III. Types de données abstraits

Éléments de conception d'algorithmes

III.1 Notion de type abstrait

Très rapidement lorsqu'on écrit des programmes, et a fortiori des algorithmes, on a besoin de regrouper dans une variable différentes valeurs de différents types représentant une seule et même entité.

On pourra par exemple de représenter une `Personne` par son nom (`text`), son prénom (`text`), et son âge (`Int`).

Utiliser plusieurs variables pour chaque `Personne` représentée s'avère très vite limité et on a besoin de pouvoir utiliser une seule variable pour représenter l'entité `Personne`.

En outre, pour certaines valeurs, un type scalaire n'existe pas. Par exemple, comment représenter une date ? Un simple entier ou flottant ne suffit pas ! On a également besoin d'un type date.

III.1.1 Type abstrait : définition

Un *type abstrait* représente une entité proposant un ensemble d'opérations et de propriétés la représentant.

La manière dont sont codées les valeurs n'ont pas à être connue par les utilisateurs de ce type.

Même si les types de base sont aussi des types abstraits, on aura besoin de définir des types complexes et ceux-ci devront également être abstraits, c'est à dire que l'algorithme qui les utilise n'a pas à connaître son codage et ne doit travailler qu'avec des valeurs et les opérations définies pour ce type.

La façon dont sont organisées les différentes données du type s’appelle la *structure de données* ; celle-ci n'a pas à être connue par l'algorithme.

Définition : *Types abstraits*

On appelle *type abstrait* une spécification d'un ensemble de données, de leurs propriétés et de *l'ensemble des opérations* que l'on peut effectuer sur ces données.

Définition : *Structure de données*

On appelle *structure de données* un structure logique *organisant* un ensemble d'informations (les données), (les données) à stocker que l'on déduit des fonctions de la spécification.

Conclusion sur la notion de type abstrait

- Un *type abstrait* représente une famille d'objets apparentés par les opérations définies sur ses objets. Les mêmes opérations peuvent s'appliquer à chacun des objets.
- Les opérations sont choisies *selon les besoins* d'une application.
- D'une application à l'autre, les opérations peuvent être très différentes, même si les types portent le même nom : *le nom ne définit pas un type* !
- On appellera *instance*, la concrétisation physique (donnée) d'un objet de la famille.
- La façon dont sont organisées les données définissant l'objet s'appelle la *structure de données*

III.1.2 Niveaux de description

- La *Spécification Fonctionnelle* : précise l'ensemble des opérations permises sur le type de données et les caractéristiques de ces opérations

La spécification fonctionnelle définit le type abstrait

- La *Description logique* fournit une décomposition des données en objets plus élémentaires (structure logique du type abstrait) que l'on code par des propriétés ; elle fournit également les algorithmes basés sur cette décomposition logique du type abstrait, qui réalisent les opérations de la spécification fonctionnelle

La description logique définit la structure de données et les algorithmes des fonctions

- La *Représentation physique* : code la description logique en machine ; la description logique est implémentée grâce aux types scalaires, aux structures de bases offertes par le langage et aux types abstraits déjà définis ; les algorithmes sont programmés à l'aide des fonctions du langage ; La représentation physique s'attache à trouver le moyen de coder dans le langage le respect des caractéristiques de la spécification fonctionnelle et la logique de la description logique.

La représentation physique définit le programme codant le type abstrait

III.2 Spécification fonctionnelle

Définir un *type abstrait de données*

→ définir précisément de quelles opérations ou fonctions auront besoin les utilisateurs

Définir précisément ces fonctions nécessite de préciser pour chacune d'elle l'ensemble de leurs caractéristiques.

Ces caractéristiques ont un triple rôle, elles :

- participent à la définition des opérations,
- précisent leurs sémantiques,
- sont des éléments pouvant être utilisés pour la preuve d'algorithmes.

III.2 Spécification fonctionnelle

Définir un *type abstrait de données*

→ définir précisément de quelles opérations ou fonctions auront besoin les utilisateurs

Définition : *Spécification fonctionnelle*

La *spécification fonctionnelle* d'un *type abstrait* regroupe :

- la définition précise des opérations ou fonctions dont auront besoin les utilisateurs,
- l'ensemble des caractéristiques des fonctions

Exemple : voiture à vendre

Soit une voiture à vendre, le vendeur doit pouvoir calculer son prix de vente, savoir si une remise est possible, de combien et bien sûr annoncer son prix de vente avec ou sans remise.

Exemple : voiture à vendre

Soit une voiture à vendre, le vendeur doit pouvoir calculer son prix de vente, savoir si une remise est possible, de combien et bien sûr annoncer son prix de vente avec ou sans remise.

