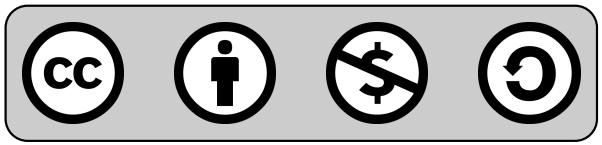
IN201 Conception et Programmation Orientées Objet

Christophe Garion
DMIA – ISAE

License CC BY-NC-SA 3.0



Citations. . .

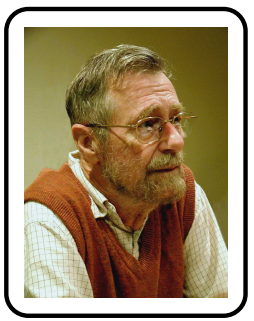
Programming is one of the most difficult branches of applied mathematics ; the poorer mathematicians had better remain pure mathematicians.

Edsger W. Dijkstra

Object-oriented programming is an exceptionally bad idea which could only have originated in California.

Edsger W. Dijkstra

E. W. Dijkstra (1930–2002) is one of the most influential computer scientist of 20^{th} century. He received the 1972 Turing Award and has written papers among the most important for CS.



Plan général du cours

1 - Introduction

Plan de la partie 1 - Introduction

Génie logiciel

Hypothèse : le « génie » (science de l'ingénieur) cherche la qualité.

Définition

Le génie logiciel est la production de logiciels de qualité.

Terme « génie » : science de l'ingénieur

- bases théoriques
- méthodes et outils validés par la pratique

Définition (génie logiciel)

Art de **spécifier**, **concevoir**, **réaliser** et **faire évoluer**, avec des moyens et dans des délais raisonnables des **programmes**, des **documentations** et des **procédures** de qualité en vue d'utiliser un ordinateur pour résoudre certains problèmes.

Qualités d'un logiciel

- externes : celles qui sont importantes !
- interne : modularité, lisibilité

Les qualités internes influent sur les qualités externes...

Ouvrage de référence (disponible à la bibliothèque) :

Meyer, B. (1997).
Object-Oriented Software Construction.
2e éd.
Prentice Hall.

Qualités d'un logiciel

correct	robuste	extensible	réutilisable
compatible	efficient	facile	portable
fonctionnel	livré à temps	vérifiable	intègre
réparable	économique	maintenable	documenté

Définition (correction)

Un logiciel est correct s'il agit conformément à ses spécifications.

- écriture précise des spécifications
- correction conditionnelle
- tests, approches formelles

Qualités d'un logiciel

correct	robuste	extensible	réutilisable
compatible	efficient	facile	portable
fonctionnel	livré à temps	vérifiable	intègre
réparable	économique	maintenable	documenté

Définition (robustesse)

Un logiciel est robuste s'il agit de façon approprié à des événements anormaux.

- cas difficiles à prévoir
- exemple typique : entrée utilisateur

```
scanf("%d", &i);
i++;
```

Qualités d'un logiciel

correct	robuste	extensible	réutilisable
compatible	efficient	facile	portable
fonctionnel	livré à temps	vérifiable	intègre
réparable	économique	maintenable	documenté

Définition (extensibilité)

Un logiciel est extensible s'il s'adapte facilement aux changements de spécifications

- *software*...
- changement de spécifications, de la façon dont on les comprend, des algorithmes, des structures de données...
- deux idées : simplicité de la conception et décentralisation

Qualités d'un logiciel

correct	robuste	extensible	réutilisable
compatible	efficient	facile	portable
fonctionnel	livré à temps	vérifiable	intègre
réparable	économique	maintenable	documenté

Définition (réutilisabilité)

Un logiciel est réutilisable s'il peut servir pour construire plusieurs applications.

- on ne réinvente pas la roue...
- permet de se concentrer sur les points durs (robustesse et correction par exemple)

Qualités d'un logiciel

correct	robuste	extensible	réutilisable
compatible	efficient	facile	portable
fonctionnel	livré à temps	vérifiable	intègre
réparable	économique	maintenable	documenté

Définition (compatibilité)

Un logiciel est compatible s'il peut se combiner facilement avec d'autres logiciels.

- standardisation : formats de fichiers, structures de données, UI, ...
- un exemple : en 1992, programme de réservation de voitures et d'hôtels par AMR, 200 ingénieurs, 47000 pages de spécifications, 165 M$ à la poubelle...

Qualités d'un logiciel

correct	robuste	extensible	réutilisable
compatible	efficient	facile	portable
fonctionnel	livré à temps	vérifiable	intègre
réparable	économique	maintenable	documenté

Définition (efficience)

Un logiciel est efficient s'il utilise aussi peu de ressources que possible.

- deux approches : optimisations fines et augmentation de la rapidité des machines
- pas forcément opposées
- peut être contradictoire avec extensible et réutilisable.

Qualités d'un logiciel

correct	robuste	extensible	réutilisable
compatible	efficient	facile	portable
fonctionnel	livré à temps	vérifiable	intègre
réparable	économique	maintenable	documenté

Définition (facilité d'utilisation)

Un logiciel est facile d'utilisation si des gens avec des qualifications et des connaissances différentes peuvent se servir du logiciel pour résoudre des problèmes.

➥ connaître l'utilisateur !

Qualités d'un logiciel

correct	robuste	extensible	réutilisable
compatible	efficient	facile	portable
fonctionnel	livré à temps	vérifiable	intègre
réparable	économique	maintenable	documenté

Définition (portablilité)

Un logiciel est portable s'il peut être utilisé sur des environnements (matériel + système d'exploitation) différents.

Qualités d'un logiciel

correct	robuste	extensible	réutilisable
compatible	efficient	facile	portable
fonctionnel	livré à temps	vérifiable	intègre
réparable	économique	maintenable	documenté

Définition (fonctionnalités)

Un logiciel est fonctionnel s'il fournit un certain nombre de possibilités.

- *creeping featurism*
- il ne faut pas se focaliser dessus et oublier les autres qualités !

Qualités d'un logiciel

correct	robuste	extensible	réutilisable
compatible	efficient	facile	portable
fonctionnel	livré à temps	vérifiable	intègre
réparable	économique	maintenable	documenté

Définition (livraison à temps)

Un logiciel est livré à temps s'il est livré quand ou **avant** que ses utilisateurs en ont besoin.

Qualités d'un logiciel

correct	robuste	extensible	réutilisable
compatible	efficient	facile	portable
fonctionnel	livré à temps	vérifiable	intègre
réparable	économique	maintenable	documenté

Définition (vérifiabilité)

Un logiciel est vérifiable si on a établi des procédures permettant de détecter des erreurs durant sa phase de validation et de mise en exploitation.

Qualités d'un logiciel

correct	robuste	extensible	réutilisable
compatible	efficient	facile	portable
fonctionnel	livré à temps	vérifiable	intègre
réparable	économique	maintenable	documenté

Définition (intégrité)

Un logiciel est intègre s'il protège ses composants de modifications et d'accès non autorisés.

Qualités d'un logiciel

correct	robuste	extensible	réutilisable
compatible	efficient	facile	portable
fonctionnel	livré à temps	vérifiable	intègre
réparable	économique	maintenable	documenté

Définition (réparabilité)

Un logiciel est réparable si l'on peut corriger ses fautes facilement.

Qualités d'un logiciel

correct	robuste	extensible	réutilisable
compatible	efficient	facile	portable
fonctionnel	livré à temps	vérifiable	intègre
réparable	économique	maintenable	documenté

Définition (économique)

Un logiciel est économique s'il coûte ce qui avait été prévu dans le budget ou moins.

Qualités d'un logiciel

correct	robuste	extensible	réutilisable
compatible	efficient	facile	portable
fonctionnel	livré à temps	vérifiable	intègre
réparable	économique	maintenable	documenté

Problème de la **maintenance** :

- la phase de maintenance commence à la livraison du logiciel
- environ 70% du coût du logiciel
- peut paraître bizarre pour un logiciel
- deux types de maintenance :
  - maintenance noble : évolution des spécifications
  - maintenance moins noble : *debug* tardif

Qualités d'un logiciel

correct	robuste	extensible	réutilisable
compatible	efficient	facile	portable
fonctionnel	livré à temps	vérifiable	intègre
réparable	économique	maintenable	documenté

- conséquence nécessaire de toutes ces qualités
- documentation externe : pour les utilisateurs
- documentation interne : pour les développeurs
- documentation sur les interfaces des modules

Quelles qualités choisir ?

Compromis à faire entre toutes ces notions

- intègre vs. facile d'utilisation
- efficient vs. portable et réutilisable
- ...

Quatre points importants actuellement

- correct et robuste : trop difficile de produire du code sans erreurs et de corriger les erreurs
  Techniques utilisées : spécifications formelles, test durant tout le processus, typage statique etc.
  ➥ logiciel **sûr**
- **extensible et réutilisable**
  ➥ logiciel **modulaire** : trouver des composants logiciels autonomes

Quelles qualités choisir ?

Compromis à faire entre toutes ces notions

- intègre vs. facile d'utilisation
- efficient vs. portable et réutilisable
- ...

Affirmation (à vérifier à la fin du cours...)

Les approches OO permettent d'améliorer ces qualités.

Plan de la partie 1 - Introduction

Qu'attend-on ?

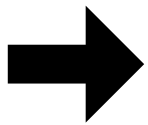
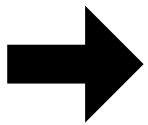
extensible et réutilisable $\rightarrow$ modulaire

Définition (informelle...)

Une méthode de construction de logiciels est modulaire si elle permet de construire des logiciels à partir de composants **autonomes** et **stables** connectés entre eux par une structure **simple**.

Module : unité de base de décomposition

Approche pour une méthode de **construction de logiciels** :

5 critères  5 règles  5 principes

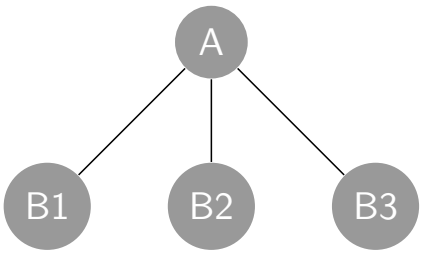
5 critères pour la modularité

- **décomposabilité**
- composabilité
- compréhension
- continuité
- protection

Définition (décomposabilité)

Une méthode de construction de logiciels satisfait le critère de décomposabilité si elle permet de décomposer un problème en plusieurs sous-problèmes moins complexes connectés par une structure simple et suffisamment indépendants.

- permet également la répartition des tâches
- il faut connaître parfaitement les dépendances entre modules



5 critères pour la modularité

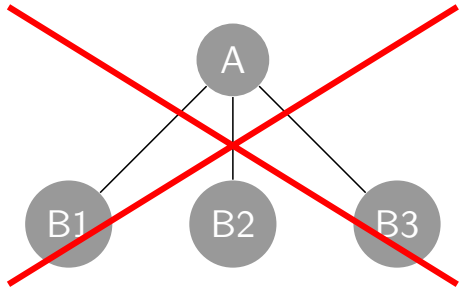
- décomposabilité
- **composabilité**
- compréhension
- continuité
- protection

Définition (composabilité)

Une méthode de construction de logiciels satisfait le critère de composabilité si elle favorise la production d'éléments logiciels qui peuvent être combinés librement entre eux pour produire de nouveaux systèmes.

- permet évidemment la réutilisabilité
- indépendant de la décomposabilité

conventions
shell Unix



5 critères pour la modularité

- décomposabilité
- composabilité
- compréhension
- continuité
- protection

Définition (compréhension)

Une méthode de construction de logiciels satisfait le critère de compréhension si elle permet de construire des logiciels pour lesquels un humain est capable de comprendre chaque module si avoir à connaître les autres modules ou un minimum.

- paraît évident pour une méthode « modulaire »...

5 critères pour la modularité

- décomposabilité
- composabilité
- compréhension
- **continuité**
- protection

Définition (continuité)

Une méthode de construction de logiciels satisfait le critère de continuité si dans les architectures qu'elle produit, un petit changement de spécification du problème amène à des changements sur un ou un petit nombre de modules.

- relié à la propriété d'extensibilité
- intuition identique à la définition mathématique
- un bon exemple : déclaration de constantes symboliques

5 critères pour la modularité

- décomposabilité
- composabilité
- compréhension
- continuité
- protection

Définition (protection)

Une méthode de construction de logiciels satisfait le critère de protection si dans les architectures qu'elle produit, les effets à l'exécution d'une condition anormale dans un module reste confinée à ce module ou dans le pire des cas ne se propage qu'à quelques modules voisins.

- les erreurs se produisent à l'exécution
- traite de la propagation des erreurs
- un bon exemple : validation d'entrées utilisateur

5 règles pour la modularité

- mapping direct
- peu d'interfaces
- couplage faible
- interfaces explicites
- masquage d'information

Règle (mapping direct)

La structure modulaire induite par le processus de construction du logiciel doit être compatible avec toute structure modulaire modélisant le problème.

- correspondance solution $\leftrightarrow$ problème
- permet d'assurer en partie la continuité

5 règles pour la modularité

- mapping direct
- peu d'interfaces
- couplage faible
- interfaces explicites
- masquage d'information

Règle (peu d'interfaces)

Chaque module ne doit communiquer qu'avec le plus petit nombre d'autres modules.

- les modules peuvent s'appeler, partager des données etc.
- pour n modules, on devrait s'approcher plus de $n-1$ connexions que de $\frac{n(n-1)}{2}\ldots$
- permet d'assurer en particulier la continuité et la protection

5 règles pour la modularité

- mapping direct
- peu d'interfaces
- couplage faible
- interfaces explicites
- masquage d'information

Règle (couplage faible)

Si deux modules communiquent, ils doivent échanger aussi peu d'information que possible.

- communication limitée
- permet d'assurer la continuité et la protection

5 règles pour la modularité

- mapping direct
- peu d'interfaces
- couplage faible
- interfaces explicites
- masquage d'information

Règle (interfaces explicites)

Si A et B communiquent, cela doit être évident en lisant le texte de A et/ou de B.

- permet de garantir la composabilité, la décomposabilité, la continuité et la compréhension
- attention aux intermédiaires (variables partagées etc.)

5 règles pour la modularité

- mapping direct
- peu d'interfaces
- couplage faible
- interfaces explicites
- masquage d'information

Règle (masquage d'information)

Le concepteur de chaque module doit sélectionner un sous-ensemble des propriétés du module comme information disponible aux modules clients.

- dualité **public/privé**
- propriétés publiques $\equiv$ **interface**
- permet d'assurer la continuité + composabilité, décomposabilité et compréhension
- **très important**

5 principes pour la modularité

unité syn.

autodoc

accès uniforme

ouvert fermé

choix unique

Principe (unité syntaxique modulaire)

Les modules doivent être les unités syntaxiques du langage utilisé.

- pour le langage de **programmation**, de **conception**, de **spécification**
- en programmation, modules compilables séparément
- correspond à tous les critères !

5 principes pour la modularité

unité syn.

autodoc

accès uniforme

ouvert fermé

choix unique

Principe (auto-documentation)

Le concepteur d'un module doit le documenter à l'intérieur de ce dernier.

- ne peut pas séparer la documentation pour qu'elle soit à jour
- c'est le rôle du concepteur/développeur
- correspond en particulier à la continuité

5 principes pour la modularité

unité syn.

autodoc

accès uniforme

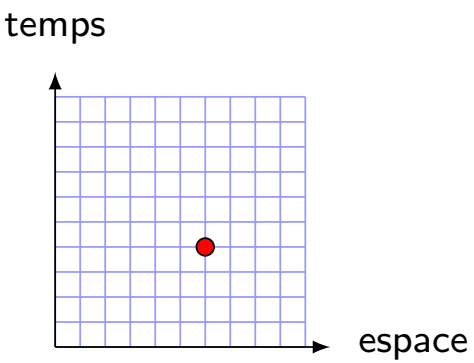
ouvert fermé

choix unique

Principe (accès uniforme)

Tous les services offerts par un module doivent être accédés par une notation unique, en particulier qu'ils soient stockés ou calculés.

- **très important**
- correspond en particulier à la continuité et au masquage d'information



Exemple classique : solde d'un compte bancaire

5 principes pour la modularité

unité syn.

autodoc

accès uniforme

ouvert fermé

choix unique

Principe (ouvert-fermé)

Les modules doivent être à la fois ouverts et fermés.

- module **ouvert** : on peut étendre son ensemble d'opérations ou ses champs de données
- module **fermé** : utilisable par d'autres modules (ses clients)

5 principes pour la modularité

unité syn.

autodoc

accès uniforme

ouvert fermé

choix unique

Principe (choix unique)

Lorsqu'un système offre plusieurs choix, ces choix doivent être connus par un et un seul module.

- exemple classique : une liste de types de données utilisables par d'autres modules

Réutilisabilité : pourquoi ?

Bénéfices attendus pour le « client »

- temps : on utilise des composants qui existent déjà
- moins de maintenance
- « sérieux »
- efficacité

Bénéfices attendus pour le « producteur »

- consistance du développement
- investissement : préserve le savoir-faire des développeurs

Réutilisabilité : quoi ?

Que doit-on réutiliser ?

- les développeurs ☺
- design et spécifications : difficile (lien fort avec l'implantation), mais il existe des *design patterns*
- code source : réponse la plus naturelle
- modules abstraits

Obstacles non-techniques à la réutilisabilité (économique, politique des entreprises etc.)

Réutilisabilité : un exemple jouet

On considère un ensemble d'éléments et on veut vérifier qu'un élément y est présent ou pas.

Algorithme 2.1 : `has(e,x)`

entrées : un ensemble e et un `ELEMENT` x
sortie : **`true`** si $x \in e$, **`false`** sinon

$pos \leftarrow$ `initial_position(`e`)` ;

tant que (`!` exhausted(pos, e**)) et (`!` found(**pos, x, e**)) faire**
  $pos \leftarrow$ `next(`pos, x, e`)` ;
fin

retourner `!` `exhausted(`pos, e`)` ;

5 exigences pour la réutilisabilité

- var. de type
- group. routines
- variation impl.
- indép. rep.
- factorisation

Variation de type

- on utilise dans has un type `ELEMENT`
- on devrait pouvoir écrire une seule « version » de has

5 exigences pour la réutilisabilité

- var. de type
- group. routines
- variation impl.
- indép. rep.
- factorisation

Regroupement de routines

- `has` dépend de la façon dont on a créé l'ensemble, comment on insère les éléments etc.
- `ins`, `del`, `create`, `has` $\in$ module

5 exigences pour la réutilisabilité

- var. de type
- group. routines
- variation impl.
- indép. rep.
- factorisation

Variation d'implantation

- ensemble : tableau, B-arbres etc.
- **famille** de modules

5 exigences pour la réutilisabilité

- var. de type
- group. routines
- variation impl.
- indép. rep.
- factorisation

Indépendance de la représentation

- point de vue **client**
- `has(e,x)` indépendant de la représentation de l'ensemble

5 exigences pour la réutilisabilité

- var. de type
- group. routines
- variation impl.
- indép. rep.
- factorisation

Factorisation de comportements communs

- point de vue **développeur**
- ensemble : séquentiel, arbres etc.
- trouver les comportements communs

Techniques pour la réutilisabilité

Généricité : paramétrer un type (ex : liste d'entiers, de réels etc)

➥ principe de dérivation générique

Surcharge **syntaxique** :

- on attache plus d'un sens à un même nom
- exemple trivial : l'addition
- exemple : `has(e,x)`, `has_binary(e,x)` $\Rightarrow$ il faut connaître les types !
- utilisation de la signature des opérations
- facilité pour le client
- un peu léger...

Techniques pour la réutilisabilité

Surcharge **sémantique**

- utilisation de la liaison dynamique

Cher ordinateur,

S'il vous plaît, regardez quel est le type de e. Ce doit être un ensemble, mais je ne sais absolument pas quel est le type réel que son créateur a choisi, et pour être honnête, je m'en fiche complètement. Après tout, mon boulot c'est de faire des maths financières/de l'aérodynamique, pas de la gestion d'ensembles. Donc débrouillez-vous tout seul pour trouver le type de e, et lorsque vous aurez la réponse, trouvez l'algorithme de has spécifique à ce type d'ensemble. Ensuite, appliquez le pour vérifier si x apparaît dans e et donnez moi le résultat. Je l'attends impatiemment pour mon calcul de produits dérivés/d'écoulement 3D.

J'ai le regret de vous informer que vous n'obtiendrez aucune autre information de ma part.

Sincèrement,

Votre dévoué développeur

Plan de la partie 1 - Introduction

Comment trouver les modules ?

Buts

Extensibilité, réutilisabilité et sûreté.

Un logiciel est décomposé en différents **modules**.

➥ quel critère pour trouver les modules ?

Actions

instructions, algorithmes

Objets

structures de données

Processeur(s)

Question

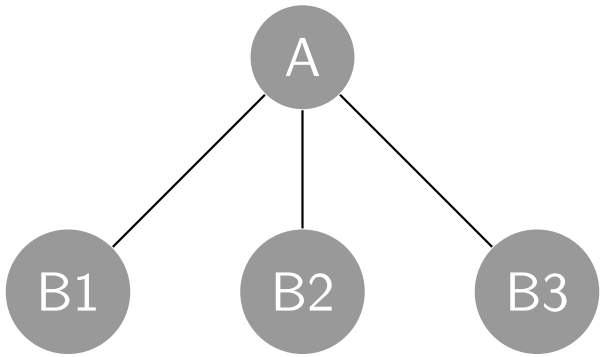
Construit-on les modules autour des actions ou des objets ?

Décomposition fonctionnelle

Point crucial

Continuité (systèmes réels, longue durée de vie)

Approche traditionnelle : décomposition fonctionnelle



- pas une seule fonction
- trouver le *top* $\Rightarrow$ très difficile sur un système réel
- met en avant les aspects extérieurs du logiciel
- l'ordonnancement des tâches peut poser problème
- réutilisabilité très mauvaise à cause du principe du *top-down*

Approche objet

Approche objet : beaucoup plus adaptée (cf. suite du cours)

Définition (développement orienté objet)

Un développement orienté objet d'un logiciel est une méthode qui fonde l'architecture d'un système informatique sur des **modules déduits des types des objets qu'il manipule** et non pas des fonctions que le système doit assurer.

Principe

Ne demandez pas ce que fait le système, mais ce **sur quoi il agit** et **pour qui il agit**.

Plan de la partie 1 - Introduction

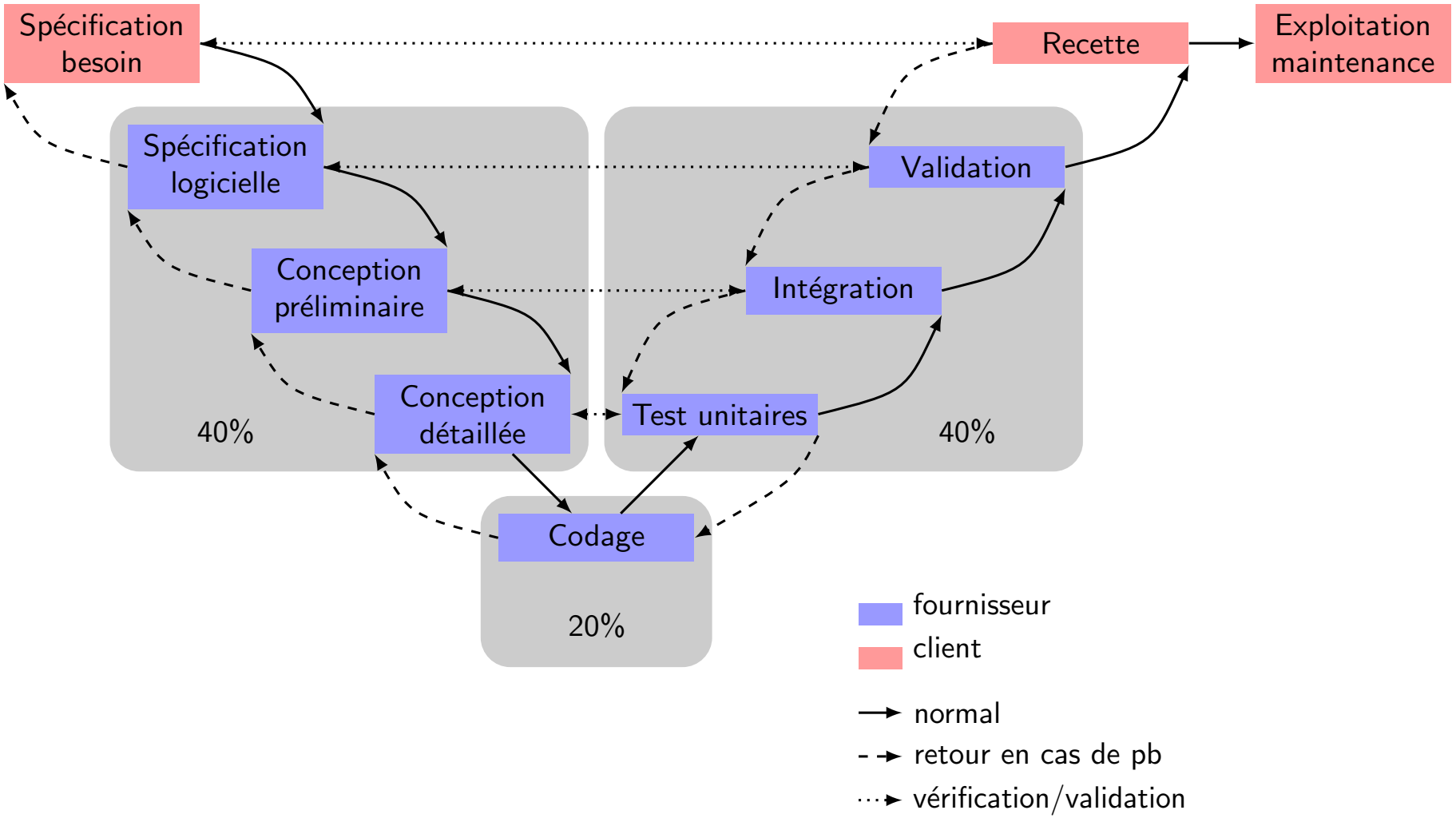
Critères demandés à une approche OO

Les **critères** demandés à une approche OO se répartissent sur trois domaines :

- méthode et langage (analyse, conception, programmation)
- implantation et environnement : les outils
  - mise à jour du système (ex : changement d'une fonction)
  - mise à jour indépendante de la taille globale du système
  - persistance (stockage)
  - documentation
- bibliothèques
  - bibliothèques de base
  - GUI
  - extensibles

Cycle de vie d'un logiciel

Consistance de l'approche objet : on couvre tout le cycle de vie



Concepts de base : représentation des objets

On centre l'approche sur les objets, i.e. les « données » que l'on manipule.

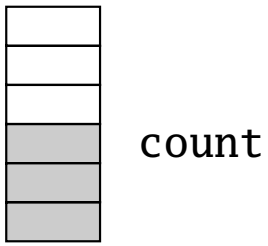
➥ comment les représenter ?

- utilisation de mathématiques élémentaires ;
- déjà aperçues (rapidement) en 1A dans le cours d'algorithmique et programmation ;
- pour décrire des objets, on a besoin d'une méthode qui respecte trois conditions :
  - description précise et sans ambiguïté
  - description complète
  - pas de **sur-spécification**
- le dernier point est le plus problématique...
  - 17% des coûts d'un logiciel provient du besoin de tenir compte des changements de format de données
  - il ne faut donc pas avoir une représentation trop proche du « physique » des structures de données

Besoin d'une représentation abstraite

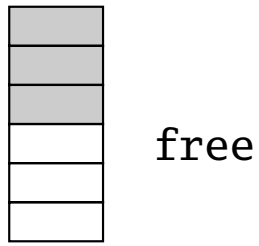
Pourquoi ce besoin de description « abstraite » ? Prenons l'exemple d'une pile (LIFO : *Last In, First Out*).

ARRAY_UP



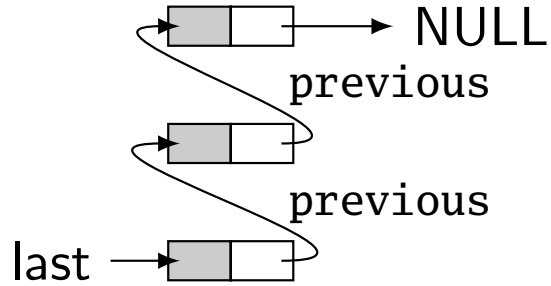
push(x)
```
count++
T[count]=x
```

ARRAY_DOWN



push(x)
```
T[free]=x
free--
```

LINKED



push(x)
```
new(n)
n.item=x
n.previous=last
last=n
```

Ces types sont des exemples de **sur-spécification** si on en choisit un comme représentation d'une pile.

Vers une vue abstraite des objets

Comment obtenir la complétude, la précision et la non ambiguïté en évitant la sur-spécification ?

Idée

Utiliser les opérations comme représentation de la pile.

On peut utiliser les opérations suivantes sur une pile :

- un **créateur** `make` qui construit une pile vide
- une **commande** `put` qui permet de mettre un élément dans la pile
- une **commande** `remove` qui retire un élément si la pile n'est pas vide
- une **requête** `item` qui renvoie l'élément du haut de la pile si elle n'est pas vide
- une **requête** `empty` qui permet de savoir si la pile est vide

Vers une vue abstraite des objets

Représentation traditionnelle (« fonctionnelle ») : on définit des données et ces opérations sont des routines utilisant des données.

On renverse le point de vue :

Définition (pile)

Une pile est une structure sur laquelle des clients peuvent appliquer les opérations listées ci-dessus.

Formaliser la spécification

Les descriptions informelles ne suffisent pas. Une spécification de TDA (Type de Données Abstrait) est définie par 4 éléments :

- types
  - un TDA comme `Stack` est un ensemble d'objets (toutes les piles en fait).
  - tout objet appartenant à l'ensemble d'objets décrit par une spécification de TDA est une **instance** de ce TDA.
  - pour la pile :

Types

$Stack[G]$

  - `G` est un paramètre **générique**

Formaliser la spécification : fonctions

- fonctions
  - on décrit les **services** possibles.
  - on définit les fonctions comme des fonctions mathématiques (domaine et codomaine).
  - on décrit les signatures des fonctions.
  - pas d'effet de bord !

Fonctions

$$
\begin{aligned}
put &: Stack[G] * G \rightarrow Stack[G] \\
remove &: Stack[G] \not\rightarrow Stack[G] \\
item &: Stack[G] \not\rightarrow G \\
empty &: Stack[G] \rightarrow BOOLEAN \\
new &: Stack[G]
\end{aligned}
$$

  - on retrouve les trois types de fonctions. . .

Formaliser la spécification : axiomes

- axiomes
  - les fonctions ne suffisent pas pour décrire une pile ;
  - on définit des axiomes pour le comportement de la pile ;
  - on ne cherche pas à calculer le « résultat » des fonctions ;

Axiomes

Pour toute pile s et tout élément x :

$$
\begin{aligned}
&item(put(s,x)) = x \\
&remove(put(s,x)) = s \\
&empty(new) \\
&not\ empty(put(s,x))
\end{aligned}
$$

Formaliser la spécification : préconditions

- préconditions
  - il faut préciser le domaine des fonctions **partielles**.
  - on utilise une clause **require** pour faire cela :

Préconditions

$item(s : Stack[G])$ `require not` $empty(s)$
$remove(s : Stack[G])$ `require not` $empty(s)$

Il s'agit de la fonction caractéristique du domaine de la fonction.

Et ça marche...on capture les propriétés essentielles :

$$item(remove(put(remove(put(put(new, x_1), x_2)), x_3))) = x_1$$

C'est un modèle purement **mathématique** d'un programme et de son exécution.

Des TDA aux classes

Notre objectif : informatique, pas des maths. . .

Définition (classe)

Une classe est un TDA avec une implantation qui peut être partielle.

On dispose de trois éléments :

- E1 : un TDA et sa spécification ;
- E2 : un choix de représentation ;
- E3 : un mapping des fonctions de E1 vers E2 sous la forme d'un ensemble d'**opérations**, chacune implantant une des fonctions en terme de représentation et satisfaisant les axiomes et les préconditions.

Définition (construction de logiciels OO)

La construction de logiciels OO est la construction de systèmes logiciels comme une collection structurée d'implantations (partielles) de TDA.

Mécanismes de base utilisés

Appel d'opérations : un seul mécanisme computationnel

- étant donné un objet **instance** d'une classe, on appelle une opération de cette classe sur cet objet ;
- une classe contenant un appel à une opération d'une classe `C` est dite **cliente** de `C` ;
- on parle aussi de **passage de messages** ;
- ex : appel d'une opération `crediter` avec un paramètre 5000 sur un objet `compteGarion` de la classe `CompteBancaire`.

Masquage d'information : public/protégé/privé

➥ pas de variable globale

Gestion des **exceptions**

Mécanismes de base utilisés

Typage **statique**

- chaque entité (nom utilisé par le logiciel référant un objet) est **déclarée comme étant d'un certain type** ;
- chaque appel d'opération sur une entité « est » une opération de la classe de l'objet ;
- l'assignation et le passage de paramètres sont assujettis à une règle de conformité.

Gestion mémoire

Concepts **avancés** : héritage, polymorphisme, généricité etc.

Plan de la partie 1 - Introduction

Plateforme Java

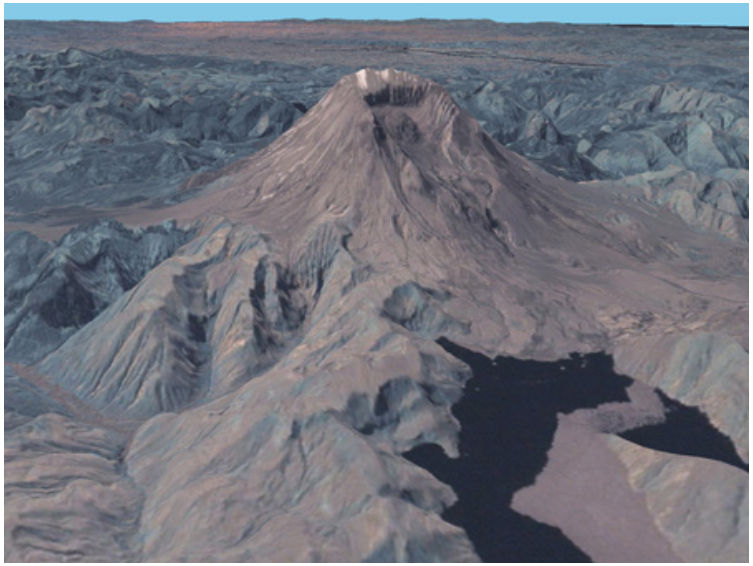
Une plateforme Java est constitué de trois éléments :

- **la machine virtuelle Java** ou JVM. Il s'agit en fait d'une spécification, implantée sur différentes plateformes ;
- **le langage Java** lui-même. Il a évolué principalement jusqu'à la version 1.2. Nous utiliserons la version 1.8 ;
- **la bibliothèque standard Java**. C'est elle qui évolue le plus. Pour améliorer la portabilité, elle est principalement développée en Java.

Le **JDK** (*Java Developer Kit*) fourni par Oracle fournit plusieurs outils parmi lesquels :

- `javac`, un compilateur de sources Java
- `java`, une machine virtuelle Java
- `javadoc`, un générateur de documentation HTML

Mais ça sert pour de vrai Java ?



