iUT ORSAY

Université Paris XI
I.U.T. d'Orsay
Département Informatique
Année scolaire 2003-2004

Algorithmique : Volume 3

- Agrégats
- Classes

Cécile Balkanski, Nelly Bensimon, Gérard Ligozat

Types de base, types complexes

- les types de base :
  - entier, réel, booléen, caractère, chaîne
- un type plus complexe :
  - tableau
- suffisant pour représenter :
  - les notes d'une promotion (*tableau d'entiers*)
  - un mot, une phrase (*tableau de caractères)*
  - les élèves d'une promotion (*tableau de chaînes*)
- mais comment représenter, par exemple, les clients d'une société?
  - un client : un nom, un prénom, un numéro, une adresse, etc.
  - les clients d'une société ➔ ***tableau de clients***

Nouveaux types : les agrégats

- **Agrégats "homogènes"** (types identiques)

- exemples : point dans le plan :

type Point = agrégat
abscisse : réel
ordonnée : réel
fin

- identité d'une personne

type Identité = agrégat
nom : chaîne
prénom : chaîne
fin

- **Agrégats " hétérogènes "** (types différents)

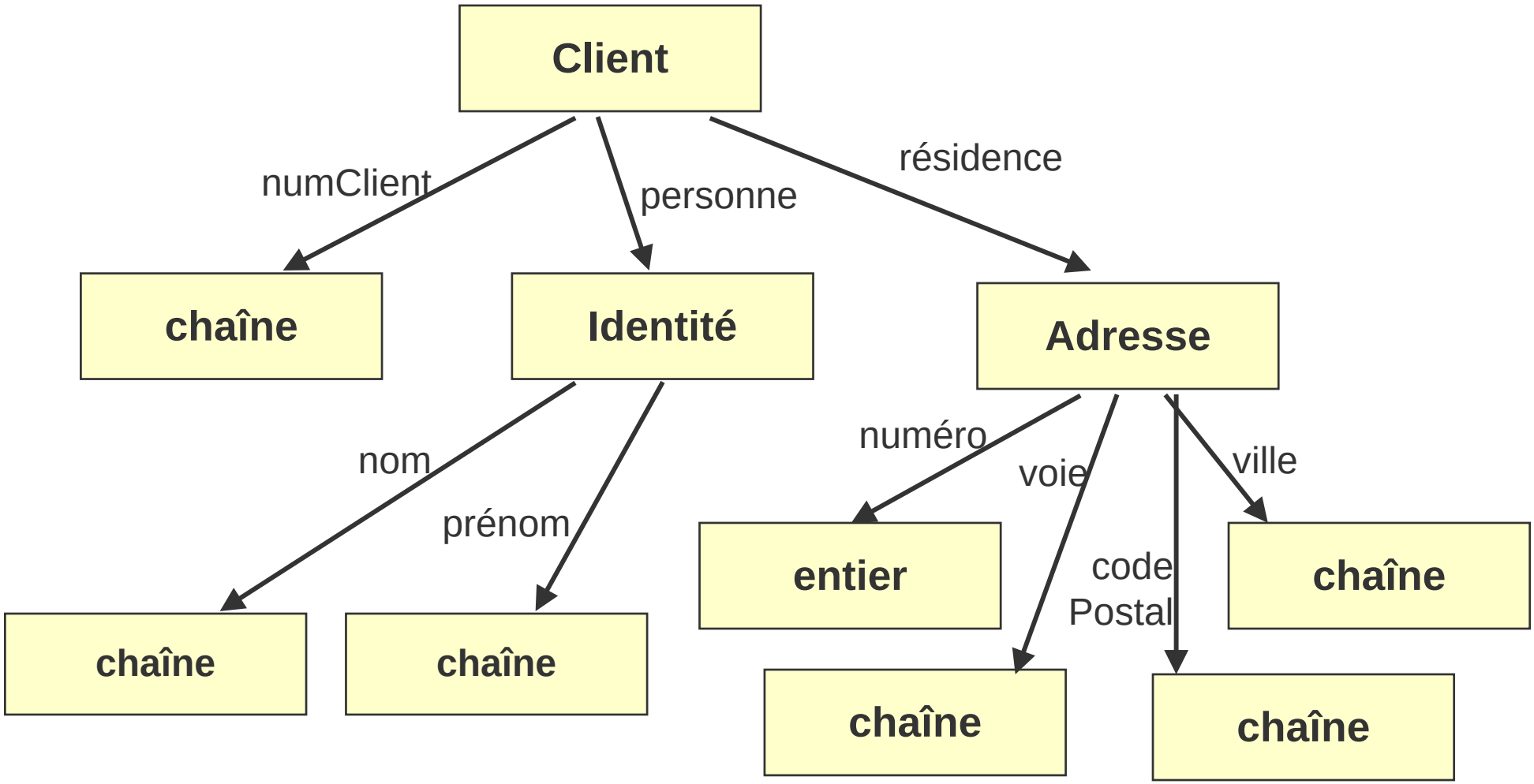
- exemples : adresse d'une personne:

type Adresse = agrégat
numéro : entier
voie : chaîne
codePostal : chaîne
ville : chaîne
fin

- client :

type Client = agrégat
numClient : chaîne
personne : Identité
résidence : Adresse
fin

Hiérarchie de types



Définir un agrégat

type <nomAgrégat> = **agrégat**
attribut1 : type1
attribut2 : type2
…
attributK : typeK
fin

- Déclaration de variables de type agrégat :

variables mrA : **Client**
pointA, pointB : **Points**

Distinction type - variable

- Un type :
  - Un modèle de structure de données
  - Un "moule" utilisé pour la déclaration de variables d'un algorithme
  - Il permet la déclaration de paramètres dans les sous-algorithmes

- Une variable:
  - Une entité "déclarée" en se référant à un modèle fourni par un type
  - Entité effectivement opérationnelle dans un algorithme