Les fonctionnalités dont à besoin le vendeur sont :

- quel est le modèle de la voiture
- quel est son prix de vente ?
- une remise est-t-elle possible ?
- quel est montant max de la remise ?
- quel est le prix remisé ?

Exemple : voiture à vendre

Soit une voiture à vendre, le vendeur doit pouvoir calculer son prix de vente, savoir si une remise est possible, de combien et bien sûr annoncer son prix de vente avec ou sans remise.

nom:	`Voiture→Text`	*// nom de la voiture*
prix:	`Voiture→Float`	*// prix de la voiture (non remisé)*
a_remise:	`Voiture→Bool`	*// remise possible ?*
remise_max:	`Voiture→Float`	*// remise maximum (en %)*
remise_max:	`Voiture x Float→Float`	*// modifier la remise max*
prix_remise:	`Voiture x Float→Float`	*// prix de la voiture (avec remise)*

Exemple : voiture à vendre

Soit une voiture à vendre, le vendeur doit pouvoir calculer son prix de vente, savoir si une remise est possible, de combien et bien sûr annoncer son prix de vente avec ou sans remise.

nom:	`Voiture→Text`	*(1) nom(v)⇒prix(v)>0*
prix:	`Voiture→Float`	*(2) prix(v)>0*
a_remise:	`Voiture→Bool`	*(3) remise(v)⇔remise_max(v)>0*
remise_max:	`Voiture→Float`	*(4) 0 ≤ remise_max(v) < 1*
remise_max:	`Voiture x Float→Float`	*(4) remise_max(remise_max(v,r))==r*
prix_remise:	`Voiture x Float→Float`	*(5) prix_remise(v,r)* *(6) prix_remise(v,r) ≤ prix(v)-prix(v)*remise_max(v)* *(7) prix_remise(v,r)⇒r<remise_max(v)*

Pour un types abstrait de données T, elles sont de la forme :

$$f : X_1 \times \ldots \times X_n \rightarrow Y_1 \times \ldots \times Y_m$$

Pour un types abstrait de données T, elles sont de la forme :

$$f : X_1 \times \ldots \times X_n \rightarrow Y_1 \times \ldots \times Y_m$$

- les *fonctions d'accès* qui sont celles où T apparaît seulement à gauche de la flèche : elles permettent d'accéder aux valeurs des informations associées au type T; deux types :
  - les *propriétés* : seul le type apparait à gauche
  - les *requêtes* : le type + d'autres paramètres sont à gauche

Pour un types abstrait de données T, elles sont de la forme :

$$f : X_1 \times \ldots \times X_n \rightarrow Y_1 \times \ldots \times Y_m$$

- les *fonctions d'accès* qui sont celles où T apparaît seulement à gauche de la flèche : elles permettent d'accéder aux valeurs des informations associées au type T; deux types :
  - les *propriétés* : seul le type apparait à gauche
  - les *requêtes* : le type + d'autres paramètres sont à gauche
- les *fonctions de modification* qui sont celle où T apparaît à droite et à gauche de la flèche : elles permettent de modifier des objets de type T ;

Pour un types abstrait de données T, elles sont de la forme :

$$f : X_1 \times \ldots \times X_n \rightarrow Y_1 \times \ldots \times Y_m$$

- les *fonctions d'accès* qui sont celles où T apparaît seulement à gauche de la flèche : elles permettent d'accéder aux valeurs des informations associées au type T; deux types :
  - les *propriétés* : seul le type apparait à gauche
  - les *requêtes* : le type + d'autres paramètres sont à gauche
- les *fonctions de modification* qui sont celle où T apparaît à droite et à gauche de la flèche : elles permettent de modifier des objets de type T ;
- les *fonctions de création* qui sont celles où T apparaît seulement à droite de la flèche : elles permettent de créer des objets de types T.

nom:	`Voiture→Text`	*(1) nom(v)⇒prix(v)>0*
prix:	`Voiture→Float`	*(2) prix(v)>0*
a_remise:	`Voiture→Bool`	*(3) remise(v)⇔remise_max(v)>0*
remise_max:	`Voiture→Float`	*(4) 0 ≤ remise_max(v) < 1*
remise_max:	`Voiture x Float→Float`	*(4) remise_max(remise_max(v,r))==r*
prix_remise:	`Voiture x Float→Float`	*(5) prix_remise(v,r)* *(6) prix_remise(v,r) ≤ prix(v)-prix(v)*remise_max(v)* *(7) prix_remise(v,r)⇒r<remise_max(v)*

