



U.F.R SCIENCES ET TECHNIQUES

Département d'Informatique

B.P. 1155

64013 PAU CEDEX

Téléphone secrétariat : 05.59.40.79.64

Télécopie : 05.59.40.76.54

TYPE ABSTRAIT DE DONNEES

Partie I

- I- Type et structure de données
- II- Notion de type abstrait
- III- Spécification d'un type abstrait
- IV- Réutilisation et hiérarchie des types abstraits
- V- Validation de la spécification d'un type abstrait
- VI- Vérification de l'implémentation d'un type abstrait

I-Type et structure de données

«**Algorithme + Structures de données = Programme**»

C'est le titre d'un livre publié par N. **Wirth**, en 1976 à Zurich.

Ce titre pose le postulat selon lequel :

- dans un **programme**,
- un **algorithme** n'est rien

s'il n'est pas accompagné de **structures de données** appropriées.

Ces **structures de données** appropriées sont destinées à :

- **stocker** les données
- et **y accéder** pour les **manipuler**

Exemple simple d'illustration

Soit l'opération bancaire qui permet d'examiner dans un fichier d'entrée nommé **inFile** le solde de tous les comptes clients

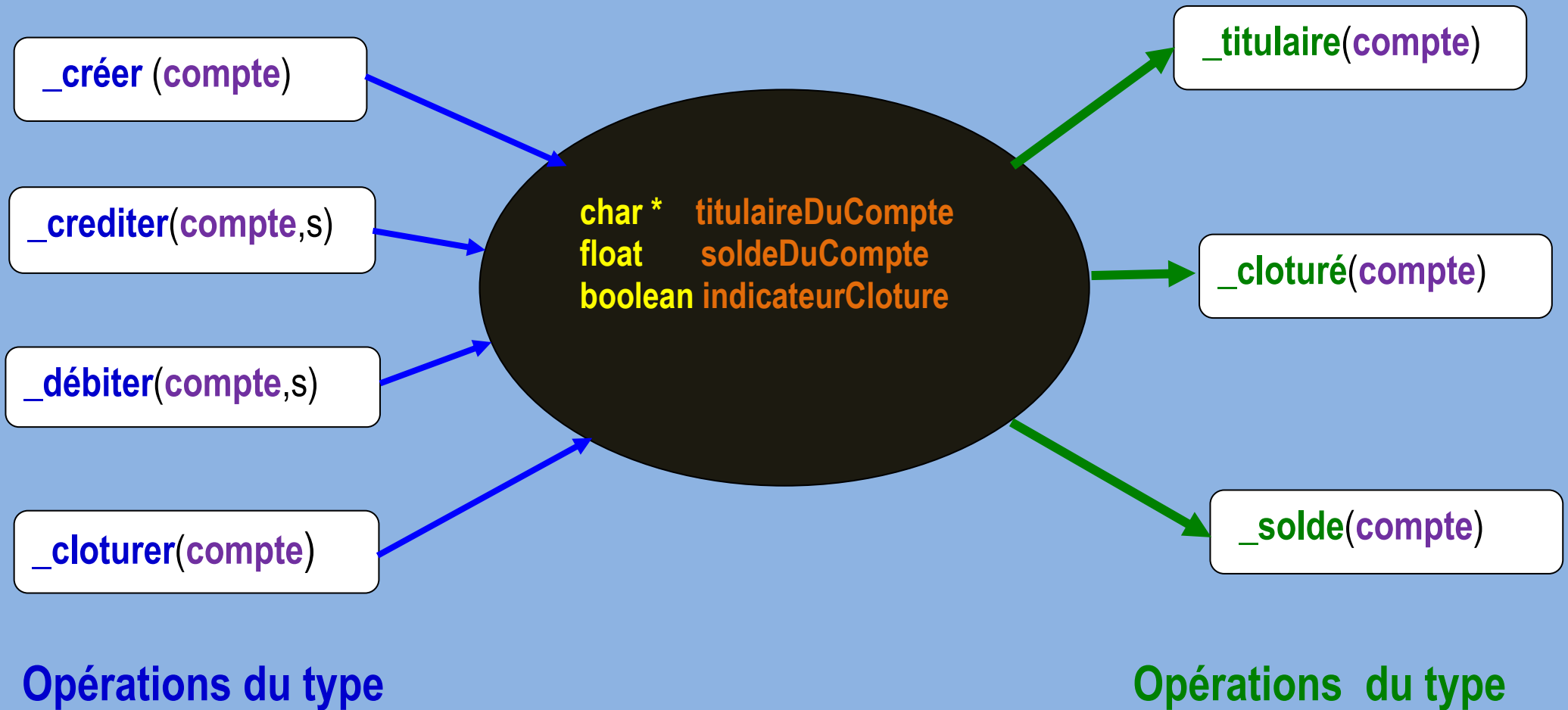
- non **clôturés**
- dont le **solde** est supérieur au montant **s**.

Une taxe est **débitée** de ces comptes dont le **titulaire** et le nouveau **solde** sont enregistrés dans le fichier nommé **outFile**.

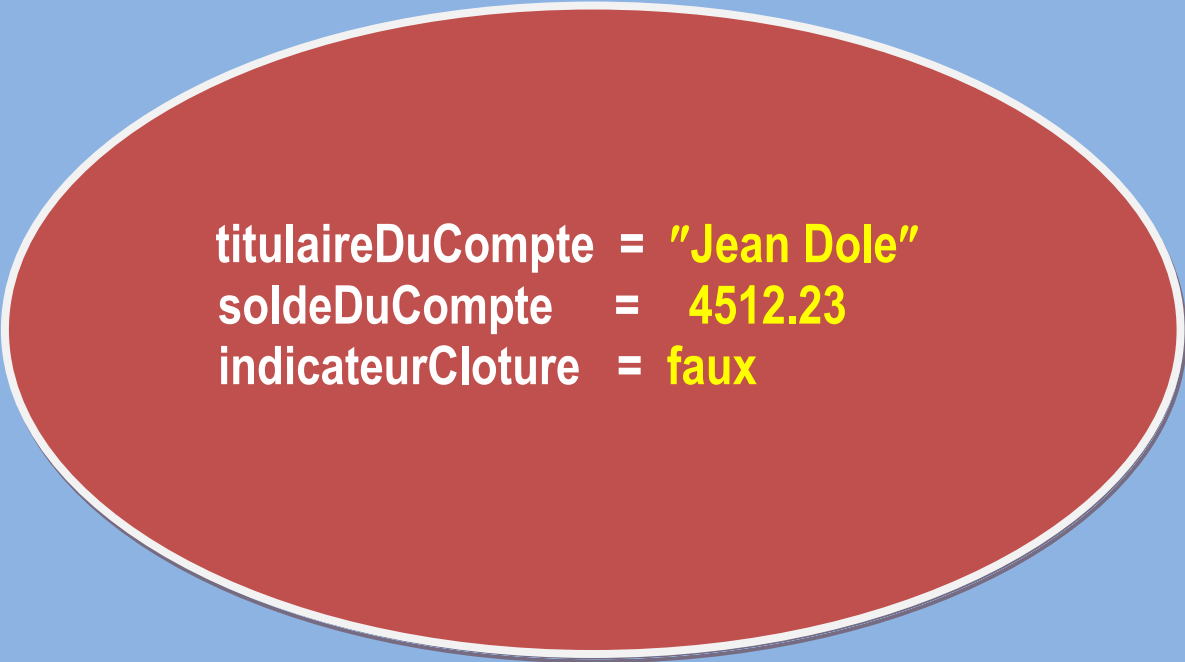
Voici une implémentation possible de cette opération

```
Opbancaire( )
  c:COMPTE,      /*déclarer une structure de données de type COMPTE * /
  open(_inFile,  read);
  open(_outFile, write);
  while not endfile(_inFile) do
    readfile (_inFile, c)
    if _cloturé(c) and _solde(c) > s then
      _débiter(c, taxe) ;
      writefile(_outFile, _titulaire(c) , _solde(c))
    endif
  endwhile
  close()
end
```

Voici le type COMPTE



Voici un **objet** de type **COMPTE**



```
titulaireDuCompte = "Jean Dole"  
soldeDuCompte     = 4512.23  
indicateurCloture = faux
```


Voici une **structure de données** de type **compte**



char *	titulaireDuCompte
float	soldeDuCompte
boolean	indicateurCloture

Le principe de base

Le principe de base d'une **structure de données** est une méthode de :

- de **stockage** des données
- d'**organisation** des données

pour en faciliter l'**accès** et la **manipulation**.

Une **structure de données** « crée un lien » entre :

- des **données** à gérer
- et un **ensemble d'opérations** pour les manipuler.

Dans l'exemple donné, les opérations de base suivantes :

_cloturé(c)

_solde(c)

_débiter(c, taxe)

_titulaite(c)

sont **exportées** par le type **COMPTE**

C'est ce type **COMPTE** qui **définit** et **détaille** les opérations de base suivantes :

- _cloturé(c)**
- _solde(c)**
- _débiter(c, taxe)**
- _titulaite(c)**

Conséquence

La **conception** et l'**expression** d'un algorithme est rendu :

- d'autant plus **simple**,
- que les **traitements de base** sur les données manipulées ne sont «**pas détaillés**».

Systematiquement, ces **traitement de base** :

- sont seulement **utilisés** dans l'**algorithme**
- mais **détaillés** et **exportés** à partir d'un **type**.

Exemple d'utilisation des structures de données

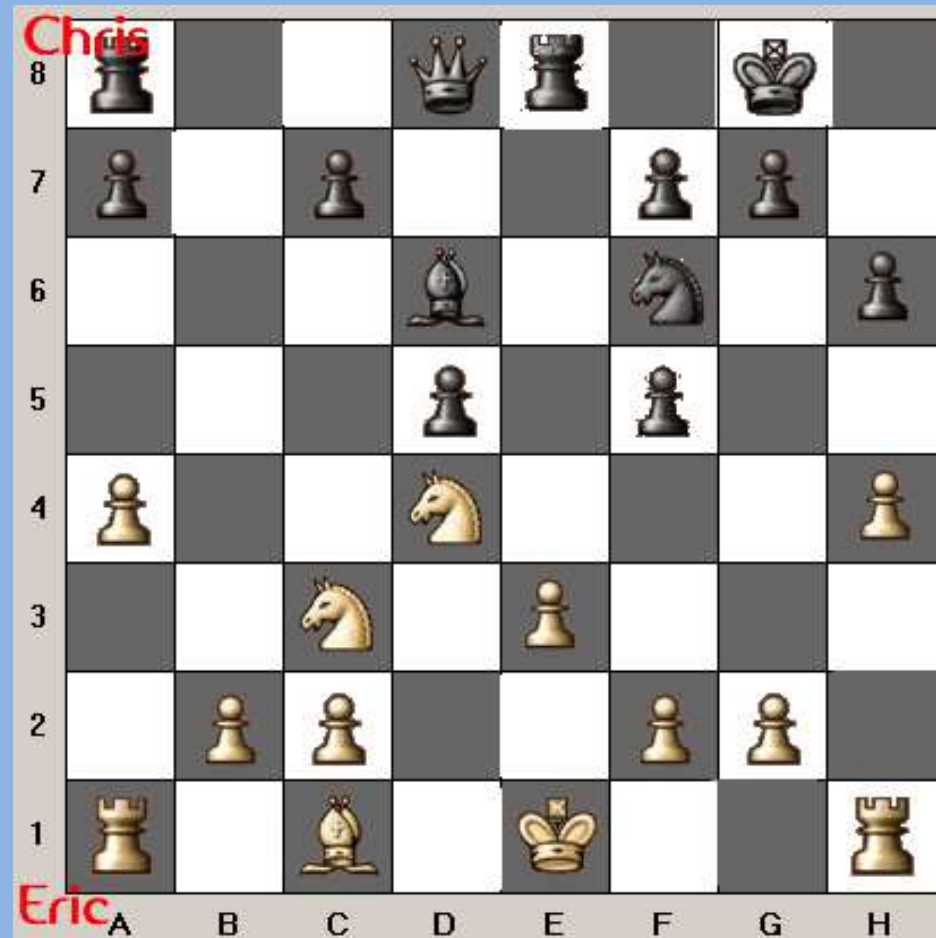
Structure de données «fantôme»



Un_fantôme

- couleur
- position
- direction déplacement
- agressif ou pacifique

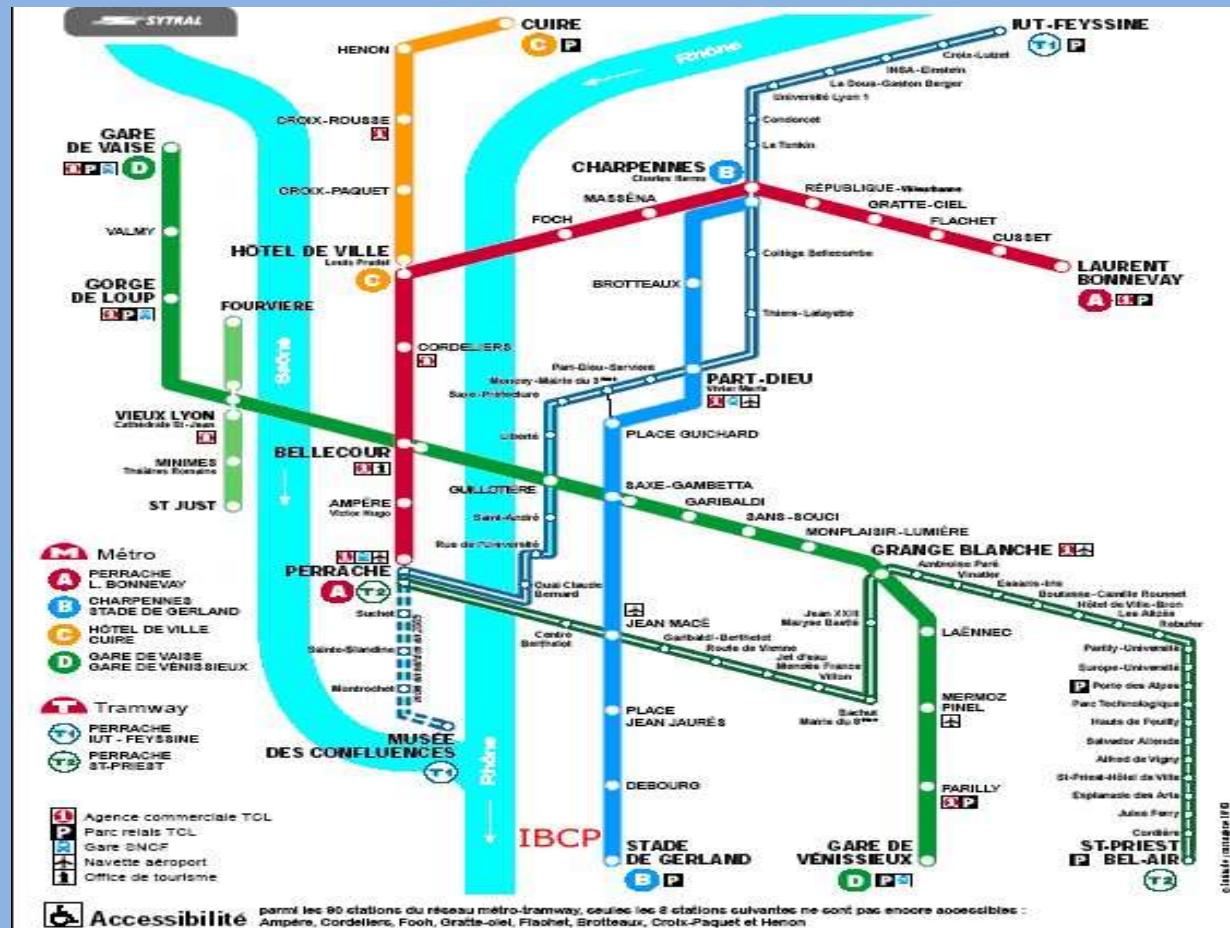
Structure de données « Jeu d'échec »



Structure de données «**File** de gestion d'accès »



Structure de données « Graphe » pour planifier un trajet



Notion type concret

L'idée de concevoir un **modèle** destiné:

- à contenir des **données**,
- à leur associer les **opérations** permettant de les **manipuler** ces données,

est à l'origine de la notion **type**.