Accès aux agrégats

- Accès en écriture et en lecture :

  saisir(pointA.abscisse) ;
  saisir(mrA.numClient) ;
  saisir(mrA. résidence.numéro) ;

  afficher(pointA.abscisse)
  afficher(mrA.numClient)
  afficher(mrA.résidence.numéro)

- Affectation d'un attribut

  pointA.abscisse ← 3
  mrA.numClient ← "AZ345"
  mrA.résidence.numéro ← 45

- Affectation globale : par affectation de tous les attributs homonymes

  mrA, jojo : Client
  jojo $\leftarrow_{\text{Client}}$ mrA

  possible seulement si le client mrA est entièrement initialisé

- Comparaison :

  si pointA.abscisse = pointB.abscisse **alors …** *champ par champ*
  si pointA.abscisse = pointB.ordonnée **alors …** *ok si champs de même type*
  si jojo $=_{Client}$ mrA **alors …** *ok si même agrégats*

- Concrètement, on doit construire les fonctions correspondantes
  - afficher, saisir :

    afficherX(<agrégat X>) ex: afficherPoint(unPoint)
    saisirX(<agrégat X>) ex: saisirClient(unClient)

- Attention :
  - Ne pas utiliser sans précaution affichage, saisie, affectation, et comparaison
  - Une bonne organisation de la programmation doit fournir à l'utilisateur les fonctions correspondantes

Exemple

Algorithme ManipPoints
{manipulation de points}
variables pointA, pointB : **Point**
début
saisirPoint(pointA)
saisirPoint(pointB)
si pointA $=_{Point}$ pointB **alors** déplacerPoint(pointA, 3, 3)
fin

Procédure saisirPoint(unPoint)
{saisie d'un point}
paramètre (R) unPoint : Point
début
afficher("Abscisse du point :")
saisir(unPoint.abscisse)
afficher("Ordonnée du point :")
saisir(unPoint.ordonnée)
fin

Procédure déplacerPoint(unPoint, x, y)
{ajoute x (y) à l'abscisse (l'ordonnée) d'un point}
paramètres (D/R) unPoint : Point
(D) x, y : entiers
début
unPoint.abscisse ← unPoint.abscisse + x
unPoint.ordonnée ← unPoint.ordonnée + y
fin

Tableaux d'agrégats

Exemple : un tableau contenant les clients d'une société

- Déclaration tabClients : tableau [1, TMAX] de Clients
- Accès

  tabClients[1] → *le client enregistré dans la 1ère case du tableau*

  tabClients[3].numClient → *le numéro de client du client enregistré dans la 3ème case du tableau*

  tabClients[2].résidence.ville → *la ville où réside le client enregistré dans la 2ème case du tableau*

- Attention à l'ordre d'accès !

  du plus général au plus précis *(voir la hiérarchie des types)*

Algorithme Clients

{saisit un tableau de clients, et crée un tableau d'indices des clients résidant dans une ville donnée}

types **Client** = agrégat

...

fin

TableauClients = tableau [1, Tmax] de **Clients**

variables tabSociété : **TableauClients**

tabUneVille : tableau [1, Tmax] d'**entiers**

nbClients, nbClientsVille : **entiers**

uneVille : **chaîne**

début

{saisie et affichage du tableau de clients}

saisirTabClients (tabSociété, nbClients) ; **afficherTabClients** (tabSociété, nbClients)

{saisie de la ville des clients recherchés}

afficher(« Vous recherchez les clients pour quelle ville ? »); **saisir**(uneVille)

{construction du tableau contenant les indices des clients de uneVille}

clientsUneVille(tabSociété, tabUneVille, nbClients, nbClientsVille , uneVille)

{Affichage des infos pertinentes}

afficherClientsUneVille(tabUneVille, nbClientsVille, tabSociété)

fin

saisirTabClients (tabClients, nbVal)

{saisit nbVal clients et les enregistre dans le tableau tabClients}

paramètres (R) tabClient : TableauClients
(R) nbVal : entier

variable unClient : Client
continuer : booléen

début

nbVal ← 0 *{aucun client saisi au départ}*
saisirClient(unClient, continuer) *{on en saisit un}*
tant que continuer **et** nbVal < Tmax **faire** *{on continue …}*
nbVal ← nbVal + 1
tabClient[nbVal] $\leftarrow_{Client}$ unClient *{on met le client saisi dans le tableau}*
saisirClient(unClient, continuer) *{on en saisit un autre}*
ftq
si (non continuer)
afficher(« Saisie de », nbVal, « clients réussie. »)
sinon(« Saisie de », nbVal, « clients, les autres ont été ignorés, plus de place dans le tableau. »)
fsi

fin

Procédure clientsUneVille (tabInit, tabRésult, nbInit, nbRésult, laVille)

{remplit le tableau tabRésult avec l'indice des clients du tableau tabInit qui habitent dans une ville donnée}

paramètres (D) tabInit: TableauClients
(R) tabRésult : tableau [1, Tmax] d'**entiers**
(D) nbInit : entier
(R) nbRésult : entier
(D) laVille : chaîne

variable cpt : entier

début

nbRésult ← 0

{parcours du tableau initial, à la recherche de clients habitant laVille}

pour cpt ← 1 à nbInit **faire**

si (**tabInit[cpt].résidence.ville**) = laVille

{on stocke l'indice du client habitant dans laVille dans le tableau des indices}

alors nbRésult ← nbRésult + 1

tabRésult[nbRésult] ← cpt

fsi

fpour

fin

Procédure afficherTabClients (tabClient, nbVal)

{affiche les nbClients du tableau tabClient}

paramètres (D) tabClient : TableauClients
(D) nbVal : entier

variable cpt : entier

début

pour cpt ← 1 à nbVal **faire afficherClient**(tabClient[cpt])

fin

Procédure afficherClientsUneVille(tabIndices, nbIndices, tabClient)

