Programmation par objets

http://lsewww.epfl.ch/~poo

Historique des langages (à objets

Rachid Guerraoui

Laboratoire de Systèmes d'Exploitation (LSE)
Département d'Informatique
INF 239 - tél. 5272

Rachid.Guerraoui@epfl.ch

Partie I - Smalltalk

- **Concepts de base**
  programmes (expressions) = objets + messages
- **Objets**
- **Messages & méthodes**
- **Classes et instances**
- **Méta-classes et reflexivité**
- **Classes du système**

Partie II - Java

- **Classes et instances**
- **Classes et héritage**
- **Encapsulation et Masquage**
- **Exceptions**
- **Classes du système**
- **Réflexivité**

Références

- Support
  - **Copie des transparents**
- Cours basé sur
  - **Smalltalk-80: the language and its implementation**
    Adele Goldberg & David Robson, Addison-Wesley (1985)
  - **Java in a Nutshell**
    David Flanagan, O'Reilly (1997)
  - **Object-oriented analysis and design (with applications)**
    Grady Booch, Ben. Cummings (1994)
- Introduction à Smalltalk et aux objets
  - **Smalltalk-80**
    Philippe Dugerdil, PPUR (1990)

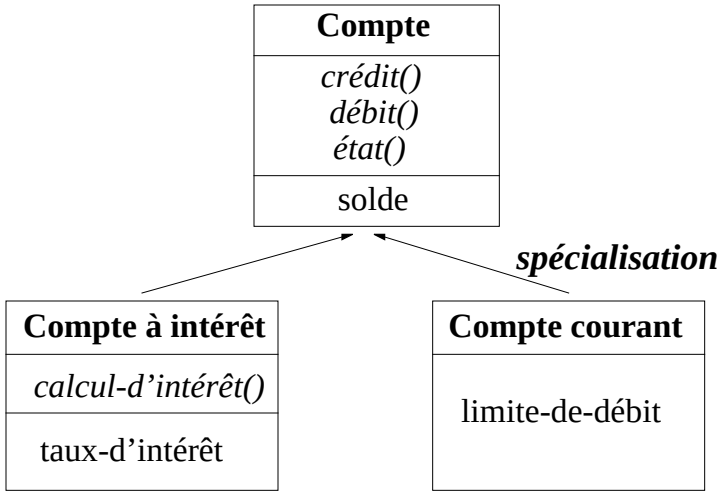
Pourquoi: Objets - Smalltalk - J

- Les nouveaux projets informatiques se font dans des langages à objets (C++, Java)
- Mieux programmer avec des objets conduit à mieux programmer avec n'importe quel langage
- En Smalltalk, on ne peut programmer qu'avec des objets
- (1) Java est à la mode; (2) Java est un langage intéressant

Programmation par objets

- Programmer par objets signifie développer des ***abstractions*** qui modélisent le problème à résoudre
- On peut voir un objet comme un ***serveur***: la mise en oeuvre d'un serveur nécessite des ***opérations*** et (parfois) des ***données***
- L'abstraction: de la carte perforée à la station de travail avec écran graphique

Des objets comptes bancaires



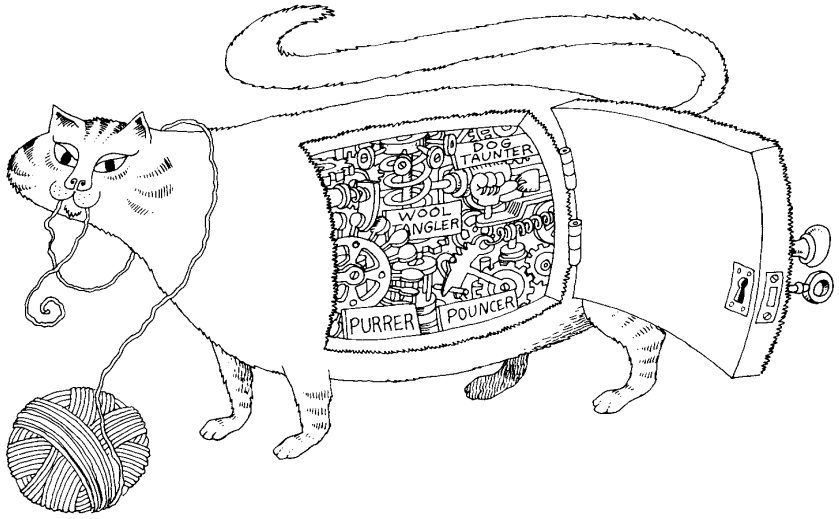
Programmation vs langage à objets

- On peut faire de la programmation par objets dans n'importe quel langage de programmation
- Un langage à objets offre des mécanismes qui facilitent la programmation par objets (mécanismes d'encapsulation, de classification, d'héritage, etc.)

Langages à objets

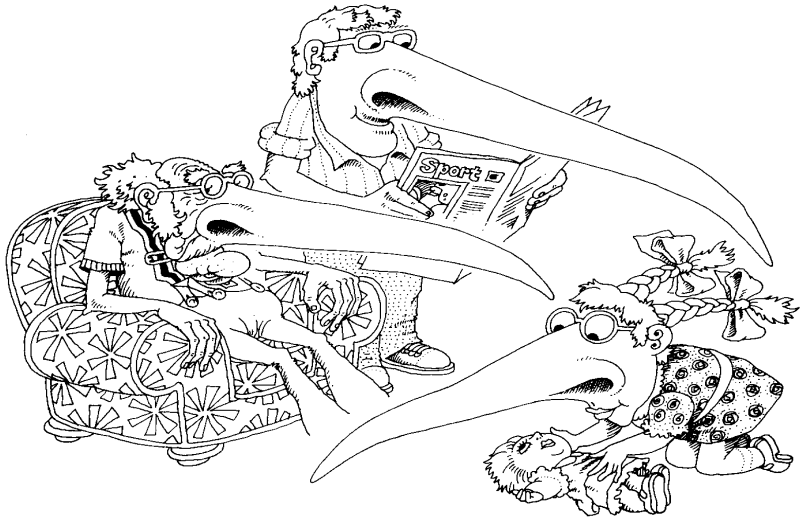
- Un objet renferme son état et ses opérations ***(encapsulation)***
- Différents objets receveurs d'un même message peuvent répondre différement ***(polymorphisme)***
- Le code à exécuter en réponse à un message est déterminé dynamiquement ***(liaison différée)***
- Les objets sont créés à partir d'un moule ***(les classes)***
- Une classe est construite par ***héritage*** d'une autre classe

Encapsulation



L'encapsulation cache les détails de l'implémentation d´un objet

Héritage



Une sous-classe peut hériter de la structure et du comportement de sa s

Bref historique des langages

- Les premiers programmes étaient écrits en langage machine et dépendaient étroitement des ordinateurs sur lesquels ils étaient mis en oeuvre

- L'évolution s'est traduite par la séparation de plus en plus nette entre les concepts manipulés dans les programmes et leurs représentations interne en machine

Historique: Fortran ~60

- Fortran est le premier langage à avoir séparé la représentation des données dans un programme de leur représentation en machine

- Les structures de contrôle d'un programme sont néanmoins calquées sur les structures de contrôle de la machine

Historique: Algo 63 - Pascal

Programme = algorithmes + structures de données

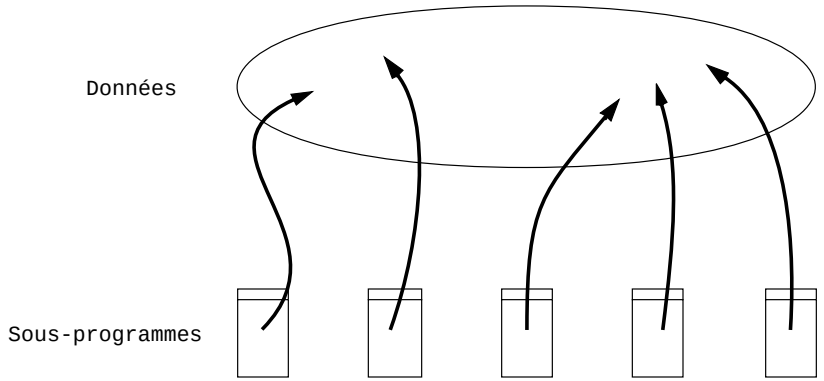
- Algol fut le premier langage à avoir séparé le modèle de contrôle du programme du modèle de contrôle de la machine

Programmation dirigée par les traitements

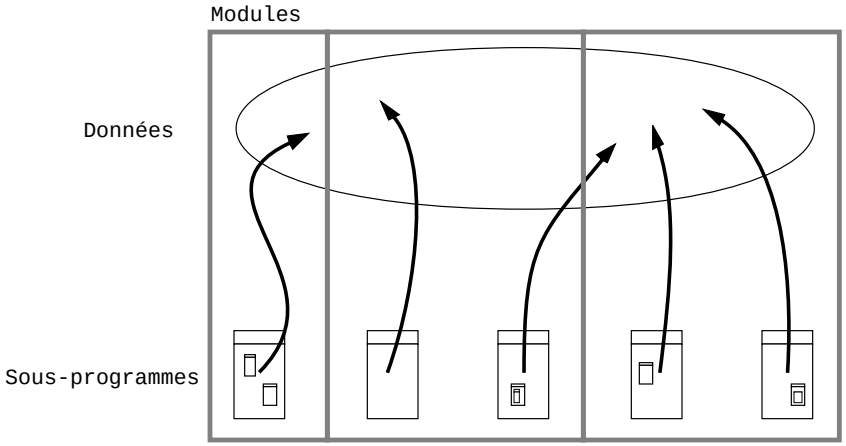
- Pascal: après la description des données, la tâche à réaliser est décomposée en un ensemble de procédures séparées

NB. Les données et les opérations sont séparées: problème lors de l'évolution des données

Historique: approche procédurale



Historique: approche procédurale (1



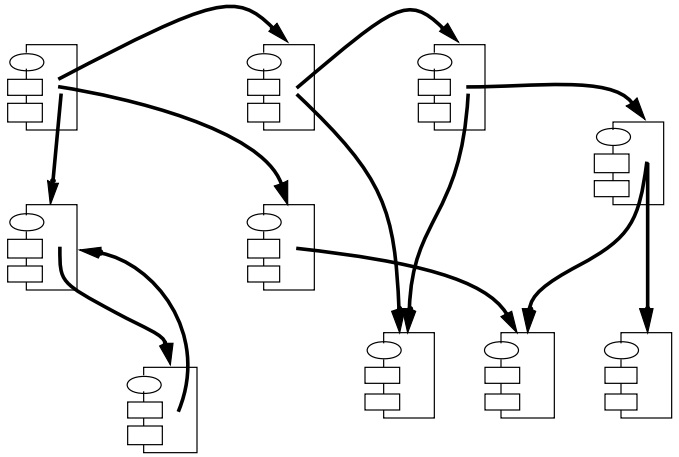
Historique: Objets

Programmation dirigée par les données et les traitements

- La tâche à réaliser est décomposée en un ensemble d'objets
- Idée fondamentale: diviser (en sous-programmes autonomes) pour mieux faire évoluer et réutiliser

Objet = données + traitements

Historique: approche orientée obje



Historique: Simula~70

- Ecole scandinave (Norwegian Computer Center, Oslo)

  Simula - un langage de simulation (Ole-Johan Dahl et Kristen Nygaard, 1966) - les programmes doivent refléter les objets du monde réel: un objet du monde réel est représenté par un objet informatique

- A partir de Simula, le langage Simula 67 a été créé comme une extension d'Algol 60 (classes, sous-classes, procédures virtuelles, garbage collector)

  Influences sur CLOS et Smalltalk

Historique: Smalltalk~72

Nous l'avons appelé Smalltalk pour que personne n'en attende rien - A. Kay

(''Smalltalk'' ~ ''Banalités'')

- Xerox: Palo Alto (Californie)

  Un langage pour programmer des interfaces graphiques

- -> Smalltalk-80 -> Macintosh -> Windows 95
- -> Objective C (B. Cox), Eiffel (B. Meyer)
- -> C++ dont le (seul) mérite a été de faire admettre qu'un langage à objets n'était pas forcément inefficace (Stroustrup 86 - AT&T)

Caractéristiques de Smalltalk

- Tout est objet (aspect uniforme): l'éditeur, le gestionnaire de la souris, l'écran, etc. sont des objets
- Les bons programmeurs Smalltalk sont des bons programmeurs C++, Java, ...
- En Smalltalk on ne peut programmer qu'avec des objets

Java

- Sun Microsystem (Java Soft)

  Le projet Oak (1990) - Bill Joy: un langage de programmation pour la télevision interactive - le langage C++ minus minus

  Oak a été abandonné en 1992

- Sun (Joy et Gosling) décida une nouvelle stratégie: un langage pour le web & un nouveau nom - Java -

  Netscape intégra une machine virtuelle Java

  Microsoft adopta Java comme langage pour ses produits Internet

Caractéristiques de Java

- La syntaxe ressemble à celle de C++, et la sémantique à celle de Smalltalk
- Java est à la mode
- Java contient des aspects originaux pour traiter la sécurité

Les objets partout

Langages: C++, Eiffel, CLOS, ADA95, Java

Bases de données à objets: Gemstone, ~Oracle

Systèmes répartis à objets: CORBA

Télécommunications: TINA

Les Concepts

à travers

Smalltalk

2 + 3 Concepts (vocabulaire)

- **Objet**
- **Message**
- Classe
- Instance
- Méthode

Programme = Objets + Messages

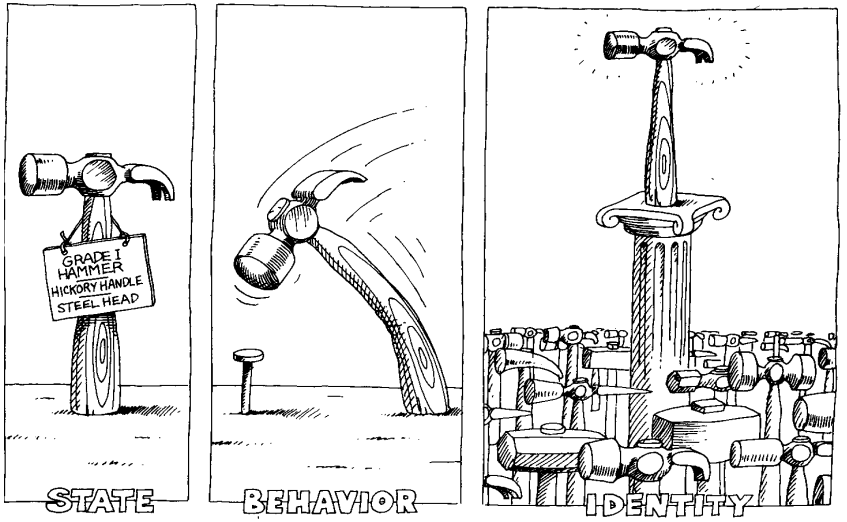
Une application est un ensemble d'objets interagissant par envois de messages

- Americain Airlines (2300 vols/jour)
- 2 + 3
- L'environnement Smalltalk

Objet

- Un objet est un serveur
- Les objets sont les nombres, les chaines de caractères, les dictionnaires, les fichiers, les comptes bancaires, le compilateur, p.ex., un objet représentant un nombre permet d'effectuer des opérations arithmétiques
- Un objet est mis en oeuvre par des données et un ensemble d'opérations

Objet



Un objet a un état, un comportement défini et une identité unique

Objet

- Un dictionnaire permet d'associer un nom à une valeur et de trouver la valeur associée à un nom particulier
- Un compte bancaire permet de stocker le solde d'un compte, puis de le débiter et de le créditer
- Penser à une utilisation ultérieure de l'objet: un dictionnaire qui permet aussi de détruire des associations est plus utile qu'un dictionnaire qui ne permet que de rajouter des associations
- Les objets inutilisés sont désalloués par le système sous-jacent, sans intervention du programmeur

Message

Un message est une requête pour un objet (receveur) de rendre un service, i.e., d'exécuter l'une de ses opérations

- Un message spécifie l'opération désirée mais ne dit rien sur la manière dont cette opération est mise en oeuvre (quoi vs. comment)

a + b

Objet & Message

Un objet peut accepter un certain nombre de messages (interface de l'objet)

- L'interface décrit les services fournis par l'objet
- Les données d'un objet ne peuvent être manipulées qu'à travers l'envoi de messages. De manière plus générale: la mise en oeuvre d'un objet n'est pas visible à d'autres objets

Classe & Instance

Une classe représente un moule d'objet (une usine)

- Une classe décrit la mise en oeuvre d'un ensemble d'objets qui représentent tous la même sorte de serveur
- Les objets d'une classe sont appelés ***instances*** de la classe
- Tout objet est instance d'une classe

Classe & Instance

Programmer consiste à définir des classes, créer des objets et envoyer des messages.