Exemple de type des comptes bancaires

Structure de données du type

char * titulaireDuCompte
float soldeDuCompte
boolean indicateurCloture

_créer (compte)

_crediter (compte,s)

_débiter (compte,s)

_cloturer (compte)

_titulaire (compte)

_cloturé (compte)

_solde (compte)

Opérations du type

Opérations du type

Exemple d'implémentation du type compte bancaire

```
typedef struct un_compte
{
    char *    titulaireDuCompte ;
    float_    soldeDuCompte ;
    boolean   indicateurCloture
} compte;
```

```
typedef struct un_compte * COMPTE ;
```

```
COMPTE _créer ()
{
    COMPTE c ;
    scanf( "Entrer le nom du titulaire %s", &c → titulaireDuCompte) ;
    c → soldeDuCompte = 0. ;
    c → indicateurCloture = FAUX ;
    return c
}
```

```
_créditer (COMPTE c, float s)
{
    c → soldeDuCompte = c → soldeDuCompte + s ;
}
```

```
_débiter (COMPTE c, float s)
{
  c→ soldeDuCompte = c→soldeDuCompte - s ;
}
```

```
_cloturer (COMPTE c)
{
  c→ indicateurCloture = VRAI ;
}
```



```
char *_titulaire (COMPTE c)
{
    return c→ titulaireDuCompte
}
```

```
boolean _cloturé (COMPTE c)
{
    return c→ indicateurCloture
}
```

```
float _solde (c :Compte)
{
    return c→ soldeDuCompte
}
```

Algorithme et Type

La description des **traitements** dans un programme implique la conception d'**algorithmes** appropriés.

La description du **comportement des données** qui y sont manipulées suppose la définition de leur **type**.

Autant que la conception d'algorithmes, la notion de **type** est fondamentale en **conception de logiciel**.

Pourquoi ?

1^{ière} raison

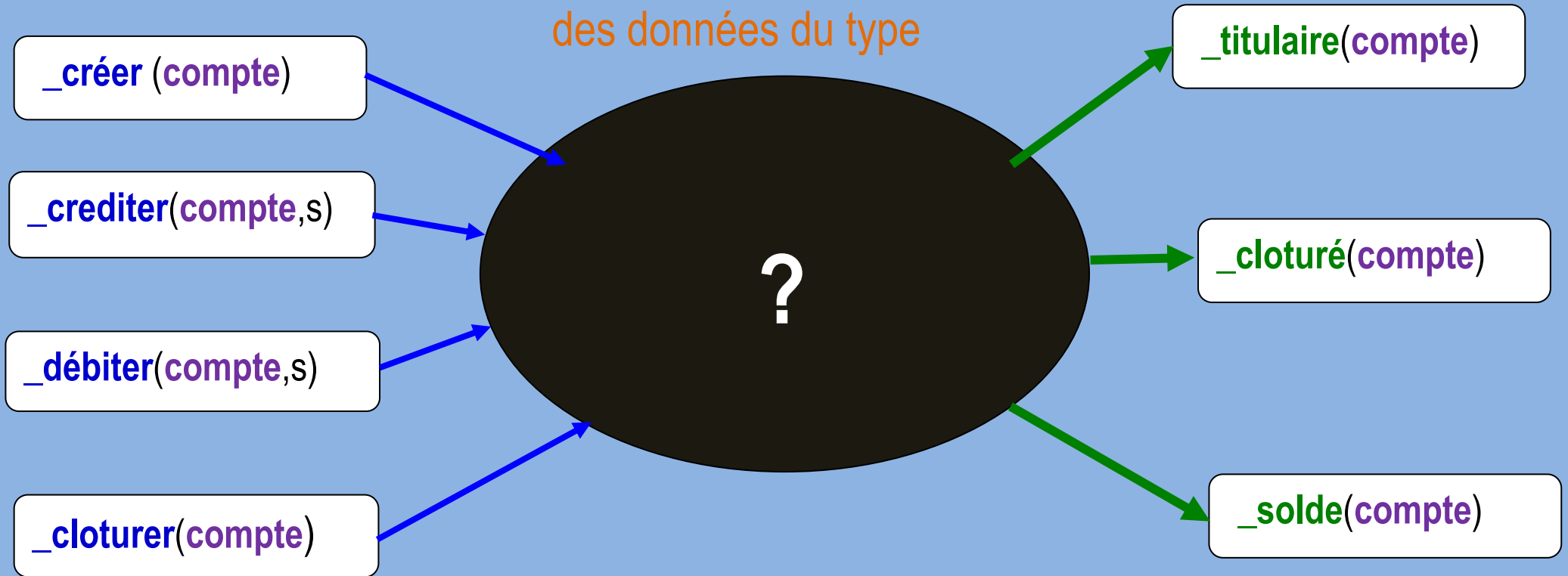
Un algorithme doit être **indépendant** de la façon dont les **données sont représentées**.

Vues d'un algorithme, les **données manipulées** doivent, donc, être considérées de manière **abstraite**.

«**Abstraite**» signifie détachée de leur **représentation interne** :
leur **implémentation**.

Cela signifie que la **façon de manipuler** ces données est
indépendante de leur **implémentation**

Abstraction de la représentation
des données du type



Opérations du type

Opérations du type

2^{ème} raison

Soit l'opération bancaire décrite précédemment :

```
Opbancaire( )
  c:COMPTE,      /*déclarer une structure de données de type COMPTE * /
  open(_inFile, read);
  open(_outFile, write);
  while not endfile(_inFile) do
    readfile (_inFile, c)
    if __cloturé(c) and __solde(c) > s then
      __débiter(c, taxe) ;
      writefile(_outFile, __titulaire(c) , __solde(c))
    endif
  endwhile
  close()
end
```

La procédure exprimant l'algorithme utilise judicieusement le type **COMPTE**.

Pourquoi ?

Réponse : le type **COMPTE** exporte les opérations de base suivantes :

_cloturé(c)
_solde(c)
_débiter(c, taxe)
_titulaite(c)

C'est ce type **COMPTE** qui **détaille** les opérations de base suivantes :

- _cloturé(c)**
- _solde(c)**
- _débiter(c, taxe)**
- _titulaite(c)**

pour manipuler la **structure de données c**.

Conclusion

Le rôle d'un **type** est de:

- définir
- et exporter

des opérations **de base** qui décrivent :

- le **comportement**
- et les **propriétés intrinsèques**

des **structures données** du type.

Notion d'objet

Dans un programme un **objet** apparaît :

- soit comme une **variable**,
- soit comme une **constante**.

Exemple

```
open(__inFile, read);
open(__outFile, write);
while not endfile(__inFile) do
    readfile (__inFile, c)
    if __cloturé(c) and __solde(c) > s then
        __débiter(c, taxe) ;
        writefile(__outFile, __titulaire(c) , __solde(c))
    endif
endwhile
```

Les objets qui sont des **variables** : **c**

Les objets qui sont des **constantes** : **inFile**, **_outFile**, **s**, **taxe**

Description d'un objet

Pour décrire un objet, il est nécessaire de décrire:

- son **état**,
- l'ensemble d'**opérations permises** pour manipuler cet état.

Etat d'un objet

L'**état** d'un objet correspond à la **valeur courante** de l'objet.

Exemple de l'**état** de l'objet décrivant un compte bancaire.



```
_titulaireDuCompte : Jean Michel  
_soldeDuCompte : 512.84  
_IndicationCloture : False
```

Objet et opérations permises

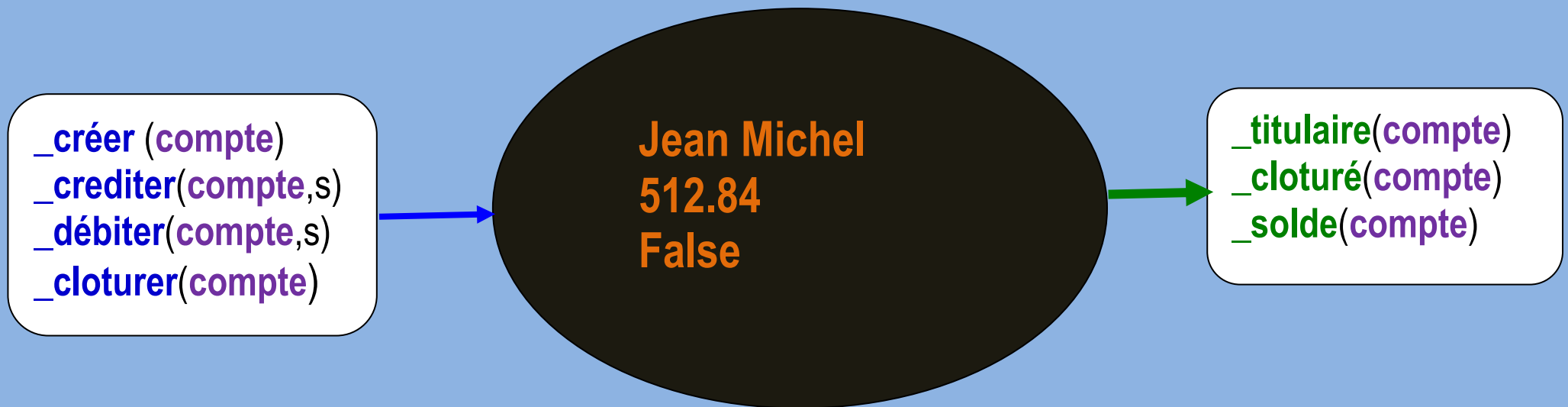
L'ensemble d'**opérations permises** sont destinées à :

- **créer** un objet : **mode constructif**
- **transformer** l'état de l'objet : **mode mutatif**
- **évaluer** l'état de l'objet : **mode observateur.**

Exemple

Ici, l'objet **unCompte** est un compte bancaire

Etat de l'objet unCompte



Opérations permises
pour **créer** ou **modifier** son état

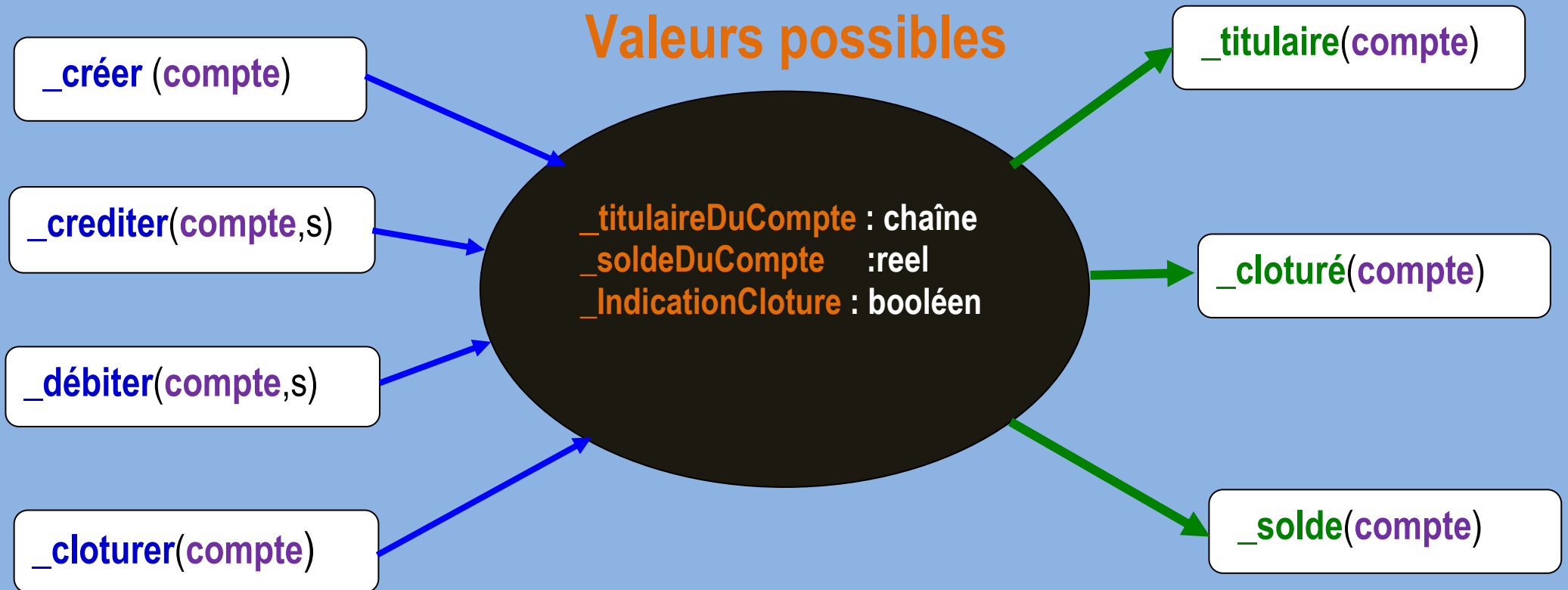
Opérations permises
pour **observer** son état

Qu'est-ce qu'un type ?