{affiche les infos concernants les clients dont les indices sont stockés dans tabIndices}

paramètres (D) tabIndices : tableau [1, Tmax] d'**entiers**
(D) nbIndices : entier
(D) tabClient : TableauClients

variable cpt : entier

début

pour cpt ← 1 à nbIndices **faire afficherClient**(tabClient[tabIndices[cpt]])

fin

Procédure saisirClient (leClient, encore)

{saisit les attributs du client leClient}

paramètres (R) leClient : **Client ;** (R) encore : **booléen**

variables réponse : booléen

début

afficher ("Nom et prénom ? ") ;
saisir (leClient.personne.nom, leClient.personne.prénom)
afficher ("numéro de client ? ") ; saisir (leClient.numClient)
afficher ("adresse ? ") ; **saisirAdresse**(leClient.résidence)
afficher ("Un autre client à saisir ? ('O' ou 'N', SVP) ") ; saisir(réponse)
encore ← (réponse = 'O' ou réponse = 'o')

fin

Procédure afficherClient (unClient)

{affiche les attributs du client unClient}

paramètres (D) unClient : **Client**

début

afficher ("Nom et prénom : ")
afficher (unClient.personne.nom) ; afficher (unClient.personne.prénom)
afficher ("numéro de client : ") ; afficher (unClient.numClient)
afficher ("adresse : ") ; **afficherAdresse**(unClient.résidence)

fin

Structuration des données

La **bibliothèque** est située dans un des **bâtiments** de l'institut. Elle est organisée en **rayons**.

Chaque **rayon** a un emplacement physique dans la bibliothèque : il est repéré par un numéro de rangée et un numéro d'étagère dans la rangée. Chaque rayon porte un code informant sur le contenu des livres exposés sur ce rayon. Sur un rayon, on peut ranger jusqu'à 50 ouvrages.

Chacun de ces **ouvrages** contient une fiche rappelant la cote de l'ouvrage, le nom de l'auteur (ou les noms des auteurs), le nombre de pages, et indiquant si l'exemplaire peut sortir ou non de la bibliothèque. Cette fiche comporte aussi des indications relatives au prêt : nom de l'emprunteur, date de l'emprunt, date de retour au plus tard.

Comment choisir une structuration?

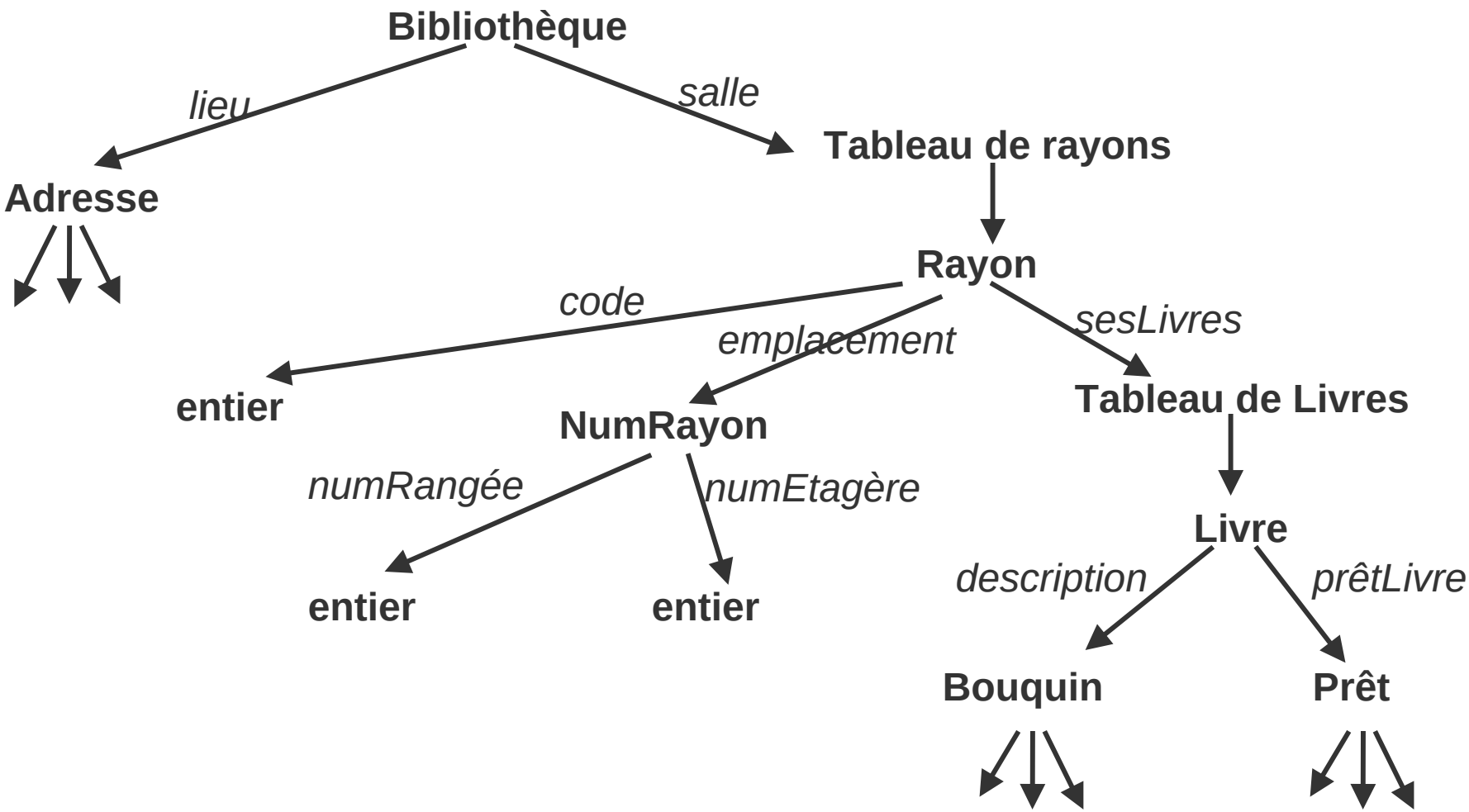
➔ **Repérer des unités de données qui ont un sens**