- Toutes les instances d'une même classe répondent aux mêmes messages.
- Les données d'une instance sont stockées dans des ***variables d'instance***: une variable d'instance est une référence à un objet (de type quelconque - langage non typé)
- Toutes les instances d'une même classe possèdent les mêmes variables d'instance.
- Deux instances d'une même classe peuvent avoir des valeurs différentes pour leurs variables d'instance. Elles peuvent donc avoir des états différents.

Méthode

- Les opérations d'un objet sont appelées ***méthodes***
- Une méthode décrit comment l'objet répond à un message
- Une méthode peut accéder aux variables d'instance
- Une méthode retourne un objet

Méthode

- Différents receveurs d'un même message peuvent répondre différement ***(polymorphisme)***
- Une variable d'instance peut désigner des objets de différentes classes ***(polymorphisme)***
- Le code de la méthode à exécuter en réponse à un message est déterminé dynamiquement ***(liaison différée)***

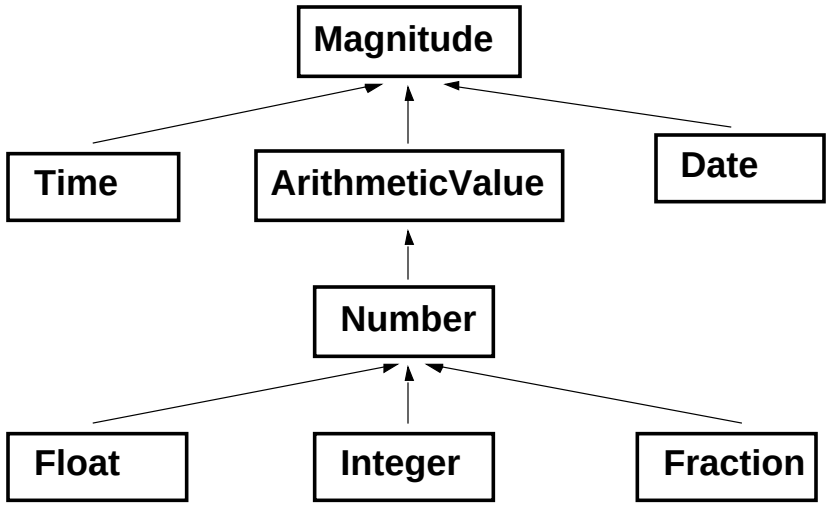
Classe (Héritage)

- Une classe peut être construite par ***héritage*** d'une autre classe: Les variables d'instance et les méthodes sont automatiquement héritées
- L'héritage peut correspondre à une extension ou à une une restriction d'une autre classe
- On parle de ***super-classe*** et de ***sous-classe***

Les classes du système

- Arithmétiques
- Structures de données (pour seule fonctionnalité de représenter des données) : différentes sortes de collections (p.ex., Dictionnaires)
- Structures de contrôle (''if-then-else'')
- Compilateur, Ecran, Fichier, etc..

Les classes arithmétiques



Résumé (Terminologie)

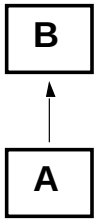
- ***Objet***: un composant du système Smalltalk-80 contenant des données et des opérations
- ***Message***: une requête à un objet pour exécuter l'une de ses méthodes
- Receveur: l'objet destinataire d'un message
- Interface: les messages auxquels un objet peut répondre

Résumé (Terminologie)

- ***Classe***: une description d'un groupe d'objets similaires
- Variable d'***instance***: une partie de la mémoire privée d'un objet
- ***Méthode***: la description de la manière dont un objet exécute ses opérations

Résumé (Terminologie)

- La classe A hérite de la classe B
  - A est une ***sous-classe*** de B
  - B est la ***super-classe*** de A



- Classes du système: l'ensemble des classes qui constituent le système Smalltalk

Résumé (Terminologie)

- Une classe est dite ***abstraite*** lorsque ses méthodes ne sont pas toutes mises en oeuvres
- Une classe ***abstraite*** ne peut pas générer d'instances (elle est utilisée principalement pour la structure)
- Une classe qui n'est pas abstraite est dite ***concrète***

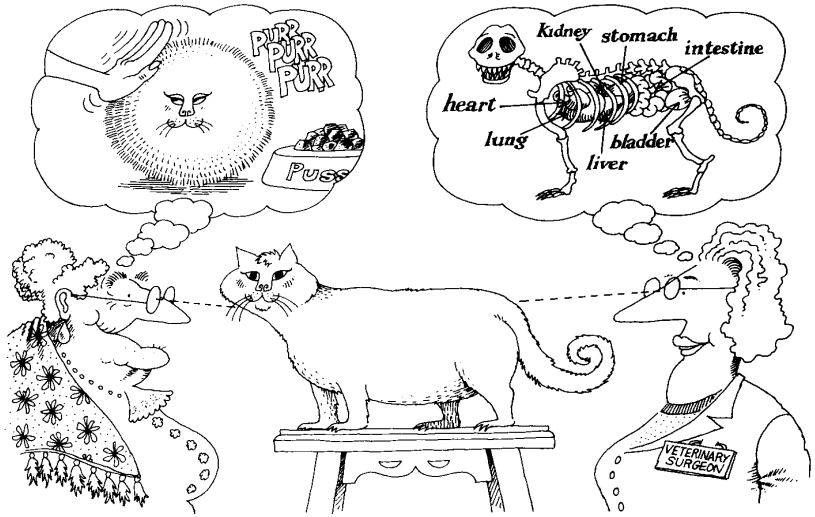
Avantages (B. Meyer)

- **Abstraction**
- **Modularité**
- **Réutilisabilité**
- **Lisibilité**

Abstraction

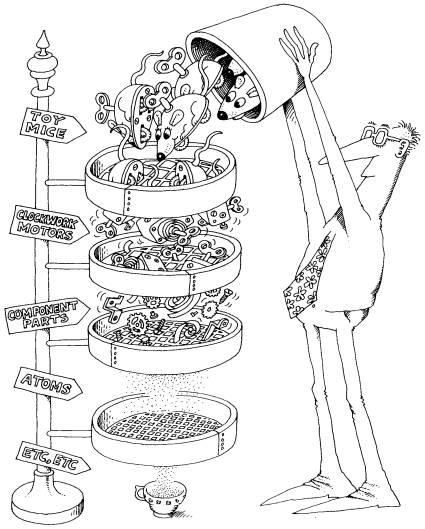
Nul besoin de connaitre la mise en oeuvre et la représentation interne d'un objet pour l'utiliser, i.e., pour lui adresser un message. On n'a pas besoin de connaitre les détails de mise en oeuvre de l'objet.

Abstraction



L'abstraction permet à un observateur de se concentrer sur les caractéri essentielles d'un objet

Abstraction



Les abstractions forment une hierarchie

Modularité

L'application est structurée en un ensemble d'objets dont il est facile de changer la mise en oeuvre avec un minimum d'impact sur les autres objets.

Modularité

La modularité consiste à découper des abstractions en composants

Réutilisabilité

Un objet est défini par un comportement grâce à une interface explicite. Il est facile de l'inclure dans une bibliothèque que tout programmeur peut ensuite utiliser (soit pour construire des objets de même sorte, soit pour construire des objets plus spécifiques par spécialisation ou composition d'objets existants)

Lisibilité

L'encapsulation, la possibilité de surcharge et la modularité renforcent la lisibilité des programmes. Les détails de mise en oeuvre sont cachés et les noms des méthodes peuvent être les plus naturels possible. Les interfaces constituent autant de modes d'emploi précis et détaillés des objets.

Objets et Messages

Syntaxe (Smalltalk)

2 + 3 Concepts (vocabulaire)

- **Objet:** les composant d'un programme sont les objets
- **Message: les objets interagissent par envois de messages**
- Classe: un moule d'objets avec mêmes méthodes et variables
- Instance: les objets sont des instances de classes
- Méthode: en recevant un message, un objet exécute une méthode

2 + 3 Concepts (vocabulaire)

- **Objet**
- **Message**
- Classe
- Instance
- Méthode

Expression

- L'expression est l'unité de programme (dans l'espace de travail Smalltalk ou dans le corps d'une méthode)
- Une expression est une séquence de caractères qui décrit un objet (de la classe String)
- Une expression légale est formée d'expressions élémentaires légales

Syntaxe (contenu d'une expression)

Expressions élémentaires légales:

- *Constantes*
  Nombres, caractères (chaines de), symboles, et tableaux.
- *Variables* (dépendantes du contexte)
  Conteneurs d'objets
- *Messages*
  Moyen d'interaction entre objets
- *Blocs*
  Séquence d'expressions (activité différée)

Constantes

- Nombres : objets qui représentent une valeur numérique et qui répondent (entre autres) à des messages de calculs de résultats mathématiques

```
3
30.45
-3
```

- Caractères: objets qui représentent des symboles de l'alphabet

```
$a
$M
$$
```


Constantes

- Chaines de caractères: objets qui répondent à des messages d'accès à des caractères, de remplacement de sous-chaines, de concatenation de chaines, etc..

```
'bonjour'
'can''t'
```

- Symboles: objets qui représentent des chaines de caractères utilisées pour les noms du système

```
#rouge
#007
#+
```


Constantes

- Tableaux: objets qui représentent des structures de données dont le contenu peut être référencé par un entier de 1 à un nombre représentant la longueur du tableau

```
#(1 2 3)
#($a $b $c (1 2 3) ())
#(('u' 'n') 'deux' ($t $r $o $i $s)
```

Les tableaux répondent à des messages pour accéder à un élément (n-ième), pour changer un élément, etc.

Variables

- Une variable désigne un objet
  - Les variables privées ne sont accessibles qu'à un seul objet
  - Les variables partagées sont accessibles à plusieurs objets

- Le nom d'une variable commence toujours par une lettre
  - privée (locale) -> minuscule
  - partagée (globale) -> majuscule

```
a
index
Rectangle
```


Variables

- Les données d'un objet (d'une classe) sont stockés dans des variables (variables d'instance)

- Le contenu des variables d'instance d'un objet ne peut être accédés que depuis les méthodes de cet objet

Variables

- L'affectation permet de changer le contenu d'une variable

```
index := 3
index := index + 1
index := indexInitial := 1
```


Pseudo-variables

- Une pseudo-variable référence un objet qui ne peut être modifié par affectation

- Certaines pseudo-variables du système sont constantes
  - `nil`
  - `true`
  - `false`
- D'autres pseudo-variables dépendent du contexte
  - `self`
  - `super`

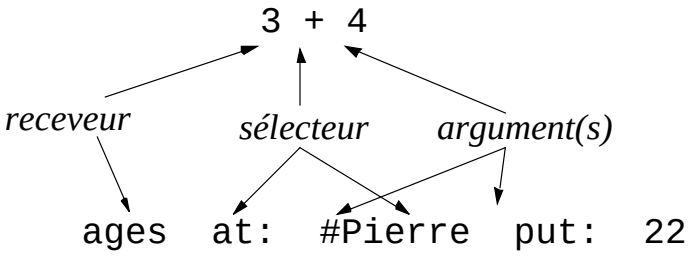
Messages

- Un message est une requête à un objet (receveur) pour exécuter l'une de ses méthodes

```
3 + 4
index + 1
theta sin
# (‘a’ ‘d’ ‘c’) at: 1 put: ‘b’
ages at: #Pierre put: 22
BankAccount new
```


Messages: Sélecteurs et Arguments

- Une expression de message décrit un *receveur*, un *sélecteur* et éventuellement des *arguments*



3 Types de Messages

- Messages unaires : messages sans arguments - seul un objet (le receveur) est concerné

```
theta sin
```

- Messages à mot-clés : messages avec un ou plusieurs arguments (pas plus de trois dans le système)

```
# (‘a’ ‘d’ ‘c’) at: 1 put: ‘b’
```

- Messages binaires: messages avec un seul argument

```
3 + 4
```


Messages “mots-clé”

```
class name
   Point
instance methods
   moveRight: value

class name
   Rectangle
instance methods
   origin: aPoint
   moveRight: value
```

Messages “mots-clé”

```
class name
   Array
instance methods
   at:index put:value
```

l La concaténation de mots-clés représente un seul sélecteur

```
   rectangle origin: pointI moveRight: 10
```

est incorrect, contrairement à l’exemple suivant

```
   # (‘a’ ‘d’ ‘c’) at: 1 put: ‘b’
```


Messages: conventions d’écritures

```
# (‘a’ ‘d’ ‘c’) at: 1 put: ‘b’

# (‘a’ ‘d’ ‘c’)
   at: 1
   put: ‘b’

 # (‘a’ ‘d’ ‘c’) at: 1
              put: ‘b’
```


Messages: Exemple

```
class name
   List
```

Permet de stocker des éléments

```
instance methods
   add: element
```

Permet de rajouter l’élément ‘‘`element`’’ dans la liste

Envoi de messages en cascade

```
temp := List new.
temp add: 1.
temp add: 2.
temp inspect.
```

est la même chose que:

```
   List new add: 1; add:2; inspect.
```

Messages: Valeurs de retour

- Le receveur d'un message retourne toujours une valeur

```
sum := 3 + 4
index := index  + 1
```

- Par défaut, le receveur retourne sa propre valeur - le retour de la valeur indique que le calcul s'est bien terminé

`3 + 4` (retourne 7)

Messages: Analyse/Evaluation

- Le receveur et les arguments d'un messages peuvent être le résultat d'un calcul (d'envois de messages à des objets)

`(x sin) sqrt` (le receveur `sqrt` est un résultat)

`(a + b) *` (le receveur de `*` est un résultat)

`a + (b sqrt)` (l'argument de `+` est un résultat)

```
# ('a' 'b') at: (1 * 1) put: 'a'
```

(le 1er argument de `at:put:` est un résultat)

Messages: Analyse/Evaluation

- Les messages unaires sont évalués de gauche à droite

`x sin sqrt` est la même chose que: `(x sin) sqrt`

- Les messages binaires sont évalués de gauche à droite

`a + b * 2` est la même chose que: `(a + b) * 2`

NB. Il n'y pas comme dans la plupart des langages un ordre dépendant des opérations (`*` et `/` vs `+` et `-`)

- Les parenthèses permettent de changer l'ordre d'une évaluation, e.g., `a + (b * 2)`

Messages: Analyse/Evaluation

- Les messages unaires sont plus prioritaires que les messages binaires

`a + b sqrt` est la même chose que: `a + (b sqrt)`

`2 * theta sin` est la même chose que: `2 * (theta sin)`

- Les parenthèses permettent de changer l'ordre d'une évaluation, e.g., `(2 *  theta) sin`

- Les parenthèses permettent d'exprimer l'opération désirée

```
rectangle origin: (pointI moveRight: 10)
(rectangle origin: pointI) moveRight: 10
```

Messages: Analyse/Evaluation

- Les messages binaires sont plus prioritaires que les messages par mot-clés

```
doubles at: x put: 2 * x
```

est la même chose que:

```
doubles at: x put: (2 * x)
```

- Priorités: (1) unaires, (2) binaires, (3) mot-clés

```
tableau at: a + 1 put: b sqrt
```

est la même chose que:

```
tableau at: (a + 1) put: (b sqrt)
```


Messages: Exemple

```
list := List new add: 3 * 4 sqrt.

list add: 3 * 0 cos; add: (3 * 0) cos.

value1 := 3 *  0 sin.
value2 := (3 * 0) cos.
list   add: value1.
list   add: value 2.
```


Résumé (expression)

- Une expression est une unité de programme (c'est aussi une séquence de caractères qui décrit un objet)

- Une expression légale est formée d'expressions élémentaires légales (constantes, variables, messages, et blocs)

- L'évaluation d'une expression se fait de gauche à droite - les messages unaires sont évalués en premier, puis les messages binaires, puis les mots-clé

Objets et Messages

Syntaxe (Smalltalk)

- Suite (blocs) -

Expression

- L'expression est l'unité de programme
  dans le *corps d'une méthode* ou
  dans l'*espace de travail* Smalltalk

NB. Une expression est une séquence de caractères qui décrit un objet (de la classe String)

- Une expression *légale* est formée d'expressions *élémentaires légales*

Syntaxe = expressions légales

Expressions élémentaires légales:

- *Constantes*
  Nombres, caractères (chaines de), symboles, et tableaux
- *Variables* (dépendantes du contexte)
  Conteneurs d'objets
- *Messages*
  Moyen d'interaction entre objets
- *Blocs*
  Séquence d'expressions ~ activité différée

Blocs

- Exemple de bloc: `[ index := index + 1. index + 1]`
  Les expressions du blocs ne sont pas directement évaluées

- Un bloc est un objet: il peut être affecté à une variable et il reçoit des messages, e.g., `value`

```
incrementBlock := [ index := index + 1. index + 1]
[ index := index ] + value.
incrementBlock value.
```

Le bloc exécute ses expressions lorsqu'il reçoit le message `value`

Blocs

- L'objet retourné par un bloc après évaluation (après réception du message `value`) est la dernière expression du bloc

```
x := 1.
y := 2.
addBlock := [ x := x + y.] x + x
z := addBlock value                    (z = 6)
```

Blocs: Exemple

```
incrementBlock := [index := index + 1]
sumBlock := [sum + (index * index)]
sum := 0.
index := 1.
sum := sumBlock value.
incrementBlock value.
sum := sumBlock value.
```


Blocs: Exemple

1. Un bloc est affecté à la variable `incrementBlock`
2. Un bloc est affecté à la variable `sumBlock`
3. Le nombre `0` est affecté à la variable `sum`
4. Le nombre 1 est affecté à la variable `index`

5. Le message `value` est envoyé au bloc `sumBlock`
6. Le message `* 1` est envoyé au nombre `1`
7. Le message `+ 1` est envoyé au nombre `0`
8. Le nombre `1` est affecté à la variable `sum`

Blocs: Exemple

9. Le message `value` est envoyé au bloc `IncrementBloc`
10. Le message `+1` est envoyé au nombre `1`
11. Le nombre `2` est affecté à la variable `index`

12. Le message `value` est envoyé au bloc `sumBloc`
10. Le message `* 2` est envoyé au nombre `2`
11. Le message `+ 4` est envoyé au nombre `1`

12. Le nombre `5` est affecté à la variable `sum`

Arguments d'un bloc: exemple

```
incrementBlock := [:i | x:= x. + i
x := 0.
incrementBlock value: 3.
y := incrementBlock value: 2.        (y = 5)
```

Le bloc est évalué avec un paramètre donné en argument

Arguments d'un bloc: exemple

```
sizeAdder := [:array | total := total + array].
total := 0.
sizeAdder value: #(1 2 3).
sizeAdder value: #($$ $r $e).
sizeAdder value: #(e f).          (total = 8)

square := [:x :y | z:= (x*x)] + (y*y)
square value: 3 value: 4.          (z = 25)
```


Messages de la classe BlockClosur

```
value
value: parameter
value:parameter1 value:parameter2
value:parameter1 value:parameter2 value: parameter3
valueWithArguments: anArrayOfParameters
```

Evalue un bloc avec les paramètres donnés en argument. Par défaut, retourne à l'envoyeur du message `value` la valeur de la dernière expression du bloc. Si une expression `^` est rencontrée, retourne la valeur de l'expression à l'envoyeur de la méthode qui contient le bloc

Blocs: Structures de contrôle

- Par défaut, les expressions dans Smalltalk sont évaluées séquentiellement - une *structure de contrôle* permet de définir l'ordre d'exécution dynamiquement

- Les structures de contrôle sont mises en oeuvre grâce à des *envois de messages à des blocs* et à l'utilisation des *blocs comme arguments de message*

NB. A la différence de beaucoup d'autres langages, les structures de contrôle de Smalltalk sont programmées en Smalltalk

Blocs: Répétition conditionnell

```
index := 0.
total := 0.
[index < collection size]
  whileTrue: [total := total +
              (collection at: (index:=index+1))].
```

Retourne la somme des nombres de la collection dans `total`

- La répétition est mise en oeuvre grâce à l'envoi de messages `whileTrue:` (resp. `whileFalse:`) à des objets blocs, qui envoient le message `value` aux blocs en argument

Messages de la classe BlockClosure

`whileFalse:`

Evalue le bloc tant qu'il retourne un booléen `false`

(resp. `aBlock whileFalse: anotherBlock`)

`whileTrue:`

Evalue le bloc tant qu'il retourne un booléen `true`

(resp. `aBlock whileTrue: anotherBlock`)

Blocs: Sélection conditionnelle

```
(number \\ 2) = 0
   ifTrue: [parity := 0]
   ifFalse: [parity := 1]
```

- La sélection est mise en oeuvre grâce à l'envoi de messages `ifTrue: ifFalse:` aux objets booléens (`true` resp. `false`) qui envoient le message `value` aux blocs en arguments

Messages de la classe Boolean

`ifTrue: aBlock`

Si le receveur est `true`, le bloc `aBlock` est évalué

(resp. `ifFalse: aBlock`)

`ifTrue: firstBlock ifFalse: secondBlock`

Si le receveur est `true`, le premier bloc `firstBlock` est évalué

Sinon, c'est le second bloc `secondBlock` qui est évalué

(resp. `ifFalse: firstBlock ifTrue: secondBlock`)

Blocs: Répétition

```
4 timesRepeat: [x := x + x]
```

- Un message est envoyé à un entier n (4) avec un bloc comme argument, pour répéter n (4) fois l'évaluation du bloc, i.e., le message `value` est envoyé 4 fois au bloc

```
1 to: 4 do: [:i | x := x + i]
```

- Le message `value:` est envoyé 4 fois au bloc avec, à chaque fois, i (de 1 à 4) comme argument

Messages de la classe Integer

`timesRepeat: aBlock`

Evalue le block `aBlock` n fois (n étant le receveur)

`to: integer do: aBlock`

Evalue le block `aBlock` i fois (i allant du receveur à integer), avec i comme paramètre

Blocs: Répétition

- Le message `do:` est utilisé par les objets collections pour appliquer un bloc à tous leurs éléments

```
sum := 0.
# (2 3 5 7 11) do:
   [:x | sum := sum + (x * x)]
```

- Le message `collect:` crée une nouvelle collection

```
collection := # (2 3 5 7 11) collect: [:x | x * x]
```


Messages de la classe Collection

`do: aBlock`

Evalue le bloc `aBlock` sur chaque i ème élément de la collection, avec à chaque fois cet élément comme argument

`collect aBlock`

Fait la même chose que `do:` et met les résultats dans une nouvelle collection

Résumé (expression)

- Une expression est une unité de programme (c'est aussi une séquence de caractères qui décrit un objet)
- Une expression légale est formée d'expressions élémentaires légales (constantes, variables, messages, et blocs)
- L'évaluation d'une expression se fait par défaut de gauche à droite - les messages unaires sont de priorité 2, les binaires de priorité 1 et les mots-clés de priorité 0

Résumé (blocs)

- Un bloc est la description d'un programme différé

- Quelques messages importants
  - `value` envoyé à un bloc
  - `value:` envoyé à un bloc avec un argument
  - `whileTrue:` envoyé à un bloc pour une répétition
  - `ifTrue: if False:` envoyé à un booléen
  - `do:` envoyé à une collection
  - `timesRepeat:` envoyé à une collection

2 + 3 Concepts (vocabulaire)

- **Objet**

- **Message**

- Classe

- Instance

- Méthode

Classes et Instances

Syntaxe (Smalltalk)

Classes

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

- Les instances d'une classe ont la même *interface*, i.e., répondent aux mêmes messages

- Une classe possède un nom qui permet d'accéder à la classe et qui permet à ses instances de s'identifier

NB. Le nom de la classe est contenu dans une variable partagée, qui désigne aussi l'objet représentant la classe

Classes

La *description d'une classe* comporte deux parties:

1. Partie externe: la *description d'un protocole* qui décrit l'interface (les messages) de la classe *(spécification)*

2. Partie interne: la *description de la mise en oeuvre* (*les variables d'instance + les méthodes*) des opérations de la classe qui permettent de répondre aux messages

NB. La description du protocole doit être terminée avant de commencer la description de la mise en oeuvre

Protocole d'une classe

- Le protocole de la classe est la liste des messages auxquels ses instances savent répondre (l'interface)

- Chaque description de message est constituée d'un modèle, accompagné d'un commentaire décrivant l'effet du message et la valeur retournée

```
credit: amount
```
Rajoute la quantité amount au solde du compte

Catégories de messages

- Les messages qui concernent des opérations similaires sont groupés dans des catégories, chacune identifiée par un nom qui décrit la fonctionnalité commune des messages de la catégorie

`BankAccount` protocol

modification

```
credit: amount
```
Rajoute la quantité amount au solde du compte

```
debit: amount
```
Retire la quantité amount du solde du compte

Catégories de messages

`BankAccount` protocol

modification

```
credit: amount
debit: amount
transferamountto:account
```

access

```
balance
updateBalance: amount
```

initialization

```
initializeBalanceamount
```

Mise en oeuvre d'une classe

La mise en oeuvre d'une classe est composée de trois parties:

1. Le *nom* de la classe

2. La déclaration des *variables (d'instance et autres)*

3. Les *méthodes (d'instance et de classe)*, utilisées pour répondre aux messages (déclarations et mises en oeuvre)

Variables d'instance

- Les variables d'instance stockent l'état des instances

  class name

  ```
  BankAccount
  ```

  instance variables

  ```
  balance
  ```

- Les instances d'une même classe ont les mêmes variables d'instances, mais avec des valeurs différentes.

Méthodes

- Une méthode décrit comment les objets de la classe répondent à un message

- Une méthode est constituée de la description (rappel) d'un modèle de message, puis d'une séquence d'expressions (séparées par un point)

```
creditDouble: amount
    balance := amount.
    balance := balance * balance
```


Méthodes

- Une classe ne peut avoir deux méthodes avec le même sélecteur (i.e., avec le même nom et nbr d'arguments)

  e.g., `#at:put:` `#to:do:` `#value`

- Les arguments d'une méthode sont considérés comme des pseudo-variables

```
writeIncrement: amount
    balance := amount.
    amout := amount + 1
```

(TEST INTERDIT)

Méthodes

- Une méthode retourne toujours une valeur

```
balance
    ^balance
balance:amount
    ^balance :=amount
```

- Par défaut, la valeur retournée est le receveur lui-même (^self)

```
creditamount
    balance := balance + amount
```


Mise en oeuvre d'une classe

class name

```
BankAccount
```

instance variables

```
balance
```

instance methods

access

```
balance
    ^balance
balance: amount
    ^balance := amount
```


Mise en oeuvre d'une classe (suite

modification

```
debitamount
    balance := balance - amount

creditamount
    balance := balance + amount

transferamountto:account
    self debit: amount.
    account credit: amount.
```


Variables accessibles à une méthod

- Une méthode peut accéder à deux sortes de variables

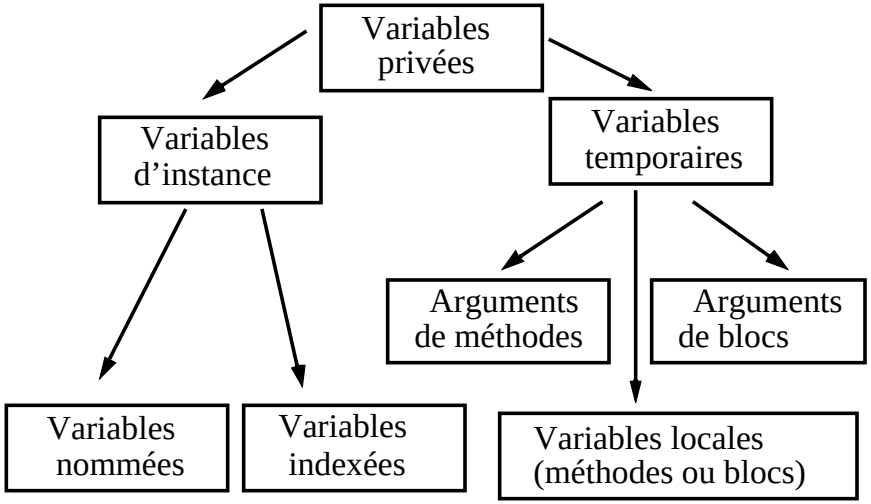
Les variables *privées* (e.g., les variables d'instance)

NB. Les variables privées commencent par une minuscule

Les variables *partagées* (e.g., les variables globales)

NB. Les variables partagées commencent par une majuscule

Variables privées d'un objet



Variables temporaires

- Les variables *temporaires* représentent un état transitoire nécessaire à l'exécution d'une activité donnée: elles sont *locales à une méthode*

  NB. La durée de vie d'une variable temporaire est (en général) l'exécution d'une méthode

```
taxedTransferamountto:account
    |temp|
  temp :=amount -((amount*TaxRate): 100).
    self debit: amount.
    account credit: temp.
```


Variables d'instance

- Les variables d'*instance* représentent l'état courant d'un objet: elles sont *globales à toutes les méthodes* de l'objet

- La création d'une nouvelle instance (message `new`) entraine la création de ses variables d'instance qui initiallement contiennent l'objet `nil`

```
Bankaccount new
```


Variables d'instance nommées

- Les variables d'instance *nommées* sont accessibles (dans une instance) à travers un nom qui les désigne

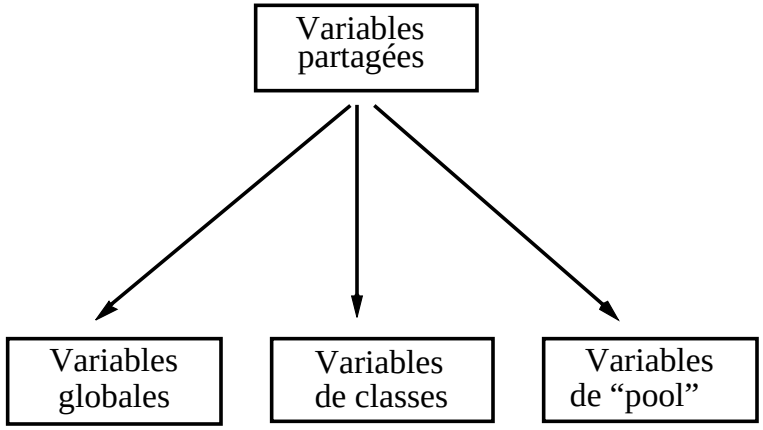
```
writeamount
    balance := amount
```

Méthodes & variable self

```
increment
   balance := self  balance + 1

factorial
   self = 0 ifTrue: [1]
   self < 0.
      ifTrue: [self error: 'factorial] inva
      ifFalse: [self * (self - 1) ]factori
```


Variables partagées



Variables partagées

- Les variables *globales* sont accessibles à toutes les instances de toutes les classes (ce sont principalement les noms de classes)
- Les variables de *classe* sont partagées par toutes les instances d’une classe
- Les variables d’*équipe* (*pool*) sont accessibles à toutes les instances d’un sous-ensemble de classes

Variables partagées

class name

`BankAccount`

instance variable names

`balance`

class variable names

`TaxRate`

shared pools

`FinancialConstants`

Résumé

- Une classe est un objet qui décrit la mise en oeuvre d'un ensemble d'objets
- Une instance est l'un des objets décrit par une classe; elle possède une mémoire et peut répondre à des messages
- Une classe est décrite à travers (1) l'interface de ses instances (le protocole) et (2) à travers ses variables et ses méthodes (la mise en oeuvre)

Remarque: méthodes primitives

- Les méthodes primitives sont celles qui effectuent de vrais calculs (une centaine en Smalltalk-80)

```
+
new
new:
```

- Utilisation de méthodes primitives :

```
< primitive #300>
```


Classes & Objets

- Le message `new`, envoyé à une classe, permet de créer une instance de la classe

```
BankAccount new
```

- Le message `now`, envoyé à la classe `Time`, retourne une instance de la classe représentant la date de l'évaluation

```
Time now
```

- Le message `today`, envoyé à la classe `Date`, retourne le jour de l'évaluation

```
Date today
```


