Plan du Cours

- Importance des SDD
- Type Abstrait : Présentation et Exemples
- Structures de Données Abstraites
- **Illustration : Type Abstrait pour la Recherche d'Information**
- Implémentation des Types Abstraits
  - Types Structurés
  - Programmation Orientée Objets

Type Abstrait pour la Recherche d'Information (1/3)

Annuaire

- Trouver le numéro de téléphone de Fred
- Supprimer l'adresse et le numéro de téléphone d'Alice
- Ajouter l'adresse et le numéro de téléphone d'Adam

Utilisation du nom comme une ***clé de recherche***

Nom	Adresse	Numéro
Ahmed	Villeurbanne	0472221537
Fred	Part Dieu	0472123478
Nora	Perrache	0472139820
Jean	Vaulx	0472093456
Alice	Grenoble	0476290763
Adam	Croix Rousse	0472089562

Type Abstrait pour la Recherche d'Information (2/3)

TypeAbs TEntrée_ann

Domaine

chaîne nom, adresse, numéro

Opérations

...

FTypeAbs TEntrée_ann

Type Abstrait pour la Recherche d'Information (2/3)

TypeAbs TAnnuaire

Domaine

TEntrée_ann tab_ann[100]

Opérations

Création_TAnnuaire : Ø → TAnnuaire

Insertion : TAnnuaire x TEntrée_ann → TAnnuaire

Suppression : TAnnuaire x chaîne → TAnnuaire

Recherche : TAnnuaire x chaîne → chaîne

FTypeAbs TAnnuaire

Plan du Cours

- Importance des SDD
- Type Abstrait : Présentation et Exemples
- Structures de Données Abstraites
- Illustration : Type Abstrait pour la Recherche d'Information
- **Implémentation des Types Abstraits**
  - **Types Structurés**
  - Programmation Orientée Objets

Implémentation des Types Abstraits : Domaine

- On a vu au premier semestre les types simples
  - entier, booléen, caractère, chaîne de caractères
- Comment gérer ?
  - TDroite
    - x, y, angle
    - a, b, c (ax+by+c=0 avec a ou b !=0)
  - TVecteur
    - x, y
  - TEntrée_ann
  - TEtudiant
    - nom, prénom, âge, classement
- Avec des **types structurés** ou **Struct**
  - Encapsulation de différentes informations

Types Structurés ou Struct

- Date = Struct (jour = int,
  mois = int,
  an = int)

- Si d est une variable de type **Date**
  - Accès aux informations de d avec l'opérateur .
    - ➔ jour_d = d.jour
    - ➔ mois_d = d.mois

- Types Droite, Vecteur, Etudiant, Entree_ann ?

Plan du Cours

- Importance des SDD
- Type Abstrait : Présentation et Exemples
- Structures de Données Abstraites
- Illustration : Type Abstrait pour la Recherche d'Information
- **Implémentation des Types Abstraits**
  - Types Structurés
  - **Programmation Orientée Objet**

Implémentation des Types Abstraits : Domaine & Opérations

- Certains langages de programmation fournissent des constructions syntaxiques spécifiques pour les Types Abstraits
- Programmation Orientée Objet : concept de Classe d'Objets

Programmation Orientée Objet

- Séparation interface vs. implémentation
- Données accessibles seulement au travers de l'interface définie (ou des opérations proposées)
- Nous en présenterons ici deux notions :
  - Encapsulation
  - Généricité

Objet

- C’est une « partie de programme » composée d’**attributs** et de **méthodes**.
- **Attributs**
  - Variables locales de l'objet : valeurs caractérisant l'objet
- **Méthodes**
  - Fonctions ou procédures que possède l'objet
  - Effectuent un traitement sur **les attributs de cet objet**

Exemple d'objet : l'objet Personne

- **Attributs**
  - nom, adresse et date de naissance d'une personne

- **Méthodes** :
  - Initialisation des attributs à partir de valeurs données en paramètres
  - Affichage des attributs de la personne à l'écran
  - Changement de son adresse
  - Comparaison du nom de la personne avec un nom donné

Classe d'objets

- **Type d'objets** ayant tous :
  - les mêmes attributs
  - les mêmes méthodes.
- Deux objets de même type :
  - différent seulement par les valeurs de leurs attributs,
  - ont les mêmes méthodes.
- Une classe a un identificateur (nom) et s'utilise comme un type ordinaire.