- regroupements d'attributs dépendants
- traitement(s) à effectuer globalement sur le regroupement d'attributs

```
type Bibliothèque = agrégat
        lieu :   Adresse
        salle:   Tableau [1, 100]
                          de Rayons
        fin
```

```
type Rayon = agrégat
        emplacement: NumRayon
        sesLivres : Tableau [1, 50]
                          de Livres
        code : entier
        fin
```

Structuration des données



type NumRayon = agrégat
numRangée, numEtagère : entiers
fin

type Livre = agrégat
description : Bouquin *{sa description : auteur, titre, édition, etc.}*
prêtLivre: Prêt *{informations relatives au prêt}*
fin

type Bouquin = agrégat
cote : entier
auteur : TabChaînes
nbPages : entier
disponible : booléen
fin

type Prêt = agrégat
emprunteur : chaîne
datedébut, datefin : chaîne
fin

Etude de cas : l'agrégat Train

Un amateur de modèles réduits de train décide de gérer sa collection en définissant un agrégat **Modèle**, et, pour chaque train qu'il constitue avec ces modèles, un agrégat **Train** :

```
type  Modèle = agrégat
        code :    entier
        nature :  chaîne
        couleur : chaîne
        marque : chaîne
        échelle : chaîne
        prix : entier
        fin
```

```
type  Train = agrégat
      numéro : entier
      composition : tableau[1, MAX]
                          de Modèles
      longueur : entier
      fin
```

constante (MAX **: entier**) ← 50

- A chaque modèle a été attribué un numéro d'ordre ***code*** ; la ***nature*** du modèle est le type de véhicule (loco électrique, wagon couvert, wagon silo, autorail, voiture corail, etc.) ; la ***couleur*** est la couleur dominante du modèle ; la ***marque*** est celle du fabricant ; ***échelle*** désigne l'échelle de réduction ; le ***prix*** est le prix d'achat.
  - *Par exemple, le modèle de code 8294 est une loco à vapeur, de couleur noire, de marque Joujou, d'échelle Rhb, de prix 125 euros.*

- Un train a un ***numéro***, une ***composition*** représentée par un tableau donnant la liste des modèles qui le composent, en allant de la tête du train vers la queue. La ***longueur*** du train est par définition le nombre de modèles qu'il contient.

1. Écrire une **fonction** booléenne qui permet de vérifier si deux modèles donnés ont même nature et même échelle.

2. Écrire une **procédure** qui calcule le prix total d'un train donné et le prix moyen des modèles le composant.

3. Écrire une **fonction** booléenne qui permet de vérifier que les tous les modèles composant un train ont une même échelle.

4. Écrire un **algorithme** qui
   - saisit puis affiche un train **leTrain** ;
   - informe l'utilisateur de la longueur de **leTrain** ;
   - permet de vérifier le bon fonctionnement des sous-algorithmes définis ci-dessus

Classes

1. **Introduction, motivation**
2. **Utilisation d'une classe**
3. **Définition d'une classe**
4. **Comparaison: agrégats, classes**
5. **Etude de cas : la classe Train**

Motivation : un exemple

- Retour à l'agrégat Client : chaque champ a besoin d'une fonction d'affichage propre
  - afficheClient , afficheChaîne, afficheAdresse
  - Mais :
    - afficheClient n'est utilisé qu'avec un paramètre de type Client
    - afficheChaîne …. paramètre de type chaîne
    - afficheAdresse …. paramètre de type Adresse

→ Il faudrait regrouper l'affichage avec le champ correspondant

- En allant plus loin : chaque champ a ses propres traitements
  - par exemple:
    - La saisie d'un client est propre aux variable de type Client
    - La comparaison de 2 adresses est propre aux variables de type Adresse

→ Il faudrait regrouper avec chaque champ les traitements qui lui sont propres

→ Notion de classe :

regroupement <u>données</u> + <u>traitements</u>

Rapprochement des types de données et des traitements correspondants

Point

abscisse, ordonnée ← *information représentée*

saisirPoint
afficherPoint
déplacerPoint } ← *traitements prévus*

Client

numClient, personne, résidence ← *information représentée*

saisirClient
afficherClient
MiseAjourAdresse } ← *traitements prévus*

→ **Au lieu de déclarer des agrégats Point et Client, on peut déclarer des classes Point et Client.**

→ **Ces classes vont associer les données et les traitements correspondants.**

Classes Point et Client

classe Point

Attributs :

abscisse, ordonnée : **réels**

Méthodes:

saisirPoint()	*{saisit un point}*
afficherPoint()	*{affiche l'abscisse et l'ordonnée d'un point}*
déplacerPoint(x,y)	*{ajoute x à l'abscisse et y à l'ordonnée d'un point}*

classe Client

Attributs :

numClient : **chaîne**
personne : **Identité**
résidence : **Adresse**

Méthodes

saisirClient()	*{saisit un Client}*
afficherClient()	*{affiche les infos relatives à un client}*
MiseAjourAdresse(nouvelleAdresse)	*{remplace l'adresse avec celle passée en paramètre}*

Les notions de Classe et Objet

- Une classe est définie par
  - La structure de ses **données** (appelées "**attributs**")

    →partie descriptive de la classe

  - Ses **méthodes** : Mprocédures, Mfonctions
    - pour manipuler ses données,
    - et coopérer avec d'autres classes

    →partie procédurale de la classe

- Un objet est une **instance** d'une classe

  Exemple :

  variables p1, p2 : **Point**

  - p1 et p2 sont déclarés des instances de la classe Point
  - ces objets ont les mêmes **attributs** que la classe et disposent de tout le "**savoir-faire**" de la classe:
    - p1 et p2 ont tous deux une abscisse et une ordonnée
    - p1 et p2 "savent" s'afficher, se déplacer…

- **Objet** : extension de la notion de variable
  - une variable est d'un certain type
  - un objet est une instance d'une certaine classe
- **Classe** : extension de la notion de type
  - avec des méthodes en plus
- Différence principale entre classes et agrégats:

➔l'encapsulation

C'est à dire la possibilité de protéger des données :

- elles sont accessibles uniquement à partir des méthodes
- elles sont "privées".