Classes et Sous-Classes

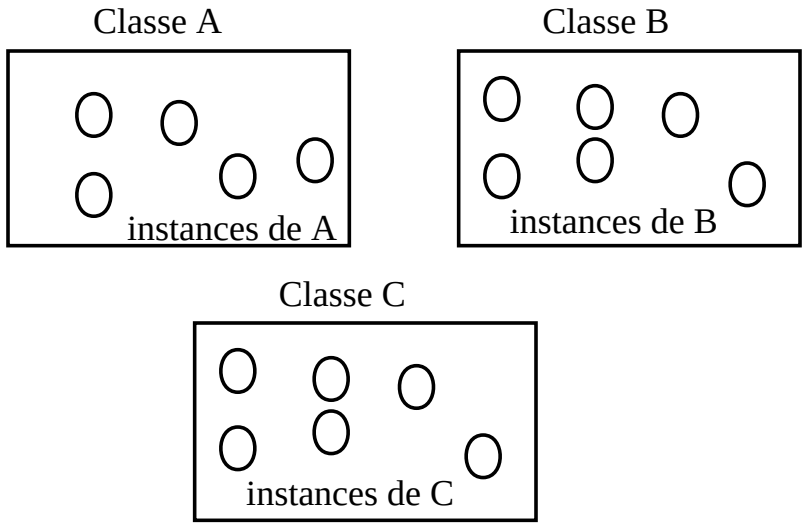
Syntaxe (Smalltalk)

Classes

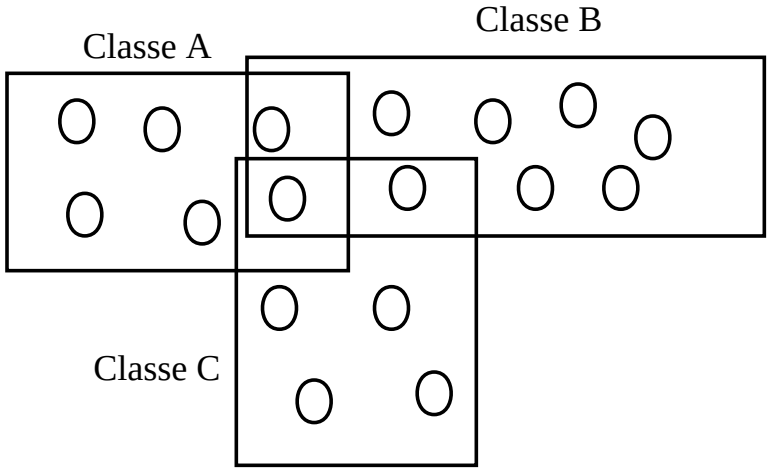
- Une classe est un *ensemble* d'objets: Chaque objet est une instance d'une classe

- Une classe est un *moule* d'objets: les instances d'une classe représentent le même type de composant (même interface et même mise en oeuvre)

Classes ~ Ensembles



Intersection entre classes



Intersection entre classes

- Un objet peut être instance de plusieurs classes

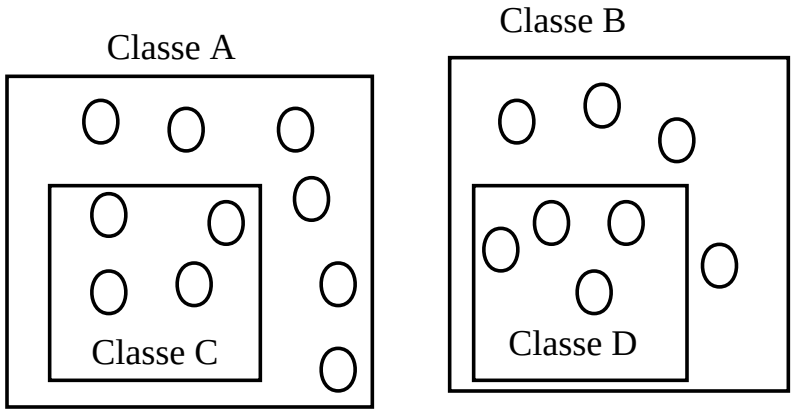
- Deux instances de classes différentes peuvent partager certaines méthodes et certaines variables

  e.g., les entiers et les réels

  e.g., les collections ordonnées et les ensembles

Héritage simple

- Une forme restreinte d'intersection



Héritage

- L'héritage simple (sous-classage) permet les inclusions: une classe A possède une *super-classe* B => les instances de A sont aussi des instances de B

NB. L'héritage multiple permet des intersections quelconques: une classe A peut avoir des super-classes B et C => les instances de A sont aussi des instances de B et de C - Smalltalk ne permet pas l'héritage multiple

Héritage

- Les instances d'une sous-classe se comportent de la même manière que les instances de la super-classe sauf pour les différences qui sont explicitement mentionnées

- Plusieurs classes peuvent avoir la même super-classe

- A part la classe `Object`, toute classe est une sous-classe d'une autre classe

- Toutes les classes sont des sous-classes (directes ou indirectes) de la classe `Object`; la classe `Object` décrit le comportement commun à tous les objets

Mise en oeuvre d'une classe

- Le nom de la classe et le nom de la super-classe

- Les variables de la nouvelle classe sont en fait des variables rajoutées

- Les méthodes de la nouvelle classe sont en fait des méthodes rajoutées

- La sous-classe peut aussi redéfinir des méthodes de la super-classe (on parle de *surcharge*)

Mise en oeuvre d'une classe

class name `PrivateBankAccount`

superclass

`BankAccount`

instance variables

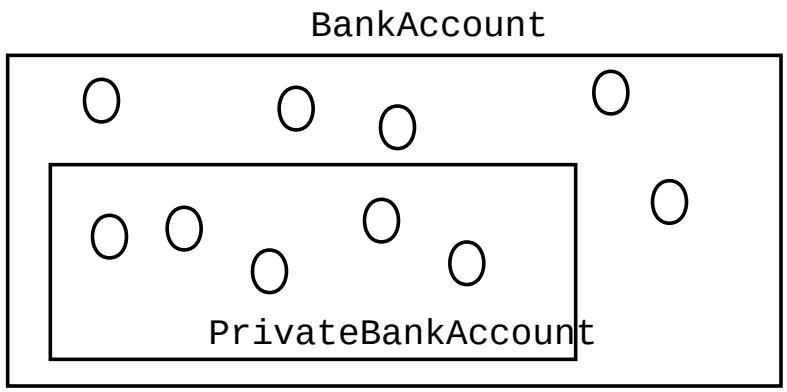
`interestRate`

instance methods

modification

```
interest
    balance := balance +
            (balance* interestRate :-1
```


Classes et Sous-classes



Recherche de méthode (lookup)

- Lorsqu'un objet recoit un message

(1) Une méthode avec un sélecteur correspondant au message est recherchée dans la classe de l'objet

(2) Si aucune méthode n'est trouvée, la recherche se poursuit dans la super-classe, puis dans la super-super-classe, etc.

(3) La classe `Object` répond à tous les messages en invoquant la méthode `doesNotUnderstand` sur le receveur

(4) La classe `Object` répond au message `doesNotUnderstand` en renvoyant un code d'erreur

Recherche de méthode (self)

- La pseudo-variable `self` désigne le receveur du message

- Lorsque la méthode d'une classe C contient un message dont le receveur est la pseudo-variable `self`, la recherche de la méthode correspondante commence dans la classe de l'objet qui a recu le message

Recherche de méthode (self)

class name `One`
instance methods

```
test^1
result1^self test
```

class name `Two`
superclass `One`
instance methods

```
test ^2
```


Recherche de méthode (self)

```
example1: = One new.
example2: = Two new.


example1 test  -> ?
example1 result1 -> ?
example2 test -> ?
example2 result1 -> ?
```


Recherche de méthode (super)

- `super` est, comme `self`, une pseudo-variable qui désigne le receveur du message
- Lorsque la méthode d'une classe C contient un message dont le receveur est la pseudo-variable `super`, la recherche de la méthode correspondante commence dans la super-classe de C
- La pseudo-variable `super` permet d'accéder aux méthodes de la super-classe même si ces méthodes ont été surchargées

Recherche de méthode (super)

class name `One`
superclass `Object`
instance methods

```
test^1
result1^self test
```

class name `Two`
superclass `One`
instance methods

```
test^2
```

class name `Three`
superclass `Two`
instance methods

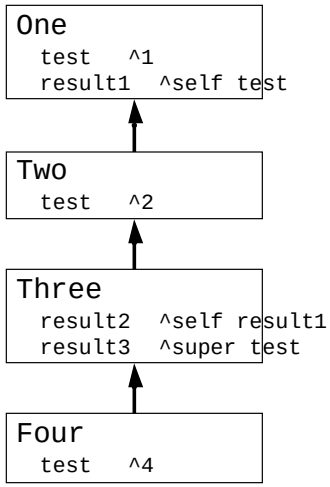
```
result2^self result1
result3^super test
```

class name `Four`
superclass `Three`
instance method

```
test^4
```

Recherche de méthode (super)

```
example3: = Three new.
example4: = Four new.
```




```
example3 test  -> ?
example4 result1 -> ?
example3 result2 -> ?
example4 result2 -> ?
example3 result3 -> ?
example4 result3 -> ?
```


Recherche de méthode (super)

- L'utilisation de `super` ne signifie pas que la recherche doit être commencée dans la super-classe du receveur, mais que la recherche doit être commencée dans la super-classe de la classe contenant la méthode dans laquelle `super` est évalué

Exemple: si on ajoute `test^3` dans la classe `Three`

```
example4 result3
```

==> `result3` est trouvée dans la classe `Three`

==> `super test` est évaluée dans la classe `Three`

==> `test` est cherché dans la classe `Two`

==> `2` est retourné (et pas `3`)

Recherche de méthode (super)

class name `One`
superclass `Object`
instance methods

```
test 1^
result1^self test + 1
```

class name `Three`
superclass `Two`
instance methods

```
result2^self result1 +
result3^super test + 3
```

class name `Two`
superclass `One`
instance methods

```
test ^2
```

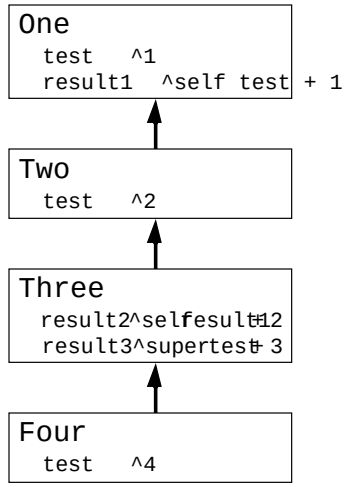
class name `Four`
superclass `Three`
instance method

```
test ^4
```


Recherche de méthode (super)

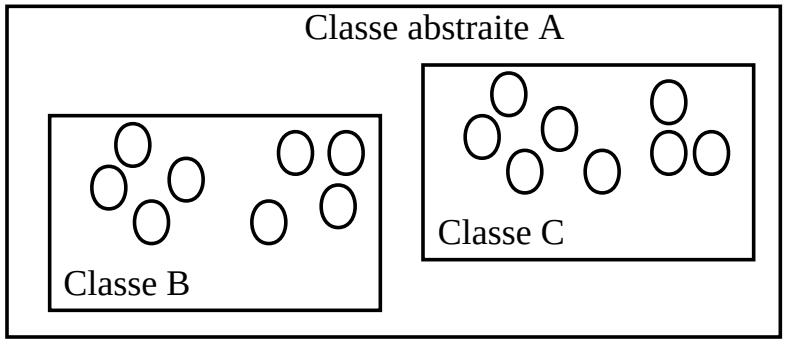
```
example1: = One new.
example2: = Two new.
example3: = Three new.
example4: = Four new.

example1 result1  -> ?
example2 result1 -> ?
example3 result2 -> ?
example4 result3 -> ?
example4 result2 -> ?
```



Classe Abstraite

- Une classe abstraite ne possède pas d'instances



Classe Abstraite

class name

```
A
```

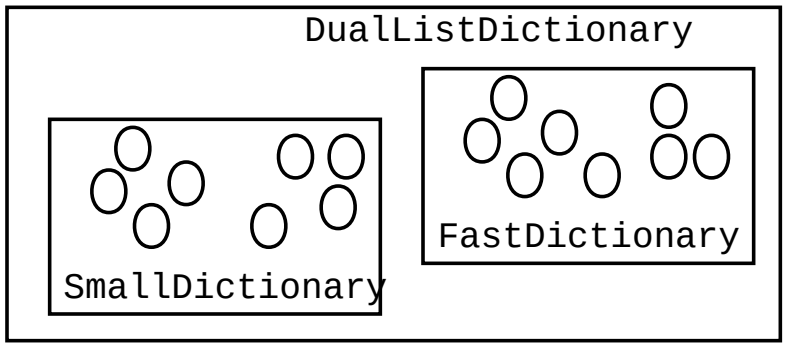
superclass

```
Object
```

instance methods

```
action1
    ^self action2
```


Classe Abstraite: Dictionnaires



Classe Abstraite: Dictionnaires

- La classe `SmallDictionary` minimise l'espace requis pour stocker son contenu
- La classe `FastDictionary` utilise une technique rapide pour retrouver les noms

  Les deux classes utilisent deux listes parallèles qui contiennent les noms et les valeurs associées

- La classe abstraite `DualListDictionary` contient les similarités entre les deux classes

Classe Abstraite: Dictionnaires

class name `DualListDictionary`
instance variable names `names values`
instance methods
accessing

```
at: name retrouve une valeur
| index |
index := self indexOf: name.
index = 0
   ifTrue: [self error: 'Name not found']
   ifFalse: [^values at: index]
```


Classe Abstraite: Dictionnaires

class name `SmallDictionary`
superclass `DualListDictionary`
instance methods

```
indexOf: name retrouve un index
1 to: names size do:
[:index | (names at: index) = name
        ifTrue: [^index]].
^0
```


Classe Abstraite: Dictionnaires

class name `FastDictionary`
superclass `DualListDictionary`
instance methods

```
indexOf: name retrouve un index
|index|
index := name hash \\ names size + 1.
[(names at: index) = name]
    whileFalse: [(names at: index) isNil ifTrue:
        [index := 0].
        index := index \\ names size + 1]
^index
```


Classes abstraites (conventions)

Utiliser les méthodes d'erreur de la classe `Object`

- La mise en oeuvre d'une méthode abstraite doit être
  `self subclassResponsibility`
  retourne une notification expliquant que la méthode doit être en fait mise en oeuvre dans la sous-classe

- La non mise en oeuvre d'une méthode abstraite doit être
  `self shouldNotImplement`
  le message n'est pas approprié à l'objet

3 Principes

(1) Chaque composant du système est un objet

- Un objet répond à des messages

(2) Chaque objet est instance d'une classe

- La relation d'instanciation détermine l'endroit où la recherche de méthodes doit débuter

(3) Chaque classe hérite d'une autre classe

- La relation d'héritage détermine la continuation de la recherche de méthodes

Résumé

- Une sous-classe hérite des variables et des méthodes de sa super-classe
- La *surcharge* est le fait de changer la mise en oeuvre d'une méthode héritée
- La racine de toute les classes est la classe `Object`
- Une classe abstraite est une classe qui ne peut pas avoir d'instances

Méta-Classes

(Smalltalk)

3 Principes

(1) Chaque composant du système est un objet

- Un objet répond à des messages

(2) Chaque objet est instance d'une classe

- La relation d'instanciation détermine l'endroit où la recherche de méthodes doit débuter

(3) Chaque classe hérite d'une autre classe

- La relation d'héritage détermine la continuation de la recherche de méthodes

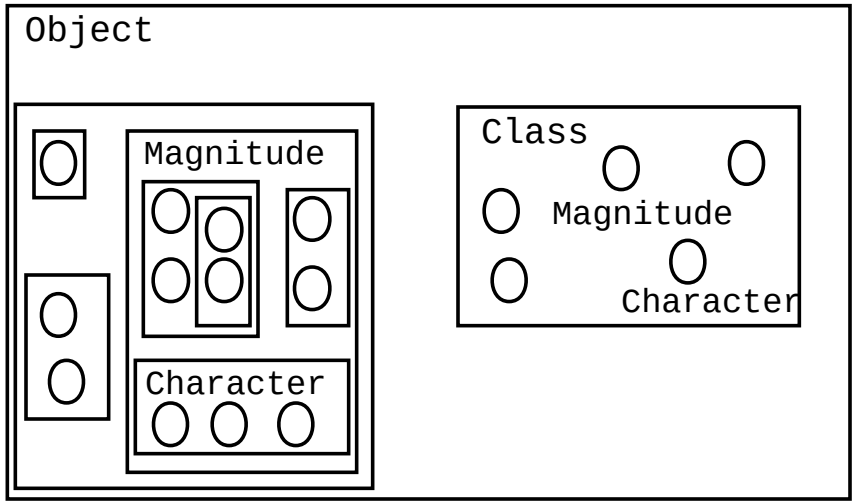
```
BankAccount new
```

- Quel est le receveur du message `new` ?
- Comment s'effectue la recherche de la méthode `new` ?