De façon **informelle**, un **type** est défini par :

- un **ensemble de valeurs possibles**,
- un ensemble d'**opérations légales** sur celle-ci.

Valeurs possibles



Opérations légales

Opérations légales

Relation objet- type

Un **objet** est une **instance** d'un certain **type**.

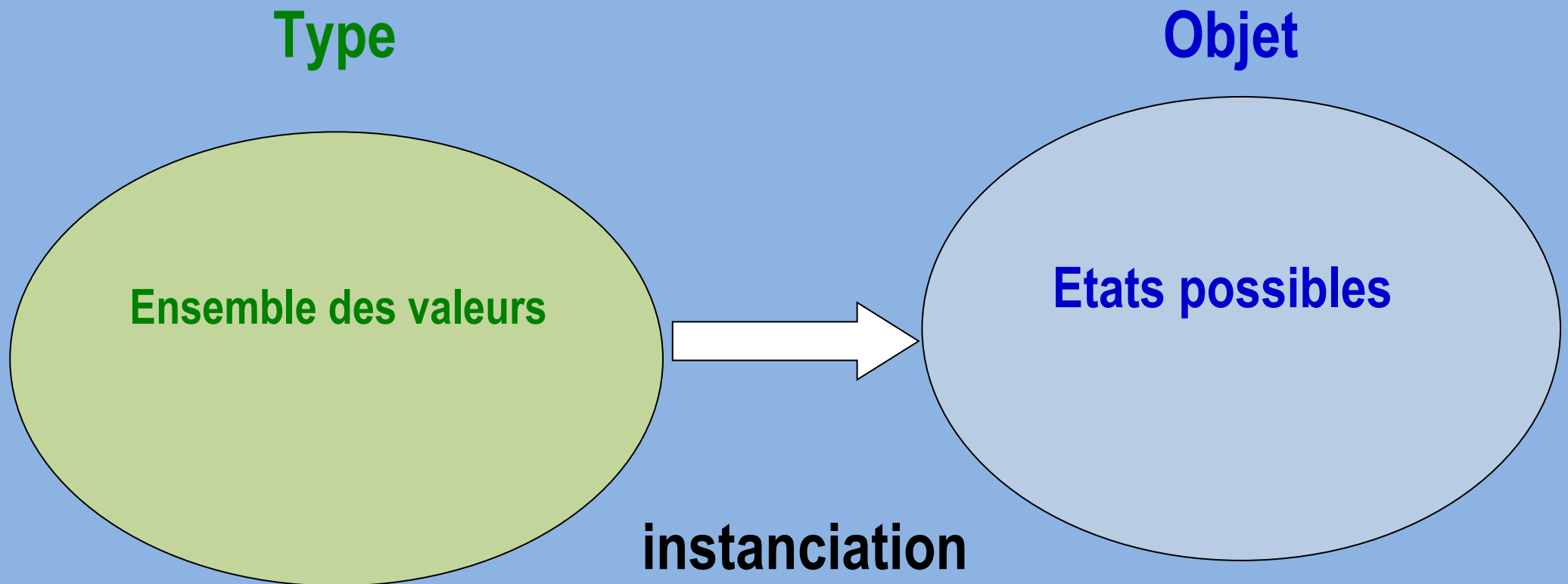
Un **objet** est créé par **instanciation** à partir d'un certain **type**.

Conséquences

1-La définition du type doit **précéder** la création d'un **objet**.

2-Aucune possibilité de création d'un objet sans avoir défini **préalablement** son type.

Que signifie le terme **instance** ?



Type

Objet

Opérations légales

Opérations permises

instantiation

II- NOTION DE TYPE ABSTRAIT

En génie logiciel, il existe deux sortes de types :

- les types **abstraits**,
- les types **concrets**.

Les **types concrets** sont :

- soit, des **types de base** du langage de codage
- soit des **types définis** à l'aide de **constructeurs**.

Exemple de types de base

```
auteur      : string ;  
editeur     : string ;  
catégorie   : char ;  
nombreDePages : integer ;  
prix: real ;  
épuisé? : boolean ;
```

Exemple de constructions de types

Ici, **record** et **seq of** sont des **constructeurs** de type :

```
type LIVRE = record
    begin
        auteur    : string;
        editeur   : string;
        prix      : real ;
        épuisé    : boolean
    end ;
```

```
type RAYON  = seq of LIVRE ;
type COLIS  = set of LIVRE ;
```

Exemple de type concret

Structure de données

_titulaireDuCompte : string
_soldeDuCompte : real
_IndicateurCloture : boolean

_créer (compte)

_crediter(compte,s)

_débiter(compte,s)

_cloturer(compte)

_titulaire(compte)

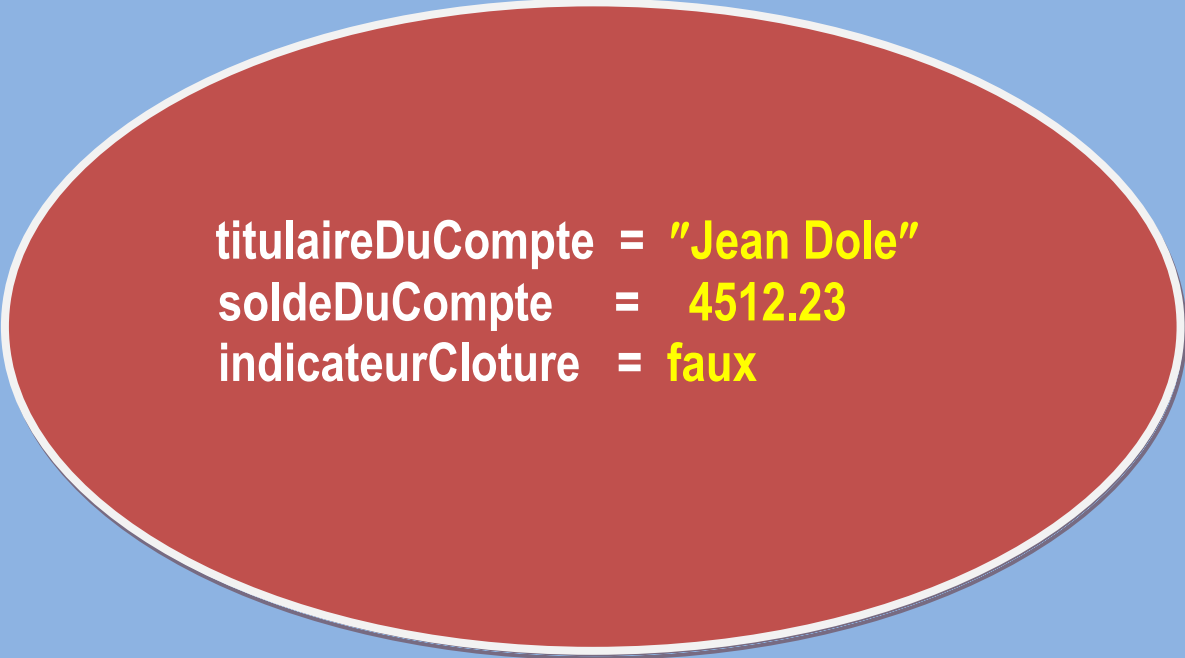
_cloturé(compte)

_solde(compte)

Implémentations
des opérations

Implémentations
des opérations

Voici un **objet** de type **COMPTE**



titulaireDuCompte = "Jean Dole"
soldeDuCompte = 4512.23
indicateurCloture = faux

Voici une définition de type concret **COMPTE**

```
compte = record  
  begin  
    _titulaireDuCompte : string ;  
    _soldeDuCompte      : real ;  
    _IndicateurCloture  : boolean  
  end
```

```
_créer () : compte  
begin  
  c:compte  
  read( c ._titulaireDuCompte) ;  
  c ._soldeDuCompte := 0. ;  
  c ._IndicateurCloture := false ;  
  return c  
end
```

```
_créditer (c :compte, s : real) : compte  
begin  
  c ._soldeDuCompte := c ._soldeDuCompte + s ;  
  return c  
end
```

```
_débiter (c : compte, s : real) : compte  
begin  
  c . _soldeDuCompte := c . _soldeDuCompte - s ;  
  return c  
end
```

```
_cloturer (c : compte) : compte  
begin  
  c . _IndicateurCloture := true ;  
  return c  
end
```



```
_titulaire (c :compte) : string  
begin  
return c ._titulaireDuCompte  
end
```

```
_cloturé (c :compte) : boolean  
begin  
return c ._IndicateurCloture  
end
```

```
_solde (c :compte) : real  
begin  
return c ._soldeDuCompte  
end
```

L'inconvénient des types concrets

On remarque que la **définition** d'un **type concret** est trop **dépendante** du langage de codage utilisé.

Donc, tout changement de langage peut **remettre en cause** cette définition.

L'**instabilité** qui en résulterait est dommageable pour les

Logiciels développés:

- fiabilité,
- réutilisation,
- maintenabilité,...

D'où l'idée de concevoir une **définition du type** qui fait **abstraction** de tout langage de codage.

On dira qu'un tel type est défini de **façon externe**.

Le qualificatif « externe » signifie de façon **indépendante** de tout langage de codage.

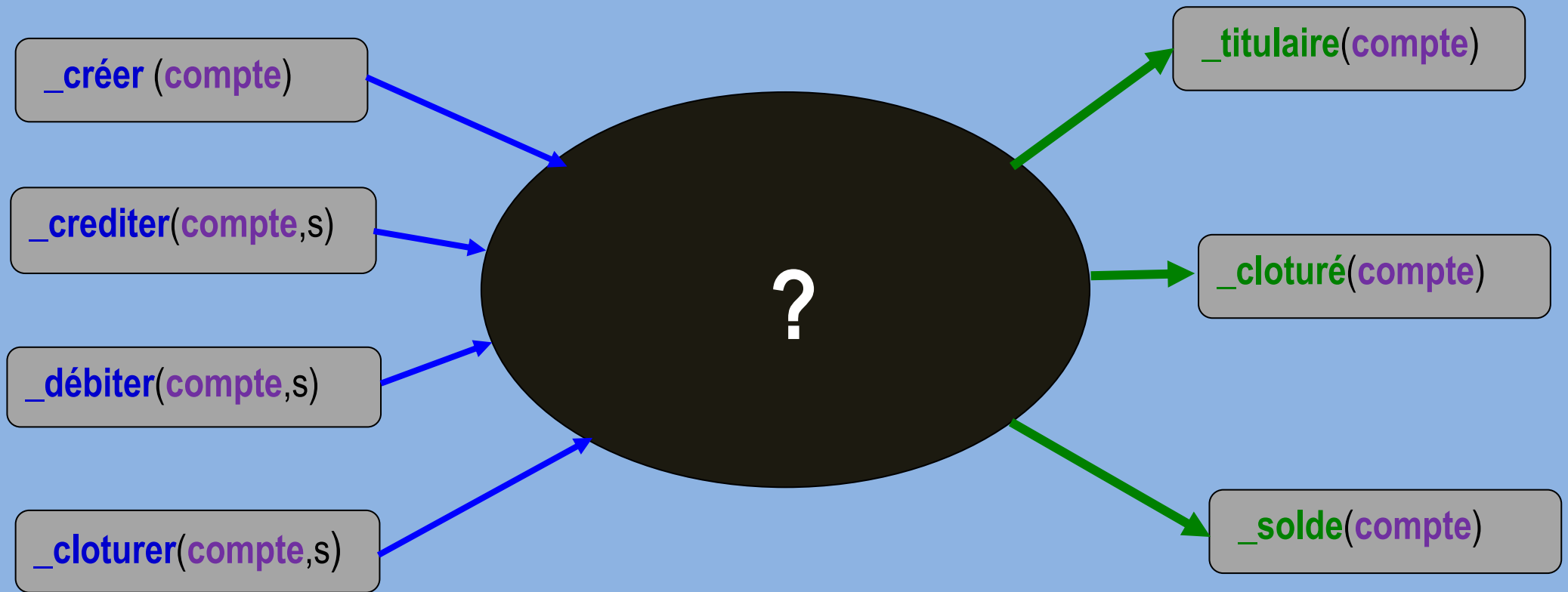
Mais, une fois un type défini, on **doit garantir** :

- qu'il aura la même **sémantique**
- **quel que soit** le langage qui sera utilisé, par la suite, pour son **implémentation**

Il résulte une grande **stabilité** des logiciels développés.

Une telle définition du type est appelé **type abstrait**.

Exemple de type abstrait des comptes bancaires



Propriétés des opérations

Propriétés des opérations

%%description type abstrait des comptes bancaires

type : **compte**

Opérations du type :

_créer () : **compte**
_crediter(**compte** , **real**) : **compte**
_debiter: (**compte** , **real**) : **compte**
_cloturer(**compte**) : **compte**
_titulaire(**compte**) : **string**
_cloturé(**compte**) : **boolean**
_solde(**compte**) : **real**

Propriétés des opérations du type

$C = \text{_créer} () \quad \Rightarrow \text{_cloturé}(C) = \text{False}$

$C' = \text{_cloturer}(C) \quad \Rightarrow \text{_cloturé}(C') = \text{True}$

$C = \text{_créer} () \quad \Rightarrow \text{_solde}(C) = 0$

$C' = \text{_crediter}(C, s) \quad \Rightarrow \text{_solde}(C') = \text{_solde}(C) + s$

$C' = \text{_débiter}(C, s) \quad \Rightarrow \text{_solde}(C') = \text{_solde}(C) - s$

$C' = \text{_crediter}(C, s) \quad \Rightarrow \text{_titulaire}(C') = \text{_titulaire}(C)$

$C' = \text{_débiter}(C, s) \quad \Rightarrow \text{_titulaire}(C') = \text{_titulaire}(C)$

$C' = \text{_cloturer}(C) \quad \Rightarrow \text{_titulaire}(C') = \text{_titulaire}(C)$

La **description** d'un type abstrait est appelée **spécification du type abstrait**.