World Wind

- développé par la NASA
- accès aux données de la NASA
- autres *providers*
- peut-être embarqué dans une autre application

Mais ça sert pour de vrai Java ?

Eglin Space Surveillance Radar

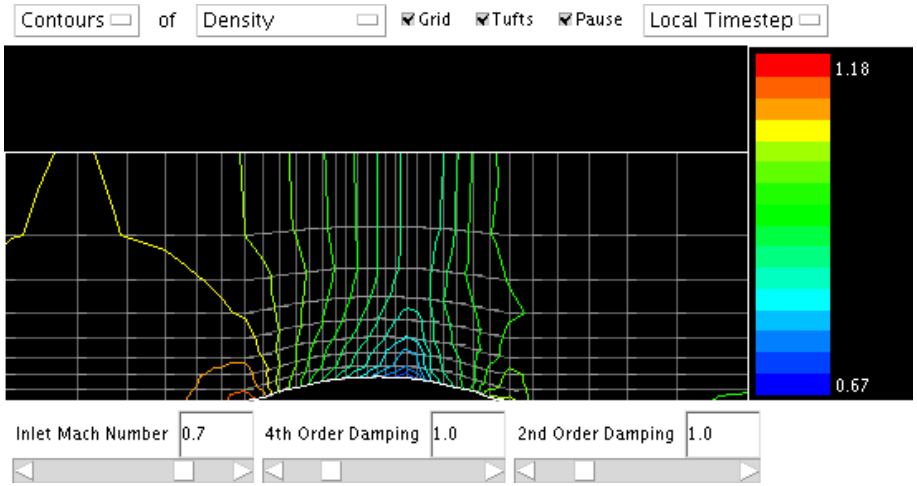
- détection et suivi d'objets dans l'espace
- utilise la technologie Java RealTime

Mais ça sert pour de vrai Java ?

Systèmes embarqués aéronautiques

- projets européens
- simulation des systèmes (Thalès Avionics)

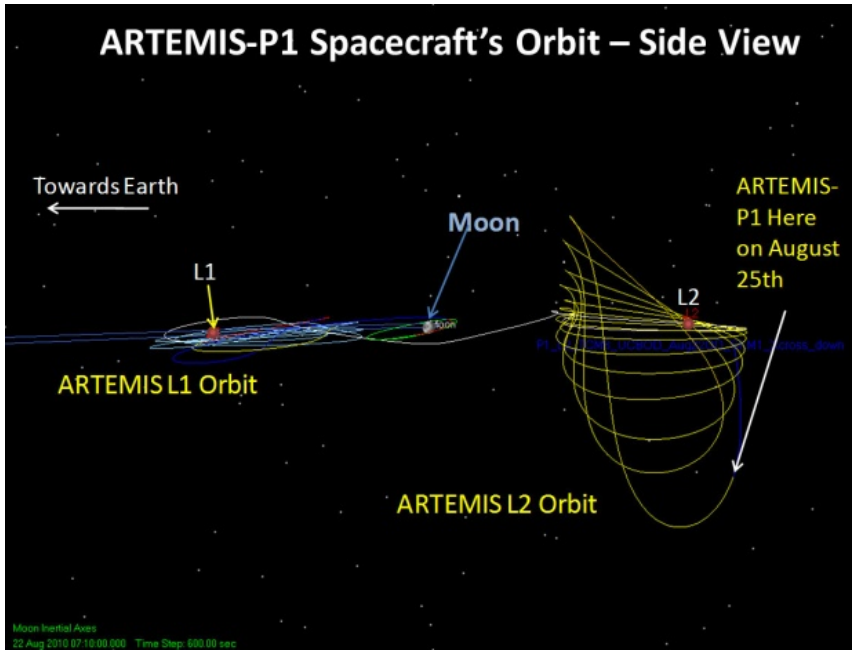
Mais ça sert pour de vrai Java ?



Java Virtual Wind Tunnel

- développé au MIT
- simulation de flux autour d'un objet 2D
- `http://raphael.mit.edu/Java/`

Mais ça sert pour de vrai Java ?



Orekit

- développé chez C-S
- projet libre et collaboratif
- choisi pour les *Flight Dynamic Systems* au CNES
- `http://www.orekit.org`

Caractéristiques de Java

Java est un langage :

- simple ;
- orienté objet ;
- **distribué** ;
- robuste ;
- **sécurisé** ;
- indépendant de l'architecture ;
- portable ;
- interprété ;
- performant ;
- **concurrent**.

Java est un langage simple

Semblable à C

- structures de contrôle ;
- types primitifs.

Simple par rapport à C++ (autre langage objet)

- pas de fichier entête ;
- pas d'arithmétique des pointeurs ;
- pas de structures ni d'unions ;
- pas de surcharge des opérateurs ;
- pas de conversion de types sans contrôle ;
- gestion de la mémoire.

Java est un langage robuste

But

Éliminer les risques d'erreurs

Moyens

- mécanisme de gestion des erreurs
  - typage statique
  - l'éditeur de lien utilise les informations de typage
  - conversion de types contrôlée
  - détection dynamique des dépassements de tableaux
- mécanisme de gestion mémoire
  - contrôle d'accès à la mémoire
  - ramasse-miettes

Java est indépendant de la plateforme

But

Exécuter le programme sur des machines différentes

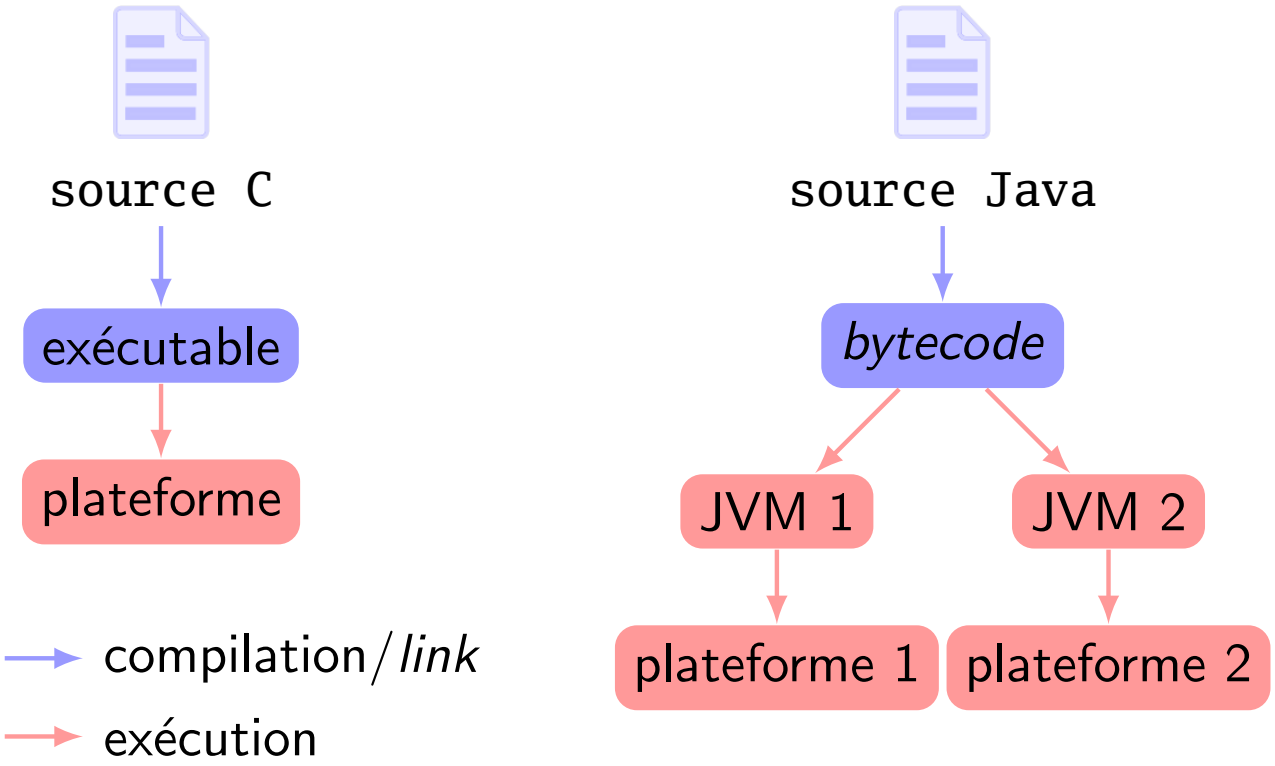
Moyens

- utilisation d'une machine virtuelle
- définition d'une bibliothèque standard instanciée pour chaque environnement

Java est indépendant de la plateforme

But

Exécuter le programme sur des machines différentes



Java est un langage portable

But

- un même code compilé sur toutes les architectures produit le même résultat
- minimiser les modifications liées au portage de la JVM

Moyens

- bibliothèque indépendante
- sémantique précise du langage
  - taille et organisation physique des données
  - valeurs par défaut, minimales et maximales
  - effets des opérateurs sur les données
  - ordre des calculs
  - effet des instructions sur la mémoire

Java est un langage dynamique

But

Accélérer le cycle de développement

- éviter les recompilations inutiles
- réduire les éditions de liens au strict nécessaire

Moyens

- édition de liens dynamique
- accès à la représentation interne des classes (réflexivité)
- chargement des classes à la demande

Java est un langage performant

Moyens

- *bytecode* adapté pour être compilé à la volée (compilateur **Just In Time**) ;
- cache mémoire pour éviter le chargement multiple d'une même classe ;
- compilation en natif disponible ;
- ramasse-miettes.

Les performances d'un langage ne se mesurent pas qu'à sa vitesse d'exécution, mais également à la **facilité de développement**. Les avantages de Java sont :

- langage simple
- vérification statique et dynamique fortes
- bibliothèque standard

Plan de la partie 1 - Introduction

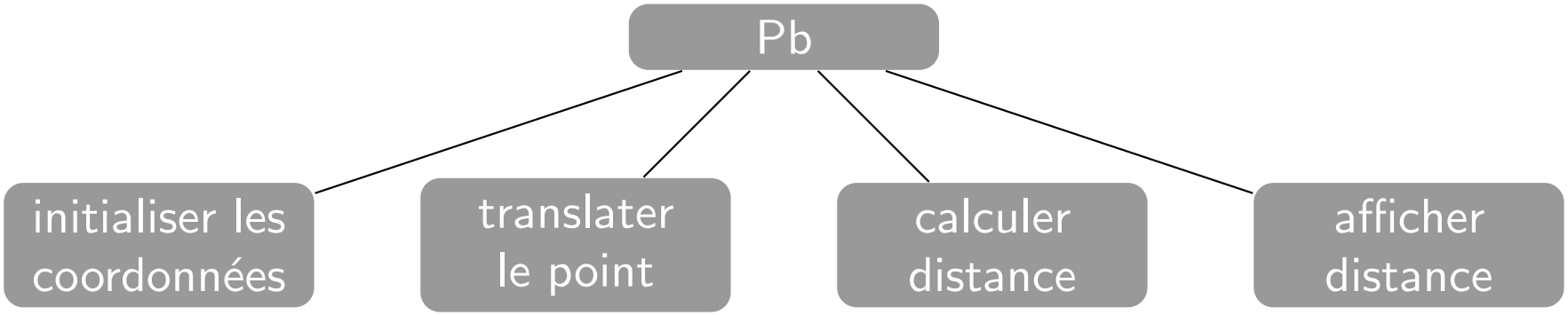
Exemple introductif

On cherche à représenter un point dans le plan avec ses coordonnées cartésiennes, à le translater et à afficher sa distance à l'origine.

On va résoudre ce problème :

- avec une approche traditionnelle (décomposition fonctionnelle) ;
- avec une approche objet.

Exemple introductif : approche top-down



Point : version top-down

PointTraditionnel.java

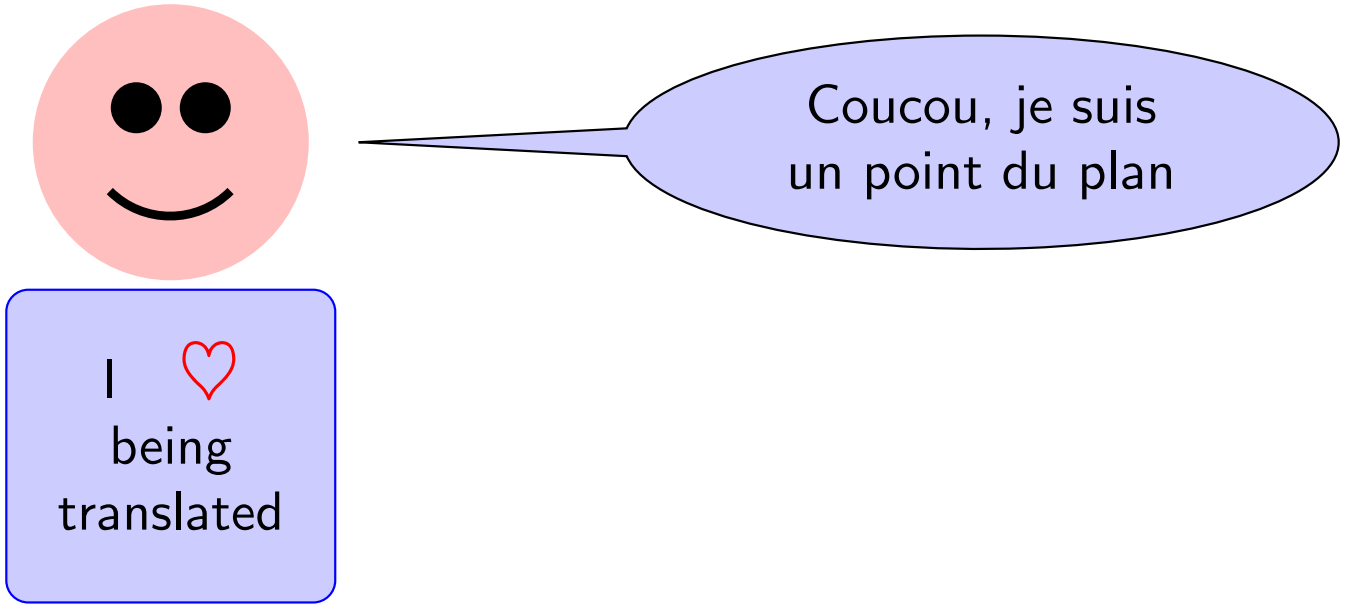
```
class PointTraditionnel {
    public static final void main(final String[] args) {
        double x, y;          // les coordonnees du point

        // initialiser les coordonnees
        x = 2.0;
        y = 3.0;

        // translater le point
        x += 5.0;
        y += -2.0;

        // calculer la distance a l'origine
        double distance = Math.sqrt(x * x + y * y);

        // afficher la distance
        System.out.println("Distance a l'origine : " + distance);
    }
}
```

Approche objet

Principe

Système $\equiv$ objets manipulés.

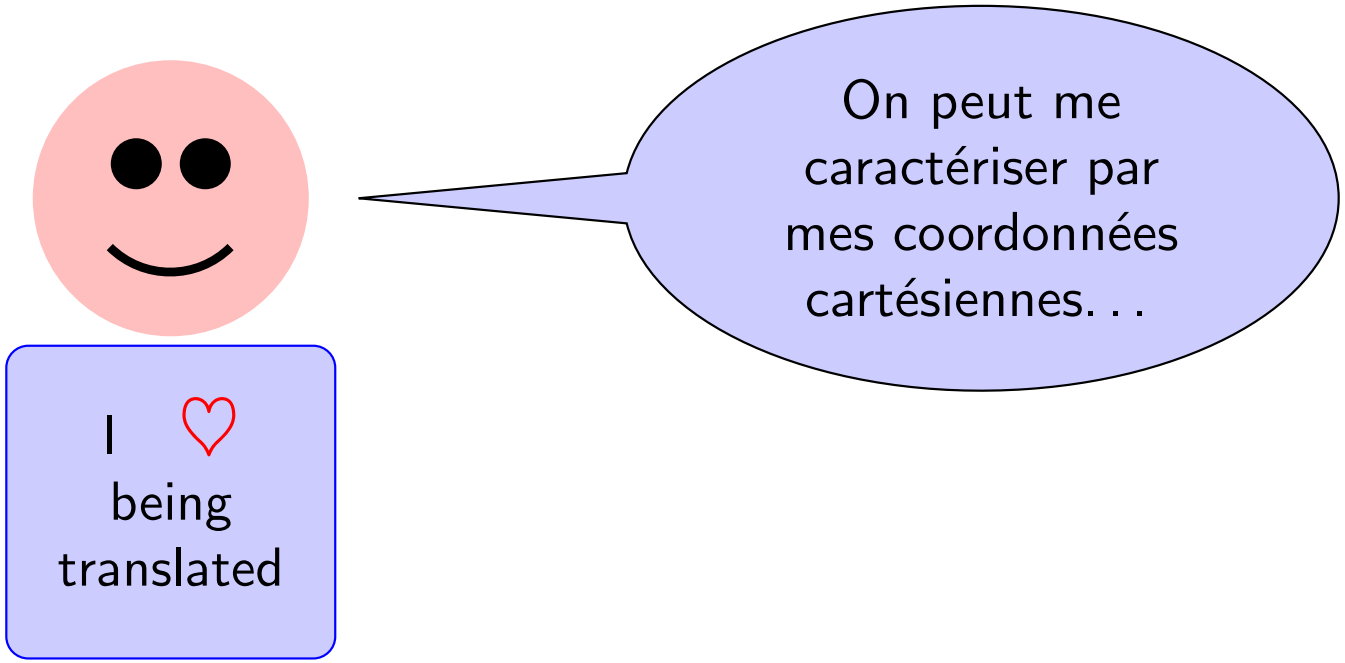
Ici, les objets manipulés sont des **points dans le plan**.



Principe

Système ≡ objets manipulés.

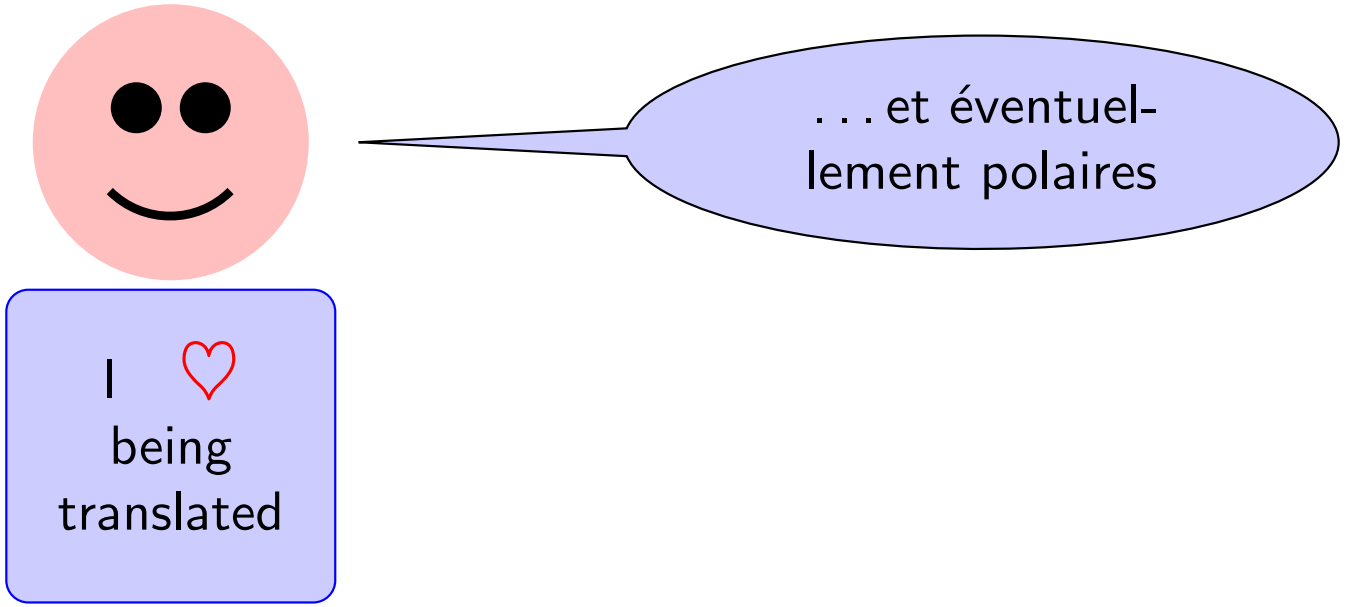
Ici, les objets manipulés sont des **points dans le plan**.



Principe

Système ≡ objets manipulés.

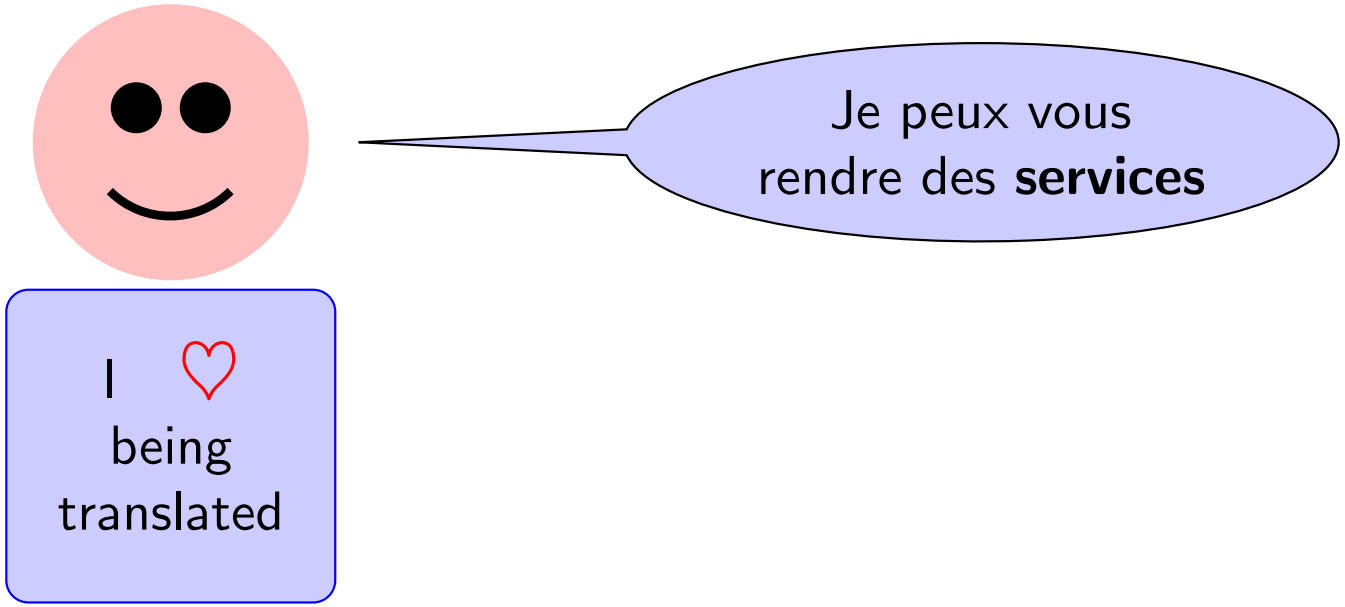
Ici, les objets manipulés sont des **points dans le plan**.



Principe

Système ≡ objets manipulés.

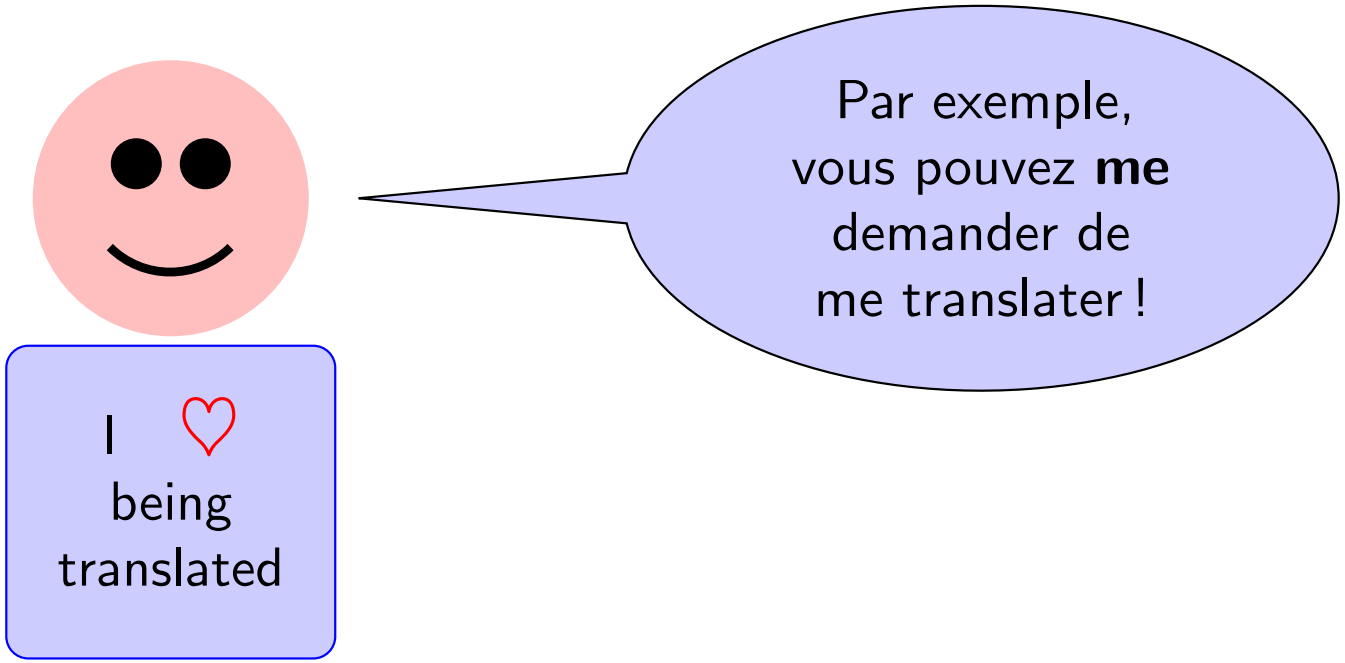
Ici, les objets manipulés sont des **points dans le plan**.



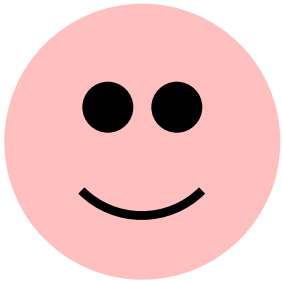
Principe

Système ≡ objets manipulés.

Ici, les objets manipulés sont des **points dans le plan**.



Approche objet : résumé



I ♡ being translated

État (attributs)

- abscisse
- ordonnée

Opérations ou méthodes

- `translater`
- `distance` à un autre point

Approche objet : classification

Il existe donc des objets qui représentent des points du plan, qui peuvent tous rendre les services `translater` et `distance` et qui se distinguent les uns des autres par leur état (coordonnées).

On va utiliser une procédure de classification pour « regrouper » les caractéristiques de ces objets dans une **classe**.

Définition (instance)

On dira qu'un objet est **instance** d'une classe si et seulement si il possède les attributs et méthodes fournies par la classe.

Équations : version (plus) objet

Point.java

```
/**
 * <code>Point</code> modelise un point du plan.
 *
 * @author <a href="mailto:garion@isae.fr">Christophe Garion</a>
 */
class Point {

    /** Abscisse du point */
    double x;

    /** Ordonnee du point **/
    double y;

    /** Translater le point */
    void translater(double dx, double dy) {
        this.x += dx;
        this.y += dy;
    }
```

Équations : version (plus) objet

Point.java

```
    /** Calculer la distance a un autre point */
    double distance(Point p) {
        double dx2 = Math.pow(this.x - p.x, 2);
        double dy2 = Math.pow(this.y - p.y, 2);

        return Math.sqrt(dx2 + dy2);
    }
}
```

Programme principal utilisant le point

TestPoint.java

```
class TestPoint {
    public static void main(String[] args) {
        Point p;            // une poignee sur un Point
        p = new Point();    // creation d'un objet Point

        // initialiser les coordonnees
        p.x = 2.0;
        p.y = 3.0;

        // translater le point
        p.translater(5.0, -2.0);

        // calculer la distance a l'origine
        double distance = p.distance(new Point());

        // afficher la distance
        System.out.println("Distance a l'origine : " + distance);
    }
}
```

Constatations

- ressemblance syntaxique avec le langage C ;
- une classe ressemble à un enregistrement dans lequel on peut mettre à la fois des champs (les attributs) et des fonctions (les méthodes) ;
- les objets sont toujours **créés dynamiquement** par l'opérateur **`new`** ;
- les objets ne sont pas accessibles directement : on utilise une **poignée** (variable d'objet) pour y accéder ;
- le type de la poignée conditionne le type des objets qui peuvent lui être attachés ;
- pas de `free`/`delete` (ramasse-miettes) ;
- on utilise la notation pointée pour accéder à un attribut ou une méthode d'un objet ;
- **important :** on distingue une classe pour **décrire** un objet et une classe **application** qui **utilise** cet objet et les services qu'il rend.

Remarque : on peut le faire en C ?

On aurait pu écrire une **structure** en C...

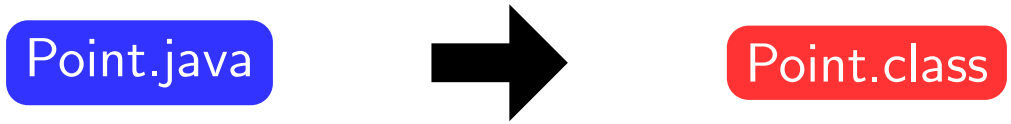
point.c

```c
typedef struct _point {
    double x;
    double y;
} point;

void translater(point p, double dx, double dy) {
    p.x += dx;
    p.y += dy;
}
```

...mais nous verrons dans la suite du cours qu'il est très difficile de mettre en œuvre certaines propriétés intéressantes en C : encapsulation, gestion mémoire, héritage, généricité...

Compilation

On utilise le compilateur `javac` pour produire du bytecode.



Java est un langage compilé. Les messages d'erreur émis par le compilateur vous aideront.

ex : erreur dans `TestPoint.java`

```
        System.out.println("Distance a l'origine : " + distance)
```

à la compilation :

```
[tof@suntof]~ $ javac TestPointErreur.java
TestPointErreur.java:17: error: ';' expected
        System.out.println("Distance a l'origine : " + distance)
                                                               ^
1 error
```

Dépendances

Java calcule les dépendances. Il suffit donc de compiler la classe « principale ».

```
[tof@suntof]~ $ javac -verbose TestPoint.java
[parsing started RegularFileObject[TestPoint.java]]
[parsing completed 17ms]
[loading ZipFileIndexFileObject[/usr/lib/jvm/jdk-7-oracle-i586/lib/ct.
[loading ZipFileIndexFileObject[/usr/lib/jvm/jdk-7-oracle-i586/lib/ct.
[checking TestPoint]
[loading ZipFileIndexFileObject[/usr/lib/jvm/jdk-7-oracle-i586/lib/ct.
[loading RegularFileObject[./Point.class]]
[loading ZipFileIndexFileObject[/usr/lib/jvm/jdk-7-oracle-i586/lib/ct.
[loading ZipFileIndexFileObject[/usr/lib/jvm/jdk-7-oracle-i586/lib/ct.
[loading ZipFileIndexFileObject[/usr/lib/jvm/jdk-7-oracle-i586/lib/ct.
[loading ZipFileIndexFileObject[/usr/lib/jvm/jdk-7-oracle-i586/lib/ct.
[loading ZipFileIndexFileObject[/usr/lib/jvm/jdk-7-oracle-i586/lib/ct.
[loading ZipFileIndexFileObject[/usr/lib/jvm/jdk-7-oracle-i586/lib/ct.
[loading ZipFileIndexFileObject[/usr/lib/jvm/jdk-7-oracle-i586/lib/ct.
[loading ZipFileIndexFileObject[/usr/lib/jvm/jdk-7-oracle-i586/lib/ct.
[loading ZipFileIndexFileObject[/usr/lib/jvm/jdk-7-oracle-i586/lib/ct.
[loading ZipFileIndexFileObject[/usr/lib/jvm/jdk-7-oracle-i586/lib/ct.
[wrote RegularFileObject[TestPoint.class]]
[total 169ms]
```

Utilisation de la machine virtuelle

On produit du bytecode après compilation. Il faut **interpréter** ce bytecode à travers une machine virtuelle répondant aux spécifications de la JVM. Le JDK fournit une machine virtuelle : `java`.

On fournit en paramètre de `java` le **nom** d'une classe Java qui doit contenir la méthode `main`.

La méthode `main` est la méthode qui va être exécutée. Elle se déclare **obligatoirement** de la manière suivante :

Syntaxe (méthode `main`)

```
public static void main (String[] args) {
    ...
}
```

Tous les mots (sauf `args`) sont importants !

Utilisation de la machine virtuelle

```
[tof@suntof]~ $ java TestPoint.java
Error: Could not find or load main class TestPoint.java
```

```
[tof@suntof]~ $ java TestPoint.class
Error: Could not find or load main class TestPoint.class
```

```
[tof@suntof]~ $ java TestPoint
Distance a l'origine : 7.0710678118654755
```

```
[tof@suntof]~ $ java Point
Error:
 Main method not found in class Point, please define the main method as:
   public static void main(String[] args)
```

Documentation automatique avec `javadoc`

`javadoc` est un utilitaire permettant de générer automatiquement la documentation des classes Java à partir de leur code source.

```
javadoc -author -version *.java
```

Le format de sortie est un ensemble de fichiers HTML.

Intérêt : la documentation est directement rédigée dans le source avec le code.

Principe

Le concepteur d'une classe doit documenter sa classe et s'astreindre à exprimer toute l'information sur la classe dans la classe elle-même.

Attention, par défaut, seules les informations publiques sont documentées (on peut également utiliser l'option –**`private`**).

Différents types de commentaires

Les commentaires en Java sont de trois types :

- commentaires à la C

```
/* Un commentaire classique de C
   qui est sur plusieurs lignes */

/* Ces commentaires ne peuvent
 * pas s'imbriquer
 */
```

- commentaires à la C++

```
// Ce commentaire se termine a la fin de la ligne
```

- ces commentaires servent à documenter le **code**. Ils ne peuvent donc pas servir de documentation pour la classe.