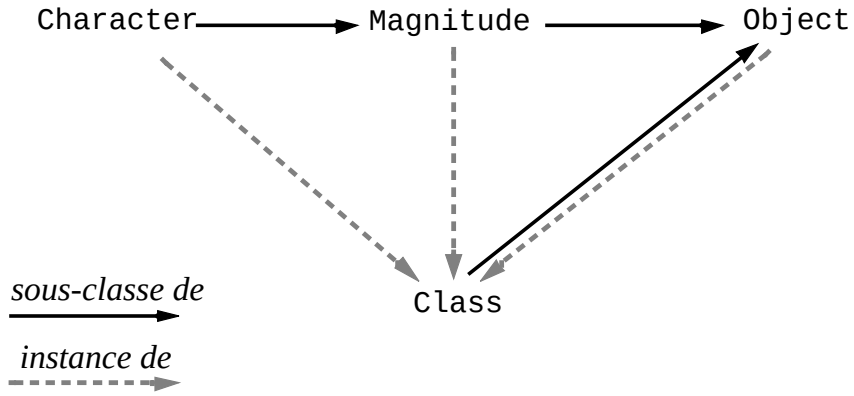
Smalltalk-76

- Toutes les classes sont instances d'une même classe, appelée `Class`
- `Class` est instance d'elle même et sous-classe de la classe `Object`

Smalltalk-76



Smalltalk-76



Smalltalk-76

```
BankAccount new
```

- La création d'une instance se fait par l'envoi du message `new` à la classe appelée `Class` (qui est instance d'elle même)
- Le message `new` est un message de la classe `Class`
- Toutes les classes répondent au message `new` de la même manière (en créant une instance dont les variables d'instances sont `nil`) - *un peu restrictif !*

Smalltalk- 80 & Méta-classes

- La restriction que toutes les classes ont le même protocole de création (i.e., la même classe) est levée
- Chaque classe *est instance de* sa propre *méta-classe* (elles sont créées en même temps)
- Comme une classe, une méta-classe contient des méthodes et des variables utilisées par son instance (la classe)
- Cependant, une méta-classe possède *une seule instance* qui est une classe

Initialisation explicite (Smallta

class name `BankAccount`

instance variable names `balance`

instance method names

access

```
balance
    ^balance
balance: amount
    ^balance := amount
```


Initialisation implicite (Smallta

class name `BankAccount`

instance variable names `balance`

class method names

instance creation

```
initialbalance: amount
    ^self new balance: amount
```

instance method names

access

```
balance  ^balance
balance: amount ^balance := amount
```

Initialisation explicite vs impl

```
account := BankAccount new  balance: 300
```

l Crée et initialise une instance de la classe `BankAccount`

L'initialisation est explicite

```
account := BankAccount  initialbalance: 300
```

l Crée et initialise une instance de la classe `BankAccount`

L'initialisation est implicite

Description d'une classe

1. Partie externe: le protocole qui décrit les messages de la classe et de la méta-classe

2. Partie interne: la mise en oeuvre qui décrit les variables d'instance, de classe et de pool, ainsi que les méthodes d'instance et de classe

Description d'une classe

class name `BankAccount`
instance variable names `balance`
class variable names `TaxRate`
shared pools `FinancialConstants`
class methods
    instance creation
        `initialbalance: amount`
instance methods
    ...

Description d'une classe

class name `PrivateBankAccount`
superclass `BankAccount`
class variables `Interest`
class methods
    instance creation
        `new  ^super new balance: 0`
    class initialization
        `initialize Interest := 10`
    class interface
        `demo  Transcript show: 'Gestion de compte'`

Messages de création/initialisati

```
Time now
```

- Retourne un objet représentant le moment de l'évaluation

```
Date today
```

- Retourne un objet représentant la date de l'évaluation

```
Rectangle
      origin: (Point x:50 y:50)
      corner: (Point x:250 y:300)
```

- Retourne un objet rectangle

Méthodes et variables de classe

- Les méthodes de classe ne peuvent pas accéder aux variables d'instances de la classe

- Les méthodes d'instance peuvent accéder aux variables de classe

Variables et méthodes de classes

- Les variables de classes sont initialisées à `nil` à la création de la classe

- La méthode de classe `initialize` est (par convention) celle qui permet d'initialiser les variables de classes - le message `initialize` doit être envoyé une seule fois (à la classe) avant la création des instances

- Les méthodes de classes qui retournent une nouvelle instance sont par convention classées dans une catégorie *''instance creation''*

Méta-classes

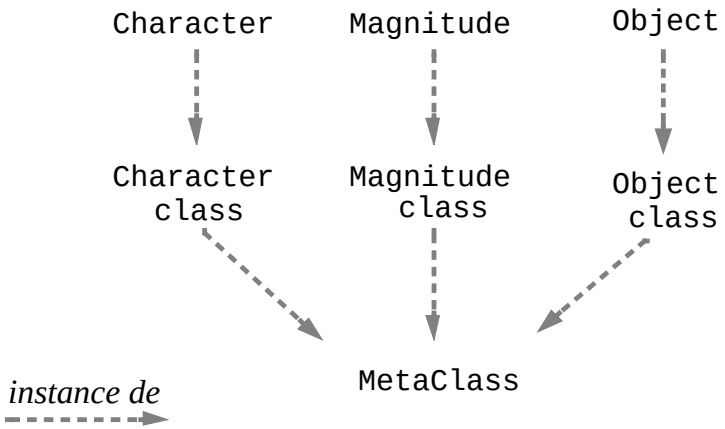
- Une méta-classe est différente d'une classe:

  (1) Une méta-classe n'a pas de méta-classe

  (2) Une méta-classe n'a pas de nom propre - la méta-classe d'une classe `C` (normale) est nommée `C class`

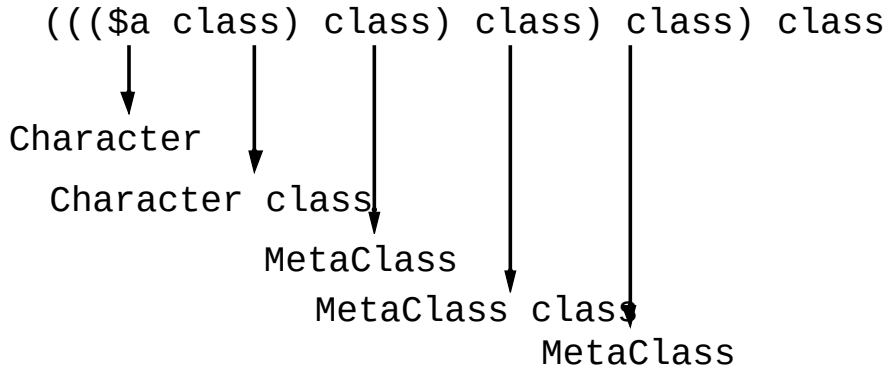
  (3) Une méta-classe a comme seule instance: la classe

Méta-classes



Méta-classes

- Le message `class` retourne la classe de l'objet
- `MetaClass` est la classe de toutes les méta-classes

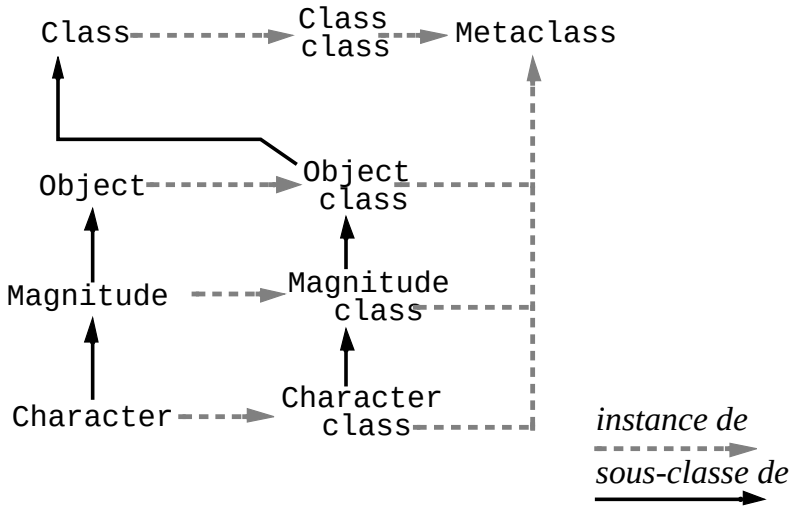


Héritage de méta-classes

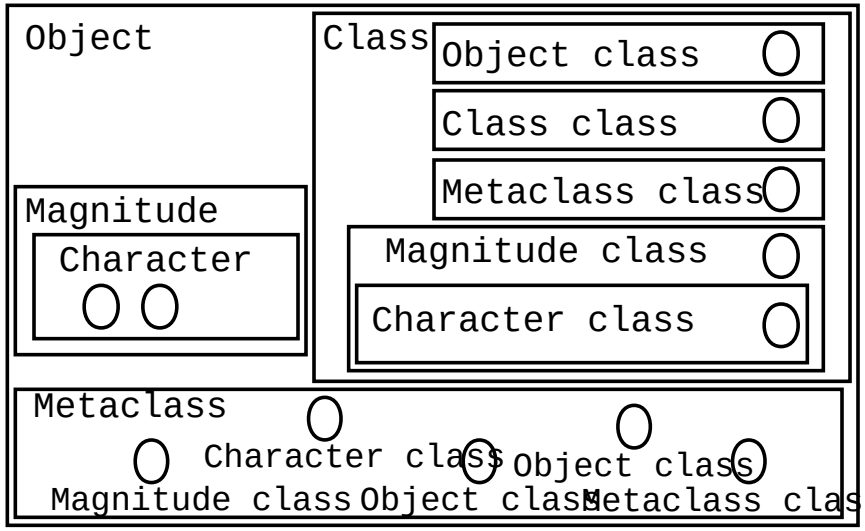
- Si une classe A hérite de la classe B, alors la méta-classe de A hérite de la méta-classe de B
- L'arbre d'héritage des classes et l'arbre d'héritage des méta-classes sont parallèles
- `Class` est la super-classe de toutes les méta-classes

NB. `Class` est la super-classe de la méta-classe de `Object`

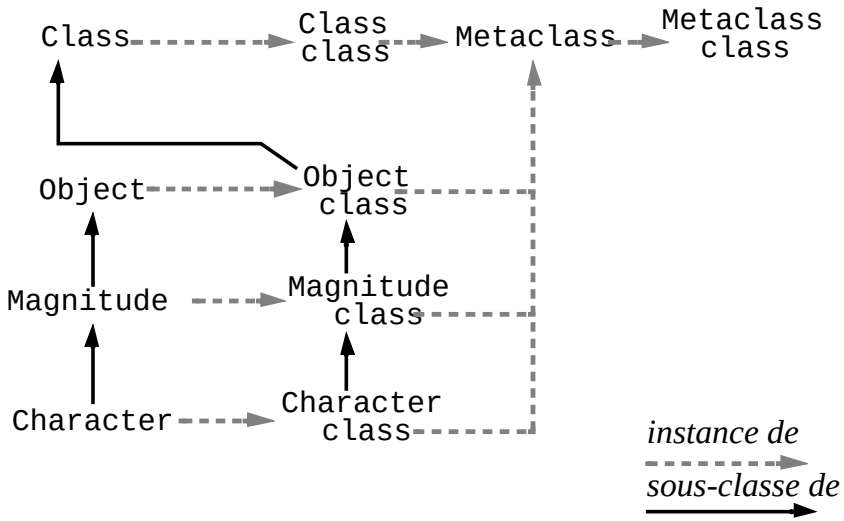
Héritage de méta-classes



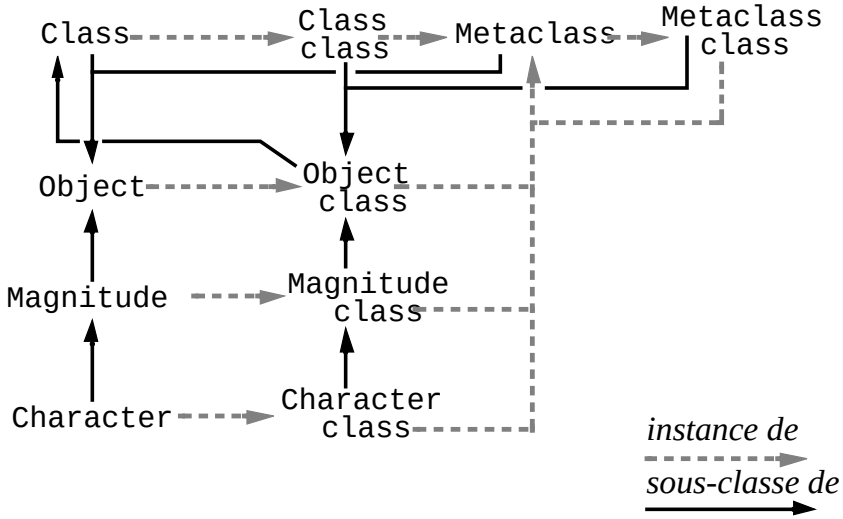
Héritage de méta-classes



Méta-classes



Méta-classes



La Classe Object

- Toutes les classes héritent de la classe `Object`
- `Object` décrit le comportement commun de tous les objets
- `Object` n'a pas de super-classe
- La super-classe de `Object class` (la méta-classe de `Object`) est `Class`

La Classe Class

- Toutes les méta-classes héritent de la classe `Class`
- `Class` décrit le comportement commun de toutes les classes (représentation en mémoire, notion de nom de classe,..)
- Chaque méta-classe rajoute (peut rajouter) un comportement spécifique pour sa classe:
  Nouvelles méthodes, e.g., `Date, Time` et `Rectangle`
  Redéfinition des méthodes `new` et `new:`

La Classe MetaClass

- Toutes les méta-classes sont des instances de `MetaClass`
- `MetaClass` permet d'initialiser les variables de classe et de créer les instances
- `MetaClass` est une instance de `MetaClass class`
- `MetaClass class` est une instance de `MetaClass`

Recherche de méthodes: self new

class name `PrivateBankAccount`

superclass `BankAccount`

class methods

instance creation

```
initialbalance: amount
    ^self new balance: amount
```

instance methods

...

Recherche de méthodes

```
account := PrivateBankAccount  initialbalance:
```

- La méthode `initialbalance:` est trouvée dans `PrivateBankAccount class`
- La méthode `new` est cherchée dans `PrivateBankAccount class`, puis dans `BankAccount class`, puis dans `Object class`, puis trouvée dans `Class`
- Une instance de `PrivateBankAccount` est créée
- Le message `balance: amount` est envoyé à l'instance créée (la méthode est trouvée dans `BankAccount`)

Recherche de méthodes: super new

class name `PrivateBankAccount`

superclass `BankAccount`

class methods

instance creation

```
new
    ^super new balance: 0
```

instance methods

...