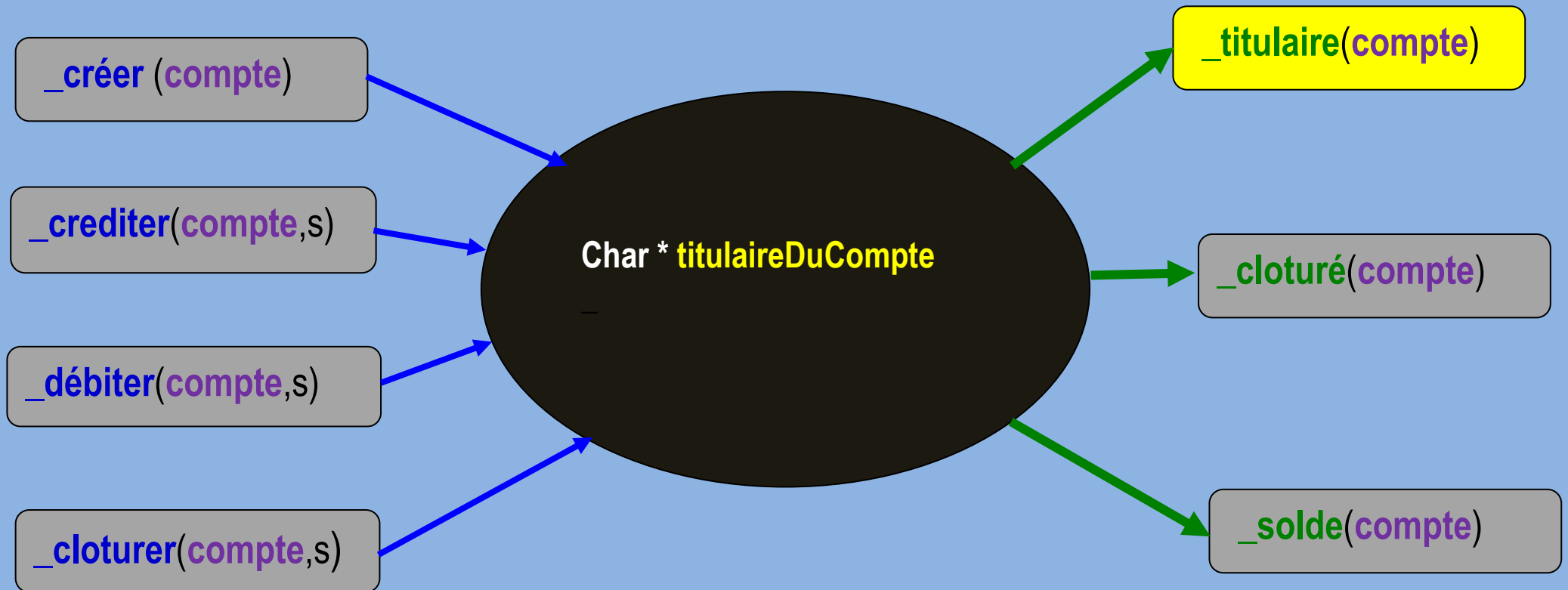
Le but d'une **spécification** du type abstrait est de décrire les **propriétés des opérations** du type.

Une telle spécification **ne décrit pas** la façon de **représenter les données** du type.

La spécification fournit une description **indépendante**
de tout **langage de programmation**.

Question : comment faire référence aux données ?

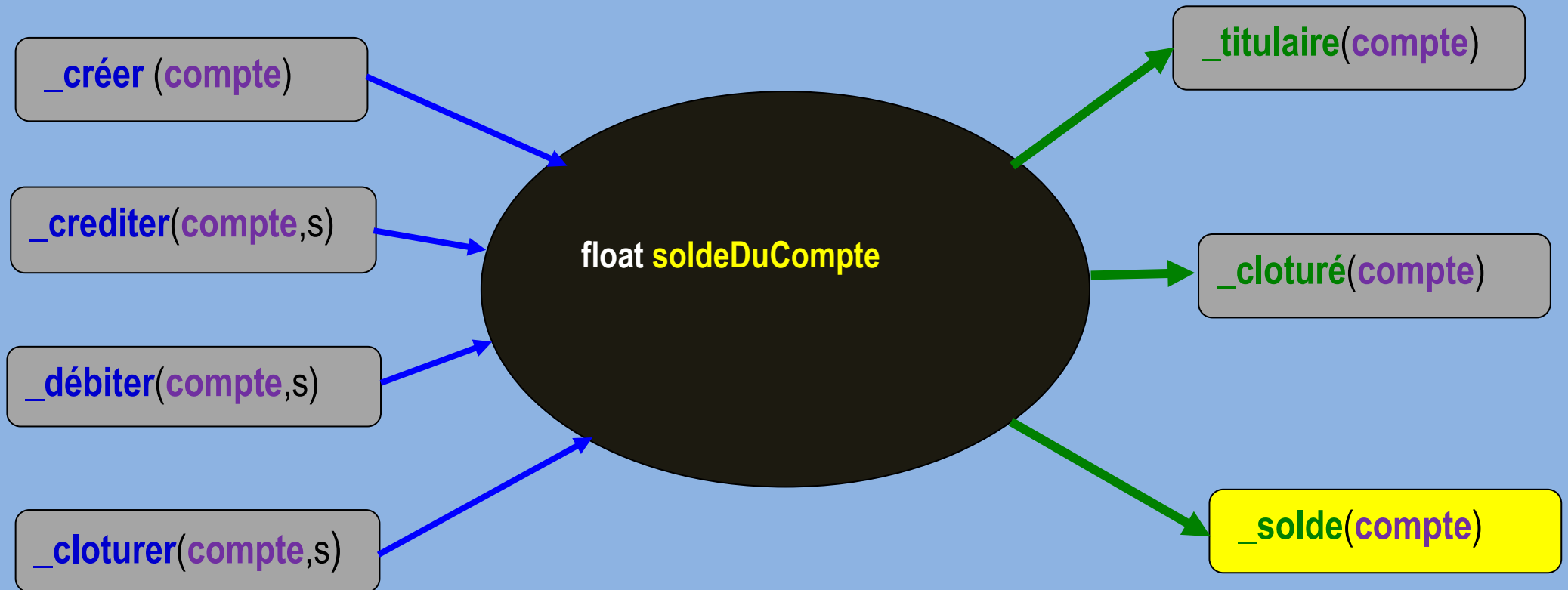
titulaire(compte)



Propriétés des opérations

Propriétés des opérations

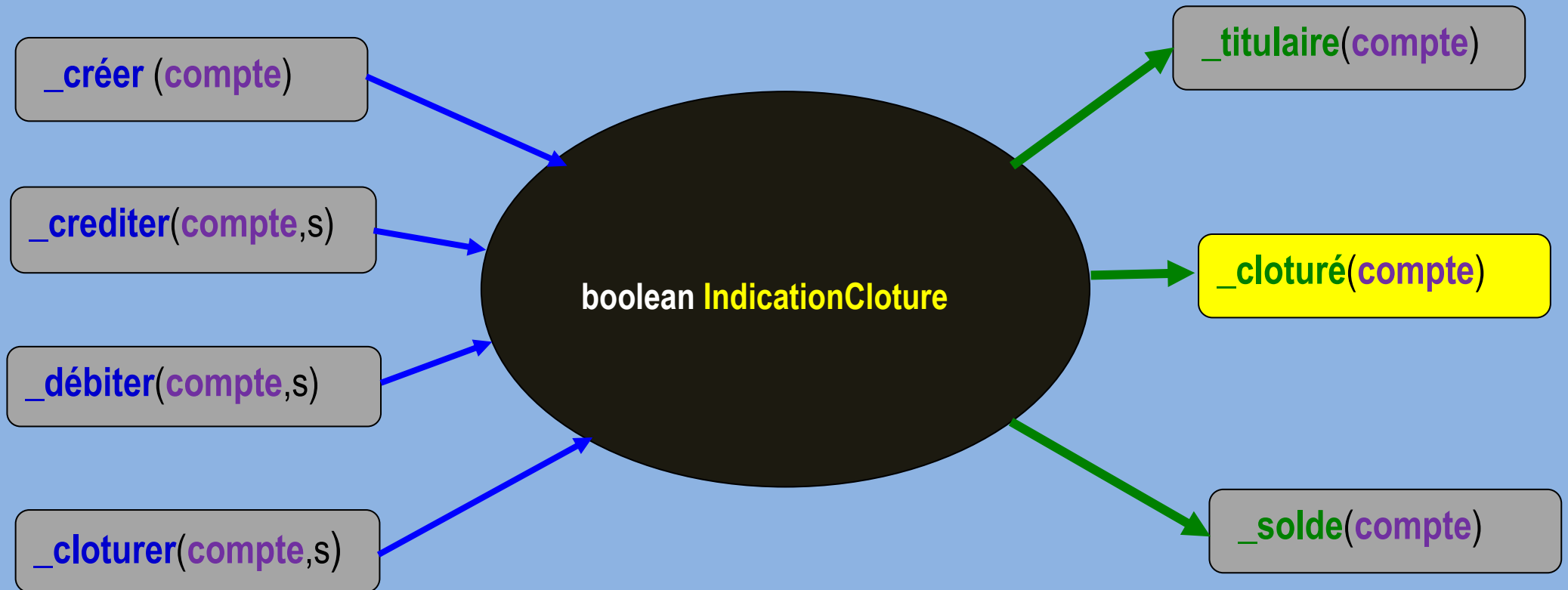
_solde(compte)



Propriétés des opérations

Propriétés des opérations

_cloturé(compte)



Propriétés des opérations

Propriétés des opérations

Problématique

1-Comment **définir** ces types abstraits ?

- comment décrire l'**état** des **objets** du type?
- comment exprimer les **propriétés** des **opérations** qui **créent** et **modifient** ces objets?

2-Quel est le **langage** approprié pour le faire ?

Un tel **langage** doit remplir deux critères:

1- être **formel** : pour apporter de la **rigueur** à la description

2-basé sur la **logique** : pour permettre d'exprimer des **propriétés**.

Spécification formelle d'un type abstrait

Une **spécification** d'un type abstrait est dite **formelle** lorsqu'elle utilise un **langage formel**.

Il existe plusieurs **langages de spécification**.

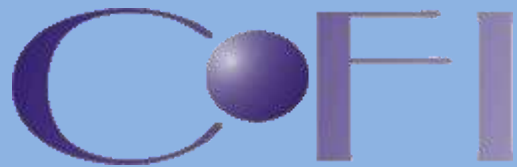
Le langage **CASL** est le langage **universel** adopté par la plupart des développeurs.

Présentation rapide de CASL

1-CoFI

A partir de **1995** une coopération ouverte a été initiée par la communauté des **chercheurs**.

CoFI



The **C**ommon **F**ramework **I**nitiative
for algebraic specification and development

<http://www.informatik.uni-bremen.de/cofi/index.php/CASL>

L'objectif est de parvenir à un **cadre de travail** commun en spécification algébrique.

Il est aujourd'hui organisé sous l'intitulé **CoFI: Common Framework Initiative**

2-CASL

Le **C**ommon **A**lgebraic **S**pecification **L**anguage (CASL), a été conçu par CoFI.

Plus qu'un langage, c'est un outil pour:

- **spécifier les exigences** exprimées dans un cahier des charges,
- **développer du logiciel.**

Approche algébrique

Une **spécification formelle** est dite **algébrique** lorsque elle s'attache :

- à définir **complètement** un **type**
- uniquement** en décrivant les **propriétés** des **opérations** qui manipulent les objets de ce type

Conséquence

1-Une spécification algébrique **ne décrit pas** la façon de **représenter les objets**.

2-C'est une description qui s'attache à décrire les **propriétés** des **opérations** qui :

- **créent** ces objets,
- **modifient et observent** leur état.

3-Une **spécification algébrique** **s'affranchit** de tout langage de programmation.

Exemple du type abstrait des comptes bancaires

A titre d'exemple, voici la spécification du type abstrait des **comptes bancaires**

La spécification est écrite en langage **Casl**.

Cahier des charges d'un « compte bancaire »

Les tâches de gestion retenues pour un compte sont les suivantes :

- **créer** un compte bancaire,
- connaître le nom de son **titulaire**: un même **titulaire** ne peut posséder plusieurs comptes,
- chercher à savoir son **solde**,
- **débiter** un compte,
- **créditer** un compte,
- **cloturer** un compte,
- tester si un compte **est clôturé** : un compte fermé ne peut être ni débité ni crédité.