fonctions d'accès

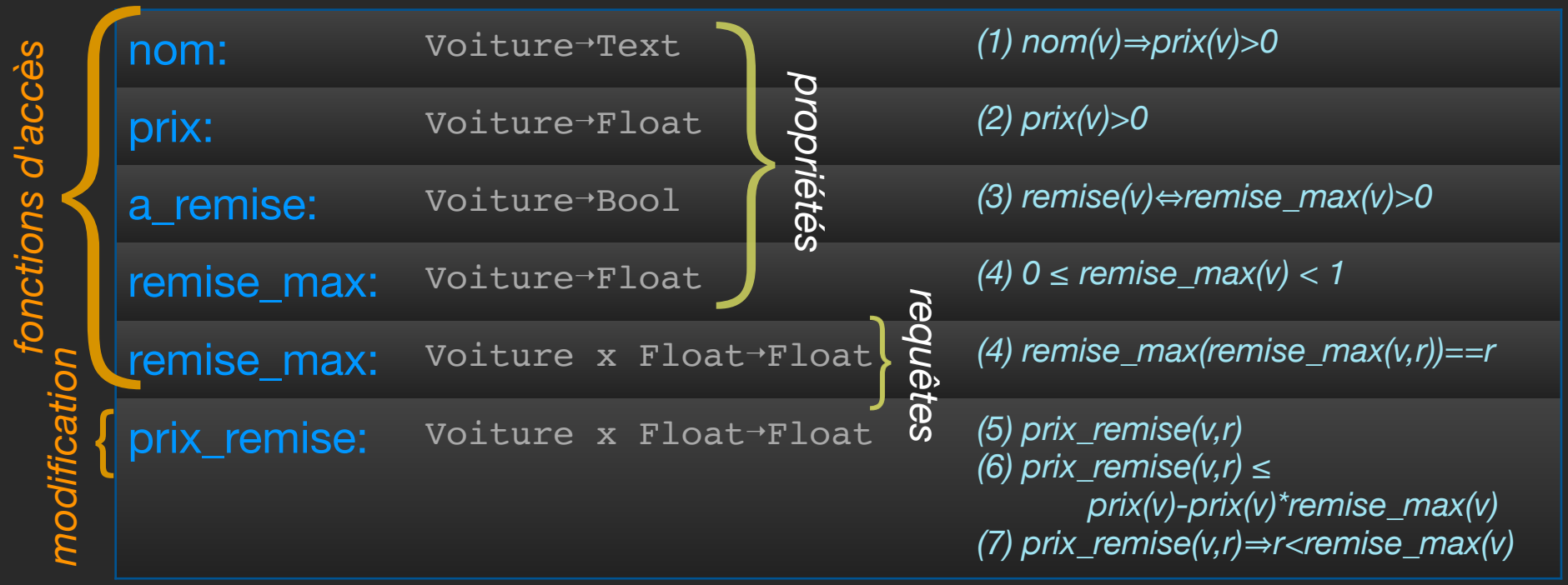
nom:	`Voiture→Text`	*(1) nom(v)⇒prix(v)>0*
prix:	`Voiture→Float`	*(2) prix(v)>0*
a_remise:	`Voiture→Bool`	*(3) remise(v)⇔remise_max(v)>0*
remise_max:	`Voiture→Float`	*(4) 0 ≤ remise_max(v) < 1*
remise_max:	`Voiture x Float→Float`	*(4) remise_max(remise_max(v,r))==r*
prix_remise:	`Voiture x Float→Float`	*(5) prix_remise(v,r)* *(6) prix_remise(v,r) ≤ prix(v)-prix(v)*remise_max(v)* *(7) prix_remise(v,r)⇒r<remise_max(v)*

fonctions d'accès		
nom:	`Voiture→Text`	*(1) nom(v)⇒prix(v)>0*
prix:	`Voiture→Float`	*(2) prix(v)>0*
a_remise:	`Voiture→Bool`	*(3) remise(v)⇔remise_max(v)>0*
remise_max:	`Voiture→Float`	*(4) 0 ≤ remise_max(v) < 1*
remise_max:	`Voiture x Float→Float`	*(4) remise_max(remise_max(v,r))==r*
modification		
prix_remise:	`Voiture x Float→Float`	*(5) prix_remise(v,r)* *(6) prix_remise(v,r) ≤ prix(v)-prix(v)*remise_max(v)* *(7) prix_remise(v,r)⇒r<remise_max(v)*

fonctions d'accès

modification

nom:	`Voiture→Text`	*propriétés*	*(1) nom(v)⇒prix(v)>0*
prix:	`Voiture→Float`		*(2) prix(v)>0*
a_remise:	`Voiture→Bool`		*(3) remise(v)⇔remise_max(v)>0*
remise_max:	`Voiture→Float`		*(4) 0 ≤ remise_max(v) < 1*
remise_max:	`Voiture x Float→Float`		*(4) remise_max(remise_max(v,r))==r*
prix_remise:	`Voiture x Float→Float`		*(5) prix_remise(v,r)* *(6) prix_remise(v,r) ≤ prix(v)-prix(v)*remise_max(v)* *(7) prix_remise(v,r)⇒r<remise_max(v)*