Syntaxe des commentaires Javadoc

Les commentaires **structurés** exploités par `javadoc` peuvent contenir :

- des étiquettes spécifiques à `javadoc` commençant par `@` :

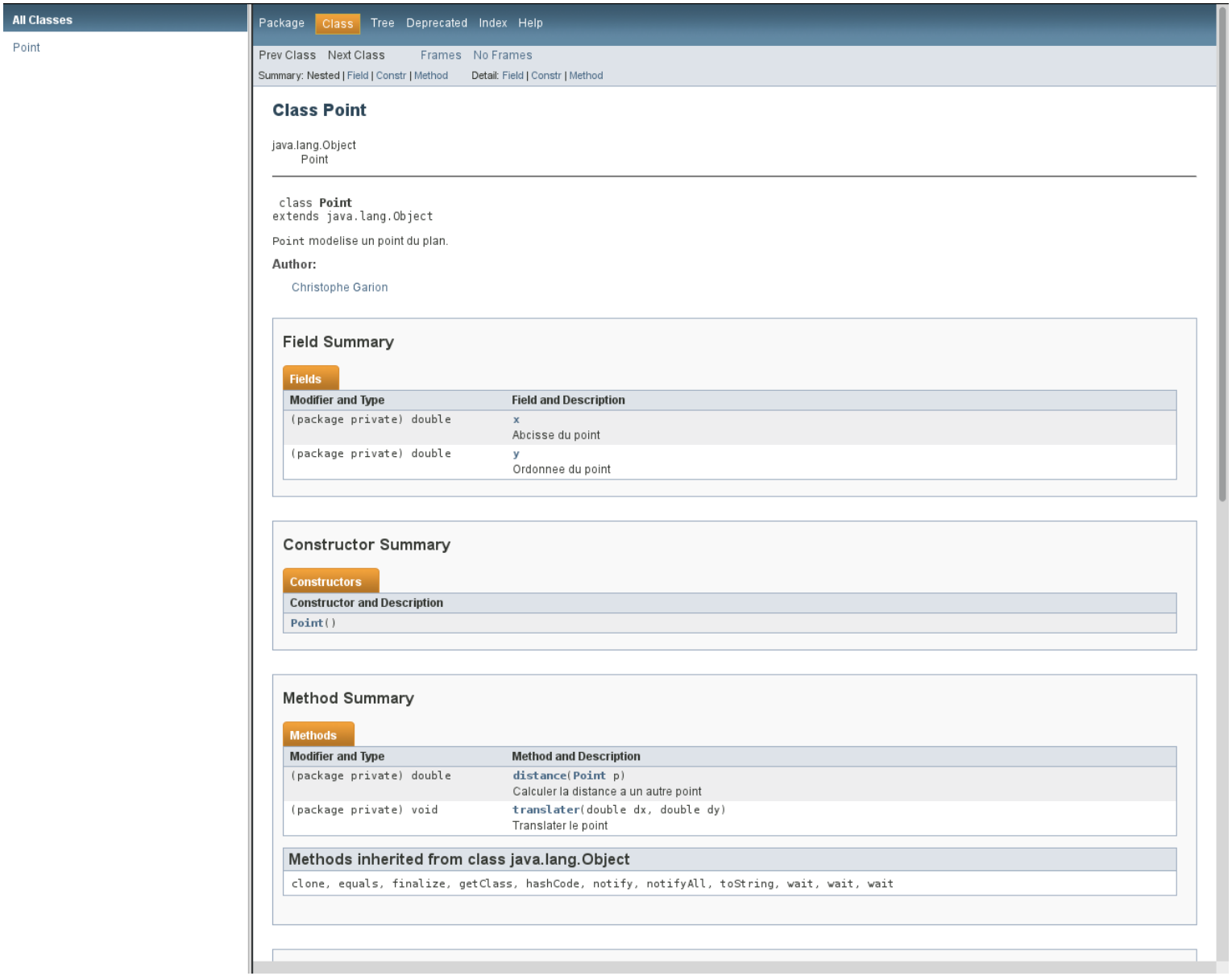
`@version`	version de la classe
`@author`	auteur(s) de la classe
`@param`	description d'un paramètre de méthode
`@return`	description du retour d'une méthode
`@exception`	description d'une exception
`@see`	renvoi vers un autre élément

- des éléments HTML

```
/** Les commentaires structures commencent par **
 *  Ils peuvent contenir des balises <strong>HTML</strong>
 */
```

Ces commentaires sont placés **avant l'entité qu'ils décrivent** : classe, attribut ou méthode.

Javadoc : exemple



All Classes

Point

Package Class Tree Deprecated Index Help

Prev Class Next Class Frames No Frames

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

Class Point

java.lang.Object
    Point

class **Point**
extends java.lang.Object

Point modelise un point du plan.

Author:

Christophe Garion

Field Summary

Fields

Modifier and Type	Field and Description
(package private) double	x Abcisse du point
(package private) double	y Ordonnee du point

Constructor Summary

Constructors

Constructor and Description
Point()

Method Summary

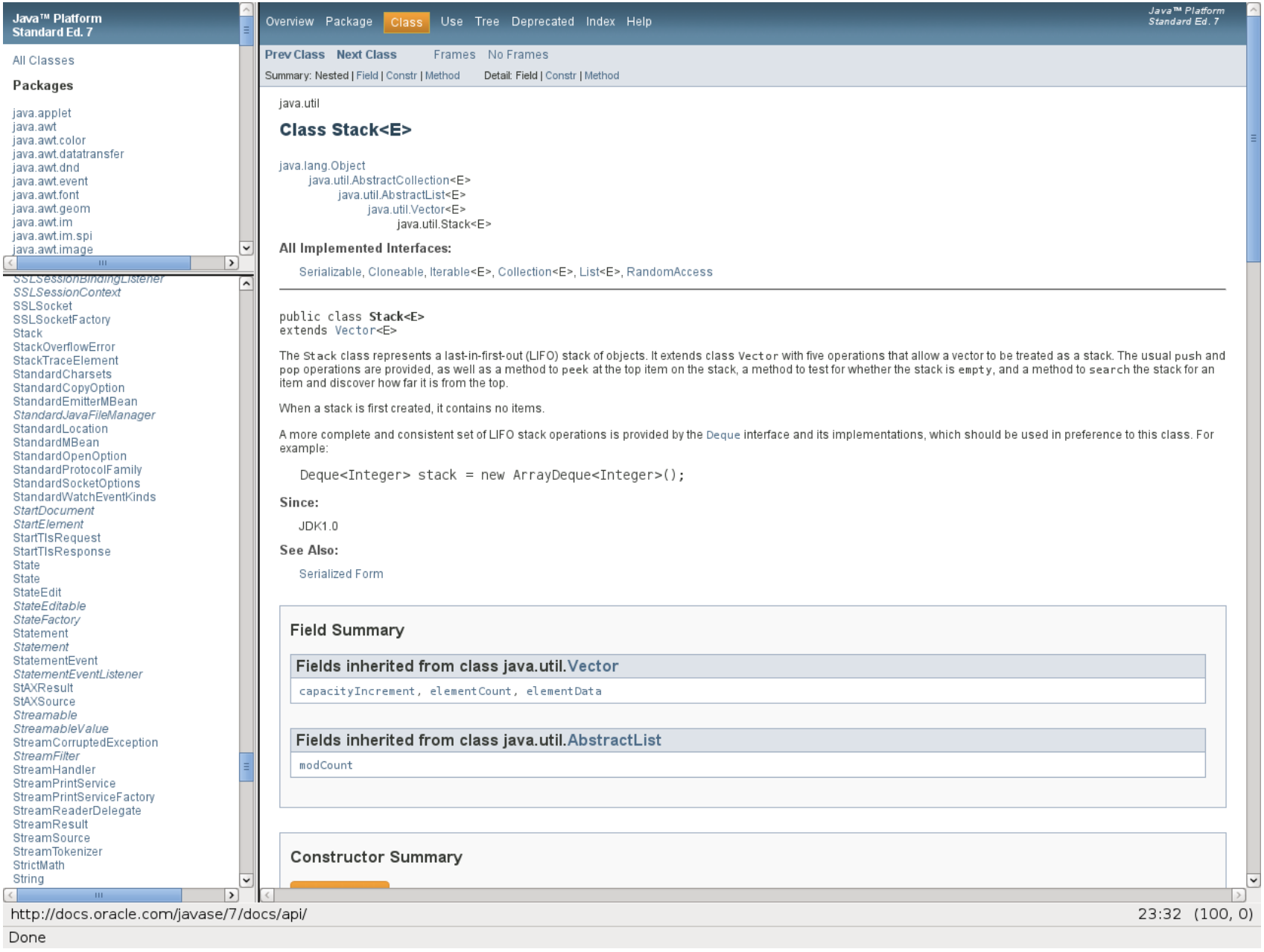
Methods

Modifier and Type	Method and Description
(package private) double	distance(Point p) Calculer la distance a un autre point
(package private) void	translater(double dx, double dy) Translater le point

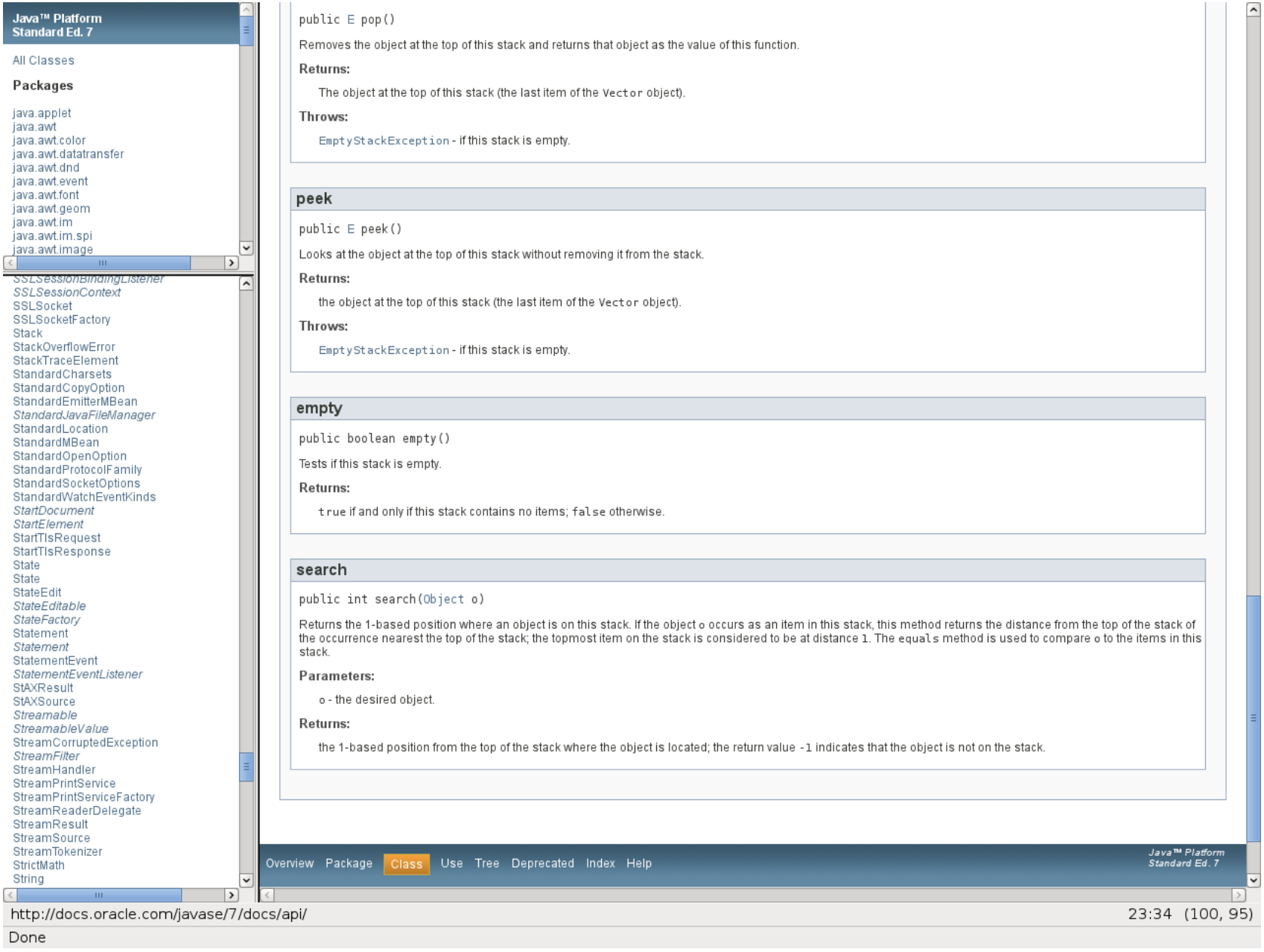
Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Javadoc : exemple de l'API



Javadoc : exemple de l'API



Syntaxe de Java

- types primitifs, opérateurs, structures de contrôle : identique à C (cf. *refcard* fournie) ;
- tous les passages et les retours de paramètres se font par **valeur** ;
- initialisation des variables locales ;
- conventions d'écriture (voir document disponible sur le site) :
  - le nom du fichier source contenant la classe `Point` doit **obligatoirement** être `Point.java` ;
  - les noms de classe commencent par une majuscule ;
  - les noms d'attributs, de méthodes et de variables commencent par une minuscule ;
  - les noms doivent être aussi explicites que possibles (pas de classe `C` ou d'attribut `a`) ;
  - on n'utilise pas de séparateur `_` : `TestPoint`, `calculeDiscriminant` etc.

Plan de la partie 1 - Introduction

Organisation

Objectifs

- étude du développement de logiciels complexes
- approches de conception OO avec UML
- programmation OO avec Java

Documentation

- notes de cours
- un site Web mis-à-jour toutes les semaines (énoncés et corrigés des TPs...) : `http://www.tofgarion.net/lectures/IN201`
- bibliothèque et sites externes

Évaluation

Examen (50% de la note finale)

- balaie l'ensemble des connaissances du cours
- si on n'a pas travaillé, on ne le réussit pas...
- depuis deux ans, plusieurs petits exercices au lieu de 3 problèmes
- toutes les annales et corrigés sont disponibles sur `http://www.tofgarion.net/lectures/IN201`

Projet (50% de la note finale)

- projet en équipe de 6
- pour l'instant, les équipes sont tirées au hasard dans les PC
- objectif : pas simplement de la programmation !

Séances (2h30 chacune)

0	C	introduction (vous y êtes ☺)
1	C/TD	classes et objets
2	TP	classes et objets
3	C/TD	tests unitaires et associations
4	TP	tests unitaires et associations
5	TP	TP récapitulatif
6	C/TP	interfaces
7	C/TD	héritage et classes abstraites
8	TP	héritage et classes abstraites
9	BE	conception projet
10	C/TP	*pattern* observateur
11	C/TP	IHM
12	BE	implantation projet
13	C/TP	exceptions
14	C/TP	généricité

Attention, certaines séances se déroulent la **même semaine** !

Contacts et aspects pratiques

Contacts

- professeur responsable : bureau 05.072, tél 8057
- votre PC(wo)man
- réunions avec les délégués de PC

Pratique

- choix des groupes définitifs ;
- conserver son binôme ;
- attention aux absences (2 points retirés sur la note finale) ;
- en cas d'absence prévue, prévenir M. Marlot, votre PCMan et moi-même par mail ;
- « évaluation » des TPs à chaque séance.

Bibliographie succincte


Meyer, B. (1997).
Object-Oriented Software Construction.
2e éd.
Prentice Hall.


Arnold, K., J. Gosling et D. Holmes (2005).
The Java Programming Language.
4e éd.
Java Series.
Addison-Wesley.


Fowler, M. (2003).
UML distilled.
Addison-Wesley.


Booch, G., J. Rumbaugh et I. Jacobson (2004).
The Unified Modeling Language reference manual.
Addison-Wesley.

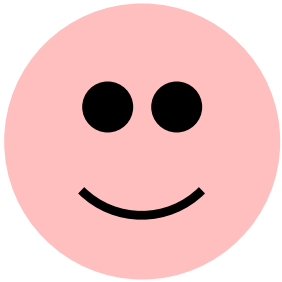
2 - Classes et objets

Plan de la partie 2 - Classes et objets

Les objets

Définition (objet)

Un objet est caractérisé par un **état** (la valeur de ses attributs), un **comportement** (les méthodes qui peuvent lui être appliquées), une **identité** qui l'identifie de manière unique (par exemple son adresse en mémoire).



I ♡ being translated

État (attributs)

- abscisse
- ordonnée

Opérations ou méthodes

- `translater` le point
- `afficher` le point
- calculer la `distance` avec un autre point
- ...

Les objets

Définition (objet)

Un objet est caractérisé par un **état** (la valeur de ses attributs), un **comportement** (les méthodes qui peuvent lui être appliquées), une **identité** qui l'identifie de manière unique (par exemple son adresse en mémoire).

Où sont définies les caractéristiques d'un objet ?

Les **caractéristiques** d'un objet (ses **attributs** et ses **méthodes**) sont définies dans une **classe**.

La classe vue comme module

Une classe définit un **module** : elle regroupe la déclaration des attributs et la définition des méthodes associées dans une même construction syntaxique.

Syntaxe (déclaration d'une classe)

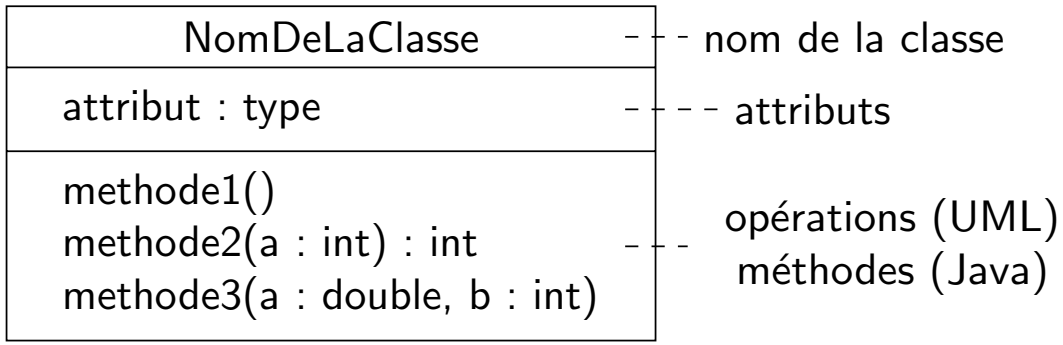
```
class NomDeLaClasse {
  // definition des caracteristiques de la classe
}
```

- les **attributs** permettent le stockage d'informations (état de l'objet)
- les **méthodes** sont des unités de calcul (fonctions ou procédures)

La classe est donc l'unité d'**encapsulation** et un **espace de nommage**.

Représentation UML d'une classe

On a besoin d'un langage pour modéliser la notion de classe : utilisation de UML (**Unified Modeling Language**).



Point
x : double y : double
translater(dx: double, dy: double) distance(p: Point): double afficher()

Trois parties :

- le nom de la classe ;
- les attributs de la classe (facultatif) ;
- les opérations de la classe (facultatif).

Attributs : déclaration

Les attributs permettent de stocker l'état de l'objet.
Ils se déclarent nécessairement **à l'intérieur d'une classe**.

Syntaxe (déclaration d'un attribut)

```
/** Documentation javadoc de l'attribut */
Type idAttribut;
```

Point.java

```
class Point {

  /** Abscisse du point */
  double x;

  ...
}
```

Méthodes : déclaration

Une méthode est une unité de calcul (fonction au sens de C) qui exploite l'état d'un objet (en accès et/ou en modification).

Une méthode est identifiée par sa classe, son nom, le nombre et le type de ses paramètres : c'est sa **signature**.
Elle possède un type de retour qui peut être **void** si elle ne retourne rien.
Elle a un code entre accolades.

Syntaxe (déclaration d'une méthode)

```
/** Documentation javadoc de la methode decrivant son objectif
 *  On documente aussi les parametres et le type de retour :
 *  @param nomParametre description de nomParametre
 *  @return description de l'information retournee
 */
TypeRetour idMethode(Type1 p1,..., TypeN pN) {
  ...
}
```

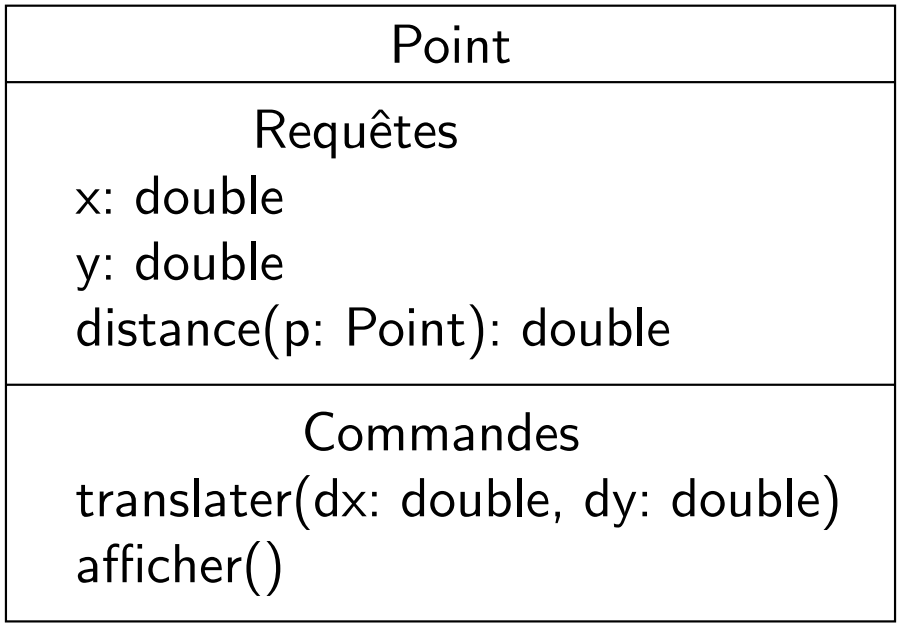
Deux types de méthodes

Traditionnellement, on peut classer les méthodes en deux catégories :

- les **commandes** qui permettent de modifier l'état d'un objet (i.e. de modifier la valeur de ses attributs)
  - elles sont **avec effet de bord** ;
  - elles ont souvent un type de retour **`void`**.
- les **requêtes** qui permettent de demander une valeur à un objet (valeur d'un attribut, résultat d'un calcul complexe que l'objet peut effectuer,...)
  - elles sont toujours **sans effet de bord** ;
  - elles ont un type de retour qui n'est pas **`void`**.

Cette classification donne une vue **utilisateur** de la classe.

Requêtes et commande : exemple avec `Point`



Le programmeur peut ensuite choisir de créer une classe où des **choix de représentation** sont faits : stockage des coordonnées dans deux attributs, stockage des coordonnées dans un tableau de deux réels etc.

Méthodes : exemple de commande

Définition d'une méthode `translater` qui translate un point à partir des coordonnées du vecteur de translation :

Point.java

```
/**
 * Translater le point.
 *
 * @param dx l'abscisse du vecteur de translation
 * @param dy l'ordonnee du vecteur de translation
 */
void translater(double dx, double dy) {
    this.x += dx;
    this.y += dy;
}
```

this est le paramètre implicite (cf. plus loin).

Méthodes : exemple de requête

Définition d'une méthode `distance` qui calcule la distance à un autre point :

Point.java

```
/**
 * Calculer la distance a un autre point.
 *
 * @param p le point servant a calculer la distance
 * @return la distance entre <code>this</code> et p. Elle
 *         est positive.
 */
double distance(Point p) {
    double dx2 = Math.pow(this.x - p.x, 2);
    double dy2 = Math.pow(this.y - p.y, 2);

    return Math.sqrt(dx2 + dy2);
}
```

La classe Point

Point.java

```java
/**
 * <code>Point</code> modelise un point du plan.
 *
 * @author <a href="mailto:garion@isae.fr">Christophe Garion</a>
 */
class Point {

    /** Abscisse du point */
    double x;

    /** Ordonnee du point **/
    double y;

    /**
     * Translater le point.
     *
     * @param dx l'abscisse du vecteur de translation
     * @param dy l'ordonnee du vecteur de translation
     */
    void translater(double dx, double dy) {
        this.x += dx;
        this.y += dy;
    }
```

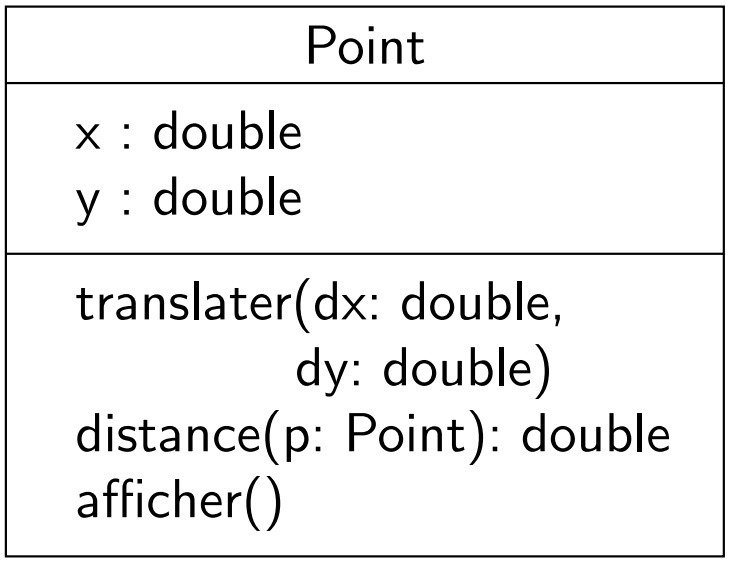
La classe Point

Point.java

```
    /**
     * Calculer la distance a un autre point.
     *
     * @param p le point servant a calculer la distance
     * @return la distance entre <code>this</code> et p. Elle
     *         est positive.
     */
    double distance(Point p) {
        double dx2 = Math.pow(this.x - p.x, 2);
        double dy2 = Math.pow(this.y - p.y, 2);

        return Math.sqrt(dx2 + dy2);
    }

    /**
     * Afficher le point sous la forme (x, y).
     */
    void afficher() {
        System.out.println("(" + this.x + "," + this.y + ")");
    }
}
```

Remarque importante

Traduction quasi-directe d'un diagramme UML d'implantation vers du code Java :

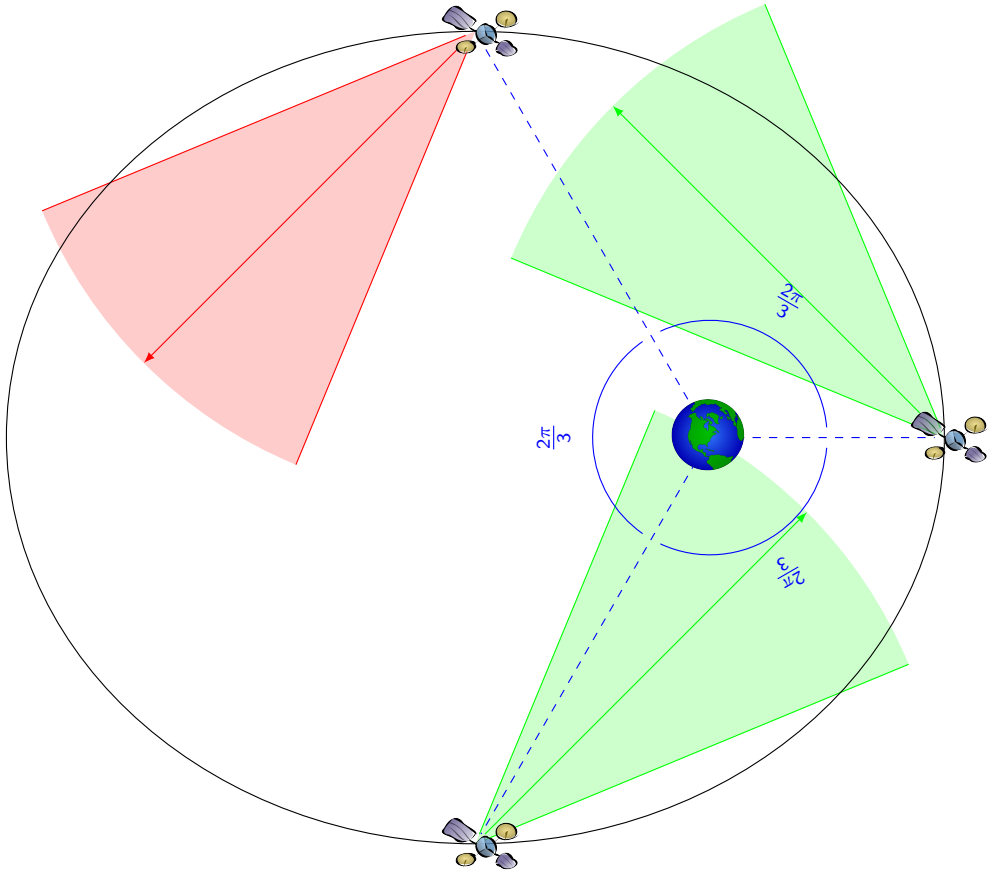



```
class Point {

  double x;
  double y;

  void translater(double dx,
                  double dy) { ... }

  double distance(Point p) { ... }

  void afficher() { ... }
}
```

Il ne reste plus qu'à implanter le corps des méthodes, **après** avoir écrit la documentation `javadoc` bien sûr !

Lost in space...

On cherche à modéliser une orbite elliptique plane autour de la Terre sur laquelle évolue trois satellites. Chaque satellite possède un instrument orienté à $\frac{\pi}{4}$ vers la Terre par rapport à la tangente à sa trajectoire et d'ouverture $\frac{\pi}{4}$ lui permettant de communiquer avec le satellite le précédant.



Lost in space...

Exercice

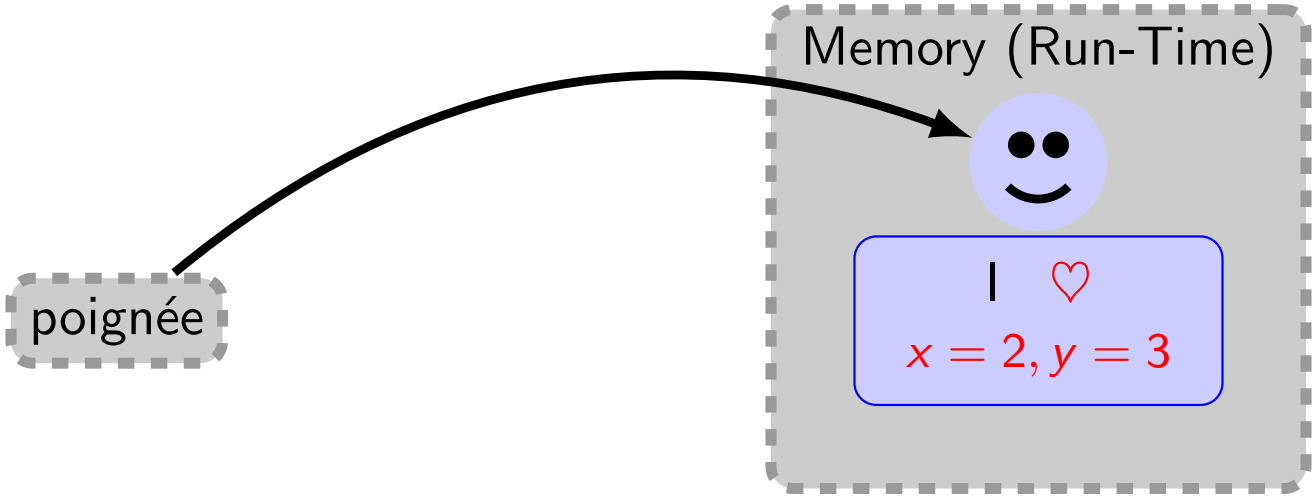
Modéliser l'orbite présentée précédemment par une classe `Orbite`.
On souhaite pouvoir :

- calculer la position (sous forme de coordonnées cartésiennes) d'un satellite en donnant l'angle au centre ou au foyer permettant de le répérer
- vérifier si un satellite voit le satellite précédent

On pourra dans un premier temps recenser les requêtes et commandes nécessaires.

Plan de la partie 2 - Classes et objets

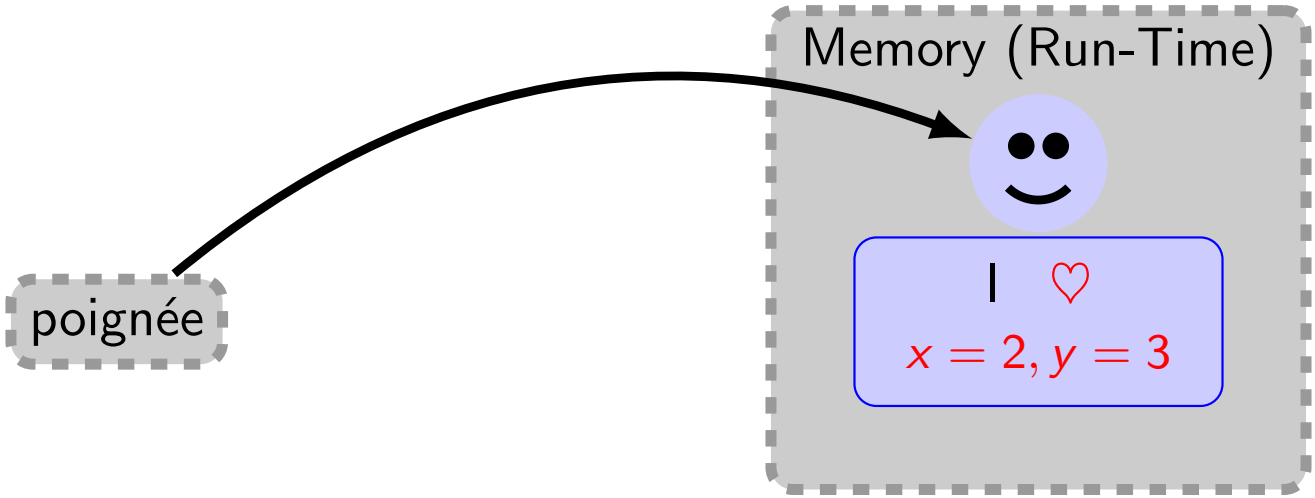
Vie d'un objet et accès à un objet



Vie d'un objet

Un objet n'a de réalité qu'à l'exécution du programme (on parle également de *run-time*).

Vie d'un objet et accès à un objet



Accès à un objet

Les objets ne sont pas accessibles directement. On y accède par l'intermédiaire de **poignées** (*handles*).

On dit alors qu'un objet est **attaché** à une poignée.

La classe vue comme un type

Une classe est un **type** qui permet de :

- **créer** des objets
- **déclarer** des poignées auxquelles sont attachés ces objets

Les poignées : création d'objet

Les objets sont créés dynamiquement par l'opérateur **new**.

Syntaxe (création d'un objet)

```
new Point();      // creation d'un objet Point
```

L'opérateur retourne **l'identité** de l'objet créé.

On peut utiliser l'identité d'un objet pour accéder à ses caractéristiques (attributs et méthodes).

Les poignées : attacher un objet

On stocke l'identité de l'objet dans une **poignée** qui est une variable dont le type est le nom d'une **classe**.

Syntaxe (déclaration d'une poignée et attachement)

```
Point p;          // declarer une poignee p de type Point
p = new Point(); // creer un objet et l'attacher a la poignee
```

La valeur par défaut d'une poignée est **null**. Elle indique qu'aucun objet n'est attaché à la poignée.

On peut regrouper déclaration de la poignée et initialisation :

```
Point p = new Point();
```

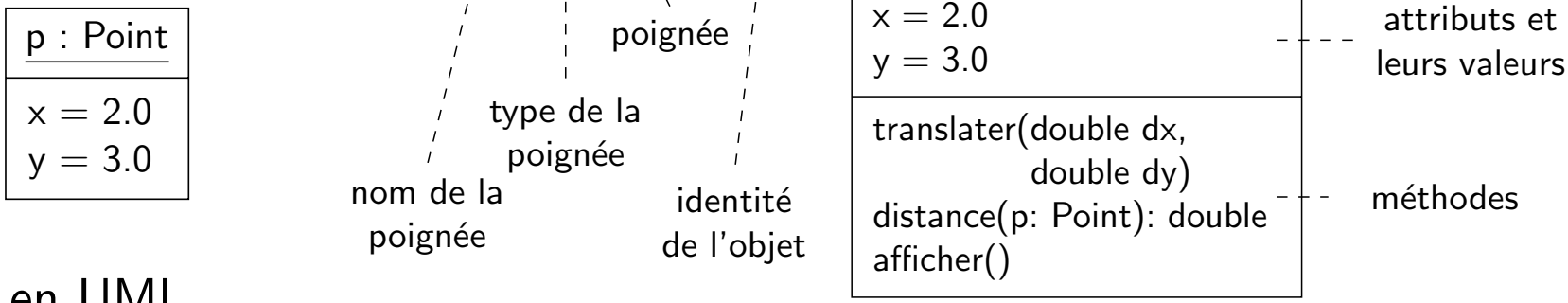
Liens entre objets, poignées, classes et types

- un objet est une **instance** d'une classe ;
- un objet est instance d'une et d'une seule classe ;
- un objet a pour type le nom de sa classe mais peut avoir d'autres types (cf. héritage et interfaces définis plus loin dans le cours) ;
- une poignée a un type qui est le nom d'une classe ;
- on peut initialiser une poignée avec toute expression dont le type est le même que celui de la poignée (ou un sous-type, voir héritage et interfaces).