Exemple de spécification en CASL du type abstrait

```
%%Spécification du type abstrait des comptes
spec COMPTE=
%% spécification par extension sur Rat et String
  Rat and String and Boolean
then
  sort Compte      %(type abstrait des comptes)%

%%profils ou signature des opérations du type
  ops
%% constructeur du type
    _créer : Compte;
%% modificateurs du type
    _crediter      :Compte x Rat →? Compte;
    _debiter       :Compte x Rat →? Compte;
    _cloturer      :Compte →? Compte;
%% accesseurs du type
```

_titulaire :Compte \rightarrow String
 _solde : Compte \rightarrow Ratl;
 _cloturé : Compte \rightarrow Boolean;

%% *domaines des fonctions partielles*

\forall c1,c2: Compte ; s1:Rat ; p1,p2:Nom

- def _crediter(c1, s1) \Leftrightarrow _cloturé(c1) = False
- def _debiter(c1, s1) \Leftrightarrow _cloturé(c1) = False
- def _cloturer(c1) \Leftrightarrow _cloturé(c1) = False

%%définition de _cloturé

- _cloturé (_créer) = False
- _cloturé (_cloturer(c1)) = True

%%définition l'opération _solde

- _solde(_créer)= 0
- _solde(_cloturer(c1)) = _solde(c1)
- _solde(_crediter(c1,s1))=_solde(c1)+ s1
- _solde(_debiter(c1,s1))= _solde(c1)-s1

%%définition l'opération ***_titulaire***

- **_itulaire**(c1) = **_titulaire**(c2) \Rightarrow c1 = c2
- **_titulaire** (**_cloturer** (c1))= **_titulaire**(c1)
- **_titulaire**(**_crediter**(c1,s1))= **_titulaire**(c1)
- **_titulaire**(**_debiter**(c1,s1))= **_titulaire**(c1)

end

Structure de la spécification en CASL du type abstrait

1- En-tête

```
%%Spécification du type abstrait des comptes
spec COMPTE=
%% spécification par extension sur REAL et STRING
    Rat and String and Boolean

    then
    sort Compte          %(type abstrait des comptes)%
```

2- Signature des opérations

%%profils ou signature des opérations du type

ops

%% constructeur du type

_créer : **Compte**;

%% modificateurs du type

_crediter : **Compte** x Rat \rightarrow ? **Compte**;

_debiter : **Compte** x Rat \rightarrow ? **Compte**;

_cloturer : **Compte** \rightarrow ? **Compte**;

%% accesseurs du type

_titulaire : **Compte** \rightarrow String

_solde : **Compte** \rightarrow Rat;

_cloturé : **Compte** \rightarrow Boolean;

%% *domaines des fonctions partielles*

$\forall c1, c2$: **Compte** ; $s1$:Rat ; $p1, p2$:Nom

- **def** **_crediter**($c1, s1$) \Leftrightarrow **_cloturé**($c1$) = False
- **def** **_debiter**($c1, s1$) \Leftrightarrow **_cloturé**($c1$) = False
- **def** **_cloturer**($c1$) \Leftrightarrow **_cloturé**($c1$) = False

3- Sémantique

%% axiomes définissant les accesseurs par leurs propriétés

*%%définition de **_cloturé***

- **_cloturé** (**_créer**) = False
- **_cloturé** (**_cloturer**(c1)) = True

*%%définition l'opération **solde***

- **_solde**(**_créer**) = 0
- **_solde**(**_cloturer**(c1)) = **_solde**(c1)
- **_solde**(**_crediter**(c1,s1)) = **_solde**(c1) + s1
- **_solde**(**_debiter**(c1,s1)) = **_solde**(c1) - s1

%%définition l'opération *titulaire*

- **_titulaire**(c1)= **_titulaire**(c2) $\Rightarrow c1 = c2$
- **_titulaire** (**_cloturer** (c1)) = **_titulaire**(c1)
- **_titulaire**(**_crediter**(c1,s1)) = **_titulaire**(c1)
- **_titulaire**(**_debiter**(c1,s1)) = **_titulaire**(c1)

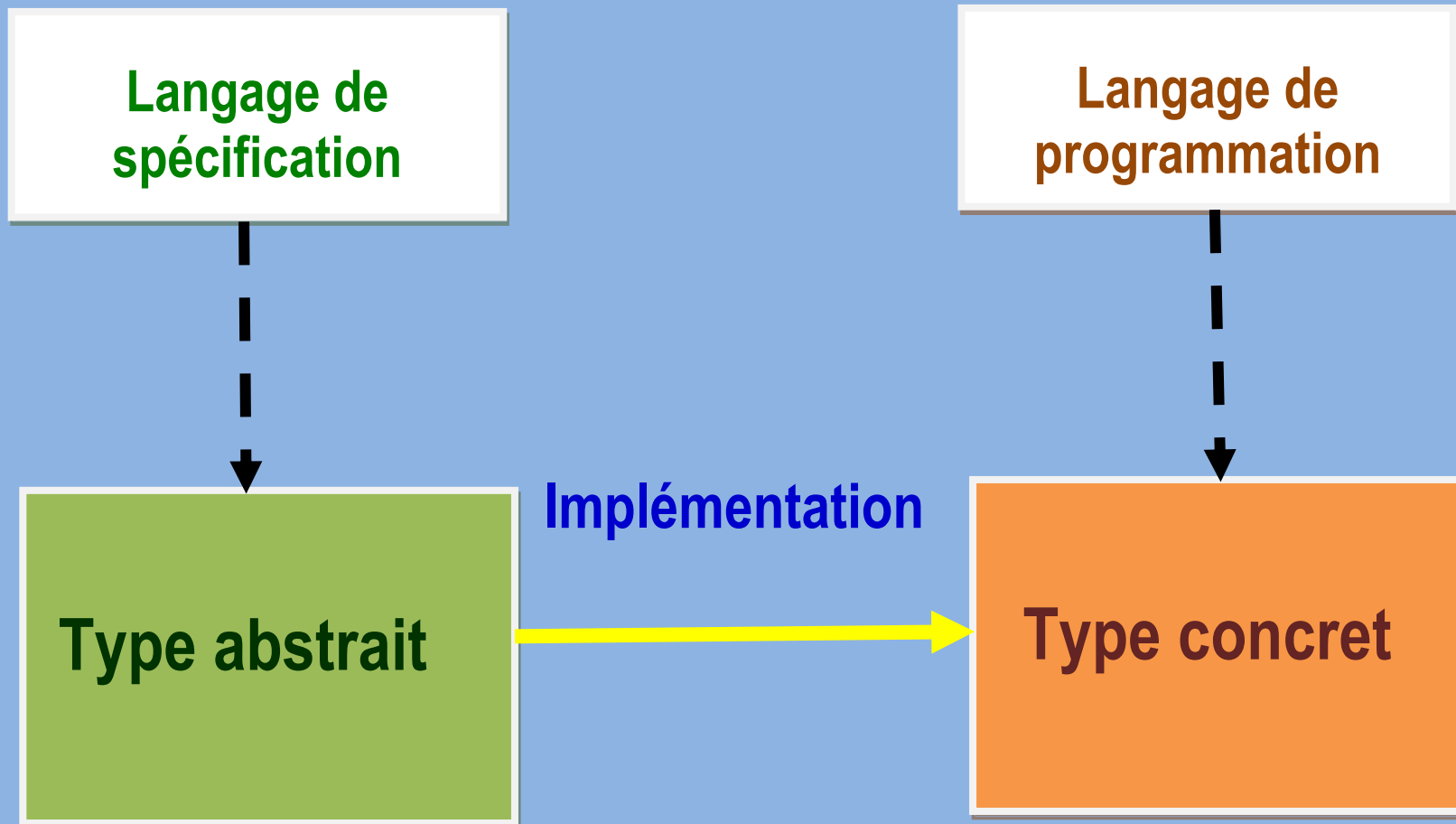
end

Implémentation d'un type

Une **implémentation** du **type abstrait** :

- est une **réalisation** du type
- en utilisant **un** langage de programmation.

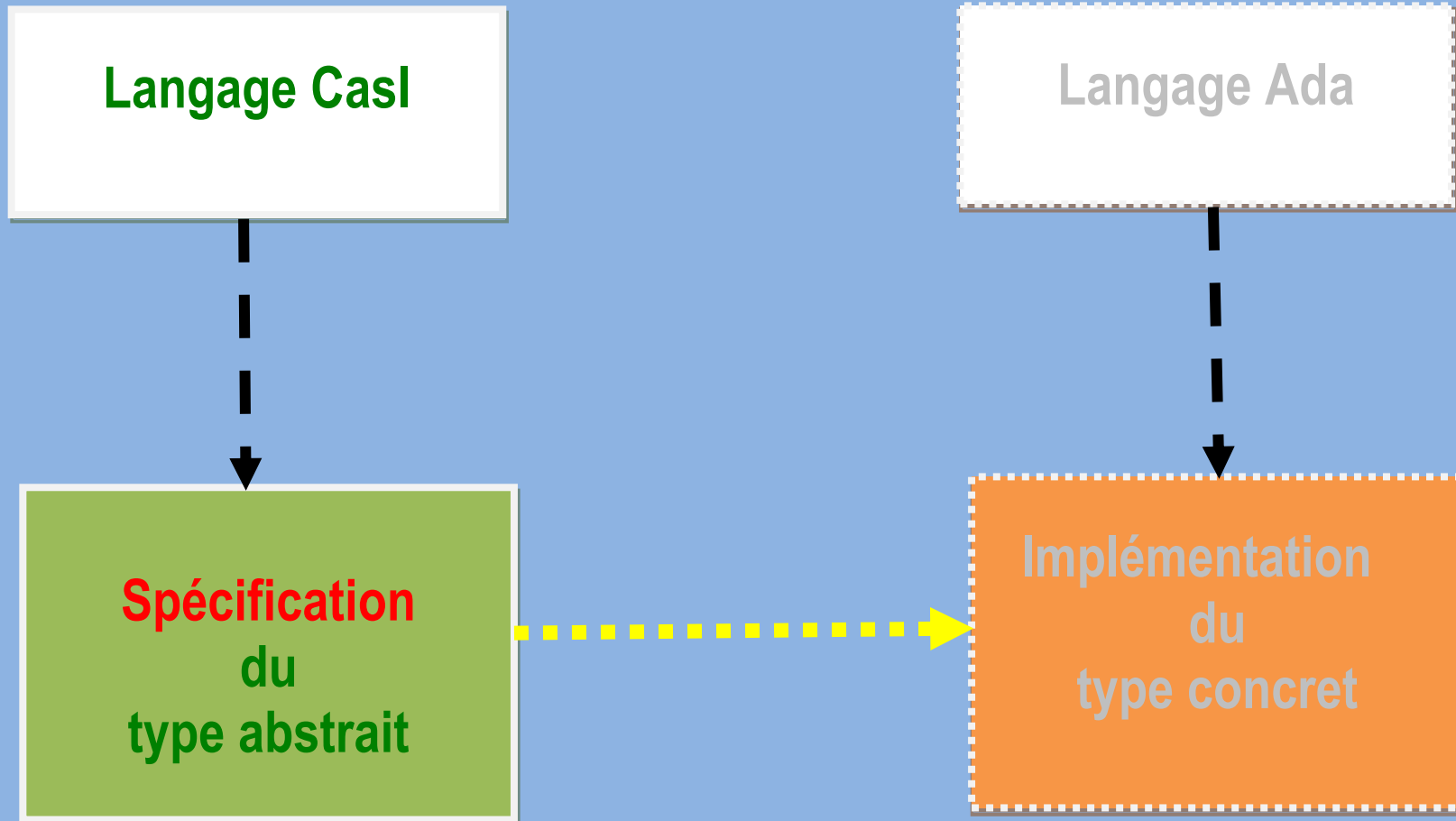
Le résultat de l'implémentation du **type abstrait** est un **type concret**.



Correspondance « type abstrait- type concret »

Le développeur de logiciel commence par établir une **spécification** du **type abstrait**.

Le **type abstrait** décrit les **propriétés** des **opérations** pour manipuler les données.



1-le développeur établit la spécification du type

%%Spécification du type abstrait des comptes

spec COMPTE=

%% spécification par extension sur REAL et STRING

Rat and String and Boolean

then

sort Compte *%(type abstrait des comptes)%*

%%profils ou signature des opérations du type

ops

%% constructeur du type

-créer : **Compte**;

%% modificateurs du type

_crediter : **Compte** x Rat →? **Compte**;

_debiter : **Compte** x Rat →? **Compte**;

_cloturer : **Compte** →? **Compte**;

%% accesseurs du type

_titulaire : **Compte** → String

```

    _solde      : Compte → Rat;
    _cloturé    : Compte → Boolean;

```

%% *domaines des fonctions partielles*

∀ c1,c2: **Compte** ; s1:Rat; p1,p2:String

- def **_crediter**(c1, s1) <=> _cloturé(c1) =False
- def **_debiter**(c1, s1) <=> _cloturé(c1) =False
- def **_cloturer**(c1) <=> _cloturé(c1) =False

%%définition de l'observateur **_cloturé**

- **_cloturé** (**_créer**) =False
- **_cloturé** (**_cloturer**(c1)) =True

%%définition l'opération **solde**

- **_solde**(**_créer**)= 0
- **_solde**(**_cloturer**(c1)) = _solde(c1)
- **_solde**(**_crediter**(c1,s1)) =_solde(c1)+ s1
- **_solde**(**_debiter**(c1,s1)) = _solde(c1)-s1

%%définition l'opération **titulaire**

- **_titulaire**(c1)= **titulaire**(c2) => c1 = c2
- **_titulaire** (**cloturer** (c1)) = _titulaire(c1)
- **_titulaire**(**crediter**(c1,s1))= _titulaire(c1)
- **_titulaire**(**debiter**(c1,s1)) = _titulaire(c1)

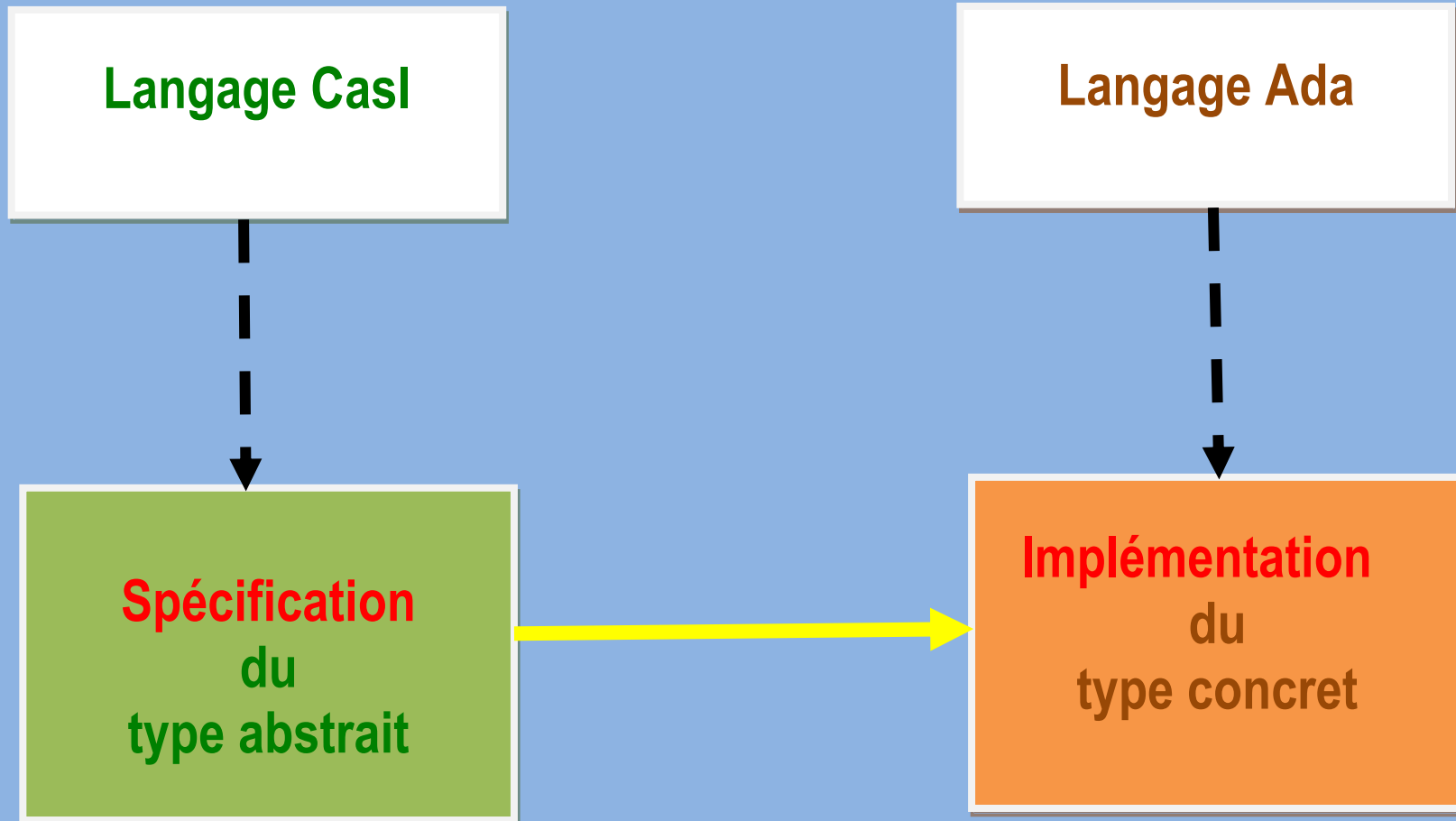
end

2-le développeur implémente la spécification du type

Le développeur **implémente**, ensuite, le **types abstrait** en **type concret** pour :

- fournir une **représentation** des données,
- décrire les **opérations**,

en utilisant un **langage de programmation**.