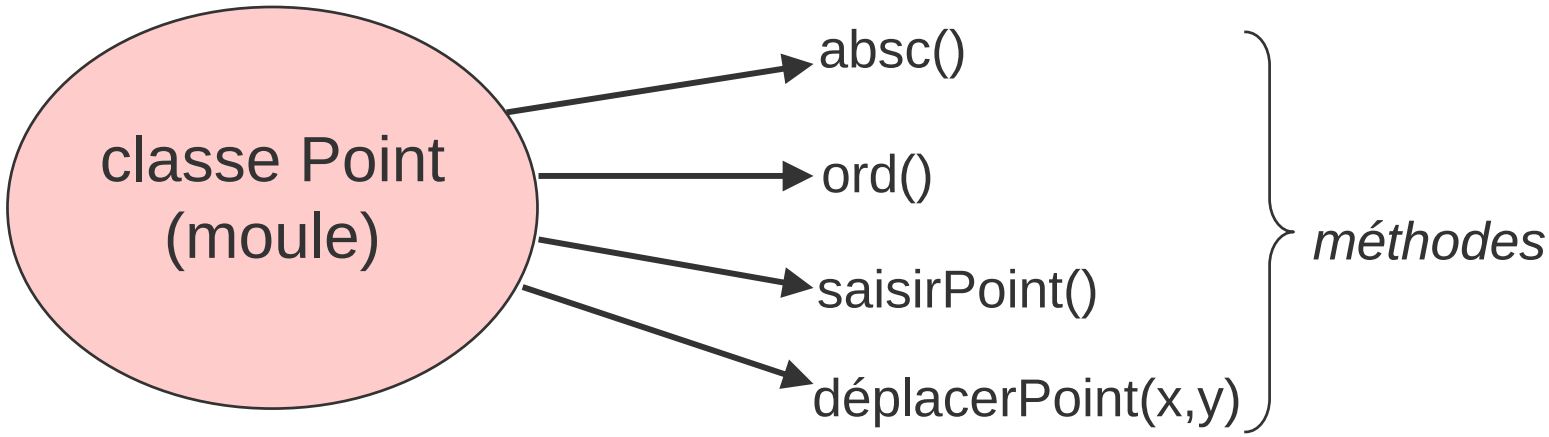
Encapsulation

Exemple : la classe Point

- Attributs : *abscisse* et *ordonnée*
- Méthodes *saisirPoint*, *afficherPoint*, *déplacerPoint*

- Soit p1 et p2 deux instances de la classe
- Un algorithme n'a pas le droit de modifier leur abscisse ou leur ordonnée dans le but de saisir ou de déplacer le point.
- Par contre,grâce à leur méthodes, p1 et p2 peuvent modifier leur propre abscisse ou ordonnée dans le but de "se" saisir, ou de se déplacer.

➔ De l'extérieur (quand on est hors classe), on ne voit, et on ne peut utiliser, que les méthodes.



- Les attributs qui servent à construire la classe sont cachés.
- De l'extérieur, on ne voit que les méthodes.
- En revanche, on peut rendre des attributs accessibles en définissant explicitement des méthodes d'accès:
    - Une méthode absc() qui retournera l'abscisse d'un point
    - Une méthode ord() qui retournera l'ordonnée d'un point
- Mais il faudra se servir des autres méthodes pour modifier ces attributs.

Récapitulation

Agrégats : types construits

(par opposition aux types de base, tels que **entier**, **réel**, **caractère**, **booléen, chaîne**)

- Toutes leurs composantes sont "publiques".
- Une variable peut être déclarée d'un type construit.

Exemple

```
type  Point = agrégat
            absc : réel
            ord : réel
            fin

variables:  p1, p2 : Point
            u, v, x,y : réels
```

```
u ← p1.absc
v ← p2. ord
afficher(p2.ord)
p1.absc ← x
p2.ord ← y
```

Classes : des catégories d'objets

- moules pour construire des objets similaires
- regroupent des données, mais aussi des fonctions et procédures (**méthodes**) utiles à la manipulation de ces données

- Toutes les données sont "privées".
  - Ceci permet la protection des données (**encapsulation**)
- Une variable peut être déclarée instance d'une classe.
  - C'est alors un **objet** de la classe

Exemple

classe Point

Attributs: absc, ord : **réels**

Méthodes: saisirPoint()

…

variables: p1, p2 : **Point**

u, v, x,y : **réels**

u ← p1.absc

v ← p2. ord

afficher(p2.ord) ← **INTERDIT !**

p1.absc ← x

p2.ord ← y

Remarques

- **Comparaison Algo/C++**
  - en algorithmique,
    - toutes les données sont privées,
    - et toutes les méthodes sont publiques
  - en C++ : on pourra déclarer
    - des données privées et des données publiques
    - des méthodes privées et des méthodes publiques
- **Suite** : il faut savoir
  - utiliser une classe
  - définir une classe

➔ Dorénavant, vous pouvez être utilisateurs de classes, ou bien programmeurs de classes

Classes

1. Introduction, motivation
2. Utilisation d'une classe
3. Définition d'une classe
4. Comparaison: agrégats, classes
5. Etude de cas : la classe Train