Représentation UML d'un objet

Exemple :

```
Point p = new Point();
...                                    // translation, calcul distance etc.
```



En UML, on peut omettre le nom de la classe ou de l'objet : c'est le « souligné » qui indique que c'est un objet.

Plan de la partie 2 - Classes et objets

Principe d'accès uniforme

Pour accéder à une caractéristique d'un objet (attribut ou méthode), on utilise la **notation pointée** :

Syntaxe (notation pointée)

```
poignee.caracteristique
```

où `poignee` est une poignee vers l'objet et `caracteristique` est la caractéristique que l'on veut accéder.

Le fait d'accéder aux attributs et aux méthodes d'un objet de la même façon permet de respecter **en partie** le **principe d'accès uniforme**.

Principe (accès uniforme)

Tous les services offerts par un module devraient être accessibles à travers une notation uniforme qui ne doit pas laisser transparaître si les services sont implantés à travers du stockage ou du calcul.

Accès à un attribut

Accès à un attribut :

```
poignee.attribut                    // forme generale

Point p = new Point();
double x = p.x;                     // acces en lecture
p.y = 1;                            // acces en modification
```

Attention à la poignée **null**

```
Point q = null;                     // pas d'objet attache a q
double x = q.x;                     // -> NullPointerException
```

Méthodes : utilisation

Comme un attribut, une méthode est appliquée sur une poignée et est exécutée sur l'objet associé à cette poignée (principe d'appel uniforme).

```
poignee.methode(p1,...,pN)      // forme generale

Point p = new Point();
Point q = new Point();
p.translater(2, 3);             // Translater le point (methode void)
double d = p.distance(q);       // Utiliser une methode non void
p.afficher();
p.distance(q);                  // Valide mais quel interet ?

Point r = null;                 // pas d'objet attache a r
r.translater(1,2);              // -> NullPointerException
```

Principe (liaison statique)

Le compilateur accepte l'appel `p.m(p1,...,pn)` ssi il existe dans la classe définissant le type de la poignée `p` une méthode `m` d'arité n telle que les types de `p1,...,pn` soient compatibles avec sa signature.

Le paramètre implicite **this**

Une méthode est toujours appliquée à un objet, appelé **récepteur**, généralement à travers une poignée.

➥ Pb : le récepteur n'apparaît pas comme paramètre d'une méthode !

C'est donc un paramètre **implicite**. On peut y faire référence en utilisant le mot clé **this**.

Point.java

```
/**
 * Translater le point.
 *
 * @param dx l'abscisse du vecteur de translation
 * @param dy l'ordonnee du vecteur de translation
 */
void translater(double dx, double dy) {
    this.x += dx;
    this.y += dy;
}
```

Le paramètre implicite `this`

Une méthode est toujours appliquée à un objet, appelé **récepteur**, généralement à travers une poignée.

➥ Pb : le récepteur n'apparaît pas comme paramètre d'une méthode !

Principe

Nous choisirons d'écrire explicitement **`this`** à chaque fois que l'on appelle une méthode ou que l'on veut accéder à un attribut de l'objet récepteur d'une méthode.

Classe cliente

Une classe `A` utilisant des objets instances d'une classe `B` est dite **classe cliente de `B`**.

Par exemple, la classe `TestPoint` est cliente de la classe `Point` :

TestPoint.java

```
class TestPoint {
    public static void main(String[] args) {
        Point p = new Point();

        p.translater(5.0, -2.0);
        double distance = p.distance(new Point());
        System.out.println("Distance a l'origine : " + distance);
    }
}
```

Classe cliente

Une classe `A` utilisant des objets instances d'une classe `B` est dite **classe cliente de `B`**.

Remarque

Attention, une classe « descriptive » peut également être cliente d'une autre classe (la classe `Orbite` par exemple...)

Plan de la partie 2 - Classes et objets

Problème posé

Supposons que l'on veuille modéliser maintenant des points qui sont **exclusivement** situés dans le quart de plan positif.
On considère le programme précédent qui utilise la classe `Point` :

TestPoint.java

```
class TestPoint {
    public static void main(String[] args) {
        Point p = new Point();

        p.x = -1.0;
        p.y = -2.0;
        double distance = p.distance(new Point());
        System.out.println("Distance a l'origine : " + distance);
    }
}
```

Comme on accède directement aux coordonnées du point depuis l'extérieur de la classe, on peut donner n'importe quelle valeur compatible avec **double** à `x` et `y`.

➥ on obtient un objet de type `Point` **incohérent** par rapport aux spécifications

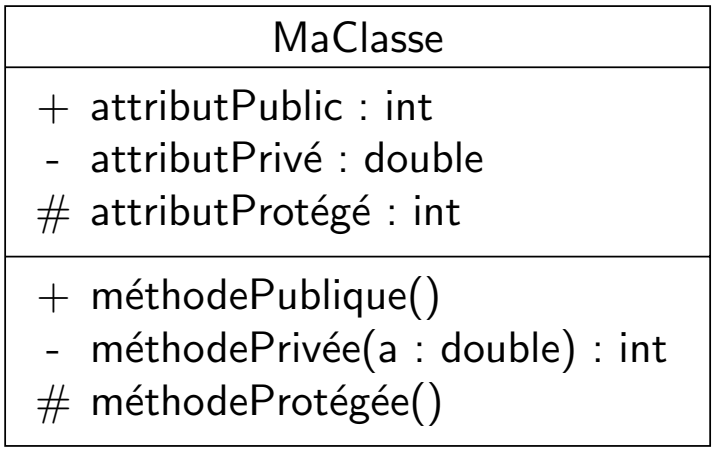
Droit d'accès des caractéristiques

Chaque caractéristique (attribut ou méthode) d'une classe a un droit d'accès. Il existe quatre niveaux de droits d'accès en Java :

- **`public`** : accessible depuis toutes les classes ;
- **`private`** : accessible seulement depuis la classe ;
- absence de modifieurs : droit d'accès de **paquetage**, accessible depuis toutes les classes du même paquetage ;
- **`protected`** : accessible du paquetage **et** des sous-classes (cf. héritage).

Accessible depuis une méthode définie dans	**Droits d'accès/Visibilité**			
	`public`	**`protected`**	**`default`**	**`private`**
La même classe	oui	oui	oui	oui
Une classe du même paquetage	oui	oui	oui	non
Une sous-classe d'un autre paquetage	oui	oui	non	non
Une autre classe d'un autre paquetage	oui	non	non	non

Représentation des droits d'accès




```
public    int    attributPublic
private   double attributPrive
protected int    attributProtege


public    void   methodePublique()
private   int    methodePrivee(double a)
protected void   methodeProtegee()
```

Les symboles + - # correspondent respectivement à **public**, **private** et **protected**.

Vérification de la visibilité

Principe

La légalité de l'accès à un attribut ou d'un appel à une méthode du point de vue de la visibilité est vérifiée à la compilation.

Vérification de la visibilité

Si l'on rend les attributs x et y privés dans la classe Point :

Point.java

```
class Point {

    private double x;
    private double y;

    ...
}
```

Vérification de la visibilité

en considérant le programme suivant :

Point.java

```
class TestPoint {
    public static void main(String[] args) {
        Point p = new Point();

        p.x = -1.0;
        p.y = -2.0;
        double distance = p.distance(new Point());
        System.out.println("Distance a l'origine : " + distance);
    }
}
```

Vérification de la visibilité

on obtient **à la compilation** :

shell

```
[tof@suntof]~ $ javac TestPoint.java
TestPoint.java:5: error: x has private access in Point
        p.x = -1.0;
         ^
TestPoint.java:6: error: y has private access in Point
        p.y = -2.0;
         ^
2 errors
```

Principe d'encapsulation

Principe (encapsulation)

Un attribut doit toujours être déclaré **`private`**.

À appliquer tout au long du cours !

Masquage d'informations

La règle précédente permet de :

- respecter le principe d'accès uniforme ;
- permettre à l'auteur de la classe de garantir la **cohérence** des objets de la classe.

La plupart des méthodes représentent des services offerts par les objets : elles sont donc **publiques**.

Les attributs privés doivent (souvent) disposer d'**accesseurs** et de **modifieurs** :

```
private double x;

public double getX() {
  return this.x;
}

public void setX(double x) {
  this.x = x;
  // cette methode ne sert pas forcement, cf. translater
}
```

Retour sur le point dans le quart de plan positif

Si l'on veut modéliser via une classe `PointPositif` :

- que faire lorsque l'utilisateur de la classe utilise un paramètre négatif dans `setX` ?
- comment vérifier que l'utilisateur n'appelle pas `setX` avec un paramètre négatif ?

Nous verrons dans la suite du cours qu'une réponse à la première question peut être apportée par les **exceptions**.
Nous ne verrons (malheureusement) pas dans le cours une réponse à la deuxième question, mais on peut utiliser des techniques de **spécification formelle** comme la **programmation par contrat** et les outils associés pour le vérifier.

Remarque

Une mauvaise solution souvent utilisée pour la première question est d'utiliser l'opposé du paramètre. Pourquoi ?

Lost in space...

Exercice

Reprendre la conception de la classe `Orbite` et appliquer le principe d'encapsulation.
Quel est l'impact sur les méthodes permettant de changer la valeur des attributs ?
Comment éviter d'écrire plusieurs fois le même code dans des méthodes différentes ?

Plan de la partie 2 - Classes et objets

Surcharge d'opérations

En Java, indiquer le nom d'une méthode n'est pas suffisant pour l'identifier. Il faut préciser :

- la classe à laquelle elle appartient ;
- son nom ;
- le nombre de ses paramètres ;
- le type de chacun de ses paramètres (de façon ordonnée).

Exemple : toutes ces méthodes sont différentes :

```
class A {
  void afficher()                          // afficher sans parametre
  void afficher(int i)                     // afficher un entier
  void afficher(long i)                    // afficher un long
  void afficher(String str)                // afficher une chaine
  void afficher(String s, int l)           // avec une longueur l
  void afficher(String s, int l, char m) // m == 'c', 'g', 'd'
  void afficher(int nbFois, String str)  // c'est une autre methode
}
```

Surcharge : résolution

Pour résoudre un appel de méthode, le compilateur s'appuie sur le nombre et le type des paramètres effectifs :

```
// on suppose que l'on est dans une methode de A
afficher(10);                    // afficher(int)
afficher(10L);                   // afficher(long)
afficher("Bonjour", 20, 'c');    // afficher(String, int, char))
afficher("Bonjour");             // afficher(String)
afficher();                      // afficher()
afficher(true);                  // erreur a la compilation
afficher(20, "Bonjour");         // afficher(int, String)
```

Intérêt : éviter les répétitions de noms.

Espace de nommage

Deux classes peuvent avoir une méthode « identique », car une classe définit un **espace de nommage** :

```
A x1;              // une poignee x1 sur un objet de type A
B x2;              // une poignee x2 sur un objet de type B
...                // les initialisations de x1 et x2
x1.afficher(10);   // afficher(int) de A
x2.afficher(5);    // afficher(int) de B
```

Lost in space...

Exercice

Peut-on utiliser la surcharge pour distinguer la méthode permettant de calculer la position d'un point à partir de l'angle au centre de celle utilisant l'angle au foyer ?

Plan de la partie 2 - Classes et objets

Problème posé

On considère le programme suivant qui utilise la classe `Point` :

TestPoint.java

```
class TestPoint {
    public static void main(String[] args) {
        Point p = new Point();

        p.x = -1.0;
        p.y = -2.0;
        double distance = p.distance(new Point());
        System.out.println("Distance a l'origine : " + distance);
    }
}
```

On construit un objet de type `Point` sans l'initialiser.

➥ est-ce vraiment ce que l'utilisateur veut ?

Constructeurs : principe

La création d'un objet nécessite deux étapes :

1. la **réservation de la zone mémoire nécessaire** qui est assurée par le compilateur ;
2. l'**initialisation de la zone mémoire**. Le compilateur ne peut utiliser que les valeurs par défaut des attributs.

Il y a alors risque d'avoir une initialisation incorrecte, ou non conforme aux souhaits de l'utilisateur.

Les **constructeurs** permettent :

- au programmeur de définir comment un objet peut être initialisé ;
- au compilateur de vérifier que tout objet créé est correctement initialisé.

Exemple : tout objet de type `Point` peut être initialisé à partir de la valeur de ses coordonnées.

Constructeurs en Java

En Java, un constructeur ressemble à une méthode, mais ce n'est pas une méthode. . .

Syntaxe (constructeur)

- un constructeur a nécessairement le même nom que sa classe ;
- un constructeur ne peut pas avoir de type de retour.

Exemple :

Point.java

```
public Point(double x, double y) {
  this.x = x;
  this.y = y;
}
```

Constructeurs en Java

En Java, un constructeur ressemble à une méthode, mais ce n'est pas une méthode. . .

Syntaxe (constructeur)

- un constructeur a nécessairement le même nom que sa classe ;
- un constructeur ne peut pas avoir de type de retour.

Un constructeur a une accessibilité, comme les méthodes et les attributs.

Remarque

Faire attention à ne pas mettre de type de retour !

Constructeur et surcharge

Le nom des constructeurs est imposé, mais la surcharge permet de définir plusieurs constructeurs pour une même classe.

Exemple (très idiot...) :

```
public Point(double valeur) {
  this(valeur, valeur);
  // appel au constructeur Point(double, double)
  // cet appel est necessairement la premiere instruction
}
```

Principe (appel de constructeur)

Un constructeur peut appeler un autre constructeur en utilisant **`this(...)`** comme **toute première instruction**. Les paramètres de **`this`** permettent de sélectionner l'autre constructeur.

Création d'un objet

Syntaxe (création d'un objet)

```
new NomClasse(parametres effectifs);
```

Les paramètres effectifs sont fournis par l'utilisateur de la classe et sont utilisés par le compilateur pour sélectionner le constructeur à appliquer (principe de liaison statique).
Exemples une fois que l'on a défini les deux constructeurs précédents :

```
new Point();                    // incorrect !
new Point(2.0, 5.0);            // OK (2.0, 5.0)
new Point(3.0);                 // OK (3.0, 3.0)
new Point(1.0, 2.0, 3.0);       // incorrect !
```

Principe (constructeur)

Le constructeur permet donc de rendre atomiques la réservation de la mémoire et son initialisation.

Vérification à la compilation

Avec le programme suivant :

Point.java

```
class TestPoint {
    public static void main(String[] args) {
        Point p = new Point();

        p.setX(-1.0);
        p.setY(-2.0);
        double distance = p.distance(new Point());
        System.out.println("Distance a l'origine : " + distance);
    }
}
```

Vérification à la compilation

on obtient à la compilation :

shell

```
[tof@suntof]~ $ javac TestPoint.java
TestPoint.java:3: error: constructor Point in class Point cannot be
applied to given types;
        Point p = new Point();
                  ^
  required: double,double
  found: no arguments
  reason: actual and formal argument lists differ in length
TestPoint.java:7: error: constructor Point in class Point cannot be
applied to given types;
        double distance = p.distance(new Point());
                                     ^
  required: double,double
  found: no arguments
  reason: actual and formal argument lists differ in length
2 errors
```

Constructeur par défaut

On appelle **constructeur par défaut** le constructeur qui ne prend pas de paramètres.
C'est le constructeur utilisé si aucun paramètre n'est fourni lors de la création d'un objet.

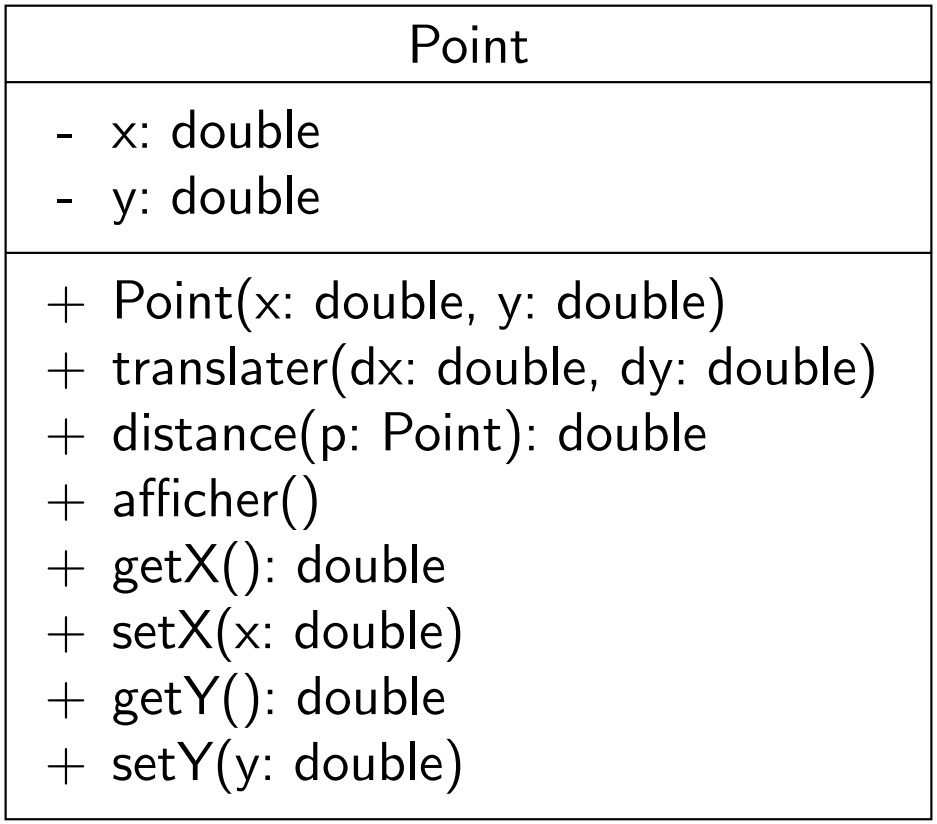
Principe (constructeur par défaut)

1. si **aucun** constructeur n'est défini sur une classe, le système synthétise un constructeur par défaut, qui ne fait rien, **le constructeur prédéfini** ;
2. dès qu'un constructeur est défini sur une classe, le constructeur par défaut synthétisé par le système disparaît.

Remarque

On peut définir un constructeur par défaut, mais...

La nouvelle version de `Point`



La nouvelle version de Point

Point.java

```
/**
 * <code>Point</code> modelise un point du plan.
 *
 * @author <a href="mailto:garion@isae.fr">Christophe Garion</a>
 */
class Point {

    private double x;

    private double y;

    /**
     * Creer un point a partir de ses coordonnees.
     *
     * @param x l'abscisse du point
     * @param y l'ordonnee du point
     */
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
```

La nouvelle version de Point

Point.java

```
    /**
     * <code>getX</code> retourne l'abscisse du point
     *
     * @return un <code>double</code>  qui est l'abscisse du point
     */
    public double getX() {
        return this.x;
    }

    /**
     * <code>getY</code> retourne l'ordonnee du point
     *
     * @return un <code>double</code> qui est l'ordonnee du point
     */
    public double getY() {
        return this.y;
    }
```

La nouvelle version de Point

Point.java

```
    /**
     * <code>setX</code> permet de modifier la valeur de l'abscisse
     * du point.
     *
     * @param x un <code>double</code> qui est la nouvelle abscisse
     */
    public void setX(double x) {
        this.x = x;
    }

    /**
     * <code>setY</code> permet de modifier la valeur de l'ordonnee
     * du point.
     *
     * @param y un <code>double</code> qui est la nouvelle ordonnee
     */
    public void setY(double y) {
        this.y = y;
    }
```

La nouvelle version de Point

Point.java

```
    /**
     * Translater le point.
     *
     * @param dx l'abscisse du vecteur de translation
     * @param dy l'ordonnee du vecteur de translation
     */
    public void translater(double dx, double dy) {
        this.x += dx;
        this.y += dy;
    }

    /**
     * Calculer la distance a un autre point.
     *
     * @param p le point servant a calculer la distance
     * @return la distance entre <code>this</code> et p. Elle
     *         est positive.
     */
    public double distance(Point p) {
        double dx2 = Math.pow(this.x - p.x, 2);
        double dy2 = Math.pow(this.y - p.y, 2);

        return Math.sqrt(dx2 + dy2);
    }
```

La nouvelle version de Point

Point.java

```
    /**
     * Afficher le point sous la forme (x, y).
     */
    public void afficher() {
        System.out.println("(" + this.x + "," + this.y + ")");
    }
}
```

Lost in space...

Exercice

Définir le ou les constructeurs de la classe `Orbite`.

Plan de la partie 2 - Classes et objets

Attributs de classe : déjà vus. . .

Que penser de l'expression suivante :

```
Math.sqrt(4); // racine carree de 4
```

- `sqrt(`**`double`**`)` est bien une méthode mais elle n'est pas appliquée à un objet mais à une classe (`Math` en l'occurence) ;
- on dit que `sqrt(`**`double`**`)` est une **méthode de classe** ;
- `sqrt(`**`double`**`)` travaille exclusivement sur son paramètre.

Méthodes de classe : déjà vus. . .

Que penser de l'instruction suivante :

```
System.out.println("Coucou");
```

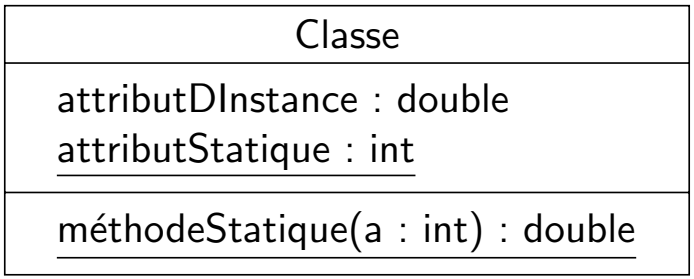
- `out` est un attribut (il n'est pas suivi de parenthèses), mais il est appliqué à une classe (`System`) ;
- il n'est pas spécifique à un objet particulier : la sortie standard est la même pour tout le monde !
- il s'agit d'un **attribut de classe**.

Attributs et méthodes de classe : représentation

On distingue :

- les attributs et méthodes d'instance, qui sont toujours appliqués à un objet (éventuellement référencé par **`this`**). On les appelle simplement attributs et méthodes ;
- attributs et méthodes de classe : appliqués à une classe.

En Java, on utilise le modifieur **`static`** qui indique si un attribut ou une méthode est de classe ou non.
En UML, on souligne l'attribut ou la méthode concerné.




```
double attributDInstance;
static double attributStatique;

static double methodeStatique(int a) {
    ...
}
```

Attributs et méthodes de classe : visibilité et utilisation

On peut appliquer des droits d'accès sur une méthode ou un attribut de classe.

```
public class Math {
  public static double sqrt(double val) {
    ...
  }
}
```

Utilisation : `NomClasse.attribut` ou `NomClasse.methode(...)`

Un exemple d'attribut de classe : compter le nombre d'instances d'une classe que l'on a créées.
Un exemple de méthode de classe particulière : la méthode principale `main`.

Lost in space...

Exercice

Pourrait-on définir des attributs ou des méthodes statiques pour la classe `Orbite` ?

Retour sur le point dans le quart de plan positif

Un des problèmes que nous avions à résoudre dans le cas du point situé dans le quart de plan positif est d'empêcher la création d'un point à partir de coordonnées dont l'une au moins est négative.

Le constructeur de `Point` ne permet pas d'éviter cela :

Point.java

```
public Point(double x, double y) {
    this.x = x;
    this.y = y;
}
```

Retour sur le point dans le quart de plan positif

Une solution est d'utiliser un patron de conception (*design pattern*) qui s'appelle **factory method**.

Une *factory method* est une méthode statique permettant la création d'un objet. Comme on a une méthode classique, on peut renvoyer **`null`** si les paramètres de construction de l'objet ne sont pas corrects.

Il suffit ensuite de rendre le constructeur privé (c'est possible !) pour forcer l'utilisateur à utiliser la *factory method*.

Retour sur le point dans le quart de plan positif

Exemple avec `PointPositif` :

PointPositif.java

```
private PointPositif(double x, double y) {
    this.x = x;
    this.y = y;
}

public static PointPositif creerPointPositif(double x, double y) {
    if (x >= 0 && y >= 0) {
        return new PointPositif(x, y);
    } else {
        return null;
    }
}
```

Retour sur le point dans le quart de plan positif

Exemple avec `PointPositif` :

PointPositif.java

```
private PointPositif(double x, double y) {
    this.x = x;
    this.y = y;
}

public static PointPositif creerPointPositif(double x, double y) {
    if (x >= 0 && y >= 0) {
        return new PointPositif(x, y);
    } else {
        return null;
    }
}
```

Exercice

Comment créer un objet de type `PointPositif` ?

3 - Gestion de version

Premier problème : gérer le changement

Problème

Comment faire pour se « souvenir » des modifications apportées sur une application ?

Pourquoi ?

- « revenir en arrière » suite à des modifications qui se révèlent être inutiles voire néfastes
- pouvoir posséder une version stable de l'application et continuer son développement
- pouvoir proposer une autre version de l'application en repartant d'une version antérieure
- ...

Premier problème : discussion...

Idée

Avoir un historique de l'application sous forme de versions

Mais...

- que doit-on gérer dans cet historique ?
  - ➥ les **codes sources**, les fichiers de configuration, de **build**
- comment gère-t-on l'historique ?
  - créer des répertoires avec les numéros de versions et recopier **tous** les fichiers concernés à chaque fois
    - ➥ ingérable !
  - ajouter des numéros de version sur les noms de fichiers
    - ➥ pas possible en Java pour les sources !
    - ➥ comment garantir la cohérence de ces numéros ?

Deuxième problème : partager le code

Problème

Comment permettre à plusieurs développeurs/concepteurs de partager du code/des documents pour travailler **en même temps** ?

Pourquoi ?

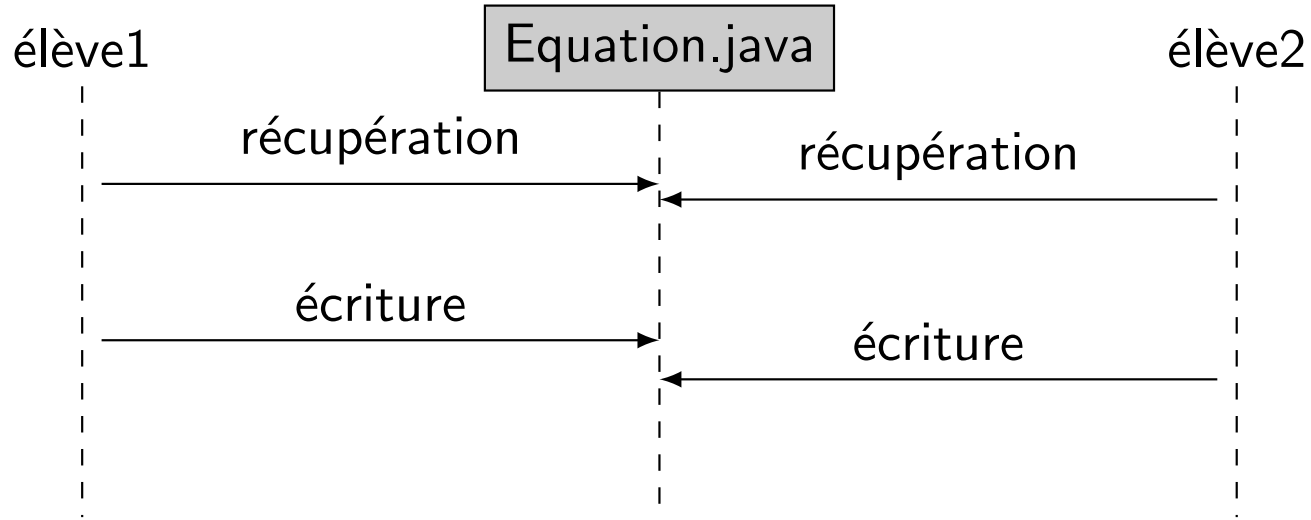
- permettre facilement à une équipe de travailler sur le même projet, en particulier sur le code source du projet
- éventuellement gérer les **conflits** lorsque deux personnes travaillent sur le même document

Deuxième problème : solutions estudiantes. . .

1. partage des documents par **envoi par mail**
   ➥ complétement ingérable

Deuxième problème : solutions estudiantes. . .

1. partage des documents par **envoi par mail**
   - ➥ complétement ingérable
2. **ouverture des droits** sur un des comptes du binôme
   - ➥ pas très sécurisé. . .
   - ➥ ne permet pas de gérer les conflits éventuels



Deuxième problème : solutions estudiantes. . .

3. utilisation de Dropbox ou d'un système équivalent
   - pas vraiment une solution
   - historique limité
   - pas de *diffs*, pas de messages de *commit*
   - gestion des conflits limitée

Plan de la partie 3 - Gestion de version

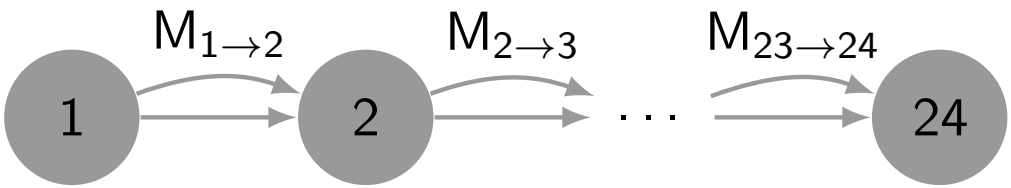
Gestion de sources

Pour pouvoir pallier les problèmes présentés précédemment, nous allons utiliser un système de **gestion de sources** (*revision control*).

Un logiciel de gestion de sources va permettre de gérer facilement :

- les modifications apportées aux fichiers du projet
- le fait que plusieurs utilisateurs peuvent contribuer au projet
- (non abordé ici) la création de branches pour le développement de fonctionnalités sans toucher à une version donnée de l'application

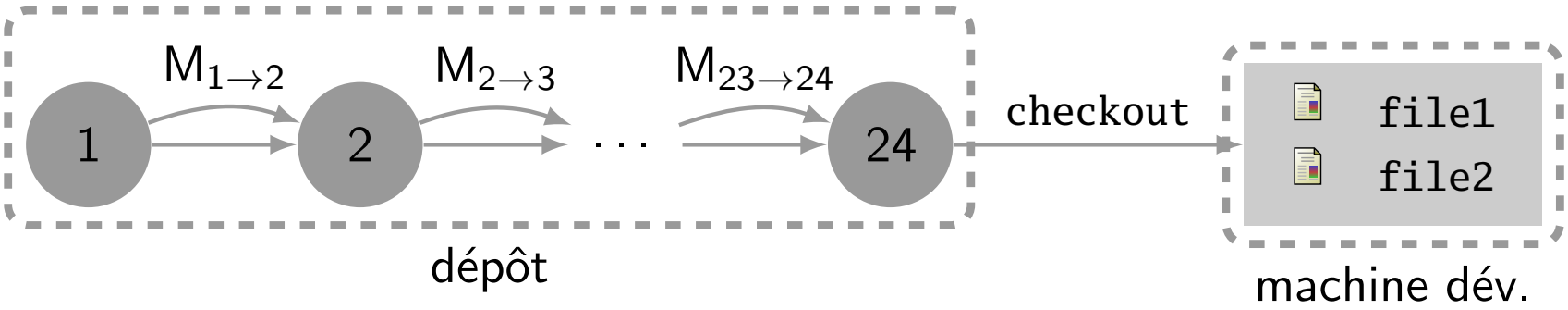
Gestion de sources : vocabulaire et concepts



Les évolutions des documents sont représentées par des **révisions**.
Les révisions sont souvent des entiers : révision 1, révision 2, etc.

Pour passer d'une révision à une autre, on applique des **modifications** aux fichiers constituant le projet.
Une modification peut concerner **plusieurs fichiers** : elle représente le passage d'un état « cohérent » du projet à un autre état « cohérent ».
Le logiciel de gestion de sources **ne conserve que les modifications**.

Gestion de sources : vocabulaire et concepts

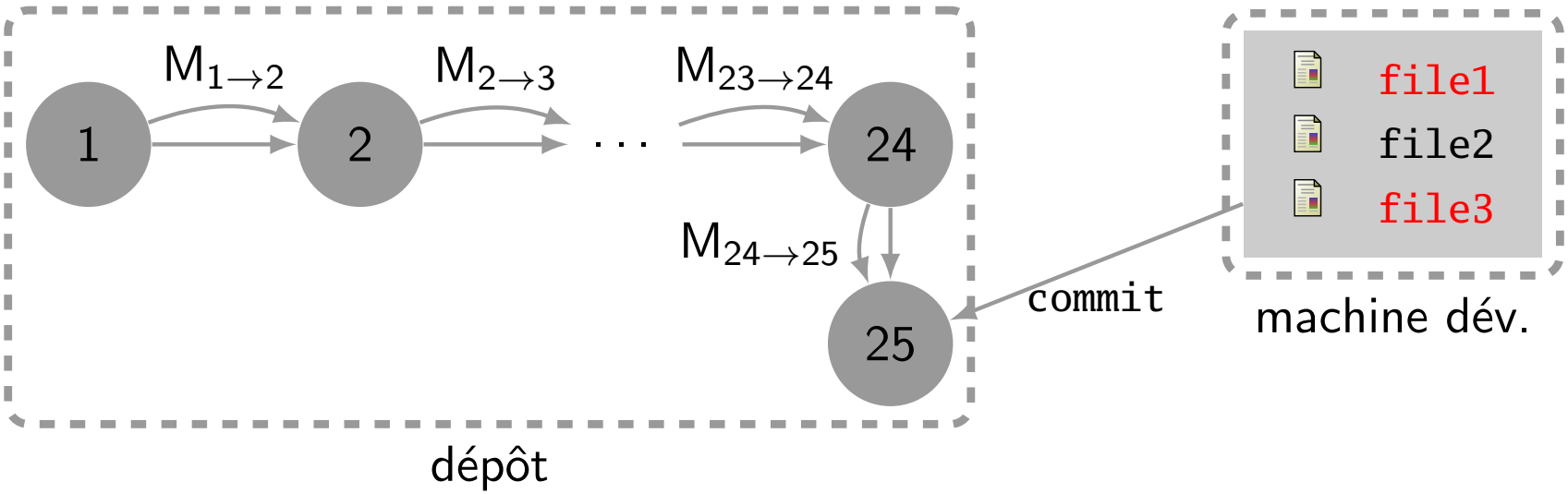


Les fichiers gérés par le logiciel de gestion de sources sont conservés dans un **dépôt** (*repository*).

Pour pouvoir travailler, le développeur fait une **copie locale** du dépôt.
Par défaut, on récupère la dernière révision du dépôt, mais on peut choisir.

On appelle cette opération un **checkout**.