Exemple d'implémentation du type concret

1- Représentation des objets de compte

```
compte = record  
  begin  
    _titulaireDuCompte : string ;  
    _soldeDuCompte      : real ;  
    _IndicateurCloture  : boolean  
  end
```

2- Implémentation des opérations du type

```
_créer () : compte  
begin  
  c:compte  
  read( c ._titulaireDuCompte) ;  
  c ._soldeDuCompte := 0. ;  
  c._IndicateurCloture := false ;  
  return c  
end
```

```
_créditer (c :compte, s : real) : compte  
begin  
  c ._soldeDuCompte := c ._soldeDuCompte + s ;  
  return c  
end
```

```
_débiter (c : compte, s : real) : compte  
begin  
  c . _soldeDuCompte := c . _soldeDuCompte - s ;  
  return c  
end
```

```
_cloturer (c : compte) : compte  
begin  
  c . _IndicateurCloture := true ;  
  return c  
end
```

```
_titulaire (c : compte) : string  
begin  
  return c . _titulaireDuCompte  
end
```

```
_cloturé (c :compte) : boolean  
  begin  
    return c ._IndicateurCloture  
  end
```

```
_solde (c :compte) : real  
  begin  
    return c ._soldeDuCompte  
  end
```

Exemple du type abstrait polynôme:

Un **cahier des charges** simplifié spécifie qu'un polynôme peut être représenté par :

- un degré, noté **n** ,
- et une suite de **n+1** coefficients, notés a_i :

$$P = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

Il définit également les opérations légales suivantes:

1- **créer** le polynôme **nul** :

$$P = 0$$

2- prendre un polynôme P de degré **$n-1$** et un terme constant **a** pour **construire** le polynôme :

$$P.x + a$$

3-calculer le **degré** d'un polynôme,

$$P = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

$$a_n \neq 0 \Rightarrow \text{degré}(P) = n$$

4-calculer le **coefficient** d'un polynôme,

$$P = a_n x_n + a_{n-1} x_{n-1} + \dots + a_0 x_0$$

$$i \in [0;n] \Rightarrow \text{coefficient}(P, i) = a_i$$

5-**tester** la nullité d'un polynôme,

$$P = a_n x_n + a_{n-1} x_{n-1} + \dots + a_0 x_0$$

$$P = \text{Nul} \Leftrightarrow \text{degré}(P) = 0 \wedge \forall i \in \mathbb{N} . [\text{coefficient}(P, i) = 0]$$

Exemple de spécification en CASL du type abstrait polynome

(à copier, éditer sous **emacs** et analyser sous **hets**)

```
library polynome
```

```
%% attention : le fichier contenant la spécification doit s'appeler polynome.casl
```

```
%% liste des importations (downloading) à partir des librairies standards
```

```
from Basic/Numbers           get   Int , Rat
```

```
from Basic/SimpleDatatypes   get   Boolean
```

```
%%Spécification du type abstrait des polynômes
```

```
spec POLYNOME =
```

```
%%Commencer par indiquer quels sont les types abstraits réutilisés: ici, les types abstraits Int et Rat
```

```
Int and Rat and Boolean
```

```
then
```

```
%% donner un nom pour désigner le type abstrait défini par la spécification
```

```
sort Polynome
```


%% donner la signature des opérations du type

ops

Nul : Polynome; %(*opération qui construit le polynome nul*)%

Construire : Polynome * Rat -> Polynome; %(*construit un polynome*)%

estNul : Polynome -> Boolean; %(*teste si un polynome est nul*)%

Degre : Polynome -> Int; %(*calcule le degre d'un polynome*)%

Coefficient : Polynome * Int -> Rat %(*calcule un coefficient d'un polynome*)%

forall p1:Polynome; a0, x0:Rat ; i:Int

%% axiomes définissant les opérations par leurs propriétés

%%définir observateur **estNul**

. **estNul**(**Nul**) = True

. **estNul**(**Construire**(p1,a0)) = True <=> estNul(p1)= True /\ a0 = 0

%%définir observateur **Degre**

. **Degre**(**Nul**) = 0

. **Degre** (**Construire**(p1,a0)) = 0 <=> **estNul**(p1)= True

. **Degre** (**Construire**(p1,a0)) = **Degre**(p1)+1 <=> **estNul**(p1)= False

%%définir observateur **Coefficient**

. **Coefficient**((**Nul**,i) = 0

. **Coefficient** (**Construire**(p1,a0), 0) = a0

. **estNul**(p1) = True => i>0 => **Coefficient** (**Construire**(p1,a0), i) = 0

. **estNul**(p1) = False =>

(i>= 1 /\ i <= **Degre**(p1) +1 => **Coefficient** (**Construire**(p1,a0), i) = **Coefficient**(p1,i-1))

. **estNul**(p1) = False => (i> **Degre**(p1) +1 => **Coefficient** (**Construire**(p1,a0),i) = 0)

end

Implémentation du type polynôme

Pour implémenter un type abstrait, le développeur doit franchir les étapes suivantes :

1-choisir un **langage de codage**,

2-proposer une **implémentation des objets** du type,

3-sur la base de cette implémentation, proposer une implémentation pour opérations du type s'appuyant sur sa spécification.

Première phase d'implémentation du type abstrait

On crée un fichier **interface** qui sera **consultable** par le futur utilisateur du type

C'est dans ce fichier interface, appelé **polynome.h** qu'il faut:

- proposer une **représentation** pour les objets de type **Polynome**,
- déclarer toutes des **opérations** du type

Exemple de fichier interface : **polynome.h**

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#define      MaxSize  10
#define      FAUX     0
#define      VRAI     1
typedef      int      BOOLEEN;
/*
Proposer un type CONCRET pour implémenter le type ABSTRAIT des polynomes
*/
typedef struct un_polynome
{
    int  sonDegre ;
    float sonCoefficient[MaxSize];
} polynome;

/* définition du type des polynômes: un type pointeur vers un objet de type polynôme */
typedef struct un_polynome * POLYNOME;
```

```

/*
    créer un polynôme nul */
POLYNOME Nul();

/*
    construire le polynôme " $p1 \cdot x + a0$ " */
POLYNOME Construire(POLYNOME p1, float a0);

/*
    calculer le degré d'un polynôme */
int Degre(POLYNOME p);

/*
    calculer le coefficient de rang i d'un polynôme */
float Coefficient(POLYNOME p, int i) ;

/*
    tester si un polynôme est un polynôme nul */
BOOLEEN estNul(POLYNOME p);

```

Seconde phase d'implémentation du type abstrait

On crée un fichier **implémentation** non consultable par le futur utilisateur du type.

Le fichier **détaille** l'implémentation des **opérations** de toutes du type.

Ici, ce fichier sera nommé **polynome.c**

Exemple de fichier implémentation : **polynome.c**

```
#include "polynome.h"

/*Créer un polynôme nul*/
POLYNOME Nul()
{
    POLYNOME p;
    int i;
    p = malloc(sizeof(struct un_polynome));
    if(p == NULL)
    {
        fprintf(stderr, "Allocation impossible \n");
        exit(EXIT_FAILURE);
    }
    else
    {
        p->sonDegre = 0;
        for(i=0; i<=MaxSize-1; i++) p-> sonCoefficient[i] = 0;
    }
    return p;
}
```



```

/*Construire un polynôme non nul */
POLYNOME Construire(POLYNOME p1, float a0)
{
    POLYNOME p; int i;
    p = malloc(sizeof(struct un_polynome));
    if(p== NULL)
    {
        fprintf(stderr,"Allocation impossible \n");
        exit(EXIT_FAILURE);
    }
else
    {
        if( estNul(p1))
        {
            {p->sonDegre = 0; p->sonCoefficient[0] = a0;
              for(i=1; i<=MaxSize-1; i++) p->sonCoefficient[i] = 0;
            }
        }
        else
        {
            {p->sonDegre = p1->sonDegre + 1; p->sonCoefficient[0] = a0 ;
              for(i=1; i<=p1->sonDegre+1; i++) p->sonCoefficient[i] =p1->sonCoefficient[i-1];
              for(i=p1->sonDegre+2; i<=MaxSize-1; i++) p->sonCoefficient[i] = 0;
            }
        }
    }
    return p;
}

```

```

/*Calculer le  degré d'un polynôme */
int Degre(POLYNOME p)
    {return p->sonDegre;
    }

/*Calculer le coefficient de rang i d'un polynôme */
float Coefficient(POLYNOME p, int i)
    {return p->sonCoefficient[i];
    }

/*Tester si le polynôme  est nul*/
BOOLEEN estNul(POLYNOME p)
    {int i;
    if (p->sonDegre != 0) return FAUX;
    for(i=0; i<= p-> sonDegre; i++)
        if (p-> sonCoefficient[i] != 0) return FAUX;
    return VRAI;
    }
}

```

II- Spécification d'un type abstrait

Formellement, un **type abstrait** est défini à l'aide d'un triplet:

$$(\Sigma, \Omega, E)$$

Σ : désigne un ensemble de symboles: les **sortes**,

Ω : une liste d'**opérations** avec leur signature,

E : une liste d'**axiomes** pour exprimer leurs **propriétés**

.

Exemple de spécification du type abstrait

%%Spécification du type abstrait des comptes

spec COMPTE=

%% spécification par extension sur REAL et STRING

Rat and **String** and **Boolean**

then

sort Compte *%(type abstrait des comptes)%*

%%profils ou signature des opérations du type

ops

%% constructeur du type

-créer : **Compte**;

%% modificateurs du type

_crediter : **Compte** x **Rat** →? **Compte**;

_debiter : **Compte** x **Real** →? **Compte**;

_cloturer : **Compte** →? **Compte**;

%% accesseurs du type

_titulaire : **Compte** → **String**

```

    _solde      : Compte → Rat;
    _cloturé    : Compte → Boolean;

```

%% *domaines des fonctions partielles*

∀ c1,c2: **Compte** ; s1:Rat ; p1,p2:String

- def **_crediter**(c1, s1) <=> ¬ _cloturé(c1)
- def **_debiter**(c1, s1) <=> ¬ _cloturé(c1)
- def **_cloturer**(c1) <=> ¬ _cloturé(c1)

%%définition de l'observateur **_cloturé**

- ¬ **_cloturé** (**_créer**)
- **_cloturé** (**_cloturer**(c1))

%%définition l'opération **solde**

- **_solde**(**_créer**)= 0
- **_solde**(**_cloturer**(c1)) = _solde(c1)
- **_solde**(**_crediter**(c1,s1)) = _solde(c1)+ s1
- **_solde**(**_debiter**(c1,s1)) = _solde(c1)-s1

%%définition l'opération **titulaire**

- **_titulaire**(c1)= **titulaire**(c2) => c1 = c2
- **_titulaire** (**cloturer** (c1)) = _titulaire(c1)
- **_titulaire**(**crediter**(c1,s1)) = _titulaire(c1)
- **_titulaire**(**debiter**(c1,s1)) = _titulaire(c1)

end

Exemple de composant Σ

%%Spécification du type abstrait des comptes

spec COMPTE=

%% *spécification par extension sur REAL et STRING*

Rat and **String** and **Boolean**

then

sort Compte *%(type abstrait des comptes)%*

Exemple de composant Ω

%% profils ou signature des opérations du type

ops

%% constructeur du type

-créer : **Compte**;

%% modificateurs du type

_crediter : **Compte** x Real $\rightarrow ?$ **Compte**;

_debiter : **Compte** x Real $\rightarrow ?$ **Compte**;

_cloturer : **Compte** $\rightarrow ?$ **Compte**;

%% accesseurs du type

_titulaire : **Compte** \rightarrow String

_solde : **Compte** \rightarrow Real;

_cloturé : **Compte** \rightarrow Boolean;

%% domaines des fonctions partielles

\forall c1,c2: **Compte** ; s1:Real ; p1,p2:String

- **def** **_crediter**(c1, s1) $\Leftarrow \Rightarrow \neg$ **_cloturé**(c1)
- **def** **_debiter**(c1, s1) $\Leftarrow \Rightarrow \neg$ **_cloturé**(c1)
- **def** **_cloturer**(c1) $\Leftarrow \Rightarrow \neg$ **_cloturé**(c1)

Exemple de composant E

%%définition de l'observateur **_cloturé**

- \neg **_cloturé** (**_créer**)
- **_cloturé** (**_cloturer**(c1))

%%définition l'opération **solde**

- **_solde**(**_créer**) = 0
- **_solde**(**_cloturer**(c1)) = **_solde**(c1)
- **_solde**(**_crediter**(c1,s1)) = **_solde**(c1) + s1
- **_solde**(**_debiter**(c1,s1)) = **_solde**(c1) - s1

%%définition l'opération **titulaire**

- **_titulaire**(c1) = **titulaire**(c2) \Rightarrow c1 = c2
- **_titulaire** (**cloturer** (c1)) = **_titulaire**(c1)
- **_titulaire**(**crediter**(c1,s1)) = **_titulaire**(c1)
- **_titulaire**(**debiter**(c1,s1)) = **_titulaire**(c1)

end

Qu'est-ce qu'une sorte ?

Une **sorte** (**sort** en CASL) est un symbole qui désigne un **type abstrait** de données, à savoir:

- un ensemble d'**objets**,
- muni d'**opérations** pour manipuler les objets

Le terme **sorte** est introduit pour éviter la **confusion** avec la notion de **type** (utilisé en programmation).

Le but d'une **spécification** est de définir une **sorte**.

Exemple

Soit l'en-tête de la spécification type abstrait des polynômes.