Un exemple

classe EnsembleEntiers

Attributs : ??? → ***cachés***

Méthodes:

Mprocédure **saisirEns()** *{saisit un ensemble d'entiers}*

Mprocédure **afficherEns()** *{affiche un ensemble}*

Mfonction **cardinal()** retourne (**entier**) *{nombre d'éléments de l'ensemble}*

Mfonction **vide()** retourne (**booléen**) *{VRAI si l'ensemble est vide, FAUX sinon}*

Mfonction **contient(elt)** retourne (**booléen**) *{VRAI si l'ensemble contient l'entier elt, FAUX sinon}*

Mfonction **ajouter(elt**) retourne (**booléen**)

{VRAI si l'entier elt a pu être ajouté à l'ensemble, FAUX sinon [manque de place]}

Mfonction **retirer(elt)** retourne (**booléen**)

{VRAI si l'entier elt a pu être retiré de l'ensemble, FAUX sinon [l'entier n'appartient pas à l'ensemble]

Algorithme Ensemble

```
variables       ensA, ensB : EnsembleEntiers
                val : entier
début
    ensA.saisirEns()
    afficher ("Voici l'ensemble saisi :")
    ensA.afficherEns()
    afficher ("Cet ensemble contient", ensA.cardinal(), "éléments.")
    si ensA.vide()
        alors   afficher ("Il est donc vide.")
        sinon   afficher ("Il n'est donc pas vide.")
    afficher ("Quel élément voulez vous rechercher ? ")
    saisir(val)
    si ensA.contient(val)
        alors   afficher ("Votre ensemble contient", val)
        sinon   afficher ("Votre ensemble ne contient pas", val)
fin
```

- **Appel des méthodes :**
  identique à l'accès aux données

Exemple **: ensA.saisirEns()**

- utilisation du point
- à sa gauche : la "**cible**", objet auquel est appliqué la méthode
- à sa droite : la méthode à appliquer

  → on applique la méthode saisirEns() à la cible ensA

  → c'est à dire : saisie de l'ensemble ensA

- Cette méthode ne retourne pas un résultat, c'est une **Mprocédure** (mais la cible est modifiée).
- Si elle retournait un résultat, ce serait une **Mfonction**.

- **Dans de nombreux langages objets, interprétation des méthodes en termes de "messages"**

Exemples :

ensA.saisirEns()

"ensemble ensA, saisis-toi "

- Le message "saisirEns" est envoyé à l'objet ensA qui réagit en affectant des valeurs à ses données privées.
- L'ensemble cible est modifié.

ensA.afficherEns()

"ensemble ensA, affiche-toi "

- Le message "afficherEns" est envoyé à l'objet ensA qui réagit en affichant ses données privées.
- L'ensemble cible n'est pas modifié.

Spécifications complètes

- Pour pouvoir utiliser les méthodes il faut connaître le nom, le type et l'attribut des paramètres.
- Exemples :

Mprocédure saisirEns()
{saisit un ensemble d'entiers}
paramètre (R) cible : Ensemble

Mfonction contient? (elt) retourne (booléen)
{retourne vrai si la cible contient elt, faux sinon}
paramètres (D) cible : Ensemble
(D) elt : réel

Mfonction ajouter(elt) retourne (booléen)
{VRAI si l'entier elt a pu être ajouté à l'ensemble, FAUX sinon [manque de place]}
paramètres (D/R) cible : Ensemble
(D) elt : réel

Remarque :
objet cible et objet paramètre

- Exemple :

  Mprocédure copier(unEns)

  {c o p i e s l e s éléments de l'ensemble unEns dans l'ensemble cible. Si l'ensemble cible contenait déjà des éléments, ceux-ci sont perdus}

  paramètres (R) cible : Ensemble
  (D) unEns : Ensemble

- Appel :

  ensA.copier(ensB)

  Les éléments de **ensB**, ensemble paramètre (qui doit être défini car paramètre (D)), sont copiés dans l'ensemble cible **ensA**. Si ensA contenait déjà des éléments, ceux-ci sont perdus.

Classes

1. Introduction, motivation
2. Utilisation d'une classe
3. **Définition d'une classe**
4. Comparaison: agrégats, classes
5. Etude de cas : la classe Train

Définition une classe

classe <NomClasse>

Attributs attribut1 : type1

...

attributK : typeK

Méthodes

Mprocédure proc1(…) {…}

...

Mprocédure procn(…) {…}

Mfonction fonc1(…) retourne(…) {…}

...

Mfonction foncn(…) retourne(…) {…}

Définition des méthodes d'une classe

paramètre implicite **cible** : l'objet auquel est envoyé la méthode.

```
Mprocédure proc (par1, … , parK)
{…}
paramètres  (D) par1 : type1
            …
            (R) parK : typeK
            (D/R) cible : NomClasse
début
   …
fin
```

paramètre **cible** pour toutes les Mprocédures et Mfonctions, (D), (R) ou (D/R).

Ex: Mprocédure afficherEns()
paramètres : (D) cible : EnsembleEntier

Un exemple

classe Point

Attributs :

abscisse, ordonnée : **réels**

Méthodes :

{Saisie et affichage}

Mprocédure saisirPoint() *{saisit un point}*

Mprocédure afficherPoint() *{affiche les coordonnées d'un point}*

{Méthodes d'accès}

Mfonction abs() retourne (réel) *{retourne l'abscisse d'un point}*

Mfonction ord() retourne (réel) *{retourne l'ordonnée d'un point}*

{Méthodes de Manipulation}

Mprocédure déplacerPoint(x,y) *{ajoute x à l'abscisse et y à l'ordonnée d'un point}*

Définition des méthodes

Mprocédure saisirPoint()
{saisit un point}
paramètre (R) cible : **Point**
début
    afficher("Donnez l'abscisse et
            l'ordonnée d'un point : ")
    saisir(cible.abscisse, cible.ordonnée)
fin

Mprocédure afficherPoint()
{affiche un point}
paramètre (D) cible : **Point**
début
    afficher("Abscisse et ordonnée du
            point : ") ;
    afficher(cible.abscisse, cible.ordonnée)
fin

Mprocédure déplacerPoint(val1, val2)
paramètre (D/R) cible : **Point** *{la cible est en donnée/résultat }*
(D) val1, val2 : **entiers**
début
    cible.abscisse ← cible.abscisse + val1
    cible.ordonnée ← cible. ordonnée + val2
fin

Remarques importantes !