Gestion de sources : vocabulaire et concepts



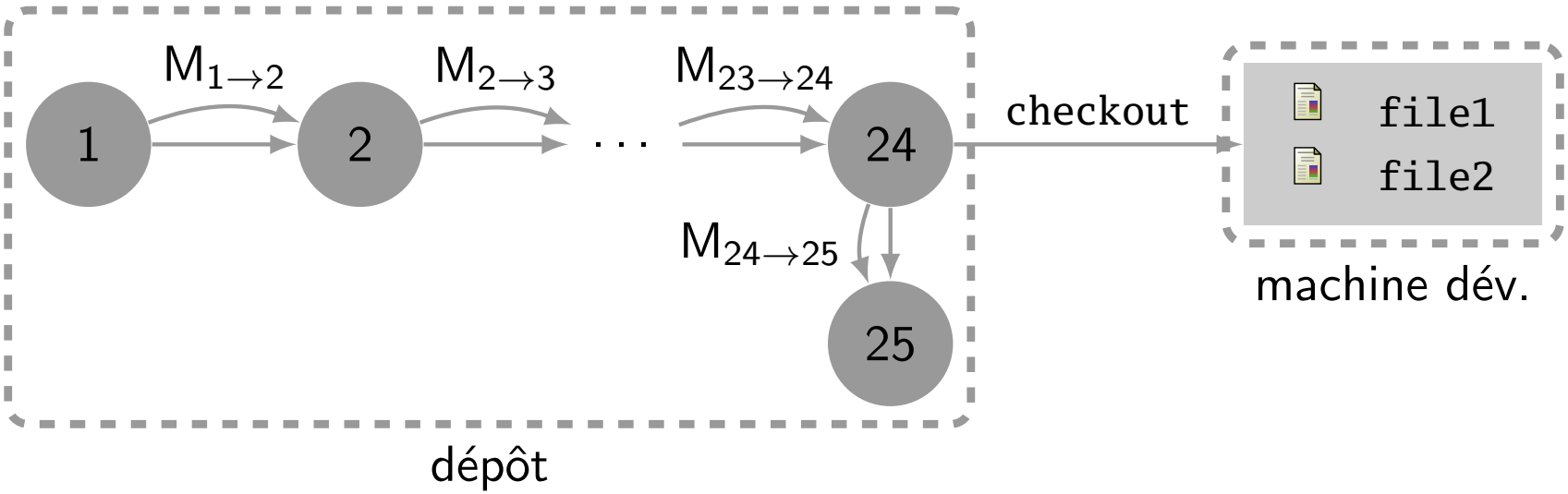
Lorsque le développeur a fait les modifications qu'il souhaitait sur les fichiers, il peut **soumettre** ses modifications au dépôt.

On appelle cette opération un **commit**.

Remarque

Des conflits peuvent apparaître lors de cette opération !

Gestion de sources : vocabulaire et concepts

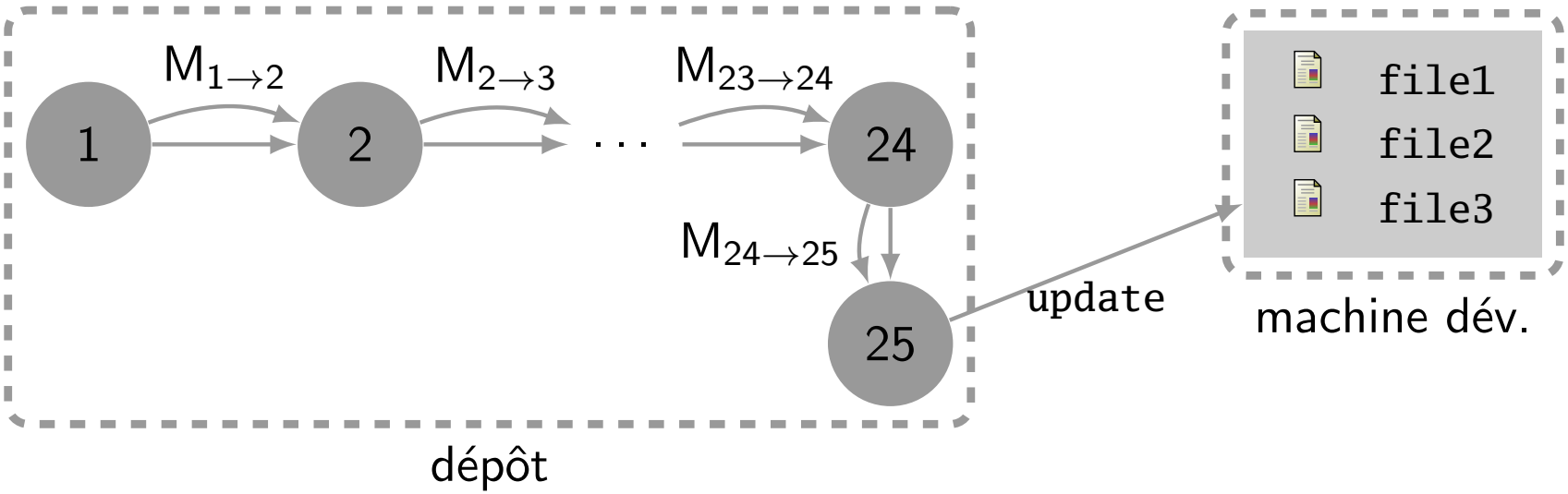


Lorsqu'un développeur veut mettre à jour sa copie locale, il effectue une opération appelée **update**.

Remarque

Des conflits peuvent apparaître lors de cette opération !

Gestion de sources : vocabulaire et concepts



Lorsqu'un développeur veut mettre à jour sa copie locale, il effectue une opération appelée **update**.

Remarque

Des conflits peuvent apparaître lors de cette opération !

Plan de la partie 3 - Gestion de version

Subversion

Nous allons utiliser comme logiciel de gestion de sources Subversion, un logiciel libre.

The Apache Software Foundation (2013).
Apache Subversion.
http://subversion.apache.org/.

Collins-Sussman, B., B. W. Fitzpatrick et C. Michael Pilato (2004).
Version control with Subversion.
O'Reilly.
http://svnbook.red-bean.com/.

Plan de la partie 3 - Gestion de version

Commandes utilisées

~alice/ - rev. 1

file1.txt

Coucou

REPOSITORY - rev. 1

file1.txt

Coucou

~bob/ - rev. 1

file1.txt

Coucou

Alice copie le dépôt (idem Bob) :

shell alice

```
[alice@computer]~ $ svn checkout URL_REPOSITORY
A    scm/alice/file1.txt
Checked out revision 1.
```

Commandes utilisées

~alice/ - rev. 1

file1.txt

Coucou

file2.txt

Hello

REPOSITORY - rev. 1

file1.txt

Coucou

~bob/ - rev. 1

file1.txt

Coucou

Alice crée un autre fichier.

Commandes utilisées

~alice/ - rev. 1

file1.txt

Coucou

file2.txt

Hello

REPOSITORY - rev. 1

file1.txt

Coucou

~bob/ - rev. 1

file1.txt

Coucou

Elle peut vérifier que sa copie locale n'est pas identique au dépôt.

shell alice

```
[alice@computer]~ $ svn status
?       file2.txt
```

Commandes utilisées

~alice/ - rev. 2

file1.txt

Coucou

file2.txt

Hello

REPOSITORY - rev. 2

file1.txt

Coucou

file2.txt

Hello

~bob/ - rev. 1

file1.txt

Coucou

Elle peut ensuite l'**ajouter** et le **soumettre** au dépôt.

shell alice

```
[alice@computer]~ $ svn add file2.txt
A         file2.txt
Adding         file2.txt
[alice@computer]~ $ svn commit -m "adding file2.txt"
Transmitting file data .
Committed revision 2.
```

Commandes utilisées

~alice/ - rev. 2

file1.txt

Coucou

file2.txt

Hello

REPOSITORY - rev. 2

file1.txt

Coucou

file2.txt

Hello

~bob/ - rev. 2

file1.txt

Coucou

file2.txt

Hello

Bob peut mettre à jour sa copie locale.

shell Bob

```
[bob@computer]~ $ svn update
Updating '.':
A    file2.txt
Updated to revision 2.
```

Commandes utilisées

~alice/ - rev. 2

file1.txt

Coucou

file2.txt

Hello

REPOSITORY - rev. 3

file1.txt

Coucou

file2.txt

Hello

file3.txt

c'est moi

~bob/ - rev. 3

file1.txt

Coucou

file2.txt

Hello

file3.txt

c'est moi

Bob ajoute un fichier et le soumet au dépôt.

Commandes utilisées

~alice/ - rev. 4

file1.txt

Coucou

file2.txt

Bonjour

REPOSITORY - rev. 4

file1.txt

Coucou

file2.txt

Bonjour

file3.txt

c'est moi

~bob/ - rev. 3

file1.txt

Coucou

file2.txt

Hello

file3.txt

c'est moi

Alice modifie `file2.txt` et le soumet (attention, la mise à jour n'est pas automatique !). **Il n'y a pas de conflits ici.**

shell Alice

```
[alice@computer]~ $ svn commit -m "changing Hello in file2.txt"
Sending        file2.txt
Transmitting file data .
Committed revision 4.
```

Plan de la partie 3 - Gestion de version

Conflits

~alice/ - rev. 1

file1.txt

Coucou

REPOSITORY - rev. 1

file1.txt

Coucou

~bob/ - rev. 1

file1.txt

Coucou

Conflits

~alice/ - rev. 2

file1.txt

Bonjour

REPOSITORY - rev. 2

file1.txt

Bonjour

~bob/ - rev. 1

file1.txt

Coucou

Alice modifie `file1.txt` et soumet sa version.

Conflits

~alice/ - rev. 2

file1.txt

Bonjour

REPOSITORY - rev. 2

file1.txt

Bonjour

~bob/ - rev. 1

file1.txt

Hello

Bob modifie `file1.txt` et veut soumettre sa version.

shell Bob

```
[bob@computer]~ $ svn commit -m "changing Coucou to Hello in file1.txt"
Sending        file1.txt
svn: E155011: Commit failed (details follow):
svn: E155011: File 'bob/file1.txt' is out of date
svn: E160028: File '/file1.txt' is out of date
```

Conflits

~alice/ - rev. 2

file1.txt

Bonjour

REPOSITORY - rev. 2

file1.txt

Bonjour

~bob/ - rev. 1

file1.txt

Hello

Bob peut mettre à jour sa copie locale.

shell Bob

Conflits

~alice/ - rev. 2

file1.txt

```
Bonjour
```

REPOSITORY - rev. 2

file1.txt

```
Bonjour
```

~bob/ - rev. 1

file1.txt

```
Hello
```

Bob peut choisir d'éditer (e). Il obtient un fichier **temporaire** qui contient les différences entre sa version et celle du dépôt.

file1.txt.tmp

```
<<<<<<< .mine
Hello
=======
Bonjour
>>>>>>> .r2
```

Conflits

~alice/ - rev. 2

file1.txt

Bonjour

REPOSITORY - rev. 2

file1.txt

Bonjour

~bob/ - rev. 2

file1.txt

Hello

Bob modifie le fichier temporaire pour conserver sa version. Il obtient alors un nouveau message de Subversion. Il choisit de signifier que le conflit est **résolu**.

shell Bob

```
[bob@computer]~ $ svn resolved file1.txt
Resolved conflicted state of 'file1.txt'
```

Conflits

~alice/ - rev. 2

file1.txt

Bonjour

REPOSITORY - rev. 3

file1.txt

Hello

~bob/ - rev. 3

file1.txt

Hello

Bob peut ensuite propager ses modifications au dépôt.

shell Bob

```
[bob@computer]~ $ svn commit -m "changing Coucou to Hello in file1.txt"
Sending        file1.txt
Transmitting file data .
Committed revision 3.
```

Conflits

~alice/ - rev. 2

file1.txt

Guten Tag

REPOSITORY - rev. 3

file1.txt

Hello

~bob/ - rev. 3

file1.txt

Hello

Alice modifie localement son fichier `file1.txt`. Elle met à jour sa copie, découvre le conflit et choisit de **reporter** (p) la gestion des conflits.

shell Alice

```
[alice@computer]~ $ svn update
Updating '.':
C    file1.txt
Updated to revision 3.
Summary of conflicts:
  Text conflicts: 1
```

Conflits

~alice/ - rev. 2

file1.txt

```
<<<<<<< .mine
Guten Tag

=======
Hello
>>>>>>> .r3
```

REPOSITORY - rev. 3

file1.txt

```
Hello
```

~bob/ - rev. 3

file1.txt

```
Hello
```

Subversion a créé un certain nombre de fichiers permettant à Alice d'avoir les différentes versions du fichier `file1.txt` : ici pour la révision 2, la révision 3 et celle de la copie courante (`file1.txt.mine`).
Le fichier `file1.txt` a la même syntaxe que celle présentée précédemment.

shell Alice

```
[alice@computer]~ $ ls
file1.txt
file1.txt.mine
file1.txt.r2
file1.txt.r3
```

Conflits

~alice/ - rev. 2

file1.txt

Guten Tag

REPOSITORY - rev. 3

file1.txt

Hello

~bob/ - rev. 3

file1.txt

Hello

Alice peut choisir le fichier qui lui convient ou modifier `file1.txt`. Elle peut indiquer directement à Subversion qu'elle souhaite conserver sa copie locale pour résoudre le conflit. Elle soumet ensuite ses changements (pas fait ici !).

shell Alice

```
[alice@computer]~ $ svn resolve --accept mine-full file1.txt
Resolved conflicted state of 'file1.txt'
```

Plan de la partie 3 - Gestion de version

Logs

Bob peut examiner les messages de *commit* relatifs à un fichier.

shell Bob

```
[bob@computer]~ $ svn log file1.txt
------------------------------------------------------------------------
r3 | bob | 2014-06-13 16:06:03 +0200 (Fri, 13 Jun 2014) | 1 line

changing Coucou to Hello in file1.txt
------------------------------------------------------------------------
r2 | alice | 2014-06-13 16:05:58 +0200 (Fri, 13 Jun 2014) | 1 line

changing Coucou to Bonjour in file1.txt
------------------------------------------------------------------------
r1 | tof | 2014-06-13 16:05:55 +0200 (Fri, 13 Jun 2014) | 1 line

initial import of file1.txt
------------------------------------------------------------------------
```

Diffs

Bob peut obtenir l'ensemble de modifications entre deux versions pour un fichier :

shell Bob

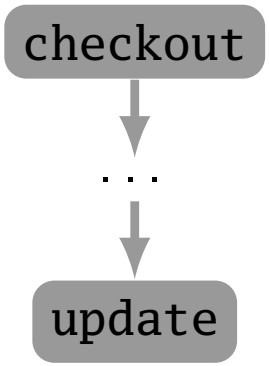
```
[bob@computer]~ $ svn diff -r2:3 file1.txt
Index: file1.txt
===================================================================
--- file1.txt   (revision 2)
+++ file1.txt   (revision 3)
@@ -1 +1 @@
-Bonjour
+Hello
```

Remarque

Le résultat de `diff` est appelé un **patch** : ce sont les changements à effectuer sur `file1.txt` pour passer de la révision 2 à la révision 3.

Plan de la partie 3 - Gestion de version

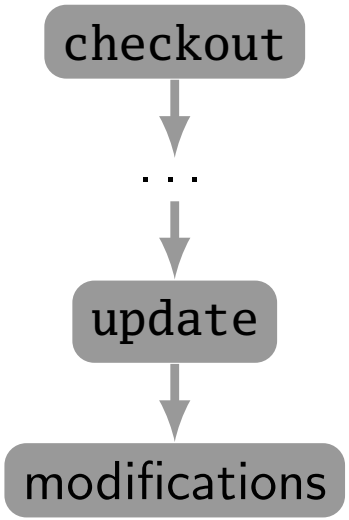
Comment utiliser subversion ?



Vérifications

- ➥ le code compile
- ➥ les tests passent

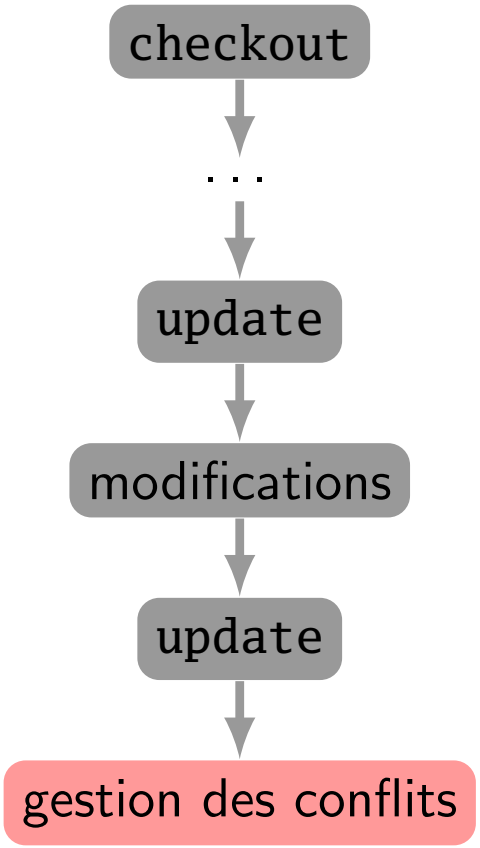
Comment utiliser subversion ?



Vérifications

- ➥ le code compile
- ➥ les tests passent

Comment utiliser subversion ?



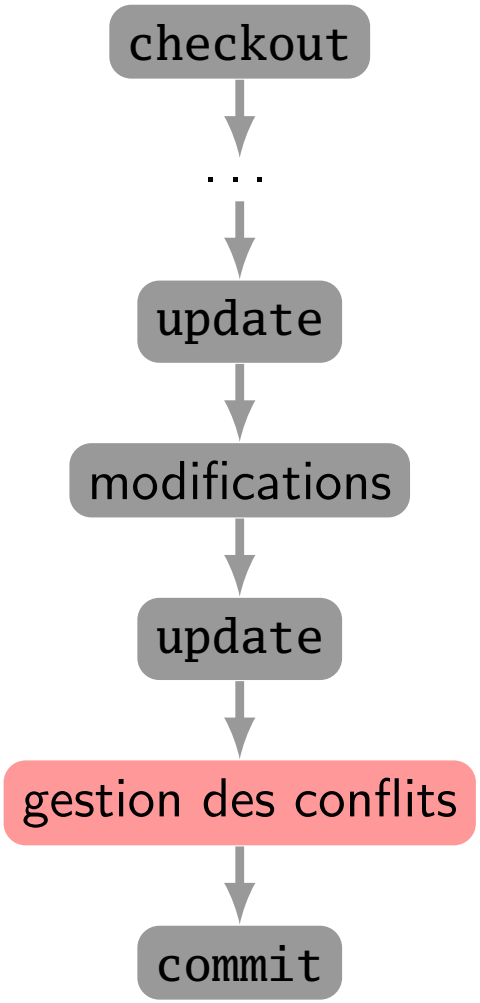
Vérifications

- ➥ le code compile
- ➥ les tests passent

Gérer les conflits

Nécessité de travailler avec les autres développeurs !

Comment utiliser subversion ?



Message

Mettre un message de `commit` explicite !
Décrire **ce qui** est concerné par le *commit*, pas **comment** on a fait les modifications (le diff/patch est là pour cela).

4 - Tests unitaires

Citations. . .

Program testing can be used to show the presence of bugs, but never to show their absence !

Edsger W. Dijkstra

I did not realize that the success of tests is that they test the programmer, not the program.

C.A.R. Hoare

Introduction

Le problème principal qui se pose en « informatique » est de pouvoir **vérifier**, voire **prouver**, que les développements que l'on fait sont **corrects** :

- au sens de ce qu'attend le client (**fonctionnalités**) ;
- que tout est **intégrable** (i.e. que les classes développées peuvent interagir entre elles) ;
- au niveau de la classe enfin (est-ce que les méthodes font bien ce que l'on veut ?).

Nous ne nous intéressons pas dans ce cours au deux premiers cas (pour l'instant) qui concernent les tests de recette, d'intégration et la validation. Nous cherchons donc à vérifier que les méthodes des classes développées ont un comportement correct.

Nous allons pour cela faire des tests unitaires avec le *framework* JUnit.

Exemple utilisé

Nous allons utiliser une classe `Point` minimale (pas de documentation en particulier !) qui possède une méthode incorrecte :

Point.java

```
public class Point {

    private double x;
    private double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() {
        return this.x;
    }

    public double getY() {
        return this.y;
    }
```

Exemple utilisé

Point.java

```
    public double getModule() {
        return Math.sqrt(Math.pow(this.x, 2) + Math.pow(this.y, 2));
    }

    public double getArgument() {
        if (this.getModule() == 0) {
            return 0;
        }

        double arg = Math.acos(this.x / this.getModule());
        if (this.y < 0) {
            return(-1 * arg);
        }
        return(arg);
    }

    public void translater(double dx, double dy) {
        this.x += dx;
        this.y += dx;
    }
```

Exemple utilisé

Point.java

```
    public void setModule(double module) {
        double ancienModule = getModule();
        if (ancienModule == 0.0) {
            this.x = module;
            }
        else {
            this.x = (this.x * module) / ancienModule;
            this.y = (this.y * module) / ancienModule;
        }
    }

    public void setArgument(double argument) {
        double module = this.getModule();
        this.x = module * Math.cos(argument);
        this.y = module * Math.sin(argument);
    }

    public double distance(Point p) {
        return Math.sqrt(Math.pow(this.x - p.x, 2) +
                         Math.pow(this.y - p.y, 2));
    }
```

Trouver l'erreur...

Exercice

Où est l'erreur dans la classe `Point` présentée précédemment ?

Plan de la partie 4 - Tests unitaires

Tester une classe

Jusqu'à présent, nous ne testions que nos méthodes via une classe possédant une méthode `main`. Par exemple :

TestPointBasique.java

```
public class TestPointBasique {
    public static void main(String[] args) {
        Point p = new Point (1,2);

        System.out.println("p.getX() = " + p.getX());
        System.out.println("p.getY() = " + p.getY());

        System.out.println("p.getArgument() = " + p.getArgument());
        System.out.println("p.getModule() = " + p.getModule());

        p.translater(3,6);
        System.out.println("p.getX() = " + p.getX());
        System.out.println("p.getY() = " + p.getY());
    }
}
```

Dans ce cas, c'est **à l'utilisateur** de vérifier que les résultats sont corrects !

Tester une classe : classe « avancée »

On peut créer une classe qui nous indique si un test échoue ou pas :

TestPointNormal.java

```
public class TestPointNormal {
    public static void main(String[] args) {
        Point p = new Point(1,2);

        verifier("getX()", 1.0, p.getX());
        verifier("getY()", 2.0, p.getY());

        verifier("getArgument()", 1.1071487177940904, p.getArgument());
        verifier("getModule()", 2.23606797749979, p.getModule());

        p.translater(3,6);
        verifier("translater(3,6), verification de Y", 8.0, p.getY());
    }

    private static void verifier(String message, double attendu,
                                 double reel) {
        String aff = ((attendu == reel) ? " OK !" : " erreur !");
        System.out.println("Test " + message + aff);
    }
}
```

Tester une classe : exécution et commentaires

L'exécution de cette classe nous donne le résultat suivant :

exécution de TestPointNormal

```
[tof@suntof]~ $ java TestPointNormal
Test getX() OK !
Test getY() OK !
Test getArgument() OK !
Test getModule() OK !
Test translater(3,6), verification de Y erreur !
```

Tester une classe : exécution et commentaires

Commentaires

- sans opérateur « humain », on ne peut pas savoir que le test a échoué ;
- en particulier, le programme se déroule normalement ;
- on ne connaît pas les conditions d'échec ;
- il faudrait pouvoir automatiser la collecte des résultats (comptage. . .).

Plan de la partie 4 - Tests unitaires

Principe des 3A

Comment tester ? Y-a-t'il une « procédure » classique de test ?

➥ on utilise les 3A : Acteur, Action et Assertion

3A

- l'**acteur** est un objet sur le lequel le test va porter. On parle également d'**OUT** (*Object Under Test*) ;
- l'**action** est une action qui consiste à appeler la méthode à tester sur l'acteur ;
- l'**assertion** est une **propriété** (vraie ou fausse) qui vérifie que l'action a bien eu l'effet escompté.

Par exemple, pour la méthode `translater` :

- on initialise un point de coordonnées $(1, 2)$;
- on le translate avec un vecteur $(3, 6)$;
- on vérifie que les nouvelles coordonnées du point sont $(4, 8)$.

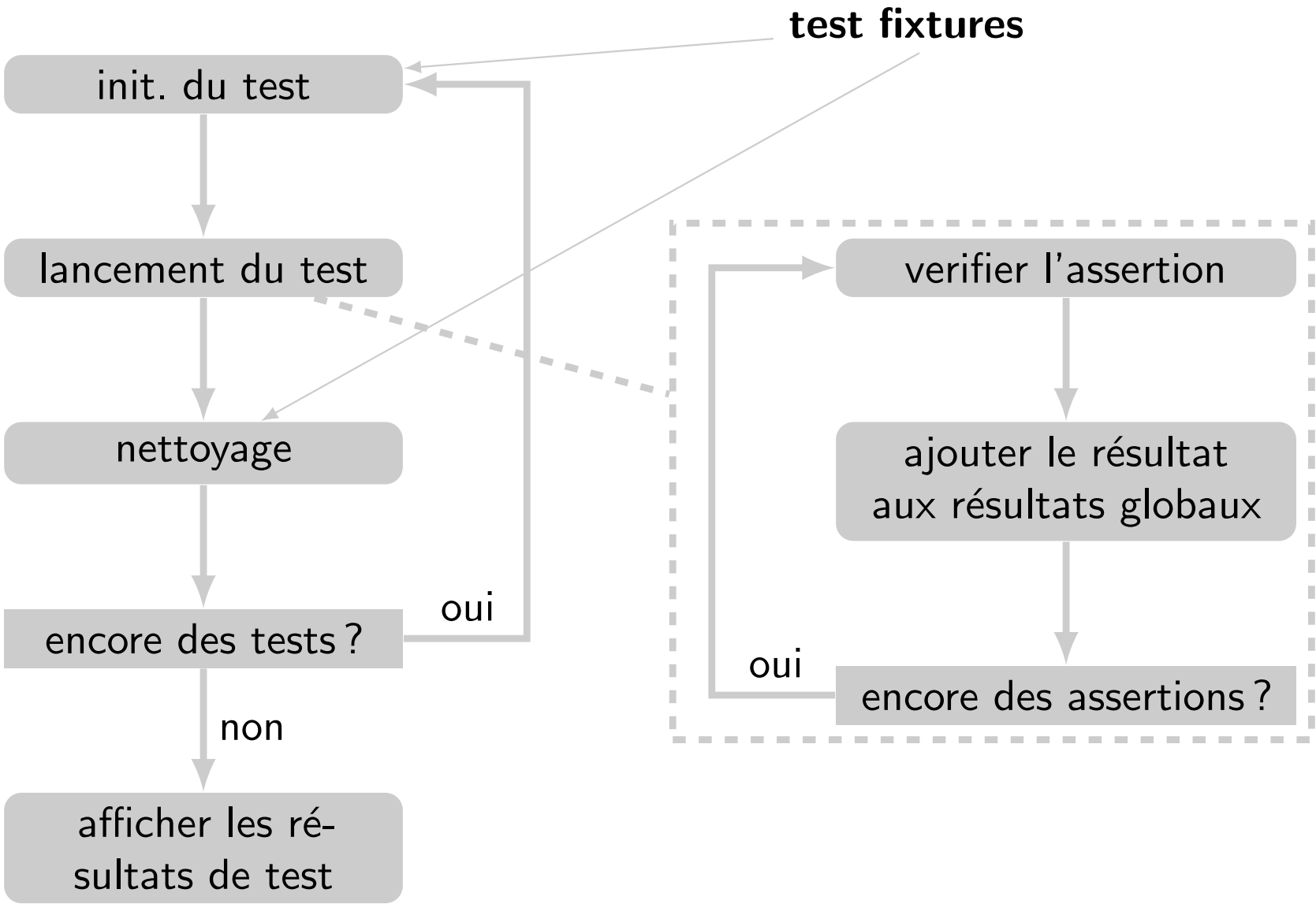
Aller vers un framework de test

On sait comment écrire « correctement » un test, mais :

- il faut pouvoir exécuter plusieurs tests avec une seule commande ;
- il faut pouvoir collecter les résultats de chaque test ;
- il ne faut pas que les tests s'interrompent si l'un d'entre eux est faux ;
- il faut pouvoir ajouter/modifier facilement de nouveaux tests.

Pour faire cela, il faut donc utiliser (ou créer) un **framework de test**, i.e. une application permettant de créer, lancer et vérifier des tests facilement.

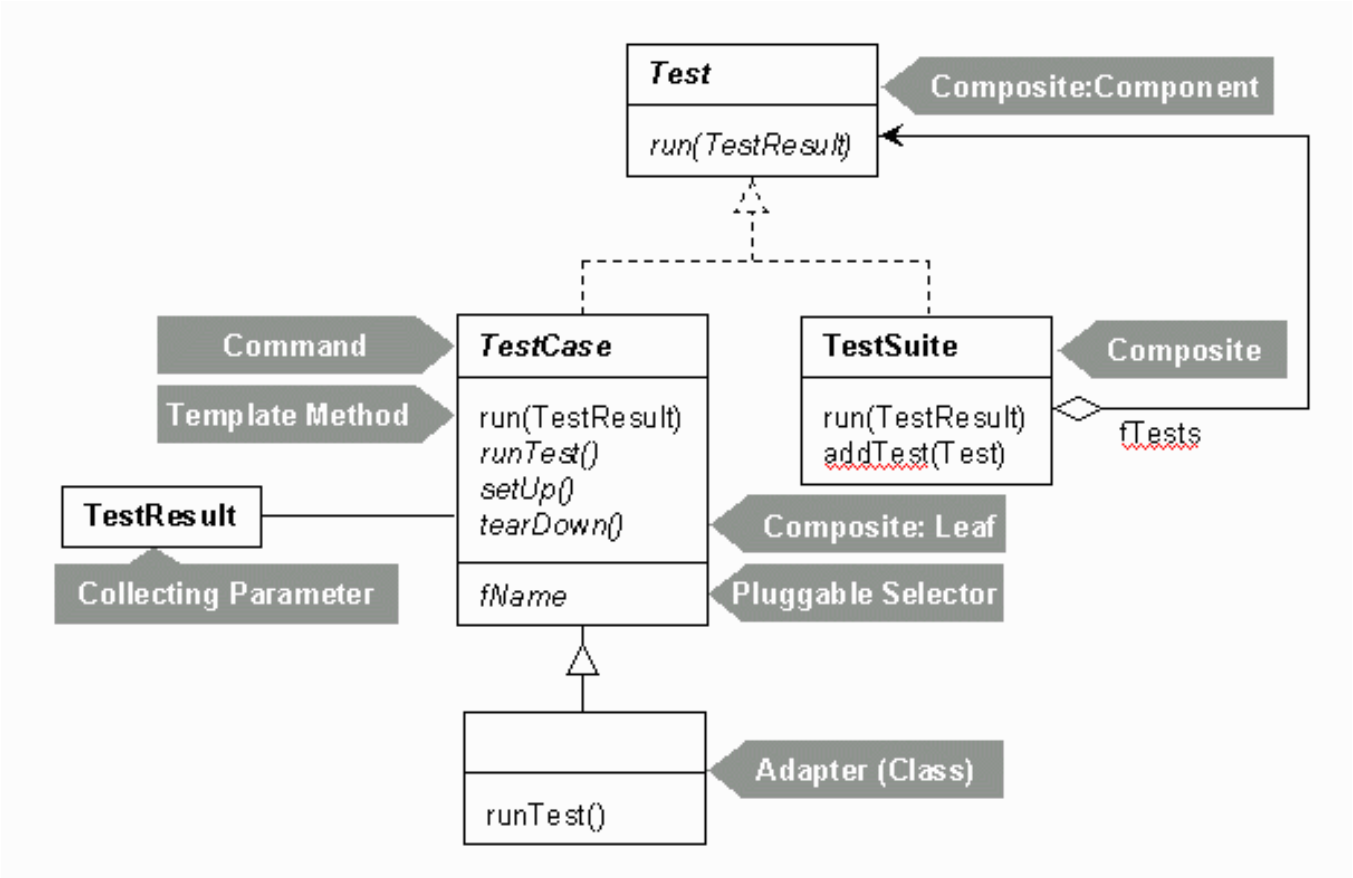
Cycle de vie d'un framework de test



Créer votre propre framework de test ?

Il est possible de créer son propre *framework* de test, mais cela est relativement compliqué (et permet aux professeurs de trouver rapidement des sujets d'examen ☺).

Par exemple, voici l'architecture de base de JUnit, le *framework* que nous allons utiliser (image prise sur `http://junit.org`) :



Plan de la partie 4 - Tests unitaires

Présentation de JUnit

JUnit est un *framework* permettant de réaliser des tests unitaires facilement avec une sortie en mode texte. Ce *framework* utilise les annotations de Java 5.0.

JUnit Team (2013).
JUnit.
http://www.junit.org.

Présentation de JUnit

JUnit est un *framework* permettant de réaliser des tests unitaires facilement avec une sortie en mode texte. Ce *framework* utilise les annotations de Java 5.0.

Pour chaque classe à tester, on construit une classe de test JUnit dont l'architecture est la suivante (exemple pour `PointTest.java`) :

PointTest.java

```
import org.junit.*;
import static org.junit.Assert.*;

public class PointTest {


}
```

JUnit : écrire un test

Lorsque l'on veut écrire une méthode de test, on écrit une méthode **obligatoirement** préfixée par le mot-clé **`@Test`**.
Toutes les méthodes préfixées par **`@Test`** seront utilisées dans la suite de tests que l'on définit dans la **classe de test**.
Chaque méthode de test devrait définir un cas de test simple.

Exemple :

PointTest.java

```
public class PointTest {

    @Test public void testTranslater() {
        Point p = new Point(1,2);

        p.translater(3,6);

        assertEquals(4.0, p.getX(), 0.0);
        assertEquals(8.0, p.getY(), 0.0);
    }
}
```

Comment écrire une assertion ?

Les assertions sont écrites au moyen des méthodes statiques de la classe `Assert` :

```
assertEquals(String expected, String actual)
assertEquals(double exp, double actual, double delta)
assertEquals(int expected, int actual)
assertFalse(boolean condition)
assertTrue(boolean condition)
assertNotNull(Object object)
assertNull(Object object)
assertSame(Object expected, Object actual)
assertNotSame(Object expected, Object actual)
...
```

Toutes ces méthodes sont **surchargées** et peuvent prendre en **premier** paramètre une chaîne de caractères qui sera affichée en cas d'erreur.

Voir la documentation sur le site et sur la carte de référence fournie.

JUnit : gérer les acteurs via les test fixtures

Les acteurs sont souvent utilisés dans plusieurs tests. Il est alors intéressant de les déclarer comme **attributs de la classe de test**, afin que les méthodes de test y aient accès.
La méthode préfixée par **@Before** permet d'initialiser les acteurs dont on aura besoin dans les tests.

PointTest.java

```
public class PointTest {

    private Point p;
    private static final double EPS = 10E-9;

    @Before public void setUp() {
        this.p = new Point(1,2);
    }


}
```

JUnit : gérer les acteurs via les test fixtures

Les acteurs sont souvent utilisés dans plusieurs tests. Il est alors intéressant de les déclarer comme **attributs de la classe de test**, afin que les méthodes de test y aient accès.
La méthode préfixée par **`@Before`** permet d'initialiser les acteurs dont on aura besoin dans les tests.