```
%%Spécification du type abstrait des polynômes%%
```

```
spec POLYNOME =
```

```
  Int and Rat and Boolean  (% réutilise Int , Rat et Boolean %)
```

```
  then
```

```
  sort Polynome
```

Comment le lire ?

POLYNOME : nom du **module** de spécification.

Le **module** de spécification appelé **POLYNOME** définit la **sorte** :

Polynome

Polynome : désigne le **type abstrait** des polynômes .

1-Signature d'un type abstrait

Le couple :

$$(\Sigma, \Omega)$$

définit la **signature** du type abstrait

La signature est la **partie visible** ou **interface** d'une spécification.

Comment construire une signature ?

La **signature** d'un type abstrait est construite en deux parties:

1-**en-tête** : une liste de **sortes** qui sont **définies** ou simplement **réutilisées** par la spécification

2- liste de **signatures** d'opérations.

Exemple de signature du type COMPTE

1-En-tête : liste de **sortes**

```
%%Spécification du type abstrait des comptes
spec COMPTE=
%% spécification par extension sur REAL et STRING
    Rat and String and Boolean

    then
    sort Compte           %(type abstrait des comptes)%
```

2- liste de **signatures** d'opérations

%% profils ou signature des opérations du type

Ops

%% constructeur du type

-créer : **Compte**;

%% modificateurs du type

_crediter : **Compte** x Real \rightarrow ? **Compte**;

_debiter : **Compte** x Real \rightarrow ? **Compte**;

_cloturer : **Compte** \rightarrow ? **Compte**;

%% accesseurs du type

_titulaire : **Compte** \rightarrow String

_solde : **Compte** \rightarrow Real;

_cloturé : **Compte** \rightarrow Boolean;

%% domaines des fonctions partielles

\forall c1,c2: **Compte** ; s1:Real ; p1,p2:Nom

- **def** **_crediter**(c1, s1) \Leftarrow \neg **_cloture**(c1)
- **def** **_debiter**(c1, s1) \Leftarrow \neg **_cloture**(c1)
- **def** **_cloturer**(c1) \Leftarrow \neg **_cloture**(c1)

Signature d'opération

La **signature** d'une opération précise:

- les **sortes** de ses arguments,
- la **sorte** de son résultat.

Il prend la forme suivante:

$$F: A_1 \times A_2 \times \dots \times A_n \rightarrow A$$

F : désigne l'opération

A_1, \dots, A_n : sortes des **arguments**,

\rightarrow : flèche symbolisant l'**application**

$\rightarrow ?$: signifie que la fonction **F** est **partielle**.

A : sorte du **résultat** ,

x : opérateur **produit cartésien** (impose un ordre)

Les trois cas des opérations

1-Cas d'opérations sans argument

_créer: **Compte**;

2-Cas d'opérations partielles

_crediter : **Compte** x Rat \rightarrow ? **Compte**;
_cloturer : **Compte** \rightarrow ? **Compte** ;

3-Cas d'opérations totales:

_solde	: Compte \rightarrow Rat;
_titulaire	: Compte \rightarrow String

Classification des sortes

Une sorte est dite **définie** s'il s'agit d'une **nouvelle sorte** que la spécification en cours est censée définir.

Une sorte est dite **exportée** s'il s'agit d'une **sorte** :

- **prédéfinie**
- et **réutilisée** dans la spécification courante.

Soit la signature du type abstrait **Vente** :

```
spec VENTE =  
  AGENCE  
  then  
  sort Vente  
  ops  
  vendre : Vehicule x Client → Vente;  
  vendu-le : Vente → Date;  
  ...
```

Dans cet exemple, la seule sorte **définie** par la spécification **VENTE** est la sorte :

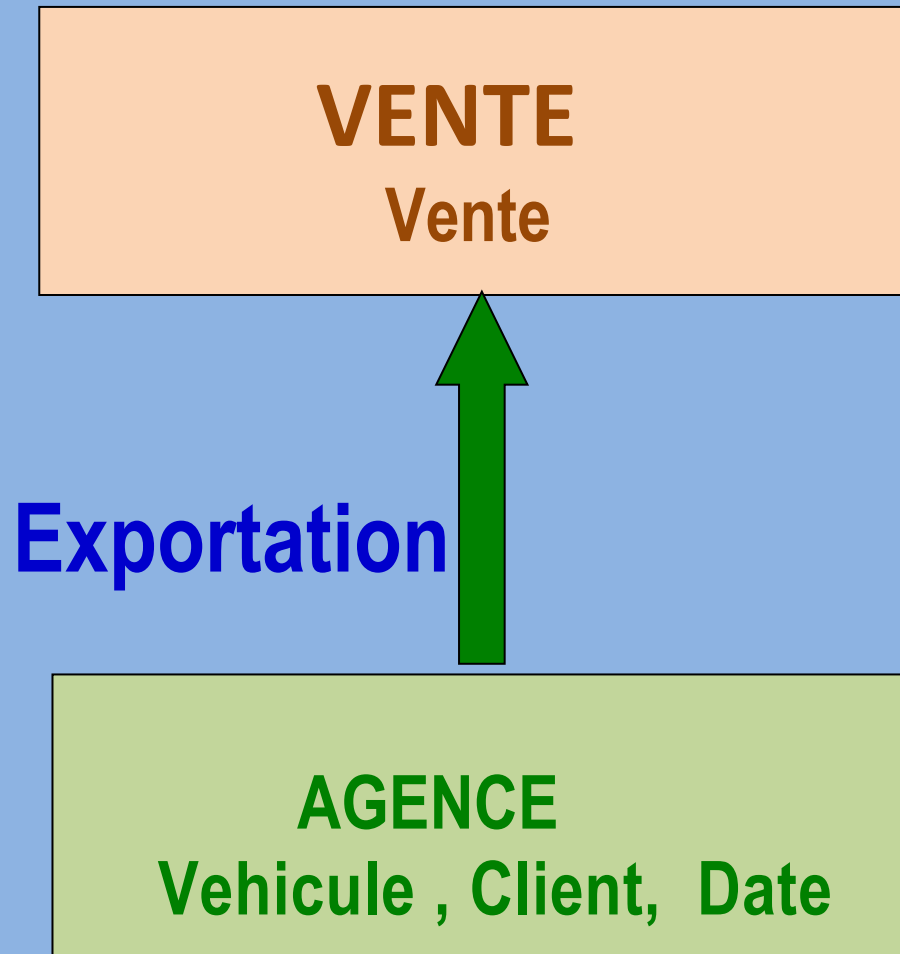
Vente.

Toutes les autres sortes, à savoir:

Vehicule , Client, Date

sont des sortes **réutilisées**.

Elles sont **exportées** par la spécification **AGENCE**.



Classification des opérations

On dira qu'une opération est un **constructeur** si :

- elle **retourne** un résultat
- résultat de **sorte définie** par la spécification courante.

Sa signature prend la forme suivante :

$$F: A_1 \times A_2 \times \dots \times A_n \longrightarrow A$$

A_1, \dots, A_n : sortes des arguments

A : sorte **définie**

Un constructeur peut être en **mode constructif**: il **crée** un nouvel objet:

```
%%profils ou signature des opérations du type  
ops  
%% constructeur du type  
_créer: Compte;
```

L'opération est sans argument.

Un constructeur peut être également en **mode mutatif**:

- **modifie l'état** d'un objet existant
- et **créer** donc un nouvel objet avec le nouvel **état**.

%% modificateurs du type

_crediter : **Compte** x Rat \rightarrow ? **Compte**;
_debiter : **Compte** x Rat \rightarrow ? **Compte**;
_cloturer : **Compte** \rightarrow ? **Compte**;

A retenir ! :

Contrainte de génération

Tout objet d'une sorte définie **doit être** le résultat de l'application d'un certain **constructeur**.

Cette contrainte est appelée **contrainte de génération**.

Accesseurs

Un **accesseur** est une opération du type qui permet :

- d'**observer** l'**état** d'un objet du type,
- sans y apporter une quelconque modification.

Une opération est un **accesseur** si et seulement si:

- elle a au moins un **argument** de sorte **définie**,
- elle rend un **résultat** de sorte **exportée**.

Sa signature prend la forme suivante :

$$F: A_1 \times A_2 \times \dots \times A_n \longrightarrow A$$

Au moins l'une des sortes A_1, \dots, A_n est une sorte définie.

A : est une sorte **exportée**.

C'est le cas des opérations suivantes:

```
_solde           : Compte -> Rat;  
_titulaire      : Compte -> String ;
```

Précondition

Dans certains cas, les opérations du type sont des **fonctions partielles**.

Le domaine de définition d'une opération partielle est spécifié à l'aide d'un **axiome**.

Cet **axiome** est appelé **précondition**.

Pour revenir à l'exemple de la sorte **Compte**, on peut alors remarquer que l'opération:

_créditer(**c** , s)

est déclarée dans la signature comme suit:

crediter: Compte x Rat \rightarrow ? Compte;

Le symbole ? signifie qu'elle n'est pas définie partout.

Elle n'est définie que **si et seulement si** le compte **C** n'est pas **cloturé**.

Ce qui est exprimé par la **précondition**:

def _créditer(**C**, s) \Leftrightarrow _cloture(**C**) = False

La précondition définit le **domaine** de l'opération

_créditer()

Ainsi, si dans la signature du type **Compte**, on a :

%% constructeurs du type

_crediter : **Compte** x Rat \rightarrow ? **Compte**;

_debiter : **Compte** x Rat \rightarrow ? **Compte**;

_cloturer : **Compte** \rightarrow ? **Compte**;

On doit avoir obligatoirement :

%% domaines des fonctions partielles

\forall c1: **Compte** ; s1:Rat

- **def** **_crediter**(c1, s1) \Leftrightarrow est_cloturé(c1) = False
- **def** **_debiter**(c1, s1) \Leftrightarrow est_cloturé(c1) = False
- **def** **_cloturer**(c1) \Leftrightarrow est_cloturé(c1) = False

III- Réutilisation et hiérarchie entre les types

Il est important de **structurer la définition** des types abstraits en se donnant la possibilité de:

- **réutilisation**
- et d'**extension**.

Le mécanisme de **réutilisation** consiste à **réutiliser** les éléments(sortes , opérations) définis par d'autres types.

Le mécanisme d'**extension** donne la possibilité de construire un type «**au-dessus** » d'autres types :

- **prédéfinis**,
- ou fournis par une **librairie**.

1-La réutilisation dans les types abstraits

La notion de **réutilisation** dans les types abstraits est un mécanisme qui rend possible:

- la **définition** de **sortes** et d'**opérations** dans certains types abstraits,
- et leur **utilisation**, ensuite, par d'autres.

Soit la signature en **Casl** de la spécification appelée **INVENTAIRE**,
définie comme suit:

```
from lib/Numbers get Nat
from lib/CharactersAndStrings get String

spec INVENTAIRE =
  String and Nat
then
  sorts Vehicule , Client ;

  . . .
```


Elle signifie la **réutilisation** des types abstraits :

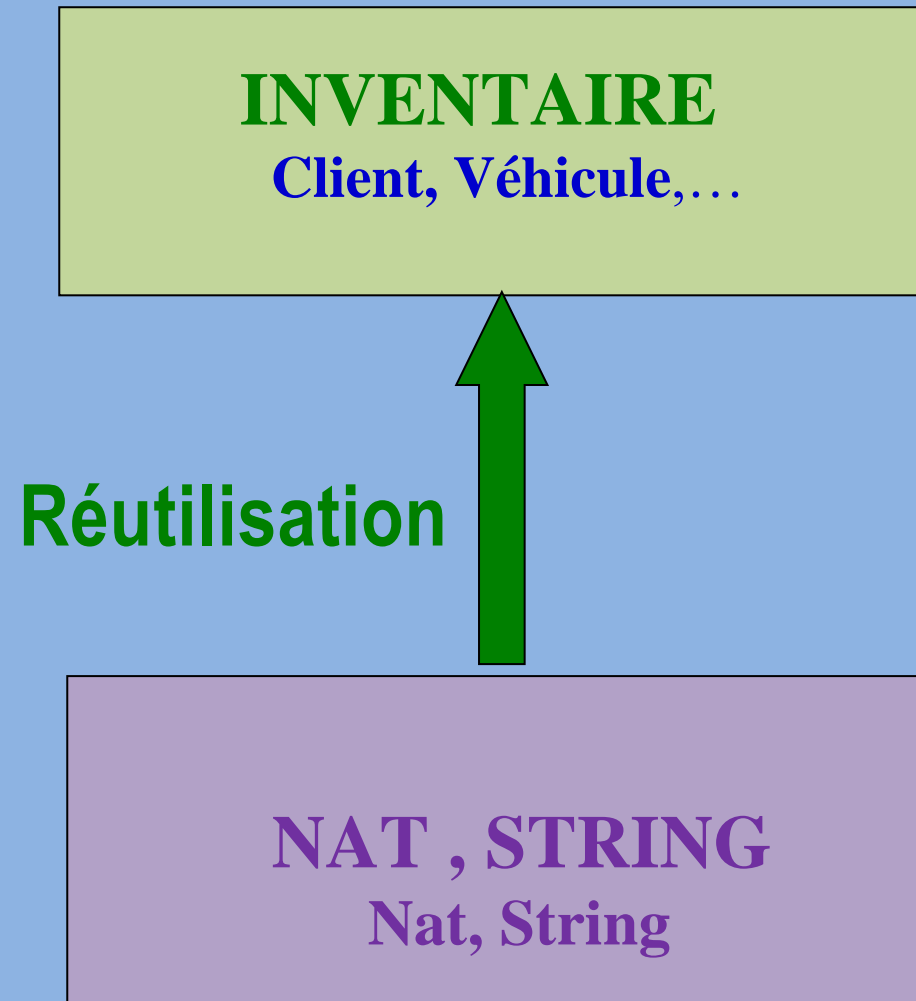
Nat et **String**

Lesquelles sont **exportées** par des librairies standards :

lib/Numbers

lib/CharactersAndStrings

Les sortes **Véhicule** et **Client** sont spécifiées en **réutilisant** les sortes **Nat** et **String**



Soit la signature du type abstrait **Vente**, définie comme suit:

```
spec VENTE =
```

```
INVENTAIRE
```

```
then
```

```
sort Vente
```

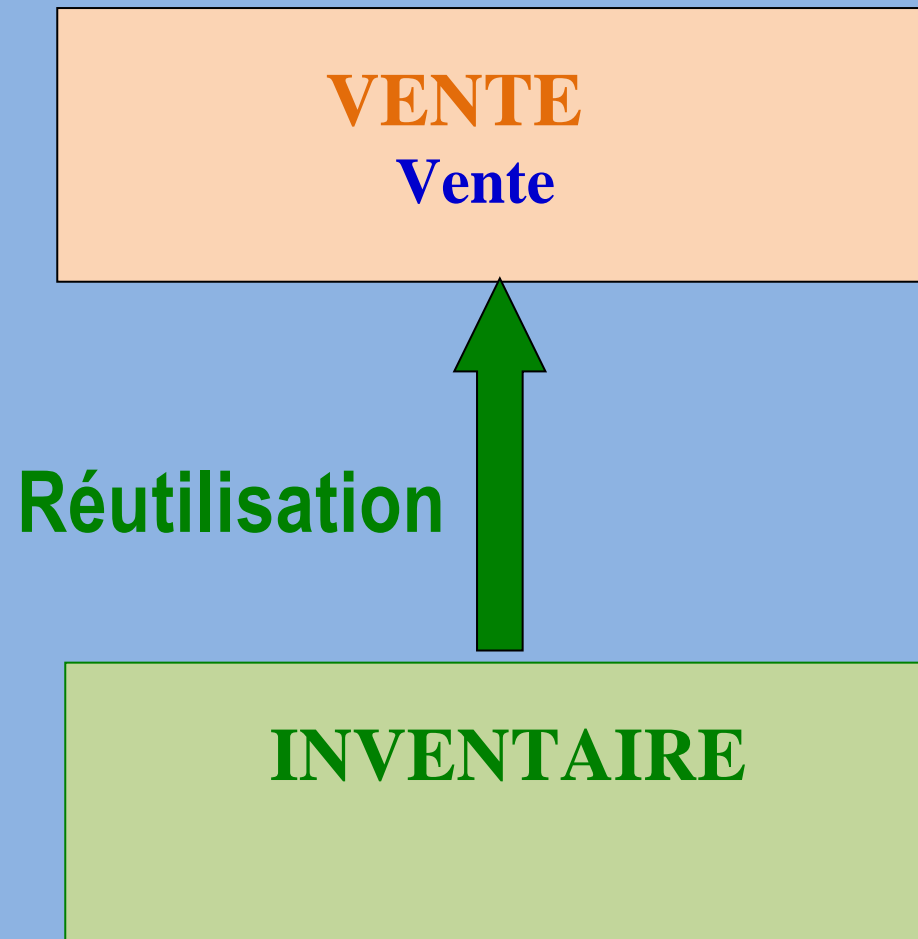
```
ops
```

```
vendre : Vehicule x Client  $\rightarrow$  Vente;
```

```
vendu_a: Vente x Véhicule  $\rightarrow$  Client;
```

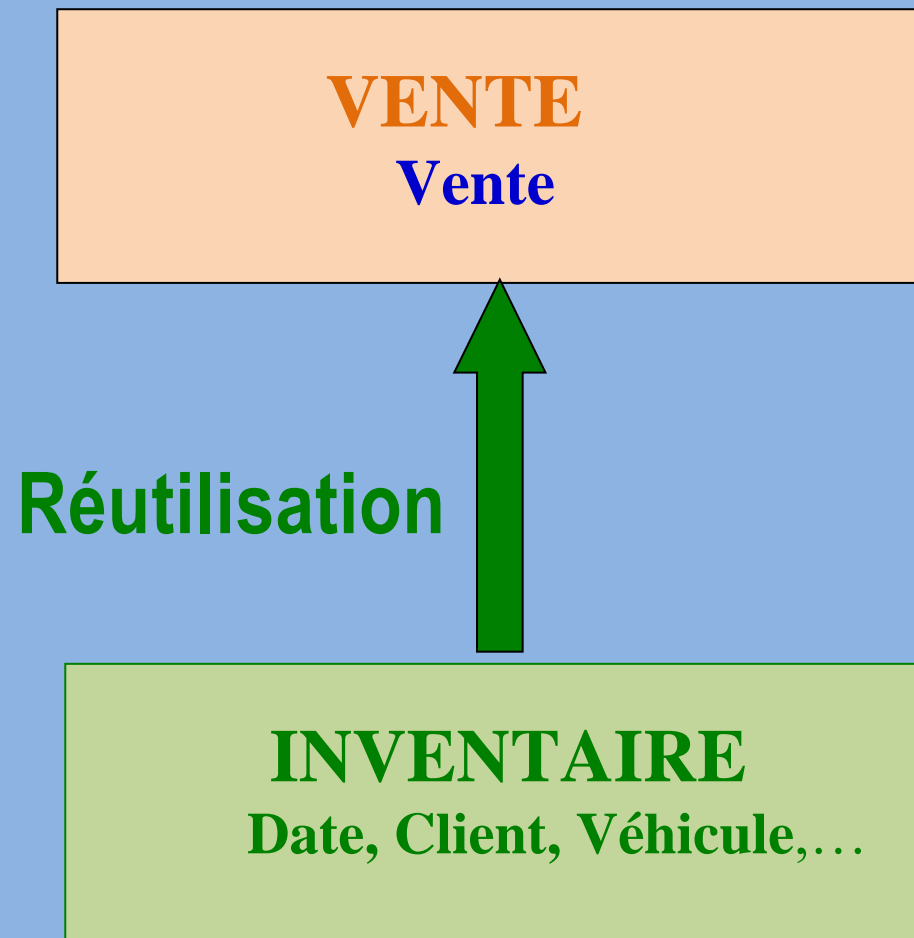
```
vendu_le: Vente x Véhicule  $\rightarrow$  Date;
```

Elle signifie la **réutilisation** des éléments définis par la spécification **INVENTAIRE**.



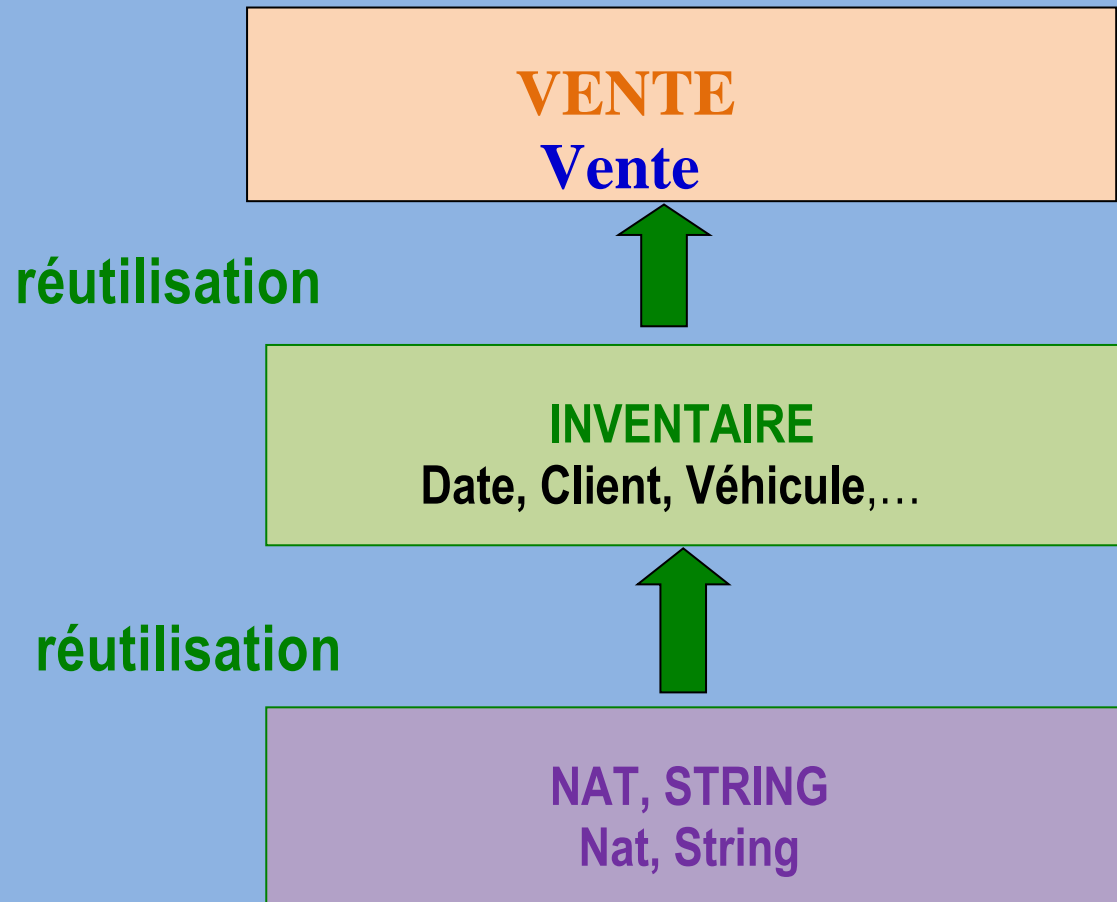
...et donc, par **héritage**, la réutilisation des sortes :

Date, Client, Véhicule,...



...et donc, par **héritage**, la réutilisation des sortes :

Nat, String



2-La hiérarchie entre les types

La notion de **hiérarchie** entre les types est fondamentale pour **structurer** la construction de types abstraits.

Par un mécanisme d'**extension**, un type abstrait est construit de façon **incrémentale** «**au-dessus**» de types abstraits **exportés**.

Ainsi des spécifications très complexes sont construites de façon :

- **structurée**
- et **incrémentale**

à partir de spécifications **moins complexes**.

Soit, par exemple, les spécifications **POLYNOME0** et **POLYNOME1** dont les signatures sont définies comme suit :

Spécification **POLYNOME0**

```
spec POLYNOME0 =  
    Int and Rat and Boolean  
  then  
    %% spécification par extension sur Nat et Rat  
    sort Polynome  
    Nul : Polynome  
    Construire : Polynome x Rat  $\rightarrow$  Polynome  
  end
```

Spécification **POLYNOME1**

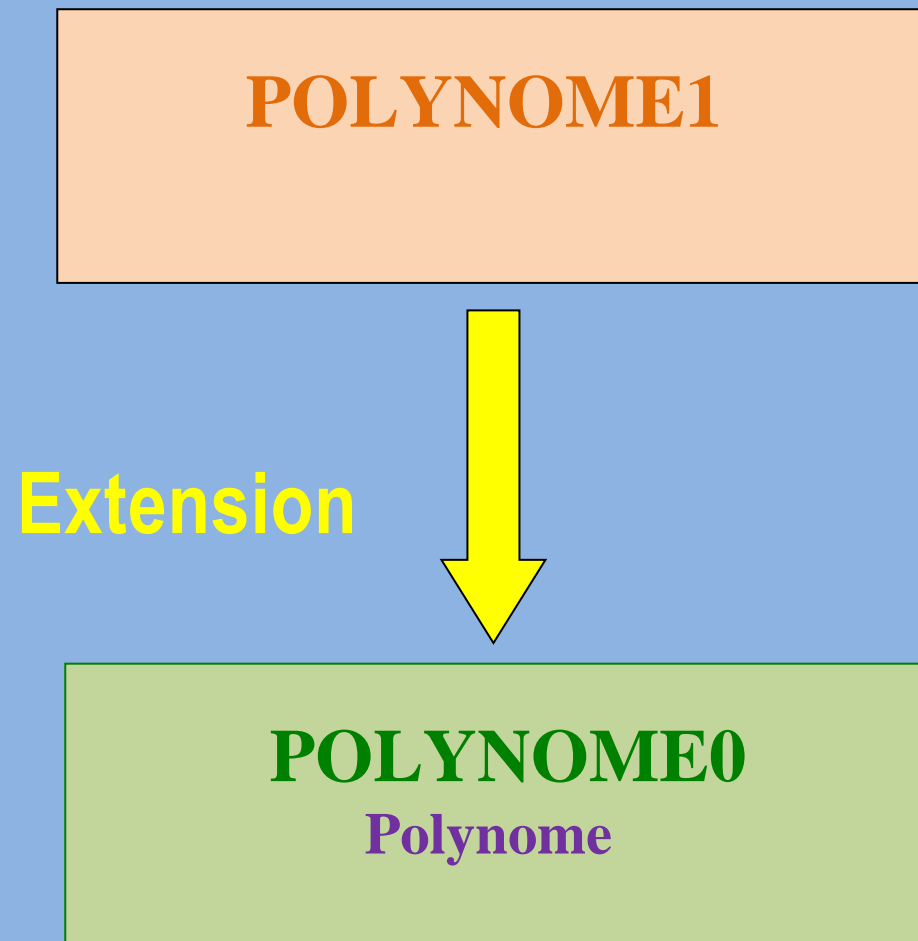
```
spec POLYNOME1 =  
    POLYNOME0  
    then  
    ops  %(on ajoute les accesseurs)%  
        estNul:      Polynome      → Boolean;  
        Degree:      Polynome      → Int  
        Coefficient: Polynome * Int → ? Rat  
        ...  
    end
```

Ces deux signatures expriment **explicitement** une **hiérarchie** établie entre les deux spécifications :

- **POLYNOME0**, d'une part,
- et **POLYNOME1** d'autre part.

Dans cette hiérarchie, le type **POLYNOME1** est "**au-dessus**" du type **POLYNOME0**

On dit que la spécification **POLYNOME1** est construite par **extension** du type **POLYNOME0**.



Vue extensive sur la spécification **POLYNOME1**

```
spec POLYNOME1 =
```

```
    Int and Rat and Boolean
```

```
then
```

```
%% spécification par extension sur Nat et Rat
```

```
sort Polynome
```

```
Nul : Polynome
```

```
Construire : Polynome x Rat → Polynome
```

```
then
```

```
ops %(les accesseurs )%
```

```
    estNul: Polynome → Boolean;
```

```
    Degre: Polynome → Int
```

```
    Coefficient: Polynome * Int →? Rat
```

```
    ...
```

```
end
```