- **Exemple de classe d'objets**
  - Classe des objets représentant des personnes.
  - Deux personnes ont mêmes attributs et mêmes méthodes,
  - Elles diffèrent seulement par la valeur des attributs *nom* et/ou *adresse*

Instances et Encapsulation

- L'utilisateur de l'objet ne devrait pas avoir a connaître sa structure interne, mais seulement la spécification de son comportement, défini par ses méthodes

- Les instances de classes contiennent également des données
  - Accessibles uniquement aux méthodes de l'objet, c'est pourquoi on les nomme attributs privés
  - En Python, on les caractérise en commençant leur nom par '_'

Envoi de message à un objet

- Syntaxe pointée pour accéder aux méthodes
- `p.changeAdresse("15 rue des fleurs")`

  est un **envoi d'un message à l'objet p**
- Ce message demande à l'objet p d'utiliser sa méthode `changeAdresse` pour **modifier son adresse**
- Un envoi de message à un objet est composé :
  - de **l'objet** auquel est envoyé le message (**`p`**)
  - d'un point (•)
  - du **nom de la méthode** à utiliser (`changeAdresse`)
  - et de **valeurs pour les paramètres** `("15 rue des fleurs")`

Paramètre implicite d'une méthode

- `p.changeAdresse("15 rue des fleurs")`

  range la chaîne "15 rue des fleurs" dans l'attribut adresse de l'objet **p**.
- `changeAdresse` a donc réellement **2 paramètres** :
  - l'objet p auquel est envoyé le message :
    - paramètre particulier
    - spécifié avec une notation particulière (**p.changeAdresse**)
  - "15 rue des fleurs", paramètre classique.
- L'objet qui reçoit le message n'apparaît pas dans la liste des paramètres de la méthode : c'est un **paramètre implicite** de la méthode

Déclaration d'une classe (Python)

- Contenue dans un bloc avec 'class', suivi du nom de la classe puis ':'
- Bloc subdivisé en deux parties : Interface et Implémentation
- Interface
  - Contient la spécification des méthodes
  - C'est la partie publique de la classe (i.e. ce qu'on a besoin de connaître pour écrire un programme utilisant la classe)
- Implémentation
  - Contient la déclaration des attributs privés et l'implémentation des méthodes (i.e. nécessaire à l'ordinateur pour l'exécution de ce programme)
  - Peut également contenir des déclarations de types structurés
  - Peut également contenir la déclaration de procédures et fonctions additionnelles dites *méthodes privées* (elles commenceront par '_')

Accès à l'objet qui reçoit le message - **self**

- La déclaration d'une méthode est similaire à la déclaration d'une procédure ou fonction comme vu au S1, à l'exception du paramètre implicite représentant l'objet dont on appelle la méthode : **self**
- **self** est toujours le premier paramètre de la méthode
- Il est indiqué sans type
- Il doit apparaître comme paramètre d'entrée, de sortie ou d'entrée-sortie, afin d'indiquer le rôle de la méthode sur l'objet auquel on l'applique :
  - Consultation
  - Modification (
  - Initialisation (
- Dans une méthode de la classe Personne, les attributs de l'objet qui reçoit le message sont donc désignés par :

  `self._nom` et `self._adresse`

Méthodes d'initialisation

- En Python, l'initialiseur est unique, et toujours nommé __init__
- Il peut avoir des paramètres d'entrée, mais self est le seul paramètre de sortie
- Il a la responsabilité d'initialiser les valeurs des attributs privés de l'objet
- Il est appelé implicitement à la création de l'objet
- Pour créer un objet, on écrit : **p = Personne()**