Remarque

La méthode préfixée par **`@Before`** est appelée avant **chaque** test.

Remarque

La méthode préfixée par **`@After`** est appelée après **chaque** test et permet de « nettoyer » éventuellement le test.

Une classe de test pour Point

Il manque des méthodes de test et la documentation !

PointTest.java

```
import org.junit.*;
import static org.junit.Assert.*;

public class PointTest {

    private Point p;
    private static final double EPS = 10E-9;

    @Before public void setUp() {
        this.p = new Point(1,2);
    }

    @Test public void testTranslaterBasic() {
        p.translater(3,6);

        assertEquals(4.0, p.getX(), EPS);
        assertEquals(8.0, p.getY(), EPS);
    }
```

Une classe de test pour Point

PointTest.java

```
    @Test public void testTranslaterNullVector() {
        p.translater(0,0);

        assertEquals(1.0, p.getX(), EPS);
        assertEquals(2.0, p.getY(), EPS);
    }

    @Test public void testTranslaterBack() {
        Point pold = this.p.clone();

        p.translater(2,3);
        p.translater(-2,-3);

        assertEquals(pold.getX(), p.getX(), EPS);
        assertEquals(pold.getY(), p.getY(), EPS);
    }
}
```

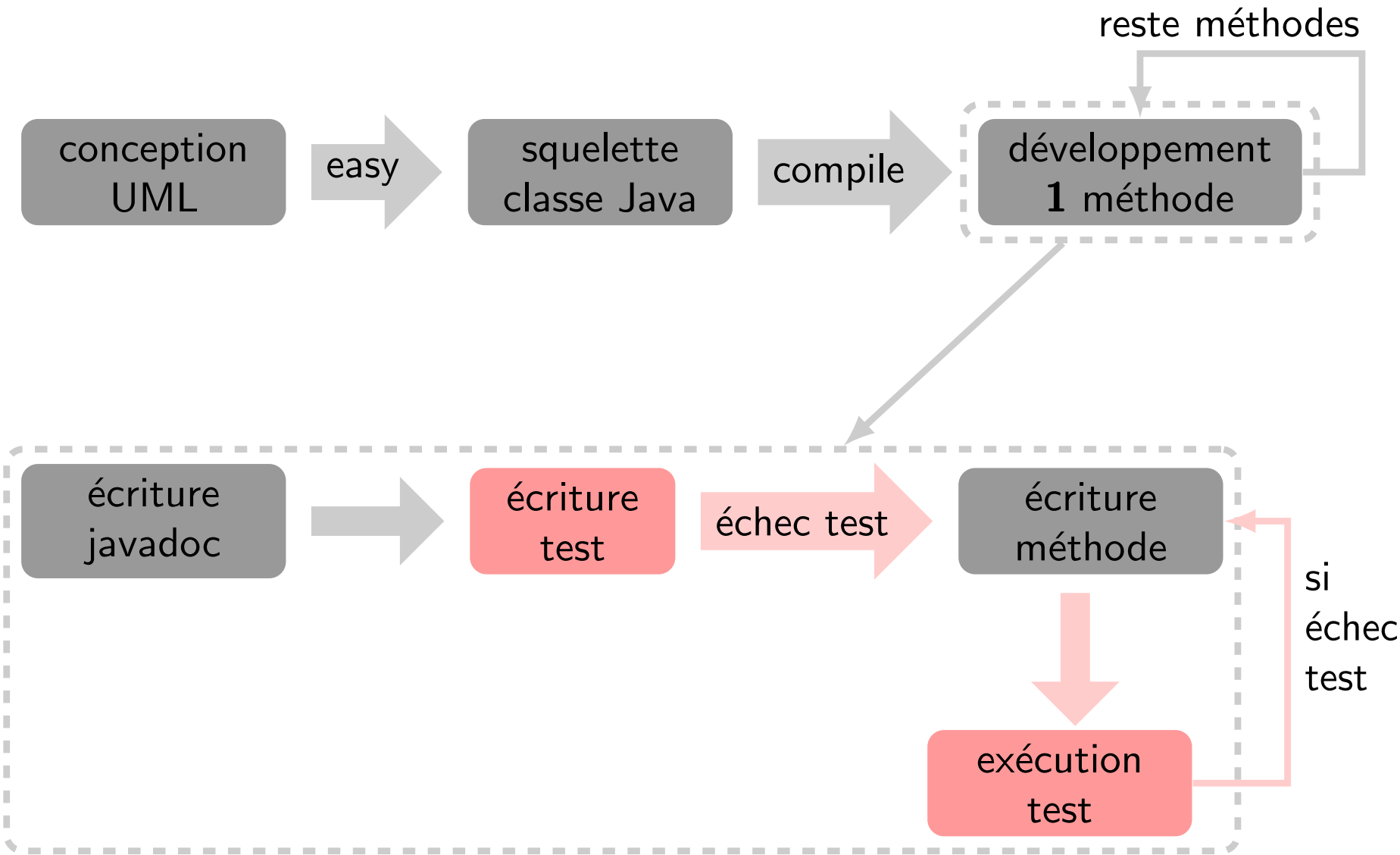
Lancement du test et résultat

Pour « exécuter » la classe de test `PointTest`, on utilise `java org.junit.runner.JUnitCore PointTest` et on obtient :

shell

```
[tof@suntof]~ $ java org.junit.runner.JUnitCore PointTest
JUnit version 4.11
.E..
Time: 0.006
There was 1 failure:
1) testTranslaterBasic(PointTest)
java.lang.AssertionError: expected:<8.0> but was:<5.0>
        at org.junit.Assert.fail(Assert.java:88)
        at org.junit.Assert.failNotEquals(Assert.java:743)
        at org.junit.Assert.assertEquals(Assert.java:494)
        at org.junit.Assert.assertEquals(Assert.java:592)
        ...
FAILURES!!!
Tests run: 3,  Failures: 1
```

Le process à utiliser...



Lost in space...

Exercice

Quels tests proposeriez-vous pour la méthode `distance` de `Point` ?

Exercice

Quels tests proposeriez-vous pour la classe `Orbite` ?

Quelques références

Hunt, A. et D. Thomas (2003).
Pragmatic Unit Testing in Java with Junit.
Pragmatic Programmers.
Pragmatic Bookshelf.

Beck, K. (2002).
Test driven development : by example.
Addison-Wesley Professional.

5 - Tableaux en Java

Les tableaux

Lorsque l'on veut utiliser un ensemble de données en Java, on peut utiliser des **tableaux**.

Les tableaux en Java ressemblent aux objets :

- ils sont accessibles par une poignée :

```
int[] tab1;     // tab1 est une poignee sur un tableau (tab1 == null)
int tab2[];     // on peut declarer des tableaux comme en C
Type[] tab;     // une poignee tab sur un tableau de Type
```

- ils sont créés dynamiquement en utilisant l'opérateur **new** :

```
tab1 = new int[5]; // creation d'un tableau de 5 entiers attache a tab1
tab = new Type[capacite];      // creation d'un tableau de capacite Type
```

Les tableaux

Lorsque l'on veut utiliser un ensemble de données en Java, on peut utiliser des **tableaux**.

Mais ce ne sont pas des objets :

- ils ont une syntaxe spécifique (les crochets) ;
- ils font partie des types génériques de Java (cf. cours sur la généricité)

Caractéristique des tableaux

Un tableau non créé ne peut pas être utilisé (`NullPointerException`).

La **capacité** du tableau peut être récupérée grâce à l'« attribut » `length` :

```
double tab[] = new double[10];
int nb = tab.length;              // nb == 10
```

L'accès à un élément se fait grâce à l'opérateur `[]`.

Les indices commencent à 0 et se terminent à `length - 1`.

En cas d'accès en dehors des limites du tableau, une exception de type `ArrayIndeOutOfBoundsException` est levée.

```
for (int i = 0; i < tab.length; i++) {
  tab[i] = i;
}
int p = tab[0];                 // premier element
int d = tab[tab.length - 1];    // dernier element
int e = tab[tab.length];        // ArrayIndexOutOfBoundsException
```

Caractéristiques des tableaux

Lorsqu'un tableau est créé avec **new**, chacun de ses éléments est initialisé avec la valeur par défaut de son type (**false**, 0, **null**).

Il est possible d'initialiser explicitement le tableau :

```
int[] nbPremiers = { 3, 5, 7, 11, 13};
int nb = nbPremiers.length;              // nb == 5
```

Les tableaux **ne peuvent pas être redimensionnés** (`length` ne peut pas changer de valeur).

On ne peut pas afficher directement un tableau avec `System.out.println` (sauf les tableaux de caractères).

Tableaux à plusieurs dimensions

Les tableaux à plusieurs dimensions sont en fait des tableaux de tableaux.

On peut les initialiser comme une « matrice » (i.e. les lignes ont toutes la même dimension) :

```
int[][] matrice = new int[2][3];
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
        matrice[i][j] = i+j;
    } // end of for ()
} // end of for ()
```

On peut manipuler une ligne par l'intermédiaire d'une poignée :

```
// permuter les deux premieres lignes
int[] ligne = matrice[0];
matrice[0] = matrice[1];
matrice[1] = ligne;
```

Tableaux à plusieurs dimensions

On peut également allouer de façon individuelle chacune des lignes :

```
// Le triangle de Pascal
// Creer le tableau de lignes
int[][] triangle = new int[10][];

// Construire la premiere ligne
triangle[0] = new int[2];
triangle[0][0] = triangle[0][1] = 1;

// Construire les autres lignes
for (int i = 1; i < triangle.length; i++) {
    // Creation de la i+1eme ligne
    triangle[i] = new int[i+2];
    triangle[i][0] = 1;
    for (int j = 1; j < triangle[i].length - 1; j++) {
        triangle[i][j] = triangle[i-1][j-1] + triangle[i-1][j];
    } // end of for (int j = 1; j < triangle[i].length - 1; j++)
    triangle[i][j+1] = 1;
} // end of for (int i = 1; i < triangle.length; i++)
```

Tableaux d'objets

On peut créer des tableaux d'objets.

Les éléments du tableau sont alors des poignées sur des objets du type précisé :

```
public class TableauEquations {
    public static void main (String[] args) {
        Equation[] system = new Equation[3];
        system[0] = new Equation(1, 5, 6);
        system[1] = new Equation(4, 4);
        for (int i = 0; i < system.length; i++) {
            system[i].resoudre();
            System.out.println("Equation " + (i+1) + " : ");
            System.out.println("x1 = " + system[i].x1);
            System.out.println("x2 = " + system[i].x2);
        } // end of for (int i = 0; i < system.length; i++)
        // attention, pour i == 2 on obtient :
        //      java.lang.NullPointerException

    } // end of main ()
}
```

6 - Associations

Plan de la partie 6 - Associations

Définition (dépendance)

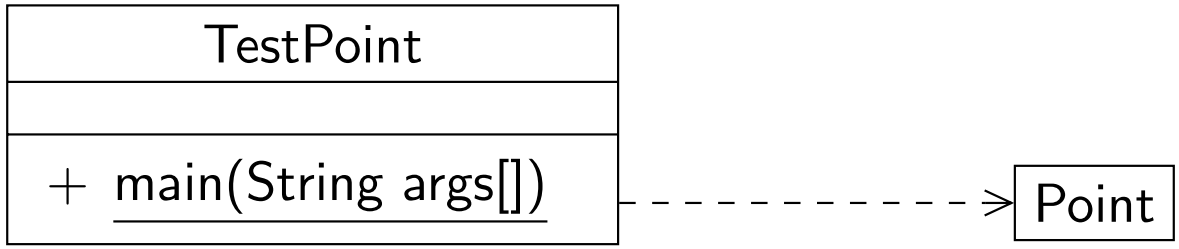
On dit qu'il y a une **relation de dépendance** entre une classe `A` et une classe `B` ssi la classe `A` fait référence à la classe `B` dans son texte.

Cette relation peut être momentanée si `B` apparaît comme :

- un paramètre d'une méthode de `A` ;
- une variable locale dans une méthode de `A` ;
- le retour d'une méthode de `A`.

Dépendance : représentation UML

On peut utiliser UML pour représenter cela (souvent peu utilisé, sauf dans le cas des exceptions) :



Si l'on modifie `Point` (surtout son interface, la signature d'une méthode...), on sera peut-être amené à modifier `TestPoint`.

Relations structurelles

Une relation est structurelle si elle dure, c'est généralement le cas quand `B` est le **type d'un attribut** de `A`.

- en Java cette relation correspond par défaut à un partage d'objet sauf si le programmeur réalise explicitement des copies ;
- en UML, on fait apparaître une relation particulière entre les classes :
  - association
  - agrégation (qui semble disparaitre dans les dernières version d'UML)
  - composition

Il existe une autre relation entre classes, très importante en conception et programmation objet, c'est la **spécialisation** (cf. cours suivants).

Plan de la partie 6 - Associations

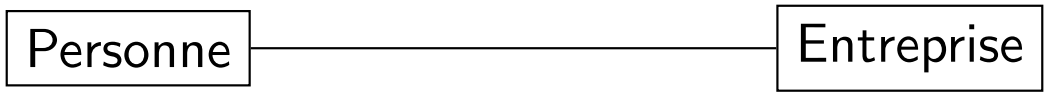
Plan de la partie 6 - Associations

Définition d'une association

Définition (association)

Une **association** représente une relation structurelle entre des classes.

Une association entre deux classes autorise la navigation des objets de l'une d'elle à des objets de l'autre.



Ici :

- on peut accéder à des objets de type `Entreprise` depuis un objet de type `Personne`
- on peut accéder à des objets de type `Personne` depuis un objet de type `Entreprise`

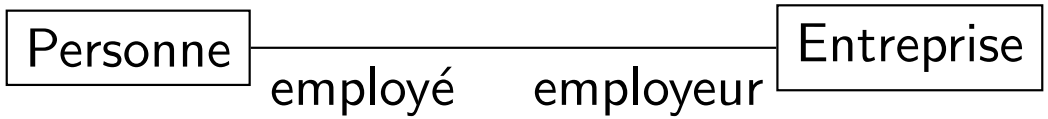
Associations : nom

Une association peut avoir un **nom** et on peut donner une **direction** à ce nom :

« Un objet de type `Personne` **travaille pour** des objets de type `Entreprise` »

Associations : rôles

Une classe participant à une association y joue un certain **rôle**. On peut le nommer explicitement.



Dans l'association liant des objets de type `Personne` à des objets de type `Entreprise` :

- un objet de type `Personne` apparaissant dans l'association joue le rôle « employé »
- un objet de type `Entreprise` apparaissant dans l'association joue le rôle « employeur »

Représentation en extension d'une association

On pourrait représenter en extension une association par un tableau (attention, ce n'est pas une notation UML !). Considérons un ensemble d'objets de type Personne et un ensemble d'objets de type Entreprise :

Personne	**Entreprise**
c.garion	ISAE
p.siron	ISAE
p.siron	ONERA
g.casalis	ONERA
j.doe	
	fausse_entreprise

On voit que certaines personnes sont liées à plusieurs entreprises, que certaines entreprises sont liées à plusieurs personnes et que certaines personnes ou entreprises ne sont liées à aucune entreprise ou personne.

Multiplicités

Il peut être important de préciser combien d'objets peuvent être reliés par une instance d'association.

Pour cela, on utilise une **multiplicité** qui précise ce nombre.

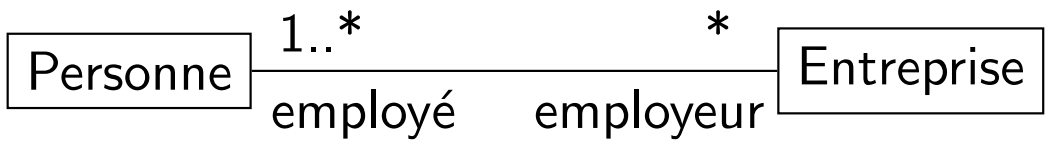
UML	Signification
1	un et un seul
0..1	zéro ou un
M..N	de M à N (entiers naturels)
*	de zéro à plusieurs
0..*	de zéro à plusieurs
1..*	de un à plusieurs

Intuition

La multiplicité associée à une classe représente pour chaque instance de cette classe le nombre d'occurences possibles de cette instance dans la table représentant l'association.

Multiplicités : exemples

- une entreprise a au moins un employé et une personne peut travailler pour plusieurs entreprises ou ne pas travailler :

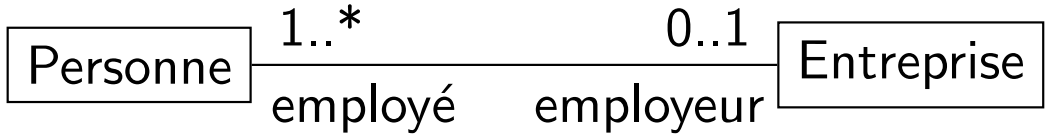


Dans ce cas :

Personne	**Entreprise**
c.garion	ISAE
p.siron	ISAE
p.siron	ONERA
g.casalis	ONERA
j.doe	
	fausse_entreprise

Multiplicités : exemples

- une entreprise a au moins un employé et une personne ne peut travailler que pour une seule entreprise ou ne pas travailler :



Dans ce cas :

Personne	**Entreprise**
c.garion	ISAE
p.siron	ISAE
p.siron	ONERA
g.casalis	ONERA
j.doe	
	fausse_entreprise

La classe Segment

Exercice

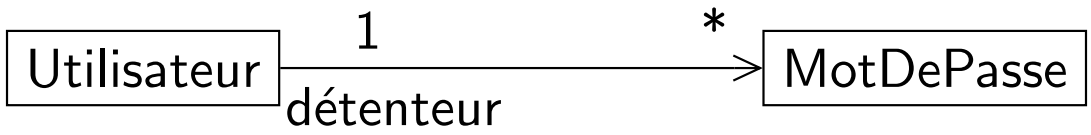
Considérons une classe `Segment`. Pour cela on utilise la classe `Point` déjà développée.

Entre la classe `Segment` et la classe `Point` il existe une **relation structurelle forte** : un objet de type `Segment` **est toujours lié** à deux objets de type `Point`, ses extrémités.

Proposer un diagramme UML faisant intervenir la classe `Segment` et la classe `Point` et la ou les associations liant les classes.

Navigabilité d'une association

On peut ajouter une **navigabilité** pour une association (par défaut elle est bidirectionnelle) :



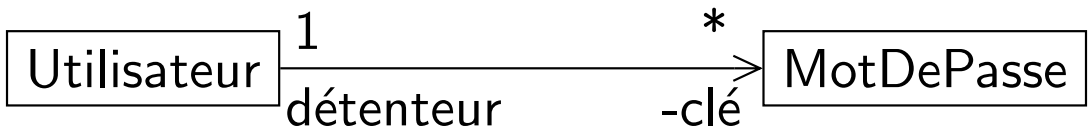
Pour chaque `Utilisateur`, il faut trouver les objets `MotDePasse` correspondants afin de vérifier que le mot de passe entré est le bon.

À partir d'un objet `MotDePasse`, on ne devrait pas pouvoir identifier l'`Utilisateur`.

La navigabilité permet de préciser qu'il est facile de passer d'un objet à un autre (souvent par le biais d'une référence).

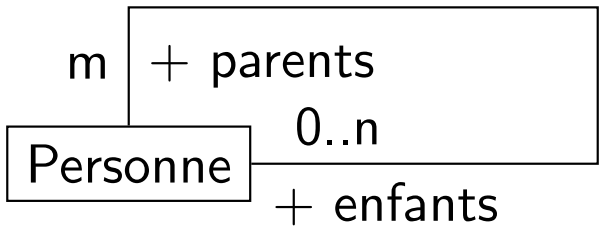
Visibilité des rôles

On peut appliquer une visibilité pour interdire l'accès à des objets d'une classe à l'extérieur de l'association :



Ici, il ne faut évidemment pas que quelqu'un puisse directement trouver un mot de passe à partir d'un utilisateur connu.

Une association peut être réflexive :



Dans ce cas, les **noms de rôles** sont très importants. On verra que grâce aux interfaces, on peut typer ces rôles.

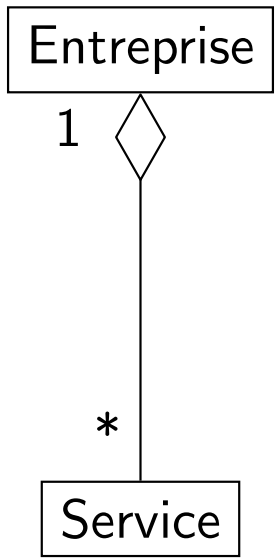
Agrégation

Dans une association « classique », les deux classes ont le même niveau d'importance.

L'**agrégation** est une association particulière dans laquelle une des classes joue le rôle de **tout** et l'autre de **partie**.

Elle représente une association « possède ».

Elle est purement conceptuelle : elle a toutes les caractéristiques d'une association.



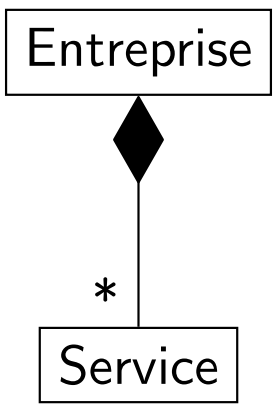
Composition

L'agrégation ne change pas la navigabilité de l'association entre le tout et ses parties.

Elle ne lie pas non plus la durée de vie du tout et de ses parties.

La **composition** est une agrégation forte :

- les parties vivent et meurent avec le tout ;
- un objet ne peut faire partie que d'une seule composition ;
- propriété forte.



Si l'entreprise est « détruite », les services qui la composent disparaissent.

Une classe `OrbiteDiscrete`

Exercice

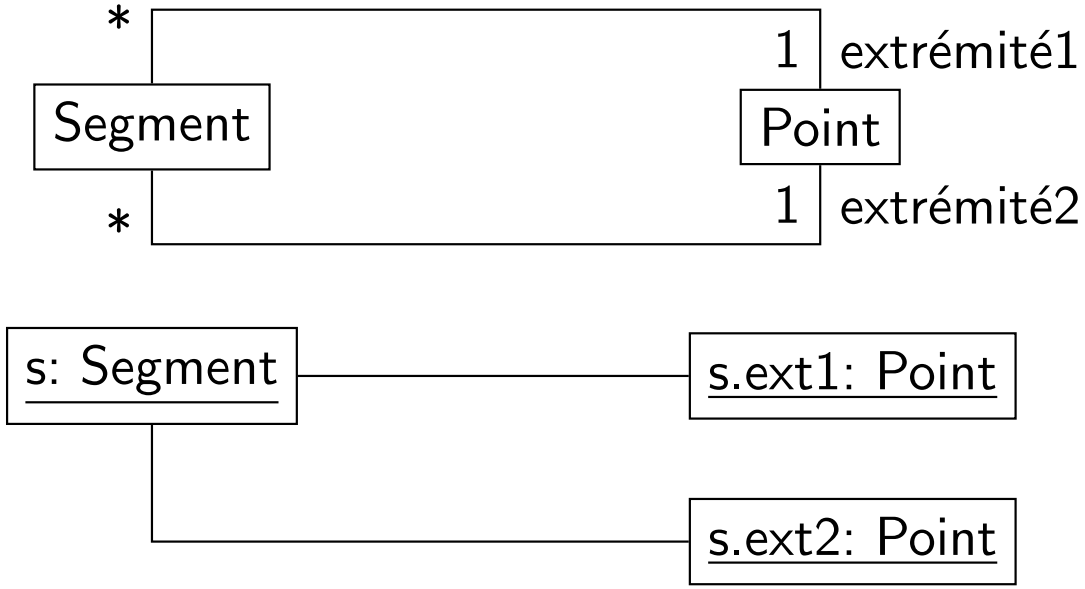
On souhaite maintenant construire une classe `OrbiteDiscrete` :

- une instance de `OrbiteDiscrete` est un ensemble de points
- on calcule l'ensemble des points appartenant à une instance de `OrbiteDiscrete` en utilisant la classe `Orbite` précédemment développée
- on peut translater une instance de `OrbiteDiscrete`
- on peut appliquer une homotéthie sur une instance de `OrbiteDiscrete`
- on peut calculer le périmètre du polygone défini par les points contenu dans l'instance de `OrbiteDiscrete`

Proposer un diagramme UML précisant les relations existant entre la classe `OrbiteDiscrete`, la classe `Point` et la classe `Orbite`. On fera apparaître les méthodes importantes de la classe `OrbiteDiscrete`.

Liens entre objets

De la même façon que les classes s'instancient en objets, les associations entre classes s'instancient en **liens** entre objets.

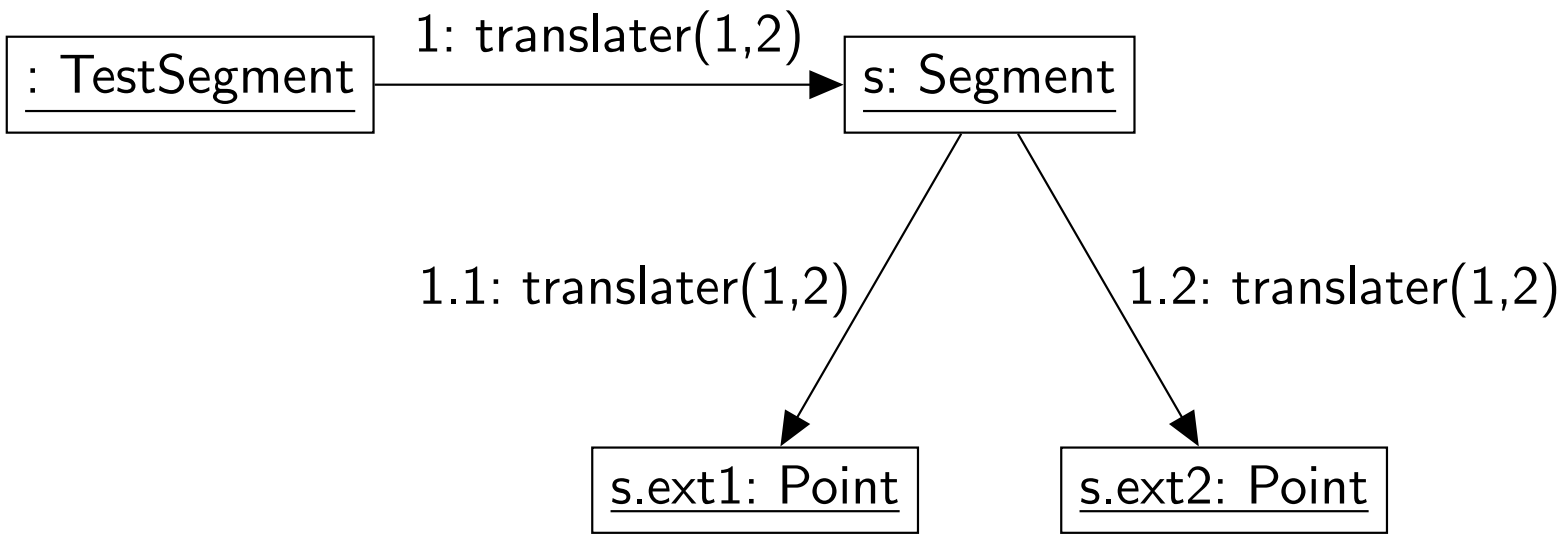


Grâce à ces liens, les objets peuvent appeler des méthodes sur d'autres objets. On parle également souvent d'**envoi de message** entre les objets. Ces liens et ces envois de messages peuvent être mis en évidence sur des **diagrammes de communication**.

Diagrammes de communication

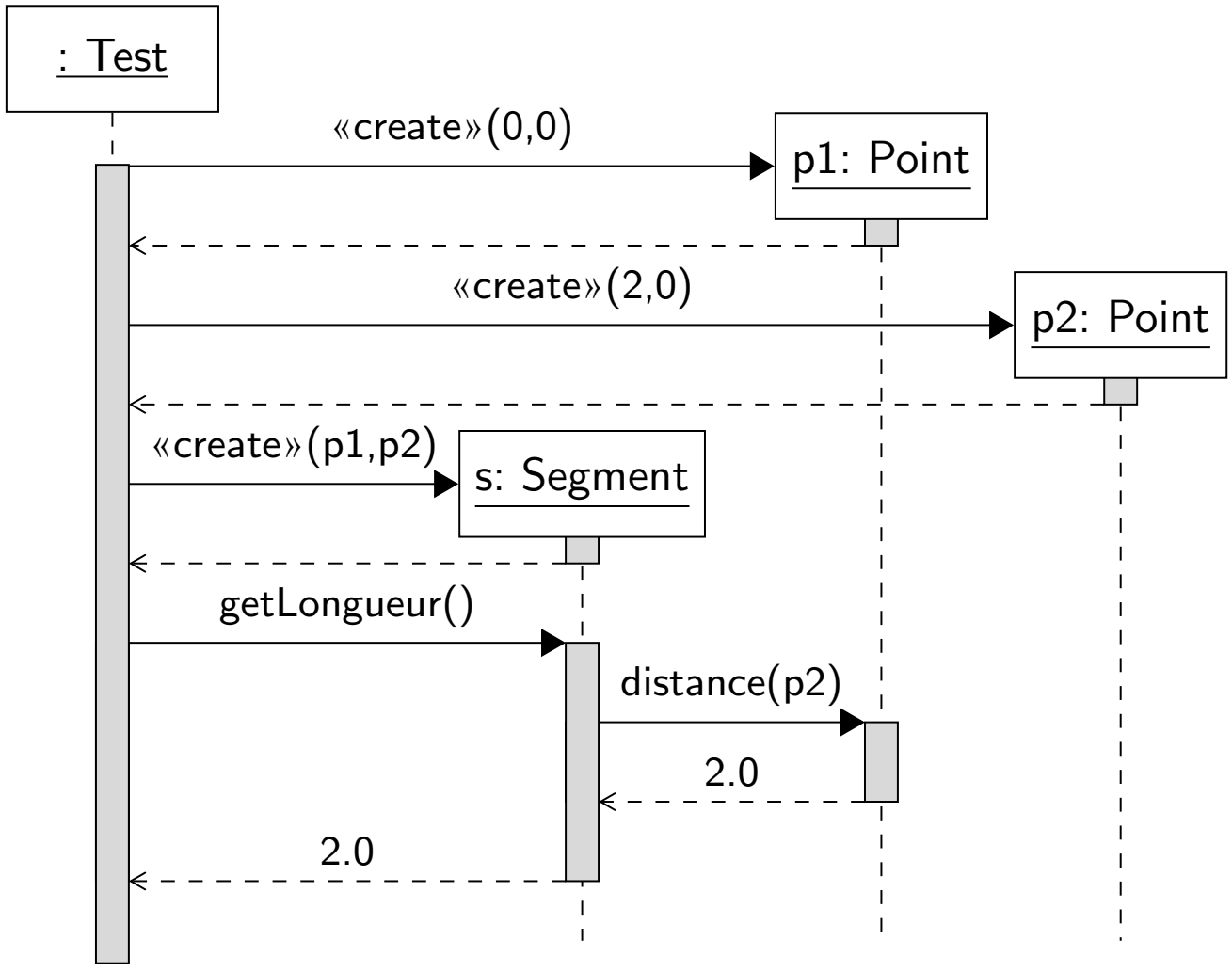
Les **diagrammes de communication** permettent de représenter les liens existant entre les objets et les messages qui s'échangent entre ces objets.

On numérote les messages pour suivre la chronologie des envois de messages.



Diagrammes de séquence

Les **diagrammes de séquence** sont un autre type de diagramme permettant de suivre la chronologie des envois de messages entre objets.



Appel de `translater` sur un segment

Exercice

On suppose que les associations entre `Segment` et `Point` sont celles définies précédemment. On suppose également que la classe `Segment` possède une méthode `translater(double dx, double dy)`.

Représenter sur un diagramme de séquence les appels de méthodes effectués lors d'un appel à `translater` sur un segment créé à partir des points $(0, 0)$ et $(2, 0)$.

Appel de `translater` sur `OrbiteDiscrete`

Exercice

Peut-on facilement représenter par un diagramme de séquence un appel à `translater` sur une instance de `OrbiteDiscrete` contenant 500 points ?

Diagrammes de séquence ou de communication ?

On peut se demander quand choisir d'utiliser les diagrammes de séquence ou de communication :

- on peut représenter des choses très compliquées avec les diagrammes de séquence (contrôle de flot etc.) ;
- les diagrammes de séquence permettent de mettre l'accent sur les **séquences d'appels** ;
- les diagrammes de communication sont plus simples et permettent de mettre l'accent sur les **liens entre objets** ;
- il s'agit avant tout d'une question d'habitude et de goût !
- on privilégiera dans le cours les diagrammes de séquence.

Fowler, M. (2003).
UML distilled.
Addison-Wesley.

Plan de la partie 6 - Associations

Implantation des associations en Java

Les langages de programmation n'ont pas la richesse sémantique de langages comme UML. En Java, les associations entre classes se traduisent par l'introduction d'attributs dans les classes :


```
public class Segment {

    private Point extremite1;
    private Point extremite2;

    ...
}
```

```
public class Point {

    // il n'y a pas d'attributs
    // de type Segment dans la
    // classe Point
    ...
}
```

Les **noms de rôles** servent de **noms d'attributs** dans la classe.

Segment composé avec Point

Exercice

On suppose maintenant que les associations liant la classe `Segment` et la classe `Point` sont des compositions :

Quels impacts cette modélisation a-t-elle sur le code du constructeur de `Segment` ? Sur le code des accesseurs et modifieurs des attributs de `Segment` ?

Agrégation et composition en Java

Comment traduire des relations comme la composition et l'agrégation en Java ?

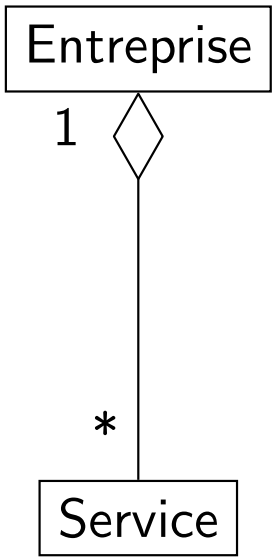
On ne dispose que d'attributs dont la sémantique est assez faible...

- cas d'un attribut de type primitif : il s'agit d'une composition, car la valeur de l'attribut « appartient » bien à l'objet et disparaitra avec lui ;
- cas d'un attribut instance d'un objet. Plusieurs problèmes se posent :
  - comment garantir la destruction des composants en même temps que le composé ?
    - ➥ ce n'est pas possible de le coder (ramasse-miettes), mais cela peut-être implicite si on n'a pas d'autre accès possible à l'attribut
  - à la création de l'agrégat, comment doit-on affecter les composants ?
    - ➥ il faut les copier (en particulier pour la composition)
  - comment copier un composé dans ce cas ?
    - ➥ il faut également copier les parties en recréant des objets (utilisation de la méthode `clone()`)

Coder les associations à multiplicité *

Lorsque l'on une association avec une multiplicité supérieure à 1, comment la traduire en Java ?

Exemple :




```
public class Service {

  private Entreprise ent;
  ...
}
```

Utilisation d'une collection

Code de la classe `Entreprise` :

- utilisation d'un tableau d'objets de type `Service` ?
- les tableaux ne sont pas faciles à manipuler : initialisation, taille fixe etc.
- en particulier, comment écrire « proprement » une méthode `ajouterService` qui permet d'ajouter un service dans l'entreprise ?