fonctions d'accès
modification
nom:
Voiture→Text
prix:
Voiture→Float
a_remise:
Voiture→Bool
remise_max:
Voiture→Float
remise_max:
Voiture x Float→Float
prix_remise:
Voiture x Float→Float
propriétés
requêtes
(1) nom(v)⇒prix(v)>0
(2) prix(v)>0
(3) remise(v)⇔remise_max(v)>0
(4) 0 ≤ remise_max(v) < 1
(4) remise_max(remise_max(v,r))==r
(5) prix_remise(v,r)
(6) prix_remise(v,r) ≤
prix(v)-prix(v)*remise_max(v)
(7) prix_remise(v,r)⇒r<remise_max(v)

III.3 Types abstraits en `Swift` : `Protocol`

Les *protocoles* `Swift` fournissent un moyen élégant pour définir un type abstrait.

Un type concret implémentant un type abstrait définit par un protocole devra être conforme aux propriétés, fonctions de requête et de modification définies par le protocole.

La syntaxe des *protocoles* pour définir un type abstrait est la suivante :

```
protocol nom_type{

  // définition des propriétés et fonctions

}
```

Définition des fonctions de création d'un type abstrait

Pour définir des fonctions de création d'une instance d'un type abstrait, il faut donner la signature de(s) fonction(s) `init()`

À la création d'une instance, est appelé la fonction `init()` correspondante.

```
protocol NomProtocol {
  init()
  init(p1: Type1, p2: Type2)
}
// on suppose que NomType est conforme au protocole NomProtocol

// utilisation des fonctions de création:
var v = NomType()
var x = NomType(p1: v1, p2: v2)
```

Définition des propriétés d'un type abstrait

Un protocole ne précise pas si la *propriété* doit être une *propriété stockée* ou une *propriété calculée*. Il spécifie seulement le nom et le type de la propriété.

Le protocole précise également si la propriété est en lecture seule ou si elle peut être également modifiée.

```
protocol NomProtocol {
  var propriétéModifiable: Type { get set }
  var propriétéLectureSeule: Type { get }
}
// on suppose que NomType est conforme au protocole NomProtocol
// utilisation :
var v = NomType()
v.propriétéModifiable = valeur
var x = v.propriétéModifiable
var y = v.propriétéLectureSeule
```

Définition des fonctions de requête d'un type abstrait

Pour définir des fonctions de *requête* d'un type abstrait, il suffit de donner la *signature* de la fonction

On appelle *signature* d'une fonction, son nom, son type et le type des paramètres.

```
protocol NomProtocol {
  func requete() -> Type
  func requeteParam(p1: Type1, p2: Type2) -> Type3
}
// on suppose que NomType est conforme au protocole NomProtocol
// utilisation :
var v = NomType()
var x = v.requete()
var y = v.requeteParam(p1: val1, p2: val2)
```

Définition des fonctions de modification d'un type abstrait

Pour définir des fonctions de *modification* d'un type abstrait, il faut donner la signature de la fonction mais aussi indiquer que celle-ci est *susceptible de modifier* l'instance

Cette indication se faire par le mot clef *mutating*.

```
protocol NomProtocol {
  mutating func modif(p1: Type1, p2: Type2) -> Type3
}
// on suppose que NomType est conforme au protocole NomProtocol
// utilisation :
var v = NomType()
var y = v.modif(p1: val1, p2: val2)
```

Exercice : Bataille navale - version 1

L'objectif est de programmer une bataille navale.

- On nous demande d'écrire un programme qui permet de jouer à la bataille navale suivant les règles suivantes :
- 5 bateaux de tailles 1 à 4, dont deux de taille 3, sont stockés à des positions prédéterminées et immuables ;
- chaque bateau occupe des cases consécutives sur la même ligne ou même colonne d'une grille 20x20
- les lignes et les colonnes sont numérotées de 0 à 19 et une position est indiquée par les coordonnées (col, lig)
- le programme doit demander à l'utilisateur une position où tirer tant que tous les bateaux ne sont pas coulés

- à chaque proposition, l'algorithme répond :
  - « *touché* » si la position est occupé par un bateau et qu'il n'a pas été encore touché à cette position
  - « *coulé* » si la position est occupé par un bateau et que c'était la dernière position du bateau non encore touchée
  - « *en vue* » si la position n'est pas occupée par un bateau ou qu'elle correspond à une position déjà touchée, et que sur la ligne ou la colonne (ou les deux) se trouve une position non touchée occupée par un bateau
  - « *à l'eau* » dans les autres cas
- le programme s'arrête quand tous les bateaux ont été coulés

Exercice

Définissez les types nécessaires à la conception de ce programme