Recherche de méthodes

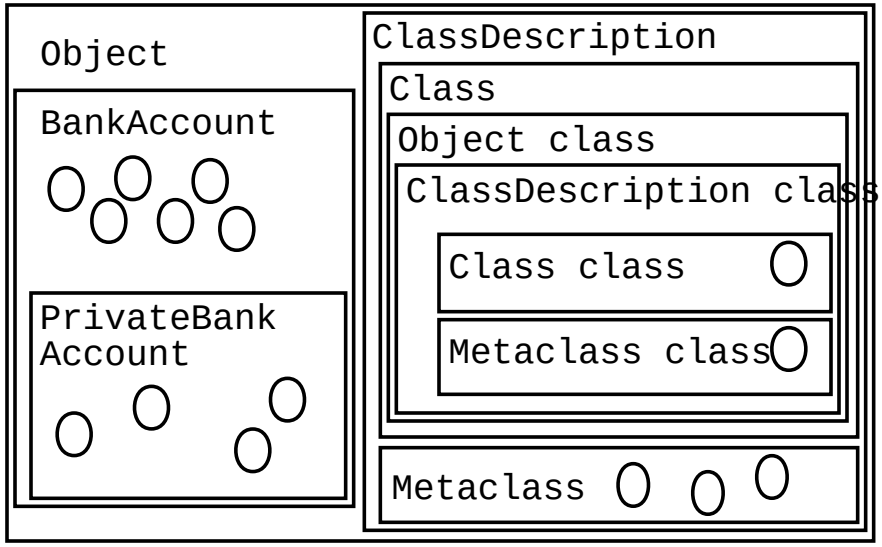
```
account := PrivateBankAccount new
```

- L'instance est créée grâce à l'évaluation de `super new`
- La méthode `new` n'est pas cherchée dans `PrivateBankAccount`, mais directement dans `BankAccount` (ou elle n'est pas mise en oeuvre.)
- La méthode `new` est trouvée dans la classe `Class`
- L'utilisation de `super` évite la récursivité

Le schéma global

- `MetaClass` et `Class` sont sous-classes de la classe `ClassDescription` qui elle-même est sous-classe de `Behavior`
- `Behavior` est une sous-classe de `Object`

Exemple



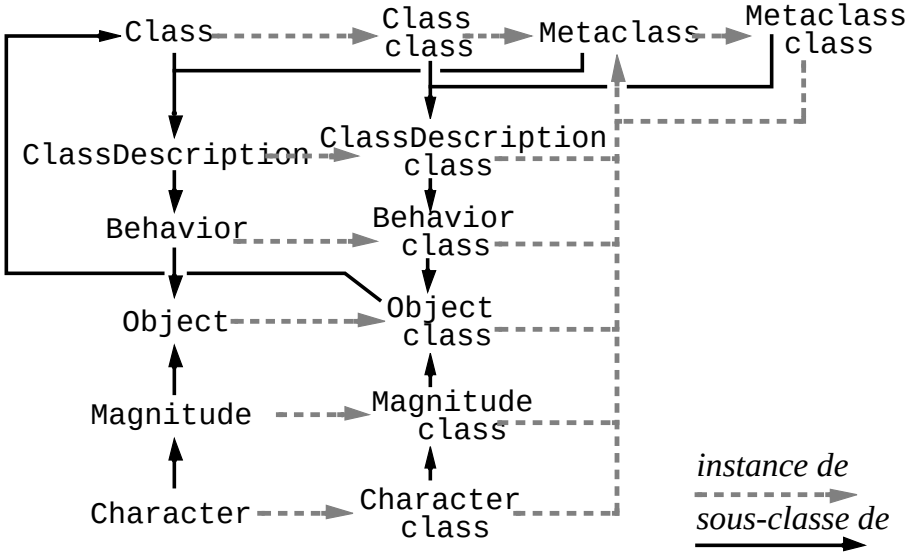
La Classe ClassDescription

- `ClassDescription` permet de gérer les noms des classes et les commentaires qui leur sont associés, les noms des variables d'instances, ainsi que les catégories de méthodes
- `MetaClass` et `Class` sont des sous-classes de `ClassDescription`
- L'environnement graphique VisualWorks est basé sur `ClassDescription`

La Classe Behavior

- `Behavior` décrit la structure des objets pouvant avoir des instances, les liens d'héritage, le dictionnaire des méthodes et la description de la structure des instances
- `ClassDescription` est une sous-classe de `Behavior`
- Le compilateur et la machine virtuelle Smalltalk sont basés sur `Behavior` qui permet de créer des instances, de compiler et de retrouver des méthodes

Le schéma global



Les Classes du Système:

La Classe OBJECT

Caractéristiques commune des objet

- Toutes les classes héritent de `Object` : l'ensemble des méthodes communes à tous les objets est mis en oeuvre dans la classe `Object`
  - *Tests de fonctionnalités*
  - *Comparaisons*
  - *Copies*
  - *Accès aux variables (indexées)*
  - *Impression & Stockage*
  - *Gestion d'erreur*
  - *Dépendances (explicites)*

Fonctionnalités d'un objet

`class` permet de retrouver la classe de l'objet

```
$a class-> Character
```

`isMemberOf: aClass` retourne `true` si `aClass` est la classe directe de l'objet (`false` sinon)

```
$a isMemberOf: Character -> true
```

`isKindOf: aClass` retourne `true` si `aClass` est la classe ou une super-classe de l'objet (`false` sinon)

```
$a isKindOf: Magnitude -> true
```


Fonctionnalités d'un objet

`respondsTo: aSymbol` retourne `true` si le dictionnaire de méthodes de la classe de l'objet contient le sélecteur `aSymbol` (`false` sinon)

`respondsTo:` permet de savoir si un objet sait répondre à un message

```
3 respondsTo: #+ -> true
3 respondsTo: #isMemberOf -> true
3 respondsTo: #blablabla -> false
```


Comparaison d'objets

- Le message == permet de tester si deux objets sont les mêmes - on parle d'identité ou d'équivalence (~~ est le contraire)
- Le message = permet de tester si deux objets ont la même valeur, - on parle d'égalité (~= est le contraire)
- Par défaut = est mis en oeuvre comme ==
- = est une méthode et peut donc être redéfini par le receveur

Comparaison d'objets (exemple)

```
'hello' == 'hello'      false
'hello' = 'hello'       true

#(1 2 3) class == Array  true
```

```
x isMemberOf: aClass
```

est équivalent à

```
x class == aClass
```


Objets immuables

- Un objet *immuable* est un objet dont l'état ne peut jamais changer.
- P.ex., les nombres entiers sont tous des objets immuables.

  Chaque nombre n'existe qu'en un seul exemplaire dans le système.

  ```
  21 = 21          true
  21 == 21         true
  ```

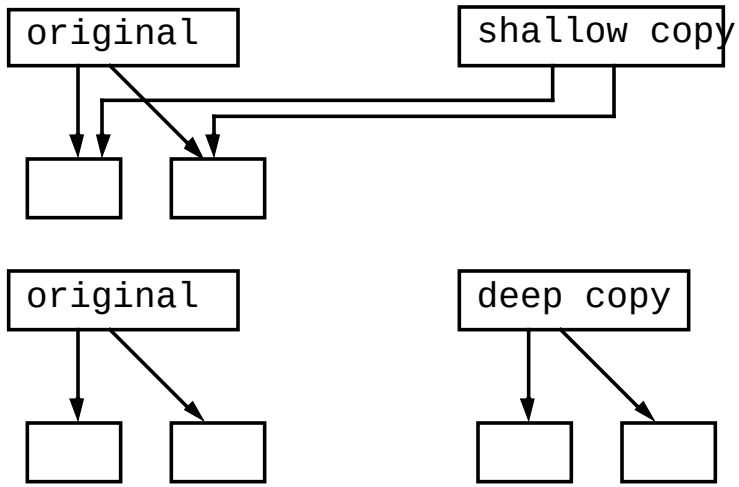
  Les opérateurs retournent un objet différent du receveur.

  ```
  3 * 7 == 21      true
  a := 3. b := a. a := b * 7.   (a = 21, b = 3)
  ```


Copies d'objets

- Il y a deux manières de copier des objets: suivant si les variables d'instances sont copiées ou pas
- Le message `shallowCopy` permet de créer une copie (superficielle) qui partage le contenu des variables d'instances
- Le message `deepCopy` permet de créer une copie avec ses propres valeurs de variables d'instances
- Par défaut `copy` est mis en oeuvre comme `shallowCopy`

Copies d'objets



Copies de tableaux (shallow)

- La copie d'un tableau référence les **mêmes** éléments que le tableau original, mais constitue un nouvel objet

```
a := #('first' 'second' 'third').
b: = a shallowCopy.        (identique à a copy)

a = b.                     -> true
a == b.                    -> false
(a at: 1) = (b at: 1).     -> true
(a at: 1) == (b at: 1).    -> true
```


Copies de tableaux (deep)

- La copie profonde d'un tableau référence des **copies** des éléments du tableau original

```
a := #('first' 'second' 'third').
b: = a deepCopy.

a = b.                     -> true
a == b.                    -> false
(a at: 1) = (b at: 1).     -> true
(a at: 1) == (b at: 1).    -> false
```


Impression/stockage d'un objet

- Imprimer une séquence de caractères qui décrit un objet

```
(set := Set new) add: #hello; add: #world
```

`printString` retourne une chaîne qui décrit le receveur

```
set printString -> 'Set (#hello #world)'
```

`printOn: aStream` met la chaîne dans `aStream`

`storeString` retourne le code Smalltalk permettant de reconstruire le receveur

```
set storeString
-> '((Set new) add: #hello; add: #world; yourself)'
```

`storeOn: aStream` met la chaîne dans `aStream`

Exceptions/Erreurs

- Lorsqu'un objet ne connait pas le message `aMessage` qui lui est envoyé, le "système" lui envoit le message `doesNotUnderstand: aMessage`

  `doesNotUnderstand: aMessage` appelle à son tour la méthode `error: aString` qui affiche un code d'erreur

- `primitiveFailed` est appellée quand une primitive (système) ne peut s'exécuter
- `subclassResponsibility` est appelé par les méthodes abstraites

Exceptions/Erreurs

- L'objet `nil` est la valeur par défaut des variables non initialisées

- `nil` est la seule instance de la classe `UndefinedObject`

- Dans le cas ou `nil` recoit un message, le code d'erreur affiché est :

`UndefinedObject` `does not understand message`

Messages spécifiques de `Object` : `isNil` et `notNil`

Dépendances

(1) Un objet est instance d'une classe

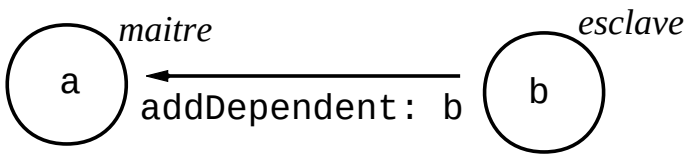
(2) Une classe hérite d'une autre classe

(3) Un objet référence un autre objet

Dépendances

- On peut coordonner des activités en introduisant des dépendances explicites entre objets

- Un objet `a` peut être lié à plusieurs autres objets `b, c, ..`
  La modification de `a` entraine une notification de `b,c,..`
  `b` *dépend* de `a`



Dépendances

- La catégorie de méthodes ''dependents access'' de la classe `Object` permet de créer des dépendances entre objets

  `addDependent: anObject` permet de créer une dépendance entre le receveur et `anObject`

  `removeDependent: anObject` permet d'annuler la dépendance

  `dependents` retourne la liste des objets qui dépendent du receveur

NB. Le receveur de ces messages est le maître

Dépendances

- La catégorie de méthodes ‘‘change and update’’ de la classe `Object` permet de mettre en oeuvre la coordination

  `changed` conduit à envoyer le message `update` à tous les dépendants

  `changed: aParameter` permet d’associer un argument au message envoyé

  `broadcast: aSymbol` diffuse à tous les dépendants le message dont le sélecteur est `aSymbol`

  `update: aParameter` est exécuté quand un dépendant (le maître) est modifié (le receveur est l’esclave)

Exemple : feux de circulation

- Un seul feu peut être allumé à un instant donné

- L’état ON-OFF d’un feu dépend de l’état ON-OFF des autres feux

  Une variable `status` représente l’état du feu

  Lorsqu’un feu s’allume, les autres doivent s’eteindre

- La classe `Light` représente le feu et la classe `TrafficLigh` représente la gestion des feux (création, etc.)

Classe Light

class name `Light`

superclass `Object`

instance variables `status`

class methods

```
setOn ^self new setOn
setOff^self new setOff
```

instance methods

private

```
setOnstatus := true
setOffstatus := false
```


Classe Light

status

```
turnOn
   ^self isOff ifTrue: [status := true. self cha
turnOff
   ^self isOn ifTrue: [status := false
```

testing

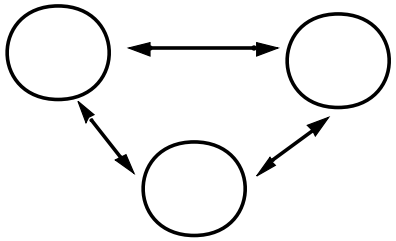
```
isOn ^status
isOff^status not
```

change

```
update: aLight
  aLight == self ifTrue: [self turnOff
```

Dépendances

- Lorsqu'un feu est allumé, il s'envoie le message `changed` qui provoque l'envoi du message `update:`aux dépendances (aux autres feux) - pour les eteindre -



Classe TrafficLight

class name `TrafficLight`

superclass `Object`

instance variables `lights`

class methods

instance creation

```
with: numberOfLights
    ^self new lights: aNumber
```

instance methods

operate

```
turnOn: aNumber(lights at: aNumber) turnO
```


Classe TrafficLight

private `lights: aNumber`

```
lights := Array new: aNumber.
lights at: 1 put: Light setOn.
2 to: aNumber do:
 [:index| lights at: index put: Light setO
lights do:
 [:light| lights do:
  [:depLight |
    light addDependent: depLight]]
```


Exemple : feux de circulation

```
trafficLight := TrafficLight with: 3.
```

- La méthode d'initialisation est `lights: numberOfLigh`
  Chaque feu est créé initialement éteint, sauf le premier feu
  Chaque feu est ensuite connecté à tous les autres

```
trafficLight turnOn: 2.
```

- Le 2ème feu est allumé (les 2 autres sont éteints)

```
trafficLight turnOn: 3.
```

- Le 3ème feu est allumé (les 2 autres sont éteints)

Messages

- Les messages sont aussi des objets (classe `Message`)
- Pour des raisons d'efficacité, les messages ne sont (en général) pas traités comme des objets
- Les messages peuvent être explicitement traités comme des objets (comme arguments) :

  `receiver perform: selector`

Messages

- La catégorie de méthodes ‘‘message handling’’ de la classe `Object` permet de traiter explicitement les messages

  `perform: aSymbol` transmet au receveur le message `aSymbol` (qui doit être sans argument)

  `perform: aSymbol with: anObject` transmet au receveur le message `aSymbol` avec l'argument `anObject` (on peut enchainer les `with`):

  `perform: aSymbol withArguments: anArray` transmet au receveur le message `aSymbol` avec plusieurs arguments (taille du tableau)

Exemple : Calculatrice

- Une calculatrice qui effecture des calculs du style:

  résultat := résultat + opérande

- Une variables d'instance : `result`

  `apply: aSymbol with: anArgument`
  exécute sur `result` l'operation désignée par `aSymbol`, avec `anArgument` pour argument

  `clear` remet le résultat à `0`

Exemple : Calculatrice

class name `Calculator`

superclass name `Object`

instance variable names `result`

class methods

instance creation

```
new
    ^super new initialize
```

Exemple : Calculatrice

instance methods

private

```
initialize
  result := 0
```

accessing

```
result
  ^result
clear
  self initialize
```

calculating

```
apply: aSymbol with: anArgument
  (result respondsTo: aSymbol)
    ifFalse: [self error: 'not understood']
  result := result perform: aSymbol
              with: anArgument
```


Exemple : Calculatrice

```
hp := Calculator new
```

- La variable d'instance est initialisée à `0`

```
hp apply: #+ with: 3.
```

- La méthode `apply:with:` exécute le code suivant

```
result := result perform: aSymbol
            with: anArgument
```

i.e.,

```
result: = 0 perform: #+ with: 3
```


Exemple : Calculatrice

```
hp := Calculator new.          result: 0
hp apply: #+ with: 3.          result: 3
hp apply: #* with: 2.          result: 6
hp apply: #- with: 4.          result: 2
hp apply: #to: with: 4.        result: (2 to: 4)
hp apply: #collect: [:x|x*2].  result: #(4 6 8)
```


Résumé

- Toutes les classes héritent de `Object` et les méthodes communes à tous les objets y sont mises en oeuvre
  - *Tests de fonctionnalités*
  - *Comparaisons*
  - *Copies*
  - *Accès aux variables (indexées)*
  - *Impression & Stockage*
  - *Gestion d'erreurs*
  - *Dépendances (explicites)*
  - *Gestion de messages*

Les Classes du Système:

Magnitude

Magnitude

- ***Les objets qui représentent une mesure comparable sont des instances indirectes de*** `Magnitude`

Parmi les sous-classes de `Magnitude`

- `Date`
- `Time`
- `Character`
- `Number`

Magnitude

- ***Les méthodes de la catégorie ‘‘comparing’’ permettent de comparer des mesures***

```
< aMagnitude
> aMagnitude
<= aMagnitude
>= aMagnitude
between: min and: max
=              (self subclassResponsibilit
```


Date

- ***Une instance de*** `Date` ***représente une date spécifique caractérisée par un jour et une année***

Méthodes de classe (catégorie ‘‘*general inquiries*’’)

```
daysInYear: yearInteger
```
-> nbr de jour ds l’année

```
nameOfMonth: monthIndex
```
-> nom d’un mois

```
daysInMonth: monthname forYear: yearInde
```

Date

Méthodes de classe (catégorie ‘‘*instance creation*’’)

```
today
```

e.g., `Date today` -> 2 December 2000

```
newDay: dayIndex
month: monthName
year: yearInteger
```


Date

Méthodes d’instances (catégorie ‘‘arithmetic’’)

`addDays: dayCount` -> une nouvelle date

```
examDate := Date today addDays: 14
```