Pour implanter une association en Java, on peut utiliser une **collection** : il s'agit d'une classe représentant un ensemble d'objets.

Depuis Java 5.0, les collections sont **génériques** : elles sont paramétrées par un type. On peut ainsi déclarer et créer un ensemble d'un type particulier (un ensemble de services par exemple).

Les collections apportent beaucoup plus de souplesse que les tableaux via des opérations de haut niveau (cf. transparent suivant). De plus, on peut « augmenter » leur taille dynamiquement.

Nous reviendrons sur les collections et les types génériques dans un prochain cours.

La classe java.util.ArrayList<E>

Nous allons utiliser ici la classe `java.util.ArrayList` qui représente une **liste d'objets**.

Le nom qualifié de cette classe est `java.util.ArrayList<E>`. `E` représente le type des objets que l'on veut stocker dans la liste.

Syntaxe (déclaration et création d'un objet de type ArrayList)

```
ArrayList<TypeObjet> nom = new ArrayList<TypeObjet>();
```

Exemple pour une liste d'objets de type `Service` :

```
ArrayList<Service> services = new ArrayList<Service>();
```

Quelques méthodes utiles de `ArrayList`

Méthodes disponibles pour un objet de type `ArrayList<Service>` :

- `add(Service s)` : pour ajouter un objet à la fin de la liste
- `add(int i, Service s)` : pour **insérer** un objet à la position `i`
- `set(int i, Service s)` : pour remplacer un objet à la position `i`
- `contains(Service s)` : pour vérifier qu'un objet est dans la liste
- `get(int i)` : renvoie l'objet situé à la position `i`
- `isEmpty()` : la liste est-elle vide ?
- `remove(Service s)` et `remove(int i)` : enlever un objet de la liste
- `size()` : le nombre d'éléments de la liste

La classe java.util.ArrayList<E>

Pour parcourir les éléments d'une liste, on peut utiliser la structure **for**.

Par exemple, pour afficher sur la console les services d'une entreprise :

```
public void afficherServices() {
  for (Service s : this.services) {
    System.out.println(s);
  }
}
```

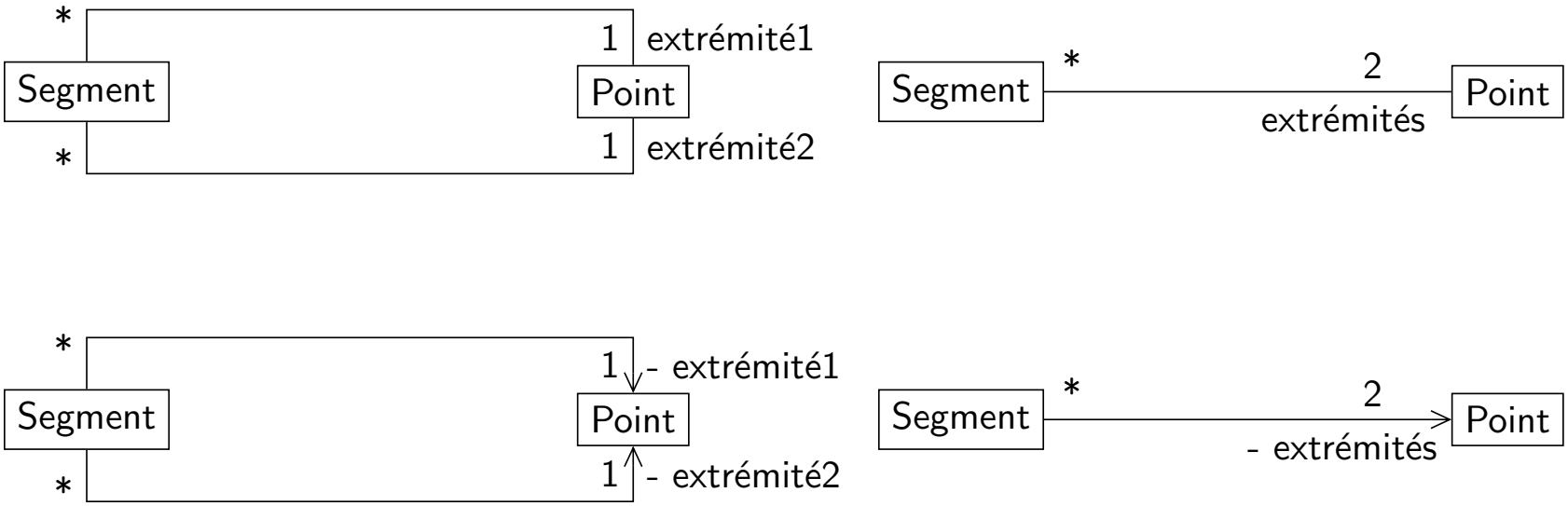
Si l'on veut utiliser la liste comme type de retour d'une méthode ou comme paramètre, on utilisera la version « instanciée » de la liste :

```
public ArrayList<Service> getServices() {
  return this.services;
}
```

Plan de la partie 6 - Associations

Niveaux de représentation en UML

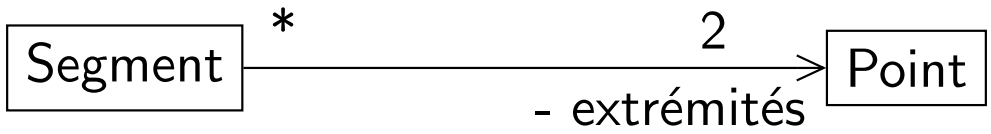
On peut utiliser UML pour modéliser le problème suivant différents niveaux de conception :

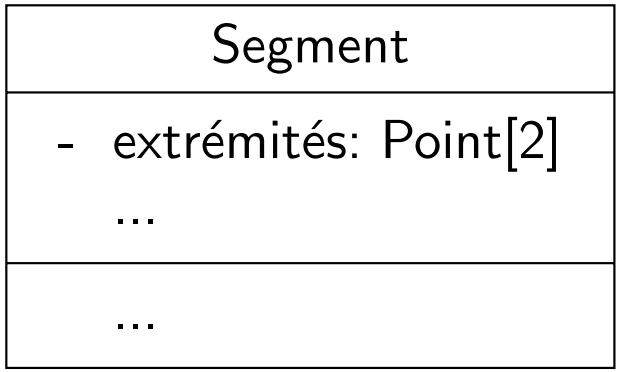


- la première ligne présente des modèles d'analyse
- la seconde ligne présente des modèles dérivés des premiers qui sont plus proches de l'implantation

Niveaux de représentation avec UML

On peut utiliser UML pour représenter des choix d'implantation :



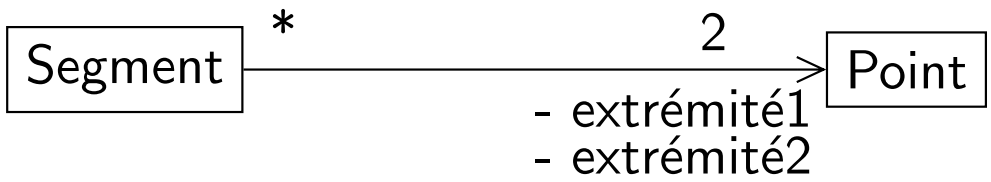


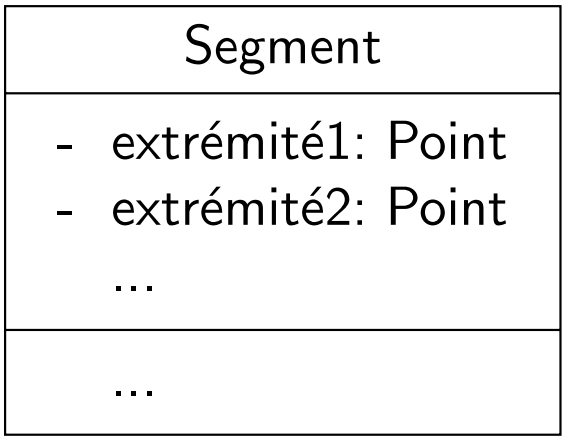
Remarque

Attention, en UML Point[2] ne signifie pas « un tableau de 2 instances de Point », mais « un ensemble de deux instances de Point » (qui peut être implanté par un tableau...).

Niveaux d'information avec UML

Avec UML, on peut choisir de représenter ou non le détail des classes :





Remarque

Ne pas se leurrer : toutes les informations sont présentes dans le premier diagramme !

Implantation de la classe `OrbiteDiscrete`

Exercice

Proposer un diagramme UML **d'implantation** précisant les relations existant entre la classe `OrbiteDiscrete`, la classe `Point`, la classe `ArrayList` et la classe `Orbite`. On fera apparaître les méthodes importantes de la classe `OrbiteDiscrete`.

Un système de réservation de vols

Exercice (merci à X. Crégut)

On s'intéresse à un système simplifié de réservation de vols pour une agence de voyage. L'interview des experts métiers a permis de résumer leurs connaissance du domaine dans les paragraphes suivants.

1. des compagnies aériennes proposent différents vols.
2. un vol a un aéroport de départ et un aéroport d'arrivée.
3. un vol a un jour et une heure de départ, un jour et une heure d'arrivée.
4. un vol peut comporter des escales dans des aéroports.
5. une escale a une heure d'arrivée et une heure de départ.
6. chaque aéroport dessert une ou plusieurs villes.
7. un vol est ouvert à la réservation et refermé sur ordre de la compagnie.
8. une réservation concerne un seul vol et un seul passager.
9. un client peut réserver un ou plusieurs vols, pour des passagers différents.

7 - Paquetages

Plan de la partie 7 - Paquetages

Paquetages : définition

En Java, on organise les classes d'un programme en **paquetages** pour regrouper par exemple des classes travaillant sur le même domaine (classes « mathématiques », classes de présentation etc,...).

Syntaxe (appartenance d'une classe à un paquetage)

```
package mon.paquetage; // paquetage d'appartenance

class A { ... }        // texte de la classe
```

Attention, **package** doit être la **première instruction** du code source.

Si on ne précise pas de paquetage pour une classe, cette classe appartient au **paquetage anonyme** (le répertoire courant, cf. plus loin).

Le nom des paquetages est écrit en minuscules et est souvent normalisé : `com.sun.java`.
Les paquetages sont organisés suivant une hiérarchie définie par `.` : `java.lang.reflect` est un sous-paquetage de `java.lang`.

Visibilité des classes

Nous avons vu qu'il existait un mécanisme de visibilité à quatre niveaux pour les caractéristiques d'une classe.

Les paquetages offrent **deux** niveaux de visibilité pour les classes :

- publique, la classe est alors accessible depuis n'importe quel paquetage ;
- de paquetage, la classe n'est accessible **que depuis son paquetage**.

Syntaxe (visibilité de classe)

```
public class MaClasse { ... }

class MaClasse { ... }
```

Utiliser une classe d'un paquetage

Il existe plusieurs façons d'utiliser une classe appartenant à un paquetage :

1. utilisation du nom complet de la classe (**nom qualifié** de la classe)

Entreprise.java

```
public class Entreprise {

    private java.util.ArrayList<Service> services;

    public Entreprise() {
        this.services = new java.util.ArrayList<Service> ();
    }

    ...
}
```

Ici, `java.util.ArrayList` signifie « la classe `ArrayList` du paquetage `java.util` »

Utiliser une classe d'un paquetage

Il existe plusieurs façons d'utiliser une classe appartenant à un paquetage :

2. **importation** de la classe (on peut alors utiliser son **nom court** dans le reste de la classe) :

Entreprise.java

```
import java.util.ArrayList;
...

public class Entreprise {

    // il s'agit de la classe ArrayList du paquetage java.util
    private ArrayList<Service> services;

    public Entreprise() {
        this.services = new ArrayList<Service> ();
    }

    ...
}
```

Utiliser une classe d'un paquetage

Il existe plusieurs façons d'utiliser une classe appartenant à un paquetage :

2. **importation** de la classe (on peut alors utiliser son **nom court** dans le reste de la classe) :

Remarque

Attention, le fait d'importer une classe n'a rien à voir avec le fait que `javac` ou `java` trouveront ou non le *bytecode* de la classe.
`import` ne sert qu'à permettre l'utilisation des noms courts !

Utiliser une classe d'un paquetage

Il existe plusieurs façons d'utiliser une classe appartenant à un paquetage :

3. **importation** du contenu entier d'un paquetage (pas recommandé) :

Entreprise.java

```
import java.util.*;
...

public class Entreprise {

    private ArrayList<Service> services;

    public Entreprise() {
        this.services = new ArrayList<Service> ();
        Iterator it = this.services.iterator();
        // il s'agit bien de la classe Iterator du paquetage java.util
    }

    ...
}
```

Remarques sur les noms des classes

1. En cas de conflit entre deux noms de classes (deux paquetages peuvent avoir des classes de même nom court), il faut utiliser les noms qualifiés.
   Par exemple, on peut utiliser deux classes `Vector` :
   - `java.util.Vector`
   - `org.apache.commons.math3.geometry.Vector`
2. les classes du paquetage auquel appartient la classe courante sont automatiquement importées : une classe peut donc utiliser le nom court de toutes les classes de son paquetage.
3. `java.lang.*` est importé par défaut (classes `Math`, `String` etc.).

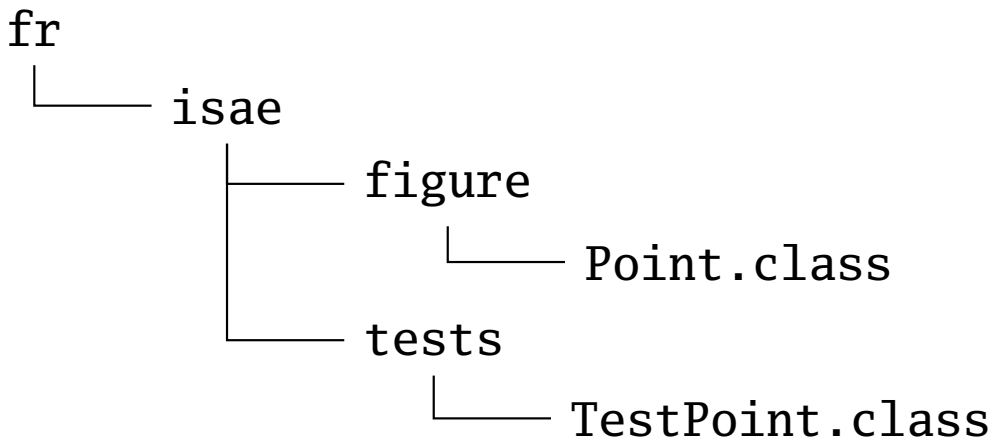
Plan de la partie 7 - Paquetages

Plan de la partie 7 - Paquetages

Correspondance avec le système de fichiers

La structure des paquetages s'appuie sur le système de fichiers : les paquetages sont des répertoires et les classes des fichiers.

Ex : `fr.isae.figure.Point` et `fr.isae.tests.TestPoint` correspondent dans la hiérarchie de fichiers à :

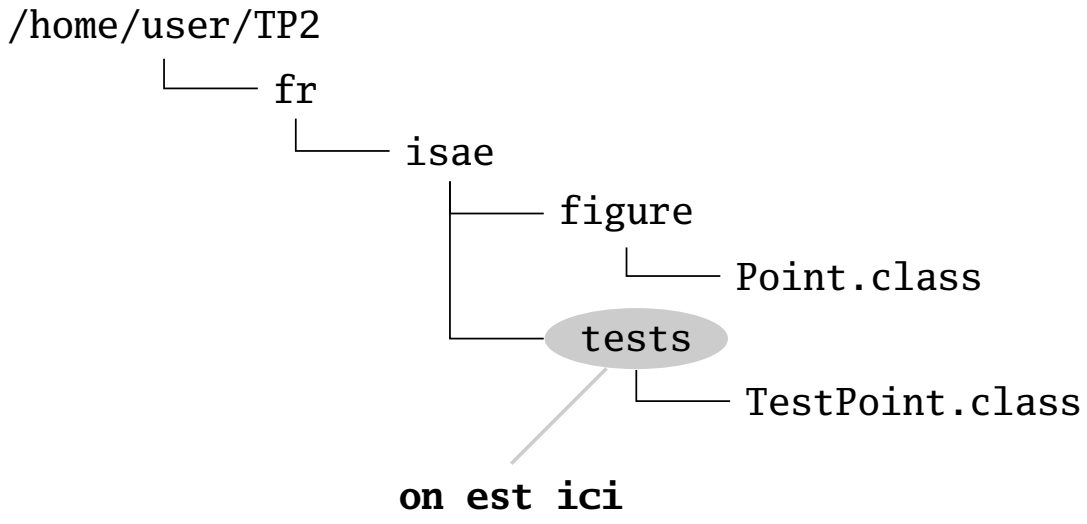


Attention

À la compilation ou à l'exécution, `javac` ou `java` doivent trouver le début de la hiérarchie (ex : le répertoire contenant le répertoire `fr`).

Correspondance avec le système de fichiers

Supposons que l'on soit dans la situation suivante :



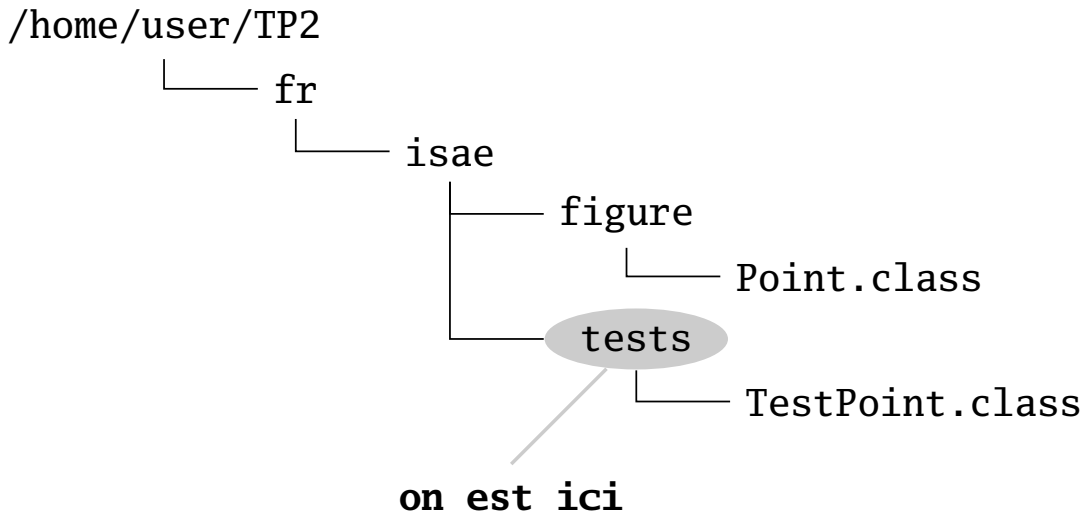
Dans ce cas, la commande suivante ne fonctionne pas (**on ne peut pas utiliser de noms courts avec la commande java**) :

shell

```
[tof@suntof] tests/ $ java TestPoint
Exception in thread "main" java.lang.NoClassDefFoundError:
 TestPoint (wrong name: fr/isae/tests/TestPoint)
```

Correspondance avec le système de fichiers

Supposons que l'on soit dans la situation suivante :



Dans ce cas, la commande suivante ne fonctionne pas (on devrait alors avoir une hiérarchie de fichiers `fr/isae/tests/TestPoint.class` sous `/home/user/fr/isae/tests`) :

shell

```
[tof@suntof] tests/ $ java fr.isae.tests.TestPoint
Error:
 Could not find or load main class fr.isae.tests.TestPoint
```

Le CLASSPATH

Pour résoudre cela, il faut préciser à `java` où trouver les *bytecodes* dont il peut avoir besoin. C'est le rôle de la variable d'environnement CLASSPATH.

Deux solutions s'offrent à nous :

1. en ligne de commande avec l'option `-cp` :

shell

```
[tof@suntof] tests/ $ java -cp /home/user/TP2 fr.isae.tests.TestPoint
```

2. en positionnant directement la variable d'environnement `CLASSPATH` :

shell

```
[tof@suntof] ~ $ export CLASSPATH=$CLASSPATH:/home/user/TP2
```

Plan de la partie 7 - Paquetages

Les archives JAR

On peut regrouper des *bytecodes* Java dans une archive sous un format spécial, JAR (Java ARchive).

Pour **créer** une archive de `Point` et `TestPoint` par exemple :

shell

```
[tof@suntof] ~ $ cd /home/user
[tof@suntof] ~ $ jar cvf point.jar isae/**/*
```

Pour **afficher** le contenu d'une archive :

shell

```
[tof@suntof] ~ $ jar tvf point.jar
```

Les archives JAR

On peut préciser quelle est la classe qui contient la méthode `main` dans un fichier `manifest.txt` pour pouvoir« exécuter » un JAR :

shell

```
[tof@suntof] ~ $ cd /home/user
[tof@suntof] ~ $ jar cvfm manifest.txt point.jar isae/**/*
[tof@suntof] ~ $ java -jar point.jar
```

avec le contenu de `manifest.txt` suivant (ne pas oublier le retour à la ligne) :

manifest.txt

```
Main-Class: fr.isae.test.TestPoint
```

Les archives JAR

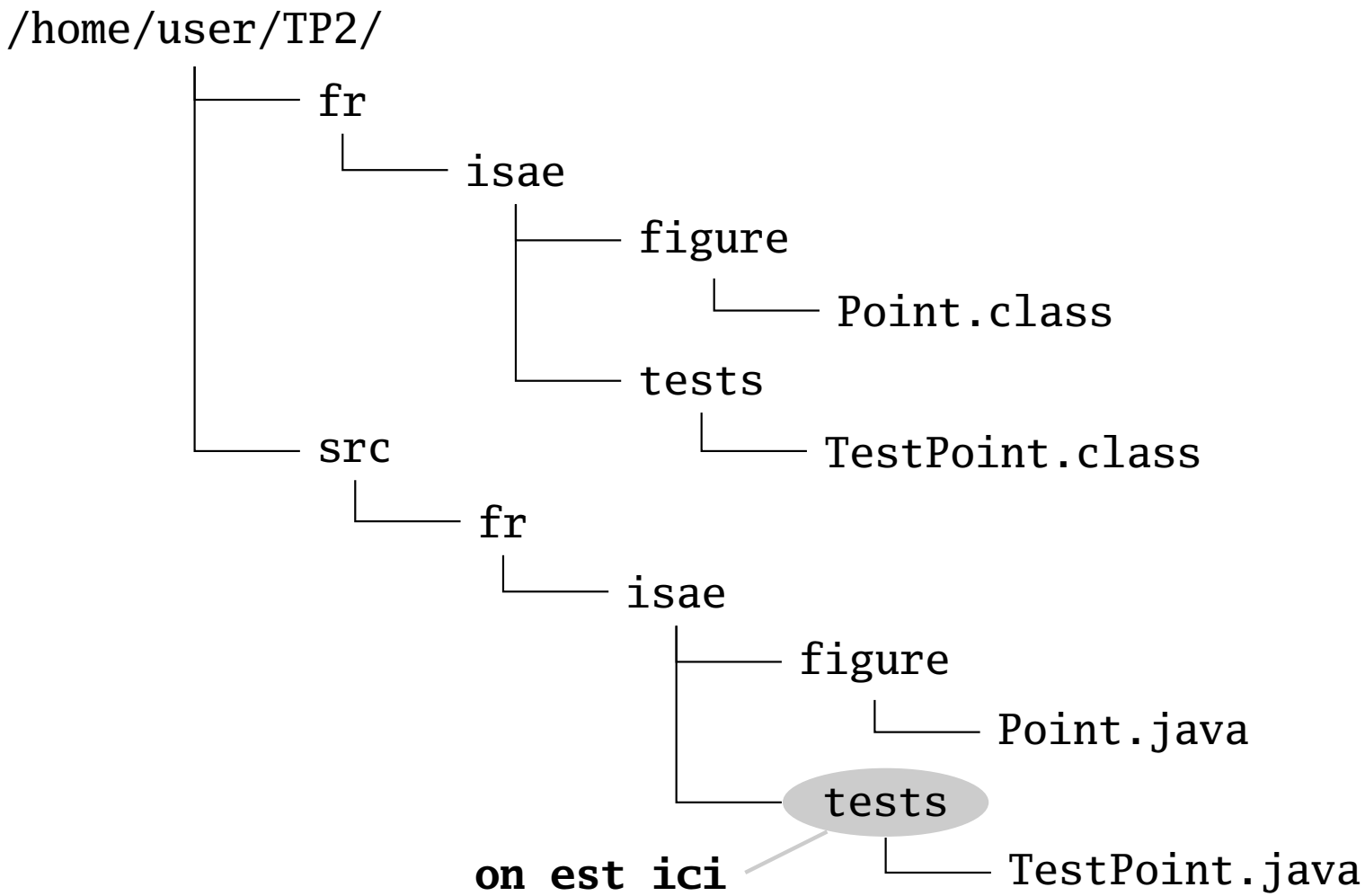
Attention

Si on utilise un JAR (archive Java contenant des bytecodes), il faut inclure **le fichier** `.jar` dans le `CLASSPATH`.

Plan de la partie 7 - Paquetages

Et si on rajoute les sources. . .

Supposons maintenant que l'on respecte également une hiérarchie pour les sources :



Et si on rajoute les sources : solution

On a besoin du *bytecode* de `Point` pour compiler `TestPoint` :

- soit on le trouve ;
- soit on trouve le source de `Point` et on le compile ;
- soit on trouve les deux et on « utilise » le plus récent.

Une solution pour compiler « proprement » :

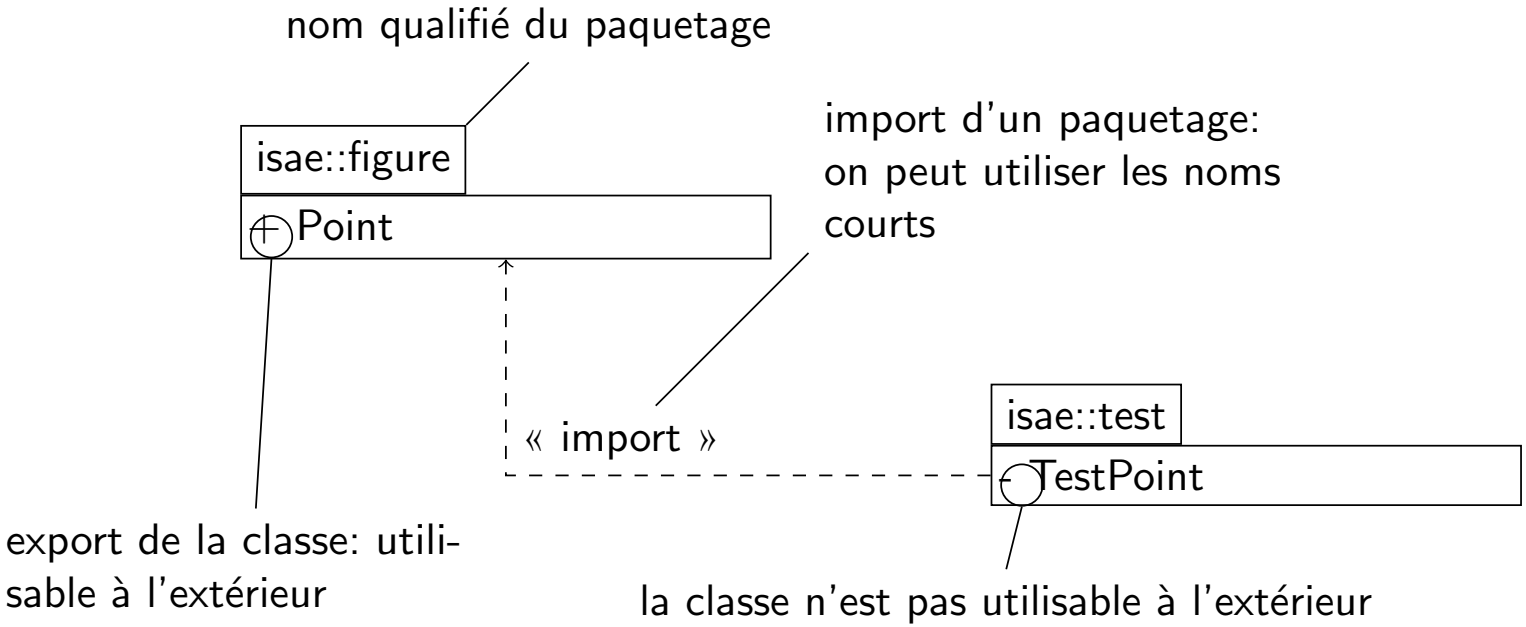
shell

```
javac -d /home/user/TP2
      -cp /home/user/TP2
      -sourcepath /home/user/TP2/src/fr/isae/figure:.
      TestPoint.java
```

- `-d /home/user/TP2` : on place le *bytecode* dans `/home/user/TP2`
- `-cp /home/user/TP2/` : on cherche le *bytecode* dans `/home/user`
- `-sourcepath /home/user/src/fr/isae/figure:.` : on cherche les sources dans `/home/user/src/fr/isae/figure` et dans le répertoire courant (représenté par `.`)

Plan de la partie 7 - Paquetages

Représentation en UML



Quelques remarques :

- le « séparateur » est ici ;
- la relation d'importation entre paquetages n'est pas transitive ;
- il existe d'autres façons de représenter l'imbrication des paquetages, cf. polycopié.

Intérêts des paquetages

Intérêts

- structurer l'application en regroupant ses constituants ;
- éviter les conflits de noms : un même nom de classe peut être utilisé dans deux paquetages différents (un paquetage définit un espace de nommage) ;
- fournir une visibilité.

Privilégier les noms qualifiés pour éviter les ambiguïtés.

Les paquetages sont très importants pour structurer correctement une application (ex : classes métiers, classes utilitaires, classes de présentation, GUI etc.).

8 - Interfaces

Introduction : deux classes « Point »

Que penser des deux classes suivantes :

PointCartesien
- x: double - y: double
+ PointCartesien(x: double, y: double) + translater(dx: double, dy: double) + distance(p: PointCartesien): double + getX(): double + getY(): double

PointPolaire
- module: double - argument: double
+ PointPolaire(m: double, a: double) + translater(dx: double, dy: double) + distance(p : PointPolaire): double + getModule(): double + getArgument(): double

Introduction : deux classes « Point »

`PointCartesien` et `PointPolaire` sont deux **implantations** d'un même concept : un point situé dans le plan.

En particulier, ces deux classes offrent des **services communs** à un utilisateur : `translater` et `distance`.

Problèmes

- comment calculer la distance entre un point modélisé avec des coordonnées cartésiennes et un point modélisé avec des coordonnées polaires ?
  ➥ la signature de `distance` l'interdit !
- comment construire un tableau ou une instance de `ArrayList` comportant à la fois des instances de `PointCartesien` et des instances de `PointPolaire` pour les translater ?
  ➥ le typage des tableaux et de `ArrayList` l'interdit !

Introduction : deux classes « Point »

`PointCartesien` et `PointPolaire` sont deux **implantations** d'un même concept : un point situé dans le plan.

En particulier, ces deux classes offrent des **services communs** à un utilisateur : `translater` et `distance`.

Solution ?

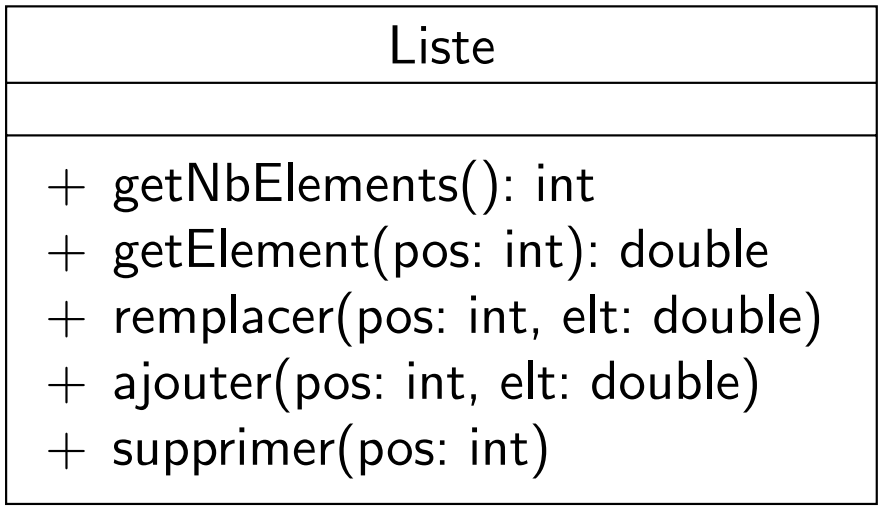
Il faudrait pouvoir déclarer les services communs à ces deux classes...

Introduction : une classe « Liste »

On considère une classe `Liste` qui représente une liste de réels et qui fournit les opérations suivantes :

- connaître le nombre d'éléments de la liste ;
- obtenir l'élément situé à la position i dans la liste ;
- remplacer le $i$ème élément de la liste ;
- ajouter un élément en position i ;
- retirer l'élément en position i.

1. on peut construire le diagramme UML de la classe `Liste` :



Introduction : une classe « Liste »

2. on peut également écrire une méthode qui calcule la somme des réels contenus dans une liste :

```
public class OutilListe {

    public static double somme(Liste l) {
        double somme = 0;
        for (int i = 0; i < l.getNbElements(); i++) {
            somme += l.getElement(i);
        }

        return somme;
    }
}
```

3. cette méthode devra rester la même quelle que soit la façon dont on implante la liste (tableau, liste chaînée,. . .).

Introduction : une classe « Liste »

Quelques constatations

- on est capable de définir une liste : on peut donner la signature de chacune des méthodes, on peut écrire un commentaire Javadoc...
- on pourrait écrire les spécifications complètes des méthodes de la classe ;
- on ne sait pas où et comment sont stockés les éléments ;
- on ne sait pas écrire le code des méthodes ;
- ce n'est donc pas une classe...

Solution

La solution à ces problèmes est d'utiliser une **interface**.

Plan de la partie 8 - Interfaces

Définition

Un utilisateur extérieur d'une classe a rarement besoin de savoir comment cette classe a été effectivement codée. Pour l'utiliser, il doit juste connaître quels sont les services (ou les opérations) qui peuvent être rendus par les instances de cette classe. On parle alors de l'**interface** d'une classe.

Définition (Interface)

Une interface est la limite entre la **spécification** des actions d'une abstraction et **comment** cette abstraction les exécute.

L'interface représente donc la limite entre vue externe et interne d'une abstraction.

Intérêts des interfaces

Intérêts

- elles permettent un **niveau de compréhension plus aisé** (on est « détaché » de l'implantation) ;
- il existe une **relation de réalisation** entre classes et interfaces ;
- elles **définissent un comportement** qui doit être respecté par ses réalisations (on doit donc correctement documenter ses méthodes !) ;
- elles **définissent un contrat** :
  - les implanteurs de la classe doivent le respecter ;
  - les utilisateurs savent comment utiliser l'interface.
- elles permettent de **diversifier l'implantation** ;
- à un moindre niveau, les interfaces permettent de faire un semblant d'« héritage » multiple (cf. cours sur l'héritage).