Classe FeuTricolore : Interface

```
class FeuTricolore:

    def __init__(self):
      """
      :sortie self:
      :post-cond: le feu est au vert
      """
    def etat(self):
      """
      :entrée self:
      :sortie e: str
      :post-cond: e est l'état courant du feu, parmi "vert", "orange" ou "rouge"
      """
    def change_etat(self):
      """
      :entrée-sortie self:
      :post-cond: passe à l'état suivant, selon le cycle "vert" -> "orange" -> "rouge" -> "vert"
      """
```

Classe FeuTricolore : Implémentation

```
def __init__(self):
  #### attribut privé
  self._num_etat = 0
  # parmi 0 (vert), 1 (orange), 2 (rouge)

def etat(self):
  if self._num_etat == 0:
     e = "vert"
  elif self._num_etat == 1:
     e = "orange"
  else:
     e = "rouge"
  return e
```

```
def change_etat(self):
  self._num_etat = self._num_etat_suivant()

#### méthode privée
def _num_etat_suivant(self):
  """
  :entrée self:
  :sortie s: int
  :post-cond: s est le num. du prochain état
  """
  s = (self._num_etat + 1) % 3
  return s
```

Exercice

- Ecrire la spécification et l'implémentation d'une classe **`Point`** permettant de manipuler un point du plan. Prévoir :
  - Un constructeur permettant d'initialiser les coordonnées du point
  - Des méthodes permettant de consulter les coordonnées (fonctions d'accès)
  - Une méthode permettant de déplacer le point par une translation (dx,dy)

Classe Point : Interface

```
def __init__(self):
  """
  :sortie self:
  :post-cond: coordonnées du point intialisées à 0
  """

def getX(self):
  """
  :entrée self:
  :sortie coordx: float
  :post-cond: consultation de la valeur de x
  """

def getY(self):
  """
  :entrée self:
  :sortie coordy: float
  :post-cond: consultation de la valeur de y
  """
```

```
def deplace(self, dx, dy):
  """
  :entrée-sortie self:
  :entrée dx: float
  :entrée dy: float
  :post-cond: déplacement du point
  courant de (dx, dy)
  """

def affiche(self):
  """
  :entrée self:
  :post-cond: affichage des
  coordonnées du point courant
  """
```

Classe Point : Implémentation

```
def __init__(self):
  self._x = 0
  self._y = 0

def getX(self):
  coordx = self._x
  return coordx

def getY(self):
  coordy = self._y
  return coordy
```

```
def deplace(self, dx, dy):
  self._x = self._x + dx
  self._y = self._y + dy

def affiche(self):
  print "les coordonnées sont %f et
  %f " % (self.getX(),self.getY())
```

Classe Vecteur : Interface

```
def __init__(self):
  """
  :sortie self:
  :post-cond: coordonnées du vecteur intialisées à 0
  """
def base(self, v):
  """
  :entrée-sortie self:
  :entrée-sortie v: Vecteur
  :post-cond: v forme une base de l'espace vectoriel
      avec le vecteur courant (vecteurs non nuls et
      colinéaires)
  """
def egaux(self, v):
  """
  :entrée self:
  :entrée v: Vecteur
  :sortie eq: bool
  :post-cond: eq est True si v et le vecteur courant
      sont égaux
  """
def nul(self):
  """
  :entrée self:
  :sortie n: bool
  :post-cond: n est True si le vecteur courant est nul
  """
def colineaires(self, v):
  """
  :entrée self:
  :entrée v: Vecteur
  :sortie co: bool
  :post-cond: co est True si v et le vecteur courant sont colinéaires
  """
def addition(self, v):
  """
  :entrée v: Vecteur
  :entrée-sortie self:
  :post-cond: v est additionné au vecteur courant
  """
def multiplication(self, k):
  """
  :entrée k: float
  :entrée-sortie self:
  :post-cond: le vecteur courant est multiplié par k
  """
```