`subtractDays: dayCount` -> une nouvelle date

```
StartDate := examDate subtractDays: 7
```

`subtractDate: aDate` -> un entier (nbr de jours)

```
leftDays := Date today subtractDate: examDate
```


Date : exemple

```
lastCourseDate :=
  (Date newDay: 25
        month: #December
        year: 1982)   subtractDays: 9
```

```
Date today < dueDate
  ifTrue: [tax := 0]
  ifFalse: [tax := 0.10 *
          (Date today subtractDate: dueDate)]
```


Time

- ***Une instance de*** `Time` ***représente une seconde spécifique dans une journée***

Méthodes de classes (catégorie ‘‘*general inquiries*’’)

`timeWords` -> le nbr de secs depuis le 01 01 1901 (Gwch)

`millisecondsToRun: timedBlock` -> le nbr de sec que met le block `timedBlock` pour retourner une valeur

Time

Méthodes de classes (catégorie ‘‘*instance creation*’’)

`now` -> la seconde à laquelle le message a été recu

Méthodes d’instances (catégorie ‘‘*arithmetic*’’)

`addTime: timeAmount` -> le temps actuel + un temps ou une date

`subtractTime: timeAmount`

Méthodes d’instances (catégorie ‘‘*conversion*’’)

`hours` -> un entier

`minutes` -> un entier

Time

```
startTime := Time now
....

moneyDue := (Time now subtractTime: startTi
            hours * 5.

((Time now subtractTime: startTime) minutes
   ifTrue: [moneyDue := moneyDue + 5]
```


Character

- ***Une instance de*** `Character` ***représente un caractère ($a)***

- `Character` est une sous-classe de `Magnitude` car les caractères sont comparables (par rapport à leur code ascii)

- Méthodes de classes (catégorie ‘‘*instance creation*’’)

  `value: anInteger` -> (e.g., 65 correspond à A)

Character

- Méthodes d’instances (catégorie ‘‘*accessing*’’)

  `asciiValue` -> (e.g., 65 correspond à A)

- Méthodes d’instances (catégorie ‘‘*testing*’’)

  `isAlphaNumeric`
  `isLowercase`
  `isSeparator`
  `isVowel`

Character : exemple

- `min: aString` associée à la classe `String`
  - `min: aString` retourne la chaine qui précède l'autre dans l'ordre alphabétique
  - Si les chaînes sont les mêmes, `min: aString` retourne la chaîne réceptrice du message

Utiliser

- `at:` pour accéder aux éléments de la chaîne
- `size` et `to: do:` pour parcourir la chaîne
- `ifTrue` pour les tests

Character : exemple

```
max: aString
 1 to: self size do:
  [:index|
   (index > aString size)
                    ifTrue[^aString].
   (self at: index) > (aString at: index)
                    ifTrue[^aString].
   (self at: index) < (aString at: index)
                    ifTrue[^self       ].
 ^self
```

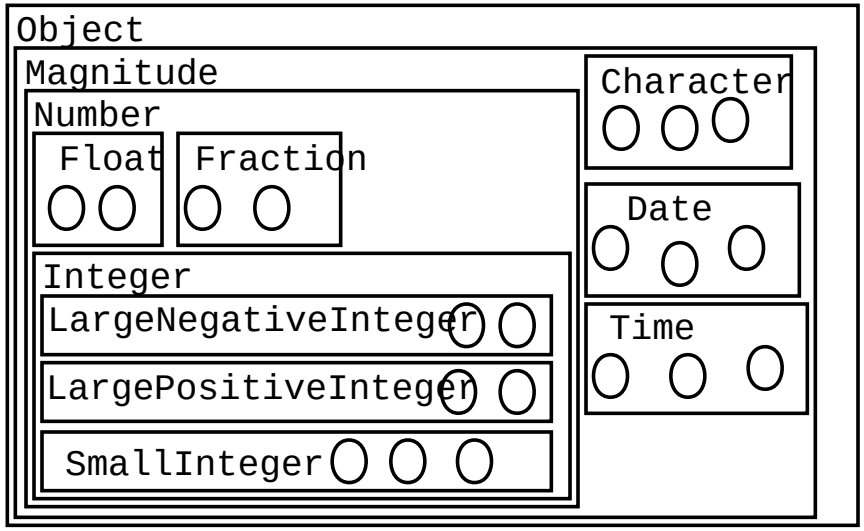

Number

- `Number` est une sous-classe abstraite de `Magnitude`
- `Float` et `Fraction` sont des sous-classes concrètes de `Number`
- `Integer` est une sous-classe abstraite de `Number`
- `LargeNegativeInteger, LargePositiveInteger` et `SmallInteger` sont des sous-classes concrètes de `Integer`

Integer

- Méthodes d'instances (catégorie ''*converting*'')
  `asFloat, asFraction, asCharacter`
- Méthodes d'instances (catégorie ''*enumerating*'')
  `timesRepeat: aBlock`
  `to: anInteger do: aBlock`

Héritage & Classes abstraites/concr



Résumé

- Toutes les classes dont les instances sont comparables héritent de `Magnitude`

`Date`
`Time`
`Character`
`Number`
  `Float`
  `Fraction`
  `Integer` `SmallInteger`, `LargePositiveInteger`, `LargeNegativeIntege`

Les Classes du Système:

Collection

Collection

- ***Les objets qui représentent un groupe d'objets sont des instances indirectes de*** `Collection`

  `#('word' 3 5 $G (1 2) tableau)`

Parmi les sous-classes de `Collection` (sans ordre)

  `Bags` (duplication), `Set` (pas de duplication)
  `Dictionaries` (paires)

Parmi les sous-classes de `Collection` (avec ordre)

  `OrderedCollection`, `Array`, `String` (ordre externe),
  `SortedCollection` (ordre interne)

Collection

Méthodes de classe (catégorie ‘‘*instance creation*’’)

`new: anInteger`

`with: anObject` -> crée une collection avec `anObject` comme (seul) élément

`with: firstObject with: secondObject...` -> (max 4)

```
Set with: $s with: $e with: $t
```


Collection

l ***Catégories de méthodes d’instance:***

messages pour rajouter des éléments

messages pour retirer des éléments

messages pour tester l’occurence d’éléments

messages pour parcourir les éléments

Collection

Méthodes d’instance (catégorie ‘‘*adding*’’)

`add: newObject` -> rajoute `newObject`

`addAll: aCollection` -> rajoute tous les éléments de `aCollection`

Méthodes d’instance (catégorie ‘‘*removing*’’)

`remove: oldObject` -> retire `oldObject`

`removeAll: aCollection` -> retire tous les éléments présents dans `aCollection`

`remove: oldObject ifAbsent: anExceptionBl`

-> retire `oldObject` s’il se trouve dans la collection et exécute le bloc sinon

Collection

Méthodes d’instance (catégorie ‘‘*testing*’’)

`includes: newObject` -> retourne un booléen

`isEmpty` -> retourne un booléen

`occurencesOf: anObject` -> retourne un entier

Méthodes d’instance (catégorie ‘‘*enumerating*’’)

`do: aBlock` -> évalue le bloc pour chaque élément

```
count :=0.
letters do: [:each | each == $a
                ifTrue: [count:=count+1]]
```

Collection

Méthodes d'instance (catégorie ''*enumerating*'')

`collect:aBlock` -> évalue le bloc pour chaque élément et met les résultats dans une nouvelle collection

```
employees collect: [:each | each salary]
```

`select:aBlock` -> crée une nouvelle collection dont les éléments sont ceux pour lesquels `aBlock` est évalué à vrai

```
(letters select: [:each | each == $a]) size
```

`reject:aBlock` -> crée une nouvelle collection dont les éléments sont ceux pour lesquels `aBlock` est évalué à faux

```
(letters reject: [:each | each ~~ $a]) size
```


Collection: exemples

```
select: aBlock
  #(1 2 3 4 5 6 7) select: [:x| x odd].
  -> #(1 3 5 7)
```

`reject:aBlock` -> crée une nouvelle collection dont les éléments sont ceux pour lesquels `aBlock` est évalué à vrai

```
  #(1 2 3 4 5 6 7) reject: [:x| x odd].
  -> #(2 4 6)
collect: aBlock
  #(1 2 3 4 5 6 7) collect: [:x| x odd].
  -> #(true false true false true false
```


Collection

Méthodes d'instance (catégorie ''*converting*'')

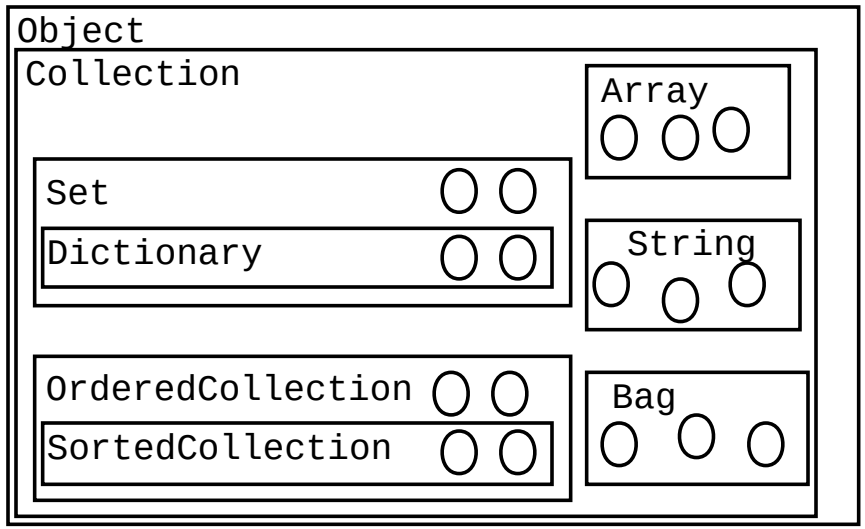
`asBag`

`asSet` (garde une copie de chaque élément)

`asOrderedCollection` (l'ordre est arbitraire)

`asSortedCollection` (l'ordre est <=)

Héritage & Classes abstraites/conc



Bag

- Le sac est la forme la plus simple de collection (pas d'ordre et pas de clé externe)

`aBag` ne répond pas à `at:` et `at:put:`

`size` retourne le nombre total d'éléments dans le sac

`add: newObject withOccurences: anInteger`

Rajoute un certain nombre (`anInteger`) de duplicas de newObject

Bag

- *Créer un sac qui contient 1 steak à 5 Fr. et 6 bières à 1.5 Fr. puis calculer le total*

Utiliser :

`class Product`

méthode de classe `of: name at: price`

méthode d'instance `price`

```
add: newObject withOccurences: anInteger
do: aBlock
```


Bag

```
basket := Bag new.
basket add: (Product of: #steak at: 5).
basket add: (Product of: #beer at: 1.5)
                withOccurences: 6.

total := 0.
basket do: [:each | total := total + each price].
```


Set

- Un ensemble est similaire à un sac, sauf qu'il ne contient pas de duplicas

`aSet` ne répond pas à `at:` et `at:put:`

`size` retourne le nombre total d'éléments dans l'ensemble

`add: newObject`

Rajoute `newObject` s'il n'existe pas déjà

Dictionary

- Un dictionnaire est une collection non ordonnée, qui permet d'accéder à des valeurs en utilisant des clés
- Un dictionnaire contient des paires (clé, valeur) : chaque paire est une instance de la classe `Association`

`size` retourne le nombre d'associations dans le dictionnaire

`at:` et `at: put:` s'appliquent aux clés du dictionnaire

`do: aBlock, collect: aBlock, select: aBlo`
`reject: aBlock`

-> évaluent le bloc sur chaque valeur du dictionnaire

Dictionary

Méthodes d'instance (catégorie ''*accessing*'' de `Object`)

`at: key ifAbsent: aBlock`

retourne la valeur correspondante à la clé `key`, et si elle n'existe pas, retourne le résultat de l'évaluation de `aBlock`

`associationAt: key`

retourne l'association correspondant à la clé `key`

`keys` -> retourne l'ensemble des clés

Dictionary

Méthodes d'instance (catégorie ''*testing*'')

`includesKey: key` -> teste si `key` est dedans

Méthodes d'instance (catégorie ''*removing*'')

`removeKey: key`

-> retire l'association correspondante à la clé `key`

Méthodes d'instance (catégorie ''*dictionary*'')

`keysDo: aBlock`

-> évalue le bloc sur chaque clé du dictionnaire

Dictionary

```
opposites := Dictionary new at: #hot put: #
        at: #stop put: #go; at: #come put
```

`opposites size` -> 3.

`opposites includes: #cold` -> true.

```
opposites do: [:word | Transcript
        show: word printString]
opposites keysDo: [:word | Transcript
        show: (opposites at: word) printS
```

OrderedCollection

- Collection ordonnée en fonction de l'ajout et du retrai.

Méthodes (catégories ''*accessing, adding removing*'')

```
indexOf: anObject
after: anObject      -> retourne l'élément après aObject
before: anObject     -> retourne l'élément avant
add: newObject after: oldObject
addFirst: newObject, addLast: newObject
removeFirst: anObject, removeLast: anObje
```


SortedCollection

- Collection ordonnée en fonction d'un critère interne (sous classe de `OrderedCollection`), le critère est <= par défaut.
- Les messages du type `addLast` ne sont pas permis.

Méthodes de classe (catégories ''*creation*'')

```
sortBlock: aBlock
```

-> Le bloc représente une fonction à deux arguments qui permet de définir l'ordre de la collection

SortedCollection: exemple

```
collection1 := SortedCollection new.
collection2 := SortedCollection sortBlock:
          [:a :b | a >= b].

collection1 add: 4; add: 5; add: 2; add: 3.
collection2 add: 4; add: 5; add: 2; add: 3.

collection1     -> (2, 3, 4, 5)
collection2     -> (5, 4, 3, 2)
```


String

- Collection ordonnée de caractères dont les clés sont des entiers

Méthodes de classe (catégories ''*creation*'')

```
fromString: aString    -> crée une copie
```

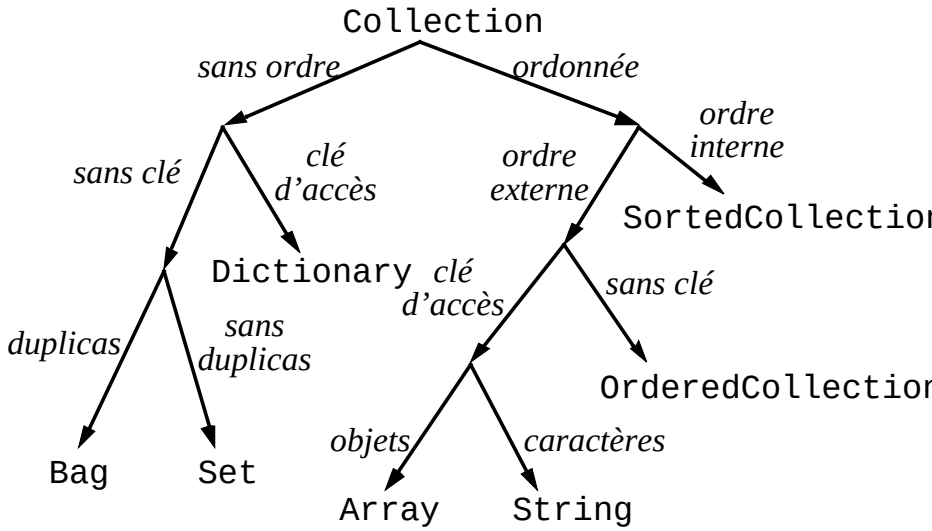
Méthodes d'instance (catégories ''*comparing*'')

```
< aString              -> comparaison alphabétique
```

Résumé

- Toutes les classes dont les instances sont des groupes d'objets héritent de `Collection`
  - `Array`
  - `Bag`
  - `String`
  - `Set`
    - `Dictionary`
  - `OrderedCollection`
    - `SortedCollection`

Quelle classe choisir ?