- Accès aux données privées (cible.abscisse et cible.ordonnée) permis car on est à l'intérieur de la classe.

- Si hors classe : faire appel aux fonctions d'accès:

  unPoint.abs() et unPoint.ord()

  ➔ Lecture alors possible, mais modifications interdites

- Paramètre implicite cible devient explicite dans l'écriture des méthodes.

  (mais reste implicite en C++)

Fonction de manipulation de points
(hors classe)

Fonction comparePoint(pt1, pt2) retourne booléen
paramètres pt1, pt2 : **points**
{retourne vrai si les points pt1 et pt2 sont identiques}
début
    retourne (pt1.abs() = pt2.abs() **et** pt1.ord() = pt2.ord())
fin

Attention

retourne (pt1.abscisse = pt2. abscisse **ET** pt1.ordonnée = pt2. ordonnée

Interdit car accès aux données privées dans un contexte hors classe

Algorithme ManipPoint

classe Point …
variables pointX, pointY : **Points**
début
pointX.saisirPoint()
pointY.saisirPoint() *{2 objets instances de la classe point}*
afficher ("Voici les coordonnées des deux points saisis : ")
pointX.afficherPoint()
pointY.afficherPoint()
{déplacement du premier point}
pointX.déplacer(3,3)

pointX.abscisse ← pointX.abscisse + 3
pointX.ordonnée ← pointX. ordonnée + 3

afficher ("Voici le premier point après un déplacement de 3,3 : ")
pointX.afficherPoint() ;
{comparaison des deux points}
si comparePoint(pointX, pointY)
alors afficher(« Les deux points sont maintenant identiques. »)
sinon afficher(« Les deux points sont distincts. »)
fsi
fin

Classes

1. Introduction, motivation
2. Utilisation d'une classe
3. Définition d'une classe
4. **Comparaison: agrégats, classes**
5. Etude de cas : la classe Train

1. Pas d'agrégats, pas de classes

Point = couple (abscisse, ordonnée) de réels

```
Procédure saisirPoint(uneAbs, uneOrd)
paramètres (R) uneAbs, uneOrd : réels
début
        ....
        saisir(uneAbs, uneOrd)
fin
```

→ **aucune notion de point**

Appel de la procédure : **saisirPoint(**abscisse, ordonnée**)**

2. Un agrégat Point

type Point = **agrégat**
abscisse, ordonnée : réels
fin

Procédure saisirPoint(unPoint)
paramètres (R) unPoint : **Point**
début
....
saisir(unPoint.abscisse, unPoint.ordonnée)
fin

→ **notion de point**

Appel de la procédure : **saisirPoint(**lePoint**)**

3. Une classe Point

classe Point

Attributs : abscisse, ordonnée : réels

Méthodes: MProcédure saisirPoint()

…

MProcédure saisirPoint()

paramètre (R) cible: **Point**

début

….

saisir(cible.abscisse, cible.ordonnée)

fin

→ notion de point, avec traitements associés

A

Récapitulation

Procédure saisirPoint(uneAbs, uneOrd)

pas de variable point

Appel : **saisirPoint(**abscisse, ordonnée**)**

Procédure saisirPoint(unPoint)

un **agrégat** point

Appel : **saisirPoint(**lePoint**)**

MProcédure saisirPoint()

un **objet** point (instance de la classe Point)

