 0.0;
   double z = 0.0; };

/* Calcul du produit scalaire de 2 Direction3D */

static double produitScalaire(Direction3D d1,
                              Direction3D d2) {
   double res = d1.x*d2.x+d1.y*d2.y+d1.z*d2.z;
   return res;
}
```

```
/* Calcul du produit vectoriel                    */
/* de 2 Direction3D                               */

static Direction3D produitVectoriel(Direction3D d1,
                                    Direction3D d2) {
   Direction3D res = new Direction3D();
  res.x = d1.y*d2.z-d1.z*d2.y;
  res.y = d1.z*d2.x-d1.x*d2.z;
  res.z = d1.x*d2.y-d1.y*d2.x;
   return res;
}
```

Produits3D.java - Exemple d'exécution

Passages de paramètres en donnée et en résultat en langage Java

- Quid des modes de passage de paramètre en Donnée, en Résultat et en Donnée/Résultat tels que définis en langage algorithmique?
- Fonctionnalité non implantée explicitement en langage Java
- Mode de passage défini implicitement par le type de paramètre

- En langage Java, mode de passage de paramètre pour les types byte, short, int, long, float, double, char, boolean et String (tous les types prédéfinis) implicitement et obligatoirement en "Donnée"
- En langage Java, mode de passage de paramètre pour les types agrégés implicitement et obligatoirement en "Donnée/résultat"
- En langage Java, pas de mode de passage de paramètre en résultat

- Tous les types de donnée supportent le mode de passage de paramètre en Donnée
  -> Pas de problème insoluble pour la traduction langage algorithmique -> Java
- Seuls les types de donnée agrégé supportent le mode de passage de paramètre en Résultat
  -> Problème pour le passage en résultat des paramètres de type prédéfini

- Designs de programmation suivant le nombre de résultats de type prédéfini:
  - Si 0 résultat:
    - Une seule possibilité, une méthode void
  - Si 1 résultat de type prédéfini A:
    - Une seule possibilité, une méthode non void avec retour d'un paramètre du type A
  - Si 1 résultat de type agrégé B:
    - Deux possibilités:
      - Une méthode non void avec retour d'un paramètre du type B

        - Une méthode void avec un paramètre d'entête de type B (implicitement passé en Donnée/Résultat)
    - Si 2 ou plus résultats de types prédéfinis, on se débrouille pour revenir à la situation précédente:
        - Construction d'un type agrégé C permettant l'agrégation des paramètres en résultat en un seul paramètre agrégé
        - Deux possibilités:
            - Une méthode non void avec retour d'un paramètre de type C
            - Une méthode void avec un paramètre d'entête de type C (implicitement en Données/Résultats)

Exemple

- Développer un sous-algorithme ayant comme résultat les coordonnées d'un point dans un espace à deux dimensions réelles (par exemple l'intersection entre deux droites du plan)
- En langage algorithmique, 3 possibilités:
    - Développement d'une action avec utilisation de deux paramètres d'entête de type réel nommés x et y passés en "Résultat"
    - Création d'un type de données agrégeant deux champs de type réel nommés x et y
      Développement d'une action avec passage d'un paramètre d'entête de ce type spécifié en "Résultat"
    - Création d'un type de données agrégeant deux champs de type réel nommés x et y
      Développement d'une fonction avec renvoi d'une valeur de ce type en retour
- En langage Java, 2 possibilités:
    - Création d'un type de données agrégeant deux champs de type réel nommés x et y
      Développement d'une méthode void avec passage d'un paramètre d'entête de ce type (implicitement en "Donnée/résultat")
    - Création d'un type de données agrégeant deux champs de type réel nommés x et y
      Développement d'une méthode non void retournant comme résultat une valeur définie selon ce type

Application Java de calcul de l'intersection de deux droites du plan:

```
public class IntersectionDroites2D {

/* Type agrege de stockage de coordonnees       */
/* reelles en deux dimensions                   */
```

```
   static class Coordonnees2D {
     double x = 0.0;
     double y = 0.0; };

/* Type agrege de stockage des composantes     */
/* reelles d'une droite du plan                */

   static class Droite2D {
     double a = 0.0;
     double b = 0.0; };

/* Calcul de l'intersection entre deux droites */
/* d'équations y = a1*x+b1 et y = a2*x+b2      */

   static Coordonnees2D intersection(Droite2D d1,
                                     Droite2D d2) {
     Coordonnees2D inter = new Coordonnees2D();
    inter.x = (d2.b-d1.b)/(d1.a-d2.a);
    inter.y = d1.a*inter.x+d1.b;
     return inter;
  }

/* Programme principal                         */

   public static void main(String [] args) {
     Droite2D d1 = new Droite2D();
     Droite2D d2 = new Droite2D();
     Coordonnees2D inter;
    Ecran.afficher
("SVP, coefficient directeur droite 1 : ");
    d1.a = Clavier.saisirDouble();
    Ecran.afficher
("SVP, ordonnée à l'origine droite 1      : ");
    d1.b = Clavier.saisirDouble();
    Ecran.afficher
("SVP, coefficient directeur droite 2 : ");
    d2.a = Clavier.saisirDouble();
    Ecran.afficher
("SVP, ordonnée à l'origine droite 2      : ");
    d2.b = Clavier.saisirDouble();
    inter = intersection(d1,d2);
    Ecran.afficherln
("Intersection : (",inter.x,",",inter.y,")");
  }
}
```

IntersectionDroites2D.java - Exemple d'exécution

Passages de paramètres agrégés en données et résultats en langage algorithmique

- Utilisation de paramètres de types agrégés en entête et en résultat d'action autorisée en langage algorithmique
- Support de tous les types de passages en paramètre (pas d'exception contrairement au langage Java):
  - En donnée
  - En résultat
  - En donnée/résultat

En langage algorithmique, action de calcul de l'intersection de deux droites du plan:

```
{ Structure de stockage de coordonnees reelles }
{ en deux dimensions                           }

structure coordonnees2D
  x : reel <- 0.0
  y : reel <- 0.0
fin structure

{ Type agrege de stockage des composantes      }
{ reelles d'une droite du plan                 }

structure droite2D
  a : reel <- 0.0
  b : reel <- 0.0
fin structure

{ Calcul de l'intersection entre deux droites  }
{ d'équations y = a1*x+b1 et y = a2*x+b2       }

coordonnees2D fonction intersection(d1,d2)
   Données
    d1 : droite2D
    d2 : droite2D
   Locales
    inter : coordonnees2D
  inter.x <- (d2.b-d1.b)/(d1.a-d2.a)
  inter.y <- d1.a*inter.x+d1.b
   retourner inter
fin fonction
```

IntersectionDroites2D.lda - Exemple d'exécution

Recommandation

- Ecrire des fonctions courtes (i.e. qui tiennent entièrement à l'écran)
  -> Plus lisible
- Privilégier les sous-algorithmes avec résultat retourné
  -> Plus clair
- Structurer les données utilisées en passage de paramètre
  -> Plus clair et plus lisible

Auteur: Nicolas JANEY
UFR Sciences et Techniques
Université de Besançon
16 Route de Gray, 25030 Besançon
nicolas.janey@univ-fcomte.fr