Queues de Priorité

et

Simulation Discrète

Illustrations en Smalltalk

Références

- **NB: Les références suivantes sont données à titre indicatif.**
- **Programmes et Objets Informatiques**
  - C. Rapin, Polycopié EPFL, 1989
- **Algorithms (2nd ed.)**
  - R. Sedgewick, Addison-Wesley, 1988
- **Simulation par événements discrets**
  - P.-J. Erard, P. Déguénon, PPUR, 1996

Queues de priorité

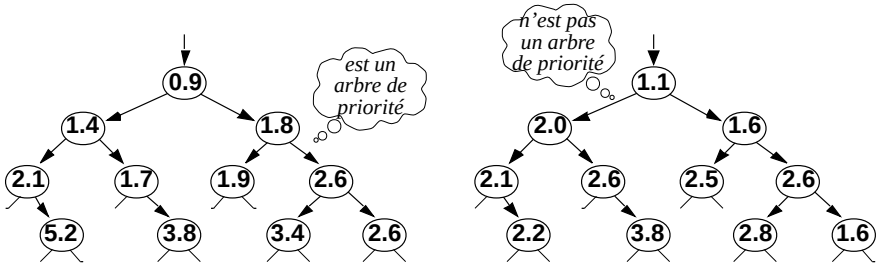
- **Définition**
  - Une queue de priorité est un ensemble de données dans lequel il est possible d'insérer de nouveaux éléments.
  - Les éléments d'une queue de priorité sont retirés de la queue selon un critère de priorité fixé à l'avance.
- **Relation de priorité (notée $\rightarrow$)**
  1. **not** ($x \rightarrow y$ **and** $y \rightarrow x$)
  2. **not** ($x \rightarrow y$ **or** $y \rightarrow x$) $\Rightarrow (z \rightarrow x \Leftrightarrow z \rightarrow y)$
  3. ($x \rightarrow y$ **and** $y \rightarrow z$) $\Rightarrow x \rightarrow z$
- **Opérations**
  - `add:` insère un nouvel élément dans la liste.
  - `first` retourne l'élément de plus grande priorité.
  - `removeFirst` élimine de la queue son élément de plus grande priorité.
  - `isEmpty` retourne vrai si la queue est vide.

Implantation

- **Liste triée**
  - Possible mais, l'insertion d'un élément est *O(n)*.
  - Le retrait de l'élément prioritaire est *O(1)*.
  - Implémentation naïve à éviter!
- **Arbre de priorité**
  - L'insertion d'un élément est *O(log n)*.
  - Le retrait de l'élément prioritaire est *O(log n)*.
  - Un arbre de priorité est un arbre binaire.
  - La valeur stockée à chaque noeud est celle qui a la plus grande priorité.
  - Les deux sous-arbres contiennent des valeurs de plus faible priorité que le noeud.

Arbre de priorité (exemple)

- **Exemple avec la relation x<y**



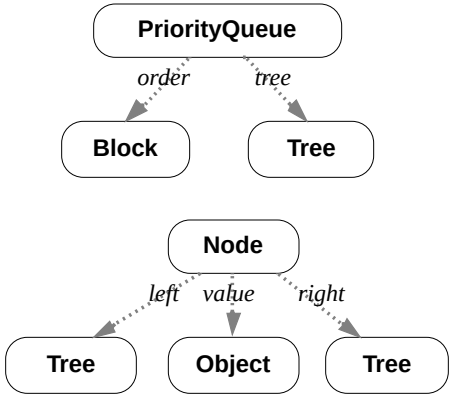
- **Remarque**
  - Dans un arbre de priorité, si on échange les deux sous-arbres d'un noeud quelconque, alors l'arbre résultant est encore un arbre de priorité.

Arbre de priorité (principe)

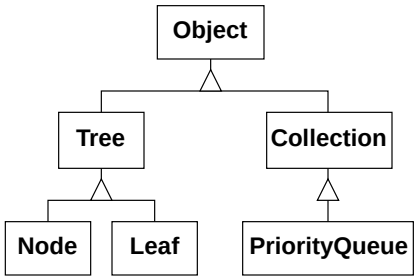
- **Idée**
  - Basé sur le principe que la fusion de deux arbres prend un temps raisonnable.
- **Ajouter un élément**
  - On fabrique un arbre de priorité avec un noeud dans lequel est stocké la nouvelle valeur.
  - On fusionne les deux arbres.
- **Retirer l'élément prioritaire**
  - On élimine le noeud à la racine de l'arbre, après en avoir récupéré la valeur.
  - On fusionne les deux sous-arbres de ce noeud (droite et gauche)
- **Fusionner deux arbres**
  - Soient *p* et *q* les deux arbres que l'ont veut fusionner.
  - Si *q* est vide, le résultat est *p*. Si *p* est vide, le résultat est *q*.
  - On fait en sorte que *p* soit prioritaire par rapport à *q*. (On les échange au besoin)
  - On échange les deux sous-arbres de *p*. (Important sinon l'arbre dégénère en liste triée!)
  - On fusionne (récursivement) l'arbre *q* avec l'un des sous-arbres de *p*.

Arbre de priorité (structure)

q **Attributs**



q **Héritage**



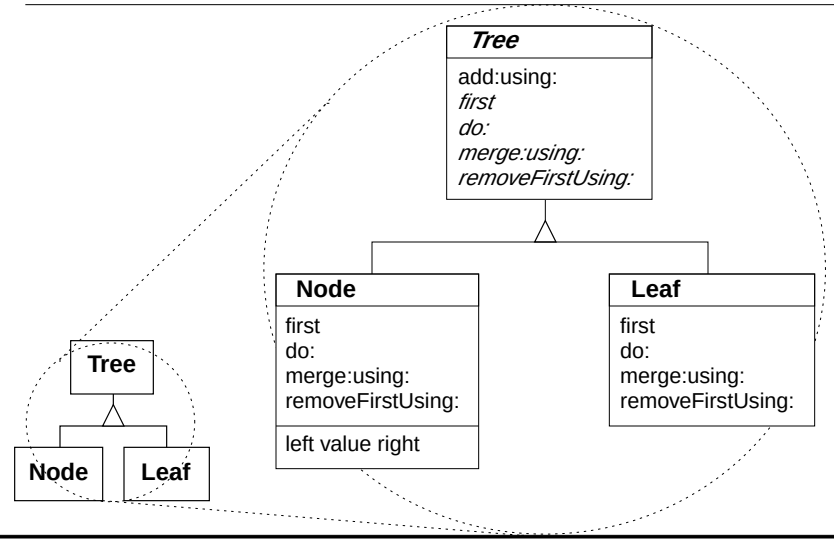
Queue de priorité

```
Collection subclass: #PriorityQueue
    instanceVariableNames: 'order tree'
class methods
    withRelation: aBlock
      ^(super new) initialize: order: aBlock
instance methods
 initialize-release
    initialize
       tree := Leaf new
       order := [:x :y | x] ≤ y
 accessing
    order: aBlock
       order := aBlock
    first          ^tree first
 testing
    isEmpty        ^tree first isNil
```


Queue de priorité (suite)

```
adding
   add: anObject
      tree := tree add: anObject usingOrder
removing
   removeFirst
      |first|
      first := tree first.
      tree := tree removeFirstUsingOrder.
      ^first
   remove: anObject ifAbsent: aBlock
      self shouldNotImplement
enumerating
   do: aBlock
      tree do: aBlock
```


Arbre de priorité



Arbre de priorité (Tree)

```
Object subclass: #Tree
    instanceVariableNames: ''
instance methods
  adding
    add: anObject using: aBloc
      ^self merge: (Node with: anObject) using: aBloc
  accessing
    first          self subclassResponsibility
  enumerating
    do: aBloc      self subclassResponsibility
  removing
    removeFirstUsing: aBloc
      self subclassResponsibility
  private
    merge: aTree using: aBloc
      self subclassResponsibility
```


Arbre de priorité (Leaf)

```
Tree subclass: #Leaf
    instanceVariableNames: ''
instance methods
  accessing
    first
      ^nil
  enumerating
    do: aBloc
      ^self
  removing
    removeFirstUsing: aBloc
      ^self
  private
    merge: aTree using: aBloc
      ^aTree
```


Arbre de priorité (Node)

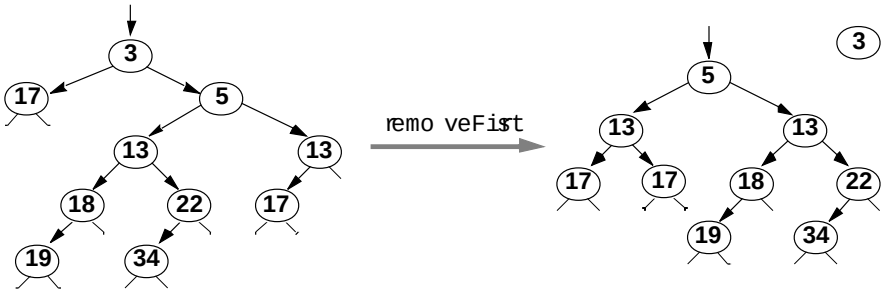
```
Tree subclass: #Node
    instanceVariableNames: 'left value right'
instance methods
  accessing
    first
      ^value
  enumerating
    do: aBloc
      aBloc value: value.
      left do: aBloc.
      right do: aBloc
  removing
    removeFirstUsing: aBloc
      ^left merge: right using: aBloc
```


Arbre de priorité (PriorityNode)

```
private
  merge: aTree using: aBloc
    aTree first isNil
      ifTrue: [^self].
    (aBloc value: aTree first value: self first)
      ifTrue: [^aTree merge: self using: aBloc].
    ^PriorityNode
        with: value
        left: (right merge: aTree using: aBloc)
        right: left
```

Exemple (queue de priorité)

```
|queue|
queue := PriorityQueue withRelation: [:x :y| x < y].
queue addAll: #(34,13,19,5,17,22,13,18,3,17).
```



Simulation

- **Objectif**
  - Simuler un système réel, au moyen d'un ordinateur.
  - Le temps intervient explicitement, il s'agit de simuler l'évolution d'un système pendant une période donnée.
  - Deux approches: *simulation continue* et *simulation discrète*.
- **Simulation continue**
  - Le système évolue de manière *continue* au cours du temps; il est représenté au moyen d'une équation différentielle, d'une équation aux dérivées partielles ou d'un système.
  - L'ordinateur intervient pour résoudre ces équations *numériquement*.
- **Simulation discrète**
  - Le système ne change d'état qu'à des instants bien déterminés; t_1, t_2, t_3, ..., t_n.
  - Chaque changement d'état du système est déclanché par un *événement*.
  - L'ordonnancement correct des événements est géré par un *échéancier*.

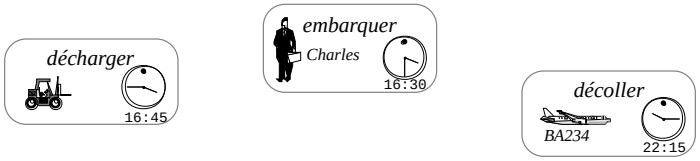
Simulation (exemples)

- **Simulation continue**
  - Simulation de l'évolution de l'atmosphère terrestre pour effectuer des prévisions météo.
  - À partir des propriétés des gaz composant l'atmosphère, son évolution est représentée au moyen d'équations aux dérivées partielles découlant des lois de la physique.
- **Simulation discrète**
  - Simulation de guichets pour optimiser les files d'attente.
  - Le système change d'état, p.ex., à chaque fois qu'un nouveau client se présente.
  - Les clients arrivent aléatoirement selon une loi de probabilité donnée.
  - Les événements sont par exemple: "nouveau client", "début transaction", "départ client".

Simulation discrète

- **Temps virtuel**
  - Le système doit gérer le temps pour le système simulé. On parle de *temps virtuel*.
- **Événements**
  - Un événement déclanche une *action*.
  - Un événement est activé à un *instant* donné. (en temps virtuel)
- **Échéancier**
  - L'échéancier gère l'ordonnancement des événements.
  - Il gère aussi l'évolution du temps virtuel.
  - Il est généralement construit autour d'une queue de priorité de notices d'événement.

Événements



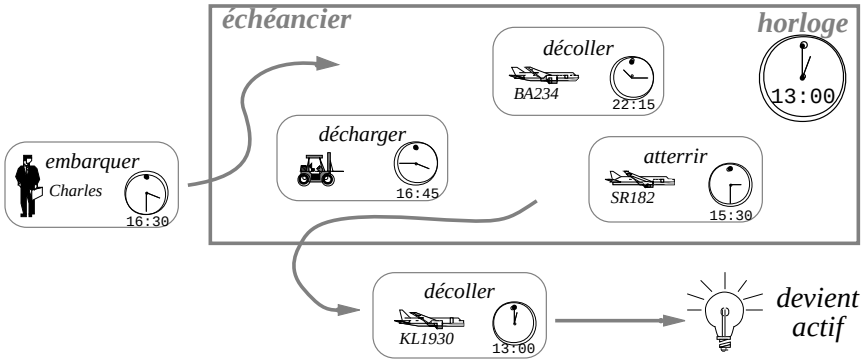
- **Action future**
  - p.ex., `[charles embarquer]`
  - On peut la représenter sous la forme d'un bloc.
  - Le bloc est évalué lors du déclanchement de l'événement.

- **Heure de déclanchement**
  - p.ex., **16:30**
  - L'action doit être déclanchée lorsque le temps virtuel est égal à l'heure de déclanchement.

Événements

```
Object subclass: #Event
    instanceVariableNames: 'time action'
class methods
    at: aTime do: aBlock
      ^(super new) time: aTime; action: aBlock; yourself
instance methods
  private
    time: aTime  time := aTime
    action: aBlock action := aBlock
  accessing
    time         ^time
  evaluating
    trigger      action value
  comparing
    precedes: anEvent
      ^(time < anEvent time)
```


Échéancier



- L'échéancier gère le temps virtuel.
- L'échéancier est responsable de gérer correctement l'ordonnancement des événements.
- Les événements sont extraits de l'échéancier en fonction de l'heure de déclanchement de l'action correspondante.

Échéancier

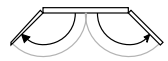
```
Object subclass: #Scheduler
    instanceVariableNames: 'queue clock'
class methods
    new
      ^(super new) initialize
instance methods
  initialize-release
    initialize
      queue := PriorityQueue withRelation: [:x :y | x precedes: y].
      clock := 0.
  scheduling
    at: aTime do: aBlock
      queue add: (Event at: aTime do: aBlock).
    wait: aDuration thenDo: aBlock
      self at: clock + aDuration do: aBlock
```

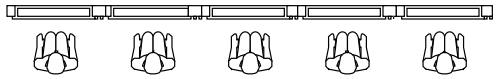
Échéancier (suite)

```
accessing
   clock
      ^clock
executing
   step
      (queue isEmpty)
         ifFalse: [self basicStep]
   run
      [queue isEmpty]
         whileFalse: [self basicStep]
private
   basicStep
      clock := queue firstTime.
      (queue removeFirst) trigger.
```


Exemple: simulation de guichets de poste

- **Événements**
  - Ouverture de la poste
  - Fermeture de la poste
  - Nouveau client
  - Début transaction
  - Fin transaction
- **Initialisation de la simulation**
  - 8:00 Ouverture de la poste
  - 12:00 Fermeture de la poste



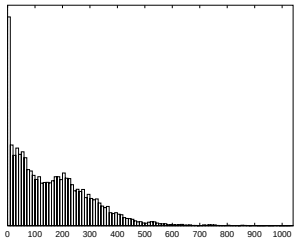


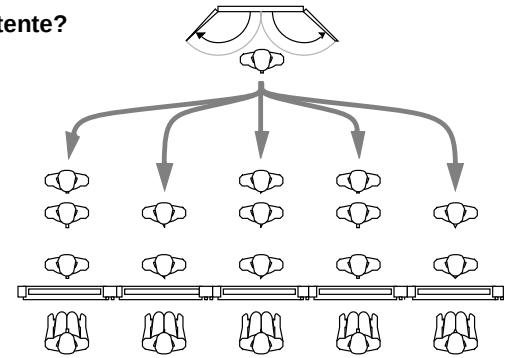
Poste (événements)

- **Ouverture de la poste**
  - Poste ouverte.
  - Générer événement "nouveau client".
- **Fermeture de la poste**
  - Poste fermée.
- **Nouveau client**
  - Si la poste est fermée: ne rien faire.
  - Générer événement "nouveau client". (le client suivant)
  - Si queue alors faire la queue.
  - Sinon, générer événement "début transaction".
- **Début transaction**
  - Générer événement "fin transaction".
- **Fin transaction**
  - Générer événement "début transaction". (pour le client suivant)

Files d'attente distribuées

- **Chaque guichet a sa propre file d'attente**
  - Modèle suisse, français, ...
- **Quel est le temps moyen d'attente?**
  - temps minimum: 0
  - temps moyen: 161.508
  - temps maximum: 1048.93





File d'attente centralisée

- **Une seule file d'attente pour tous les guichets**
  - Modèle américain, anglais, japonais, ...
- **Quel est le temps moyen d'attente?**
  - temps minimum: 0
  - temps moyen: 149.837
  - temps maximum: 398.2