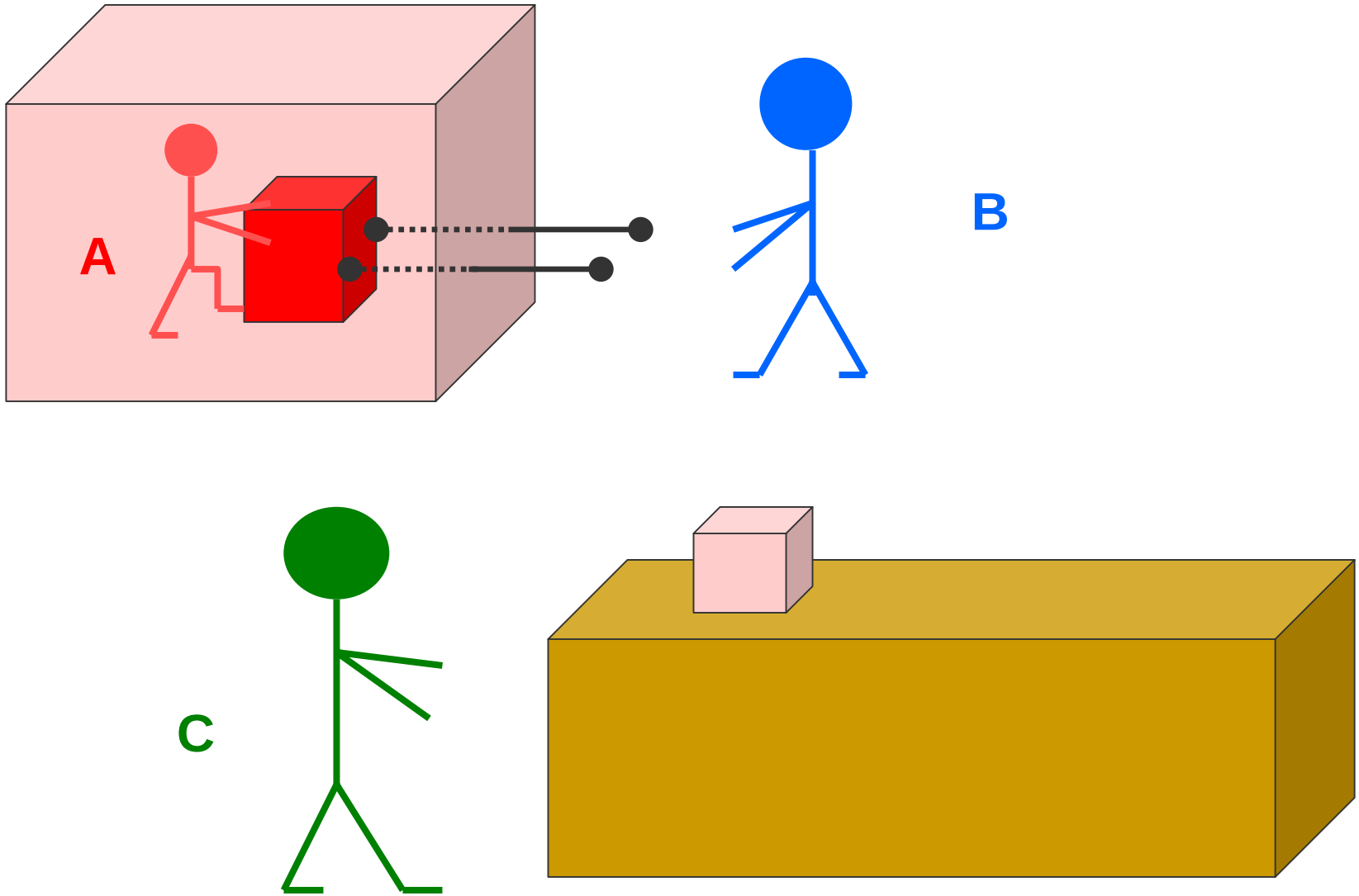
Appel : lePoint.**saisirPoint()**

Retour sur l'encapsulation

- Les détails de la construction de la classe peuvent être utilement dissimulés à l'utilisateur:
  - Celui-ci n'a pas besoin de savoir comment est construite la classe, mais seulement de savoir comment s'en servir.
  - Il n'a pas à se soucier de détails d'implémentation, mais seulement de ce que lui permet la classe.
  - Il ne peut pas faire de manœuvre imprévue.
  - Il peut raisonner à un niveau conceptuel plutôt qu'en termes d'implémentation.
- Remarque : quand on parle d'***utilisateur*** on pense aux programmeurs qui vont utiliser la classe.

Les 3 "acteurs" de la programmation objet

- Programmeur concepteur de la classe (A)
- Programmeur utilisateur de la classe (B)
  - Il n'a pas besoin de savoir comment la classe est conçue
  - Il n'a besoin que des spécification des méthodes (partie publique de la classe)
- Utilisateur de l'application (C)
  - Il n'a pas besoin de savoir comment son application est programmée


A
B
C

Classes

1. Introduction, motivation
2. Utilisation d'une classe
3. Définition d'une classe
4. Comparaison: agrégats, classes
5. **Etude de cas : la classe Train**

On reprend les agrégats **Modèle** et **Train** étudiés dans le chapitre précédent.

```
type  Modèle = agrégat
        code :    entier
        nature :  chaîne
        couleur : chaîne
        marque : chaîne
        échelle : chaîne
        prix : entier
        fin
```

```
type  Train = agrégat
      numéro : entier
      composition : tableau[1, MAX]
                    de Modèles
      longueur : entier
      fin
```

On décide d'équiper les Trains de procédures et de fonctions.
Pour cela, on remplace l'agrégat **Train** par une classe **Train** :

classe Train

attributs

numéro : **entier**

composition : **tableau**[1, MAX] de **Modèles**

longueur : **entier**

méthodes

Mprocédure saisirTrain() *{saisie d'un train}*

Mprocédure afficherTrain() *{affichage d'un train}*

... [*autres méthodes non décrites*]

1. Écrire une **Mfonction** ***nbVCC(uneCouleur)*** qui donne, en
«voiture corail» ou «voiture couchette» et de couleur ***couleur***.

2. Écrire la **Mprocédure** ***ajouter(unModèle, position, ok)*** qui ajoute le modèle ***unModèle*** à la suite de celui qui se trouve en position ***position*** (un entier entre 1 et la longueur du train), ***à condition que*** l'échelle du modèle soit celle du train (supposé ne comporter que des voitures de même échelle), et que la longueur totale ne dépasse pas MAX. Le booléen ***ok*** signale si l'ajout a pu se faire ou non (on supposera disposer d'une affectation pour les modèles, qu'on notera ←)

3. Écrire un **algorithme** qui

- saisit puis affiche un train **leTrain** ;
- informe l'utilisateur de la longueur de **leTrain** ;
- appelle la méthode ***ajouterModèle***, après avoir demandé à l'utilisateur le modèle à ajouter, et la position où doit se faire l'ajout. On supposera disposer d'une fonction ***saisirM(unModèle)*** pour la saisie d'un modèle.
- si l'ajout a pu se faire, l'algorithme affichera à nouveau **leTrain**, sinon, il signalera l'échec à l'utilisateur

4. Soit le type suivant

Type Entrepôt = agrégat

tabTrain : **tableau** [1, MAX] de **Trains**

nbTrains : **entier**

fin

Quelle est l'instruction nécessaire pour

- afficher les trains qui sont actuellement à l'entrepôt ?
- afficher juste le numéro des trains qui sont à l'entrepôt ?

5. Ecrire une fonction qui permet de déterminer si les locomotives de deux trains donnés sont de même marque.

fin Volume 3