Plan de la partie 8 - Interfaces

Représentation en UML

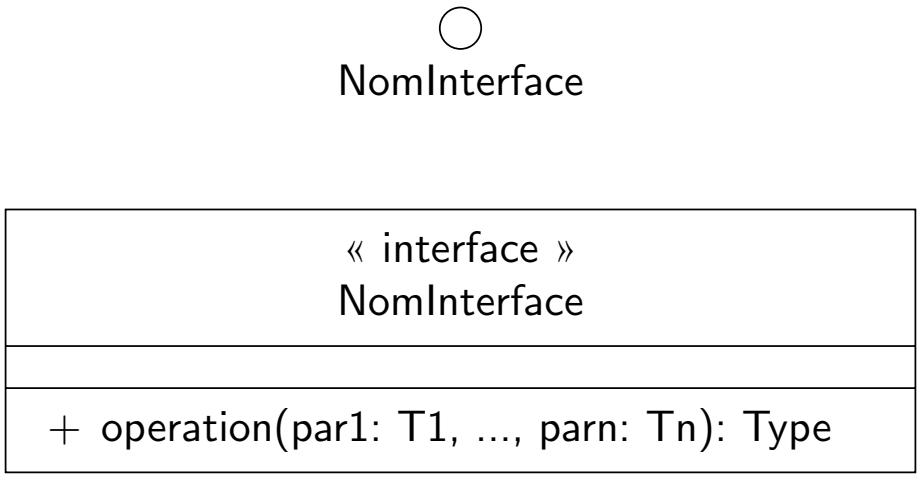
Définition (interface en UML)

Les interfaces sont un ensemble nommé d'opérations publiques.

Ces opérations représentent des **services** qui peuvent être rendus par des objets (d'où le fait qu'elles soient publiques !).

Représentation en UML

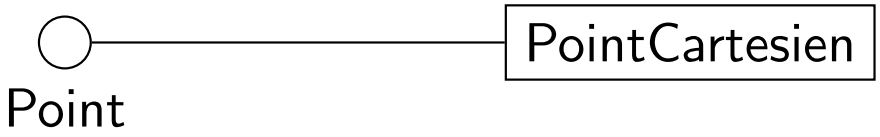
Il y a deux façons de représenter une interface (une simple et une détaillée) :

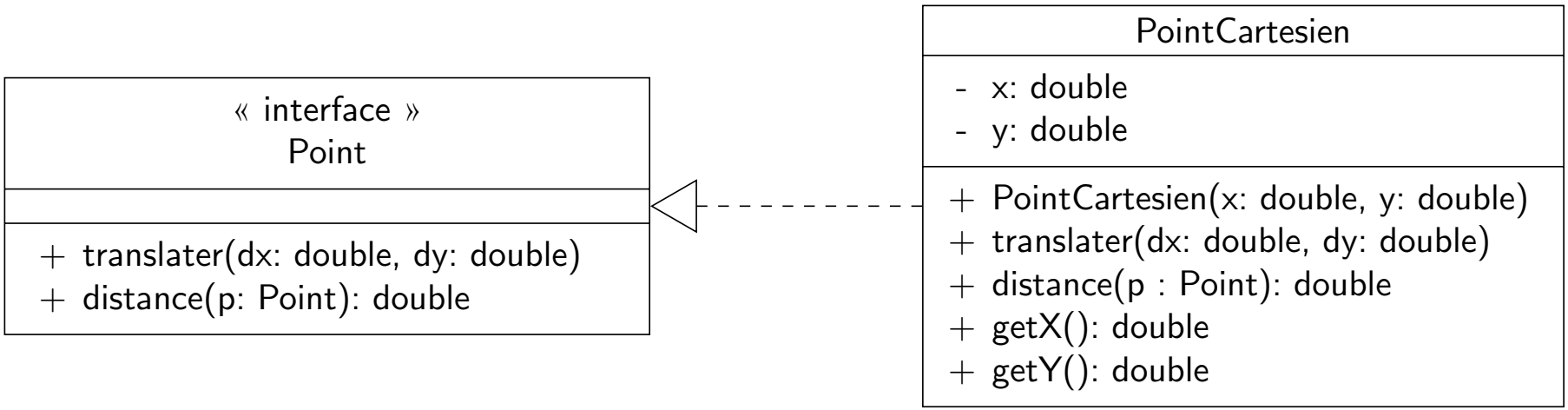


Relations entre classes et interfaces

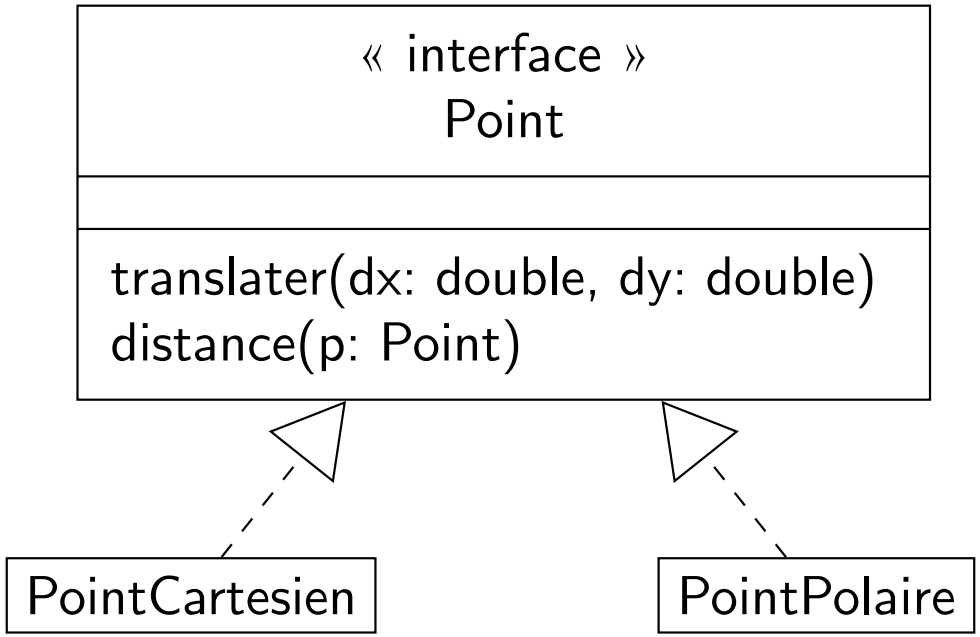
Une classe peut s'engager à **réaliser** le contrat défini par une interface : elle **définit** donc les méthodes publiques de l'interface.

On dit alors que la classe **réalise** l'interface.





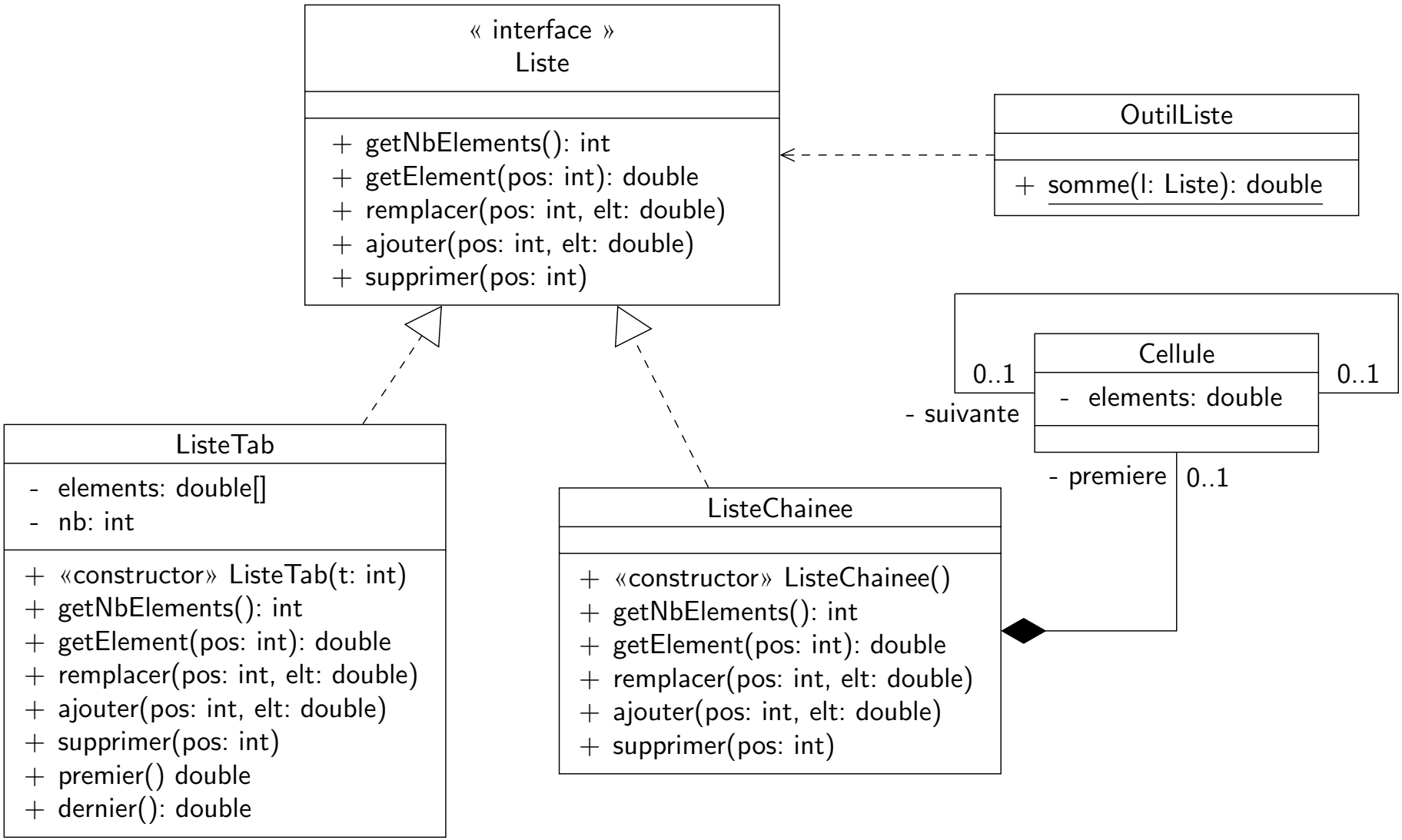
Exercice : implantation de `distance`



Question

Avec les implantations présentées précédemment de `PointCartesien` et de `PointPolaire`, peut-on implanter la méthode `distance` dans chacune de ces classes ?

Exemple avec `Liste`



Plan de la partie 8 - Interfaces

Représentation en Java

Liste.java

```java
/** Specification d'une liste */

public interface Liste {
    /**
     * Obtenir la taille de la liste.
     *
     * @return la taille de la liste
     */
    int getNbElements();

    /**
     * Obtenir un element de la liste.
     *
     * @param pos la position de l'element dans la liste
     * @return l'element
     */
    double getElement(int pos);
```

Représentation en Java

Liste.java

```java
/**
 * Remplacer un element de la liste.
 *
 * @param pos la position de l'element a remplacer
 * @param elt la nouvelle valeur
 */
void remplacer(int pos, double elt);

/**
 * Inserer un element dans la liste.
 *
 * @param pos la position de l'element a inserer
 * @param elt la valeur a inserer
 */
void ajouter(int pos, double elt);
```

Représentation en Java

Liste.java

```
    /**
     * Retirer un element de la liste.
     *
     * @param pos la position de l'element a inserer
     */
    void supprimer(int pos);
}
```

Remarques sur l'implantation en Java

Le droit d'accès des attributs et des méthodes est nécessairement et implicitement **public**.

➥ normal : quel intérêt à **spécifier** quelque chose d'inaccessible ?

En Java, une interface contient uniquement des **déclarations de méthodes** (sans leur code) : **seules la spécification et la signature des méthodes sont données**.

➥ c'est juste une **spécification**

Il n'y a pas d'attributs, sauf éventuellement des constantes :

```
static final double PI = 3.14;
```

On ne peut y déclarer un constructeur.

➥ quel en serait l'intérêt ?

Instances d'une interface ?

Peut-on créer une instance d'une interface ?

Non, car une interface définit un comportement, c'est une abstraction. Elle ne possède pas de code.

Par exemple, quelle serait la signification de ce code :

TestListeInstance.java

```
public class TestListeInstance {
    public static void main(String[] args) {
        Liste l = new Liste();
        l.ajouter(0, 25.6);   // qu'est-ce que cela signifie ?
    }
}
```

Instances d'une interface ?

Peut-on créer une instance d'une interface ?

Non, car une interface définit un comportement, c'est une abstraction. Elle ne possède pas de code.

On obtient le message suivant à la **compilation** :

shell

```
[tof@suntof]~ $ javac TestListeInstance.java
TestListeInstance.java:3: error: Liste is abstract; cannot be instantiated
        Liste l = new Liste();
                  ^
1 error
```

Instances d'une interface ?

Peut-on créer une instance d'une interface ?

Non, car une interface définit un comportement, c'est une abstraction. Elle ne possède pas de code.

Conclusion

Il faut donc créer des classes qui **réalisent** l'interface.

Relations entre classes et interfaces

Une classe peut accepter le contrat proposé par une interface. On dit que la classe **réalise** l'interface.

Définition (réalisation)

On appelle réalisation d'une interface une classe qui s'engage à implanter toutes les méthodes définies dans l'interface.

En Java, la relation de réalisation se code via le mot clé **`implements`** :

Syntaxe (réalisation)

```
public class C implements I1, I2, ... {
  ...
}
```

Exemples de réalisations

Exemple avec `ListeTab` :

ListeTab.java

```
public class ListeTab implements Liste {


}
```

Exemples de réalisations

Une classe peut réaliser un nombre quelconque d'interfaces. Exemple avec `ArrayList` :

ArrayList.java

```
public class ArrayList implements Serializable,
                                  Cloneable,
                                  Iterable<E>,
                                  Collection<E>,
                                  List<E>,
                                  RandomAccess {
  ...
}
```

Exemples de réalisations

Remarque

Une classe n'implantant pas toutes les méthodes d'une interface qu'elle réalise est dite **abstraite**. Elle ne peut pas être instanciée.

La classe ListeTab

ListeTab.java

```java
/** Realisation de Liste */
public class ListeTab implements Liste {

    private double[] elements; // les elements
    private int nb;            // la taille de la liste

    /**
     * Creer une liste vide.
     * @param capacite La capacite max de la liste
     */
    public ListeTab(int capacite) {
        this.elements = new double[capacite];
        this.nb = 0;
    }

    @Override public int getNbElements() {
        return this.nb;
    }
```

La classe ListeTab

ListeTab.java

```
    @Override public double getElement(int pos) {
        return this.elements[pos];
    }

    @Override public void remplacer(int pos, double elt) {
        this.elements[pos] = elt;
    }

    @Override public void ajouter(int pos, double elt) {
        if (this.nb == this.elements.length) {
            // on agrandit le tableau
            double[] nouveau = new double[pos + 2]; //arbitraire !
            System.arraycopy(this.elements, 0, nouveau, 0, this.nb);
            this.elements = nouveau;
        }
        System.arraycopy(this.elements, pos, this.elements, pos + 1,
                         this.nb - pos);
        this.elements[pos] = elt;
        this.nb++;
    }
```

La classe ListeTab

ListeTab.java

```
    @Override public void supprimer(int pos) {
        System.arraycopy(this.elements, pos + 1, this.elements, pos,
                         this.nb - pos - 1);
        this.nb--;
    }

    /**
     * Le premier element de la liste
     * @return le premier element de la liste
     */
    public double premier() {
        return this.elements[0];
    }

    /**
     * Le dernier element de la liste
     * @return le dernier element de la liste
     */
    public double dernier() {
        return this.elements[this.nb - 1];
    }
}
```

Le mot-clé `@Override`

Le mot-clé `@Override` placé devant une méthode signale au compilateur qu'il s'agit d'une méthode définie dans une interface réalisée par la classe.

`@Override` permet une vérification **supplémentaire** par le compilateur : on sait déjà que si l'on ne définit pas une méthode d'une interface dans une classe la réalisant, une erreur de compilation se produit.

L'utilisation systématique de `@Override` permet d'avoir un message d'erreur pour **toutes** les méthodes non correctement implantées.

Nous reparlerons de `@Override` lors du cours sur l'héritage.

@Override : exemple

FalseListe.java

```
public class FalseListe implements Liste {

    @Override public int getnbElements() {
        return 0;
    }

    @Override public double getElement(int pos) {
        return 0.0;
    }

    @Override public void replacer(int pos, double elt) {

    }

    @Override public void ajouter(int pos, double elt) {

    }

    @Override public void supprimer(int pos) {

    }
}
```

@Override : exemple

shell

```
[tof@suntof]~ $ javac FalseListe.java
FalseListe.java:1: error: FalseListe is not abstract and does not override
abstract method remplacer(int,double) in Liste
public class FalseListe implements Liste {
       ^
FalseListe.java:3: error: method does not override or implement a method
from a supertype
    @Override public int getnbElements() {
    ^
FalseListe.java:11: error: method does not override or implement a method
from a supertype
    @Override public void replacer(int pos, double elt) {
    ^
3 errors
```

Plan de la partie 8 - Interfaces

Une interface est-elle un type ?

Sans s'apesantir sur les détails formels, une interface est un **type** :

- elle représente un ensemble d'objets ;
- ces objets sont des instances des classes réalisant l'interface.

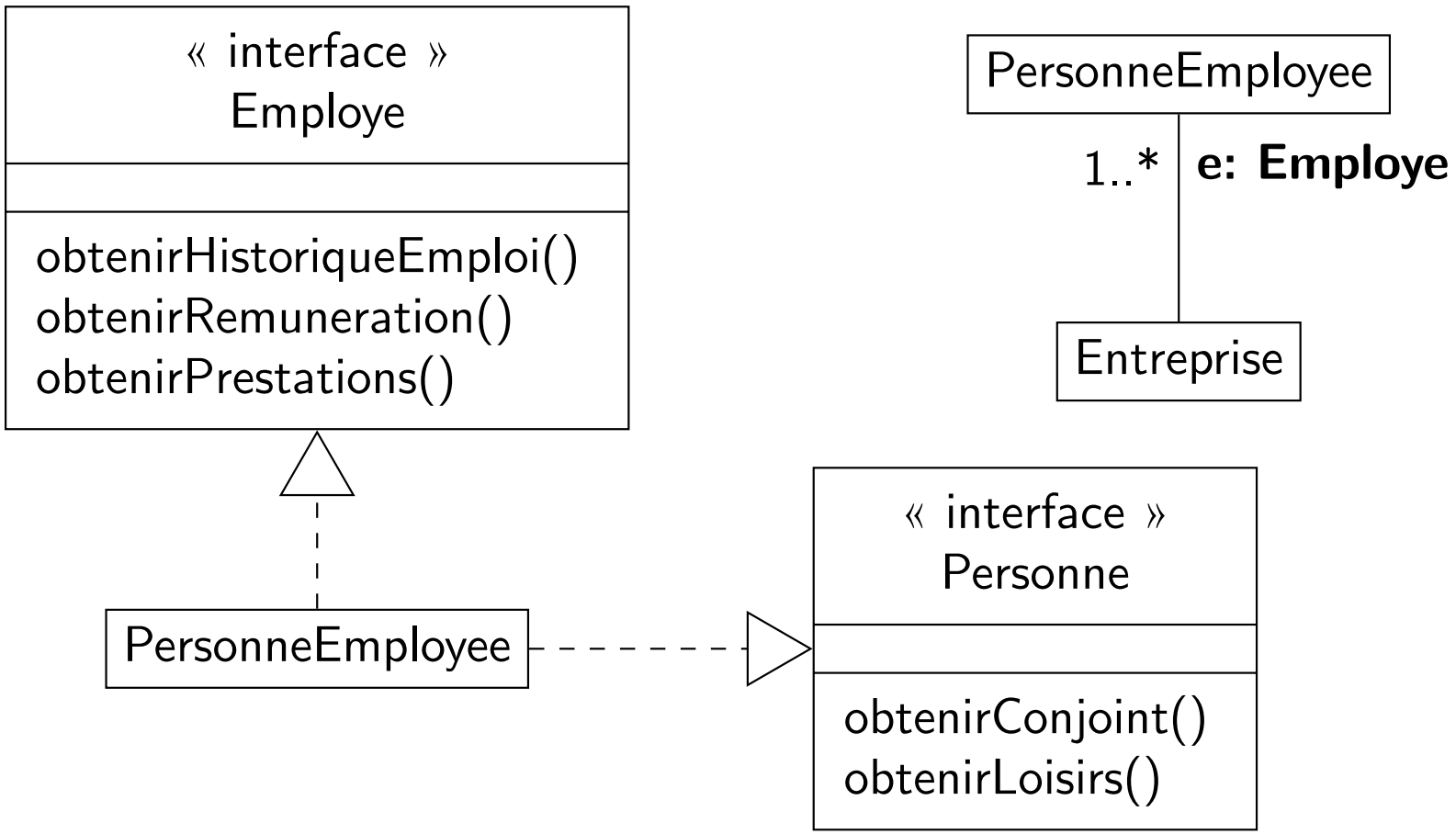
On doit donc pouvoir s'en servir :

- avec UML, pour typer les noms de rôles par exemple ;
- avec Java, pour typer une poignée.

Qu'attend-on d'une **poignée** typée par une interface ?

➥ l'**objet référencé par la poignée** fournira les services décrits dans l'interface.

Les interfaces vues comme types avec UML



Même si la classe `PersonneEmployee` possède d'autres méthodes, `Entreprise` n'a accès qu'aux méthodes de `Employe` sur `e`.

Les interfaces vues comme types avec Java

Puisque l'on peut typer une poignée avec une interface, on peut écrire des méthodes qui utilisent les interfaces sans en connaître les réalisations :

```
public class OutilListe {

    public static double somme(Liste l) {
        double somme = 0;
        for (int i = 0; i < l.getNbElements(); i++) {
            somme += l.getElement(i);
        }

        return somme;
    }
}
```

Que désigne le paramètre `l` de cette méthode ?

Types et sous-types

Définition (sous-type)

Si une classe C réalise une interface I, alors C est un sous-type de I.

Principe (substitution, Liskov)

Si un type T_1 est un sous-type de T_2 alors T_1 peut être **utilisé** partout où T_2 est **déclaré**.

On peut donc initialiser une poignée typée avec une interface avec un objet instance d'une classe réalisant l'interface.

```
Liste l1 = new ListeTab(5);    // OK, substitution
Liste l2 = new ListeChainee(); // OK, substitution
ListeTab l = new ListeTab(6);
OutilListe.somme(l);           // OK, substitution
// ListeTab lt1 = l1           // Interdit !
```

Affectation et sous-types

Principe (vérification statique)

Lors d'une affectation d'un **objet** de type T_1 à une **poignée** de type T_2, le **compilateur** vérifie que T_1 est un sous-type de T_2.
Lors d'une affectation d'une **poignée** de type T_1 à une **poignée** de type T_2, le **compilateur** vérifie que T_1 est un sous-type de T_2.

Ce principe est également appliqué lors de la vérification du type des **paramètres** d'une méthode lors d'un appel.

Liaison tardive (ou dynamique)

```
Liste l;              // declaration d'une poignee l de type Liste
l = new ListeTab(5); // creation d'un objet ListeTab attache a l
l.ajouter(0,21);     // appel de ajouter(0,21) sur l
```

Il y a deux types ici :

- le type **apparent** qui est le type de la poignée ;
- le type **réel** qui est le type de l'objet.

Remarque

La classe d'un objet ne change pas (création avec **new**) ! Par contre, un même objet peut être attaché à des poignées de types différents.

Liaison tardive (ou dynamique)

Principe

Quand une méthode est appliquée sur une poignée :

1. le **compilateur** vérifie que la méthode est déclarée dans le type de la poignée ;
2. la méthode effectivement exécutée est celle de la classe de l'objet attaché à la poignée. C'est la **liaison tardive**.

Nous reviendrons sur ces notions lors du cours sur l'héritage.

Type apparent : exemple avec `ListeTab`

Considérons le programme suivant :

TestListeTab.java

```
public class TestListeTab {
    public static void main(String[] args) {
        Liste lt = new ListeTab(10);
        lt.ajouter(0, 2.0);
        double d = lt.dernier();
    }
}
```

Type apparent : exemple avec `ListeTab`

Considérons le programme suivant :

TestListeTab.java

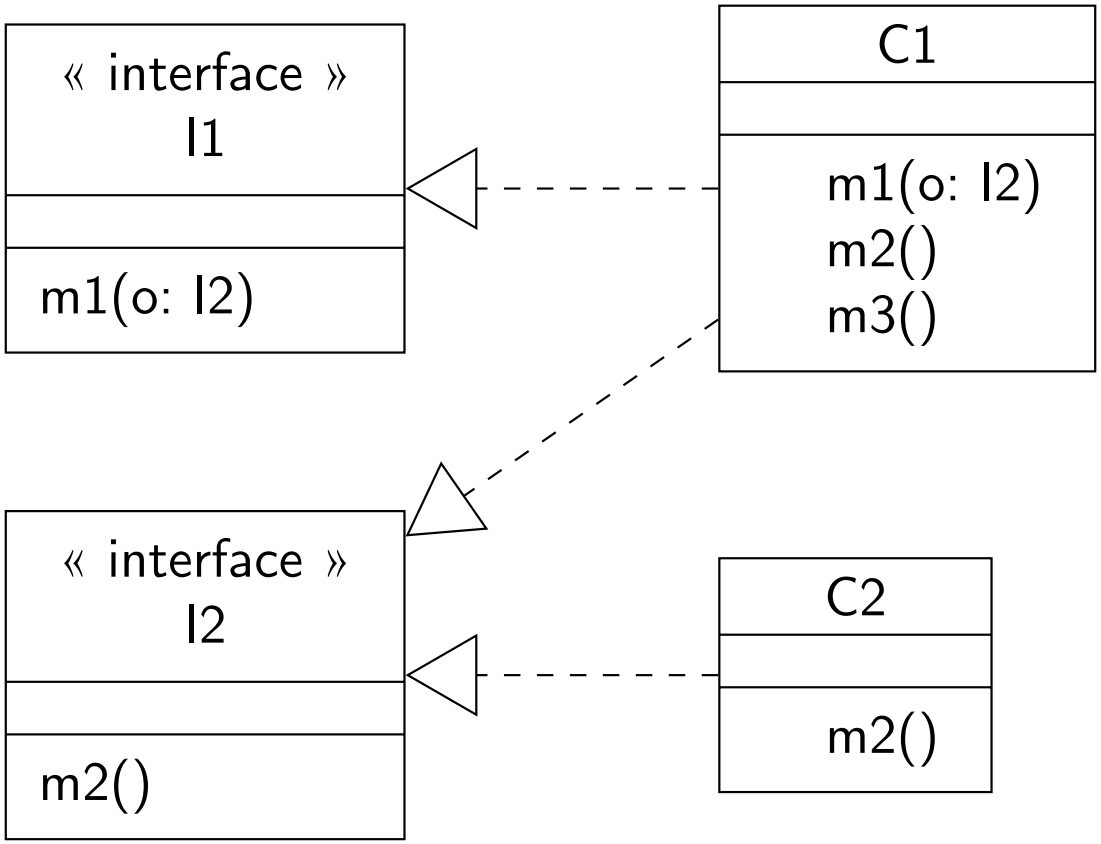
```
public class TestListeTab {
    public static void main(String[] args) {
        Liste lt = new ListeTab(10);
        lt.ajouter(0, 2.0);
        double d = lt.dernier();
    }
}
```

Erreur à la **compilation** même si `dernier` est une méthode de `ListeTab` :

shell

```
[tof@suntof]~ $ javac TestListeTab.java
TestListeTab.java:5: error: cannot find symbol
        double d = lt.dernier();
                      ^
  symbol:   method dernier()
  location: variable lt of type Liste
1 error
```

Exercice : liaison tardive



Exercice : liaison tardive

En considérant le diagramme précédent, quelles sont les lignes du programme suivant qui seront acceptées par le compilateur ?

```
I1 o1 = new C1();
I2 o2 = new C2();
o2 = o1;
o2 = new C1();

o1.m2();
o2.m3();

o1.m1(o1);
o1.m1(o2);
o1.m1(new C1());
```

Les interfaces dans Java 8 (avancé)

Depuis la version 8 de Java, il est possible de définir l'implémentation de certaines méthodes :

- des **méthodes statiques** qui permettent ainsi de définir des méthodes utilitaires dans une interface
- des **méthodes par défaut** (*default methods*) qui fournissent une implémentation par défaut de ces méthodes pour les classes réalisant l'interface.
  Par exemple pour l'interface `Point`, on pourrait définir une méthode `distance` par défaut dans l'interface :

```
default double distance(Point p) {
    return Math.sqrt(Math.pow(this.getX() - p.getX()) +
                     Math.pow(this.getY() - p.getY()));
}
```

  Les méthodes par défaut permettent par exemple d'enrichir une interface existante sans que toutes les classes réalisant cette interface ne doivent être modifiées.

Nous n'aborderons pas ces aspects dans le cours.

9 - Spécialisation et héritage

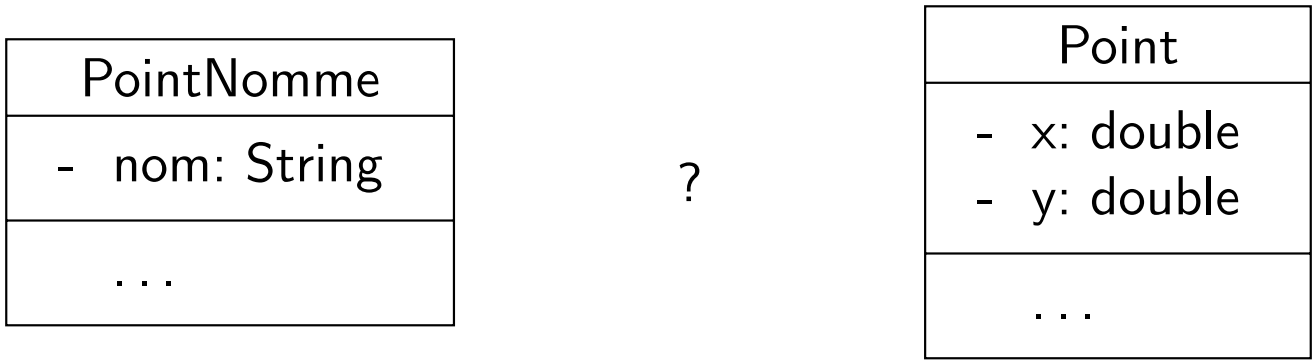
Problématique

1. on dispose d'une classe `Compte` décrivant des comptes bancaires génériques et les opérations qui y sont associées (créditer, débiter, etc.). On souhaite construire une classe représentant des comptes bancaires particuliers, par exemple des comptes rémunérés.
   - on ne dispose évidemment pas du source de la classe `Compte`, donc pas de copier-coller ;
   - même si l'on disposait du source, il faudrait répercuter les changements du code des méthodes de `Compte` à chaque fois que l'auteur de la classe la modifierait ;
   - existe-t-il une solution efficace pour résoudre ce problème ?
2. on dispose de classes décrivant des objets graphiques servant à réaliser une GUI (boutons, fenêtres, etc.) provenant de multiples concepteurs.
   - évidemment, les objets de ces classes ont des comportements en commun, comme par exemple le fait que l'on puisse les afficher ;
   - comment faire pour « factoriser » ces comportements communs ?
   - comment faire pour imposer à une nouvelle classe représentant un objet graphique ces comportements ?

Un exemple simple

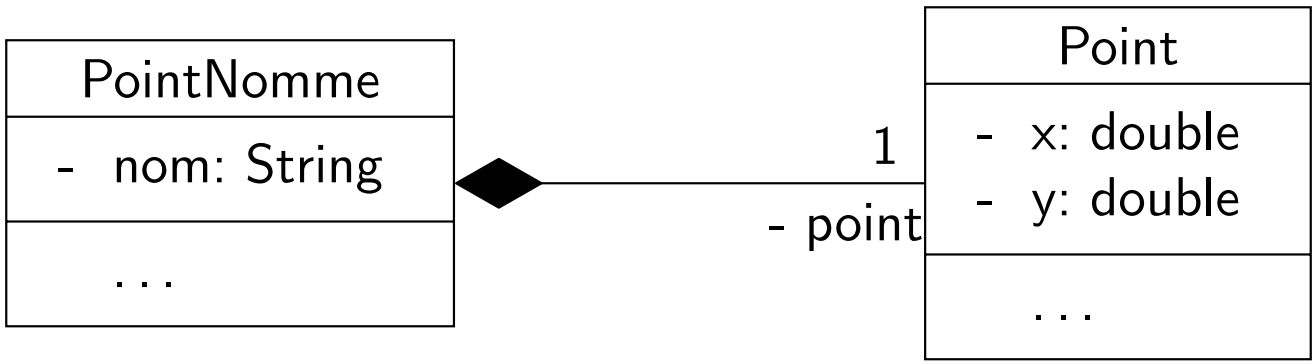
Supposons que l'on veuille créer une classe `PointNomme` à partir de la classe `Point`. Un point nommé est juste un point avec un nom.

Comment lier les classes `Point` et `PointNomme` ?



Un exemple simple

On pourrait utiliser la **composition** ...



...mais :

- un point nommé **est aussi** un point : comment le représenter ?
- on doit réécrire (avec délégation) **toutes les méthodes** de `Point`

Objectifs et principes

Buts

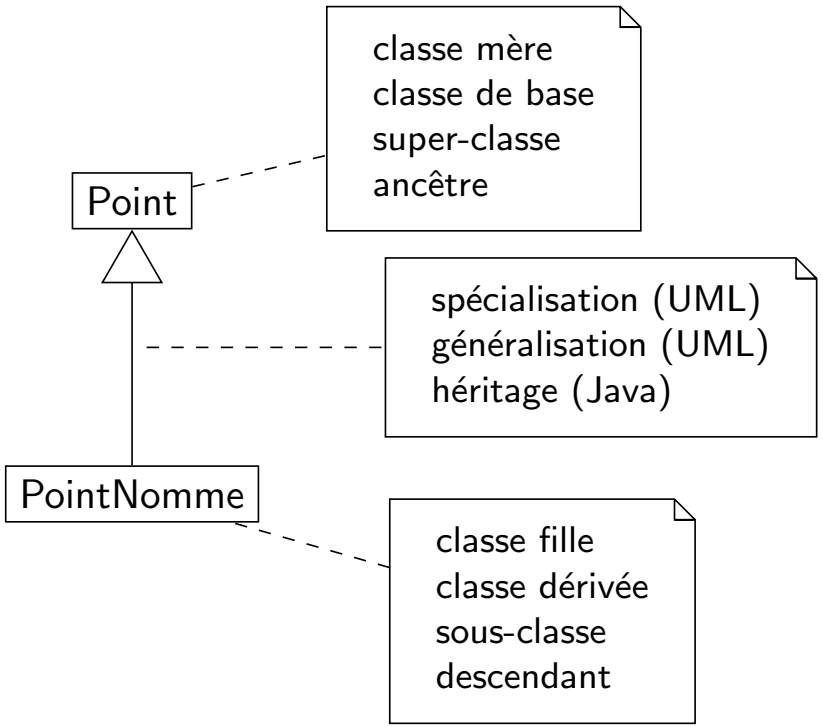
- définir une nouvelle classe comme **spécialisation** d'une classe existante ;
- factoriser des propriétés et comportements communs à plusieurs classes (**généralisation**) ;
- définir une **relation de sous-typage** entre classes (substitutionnalité) ;
- **copier virtuellement** les caractéristiques de classes existantes dans la définition d'une nouvelle classe (héritage).

Moyens

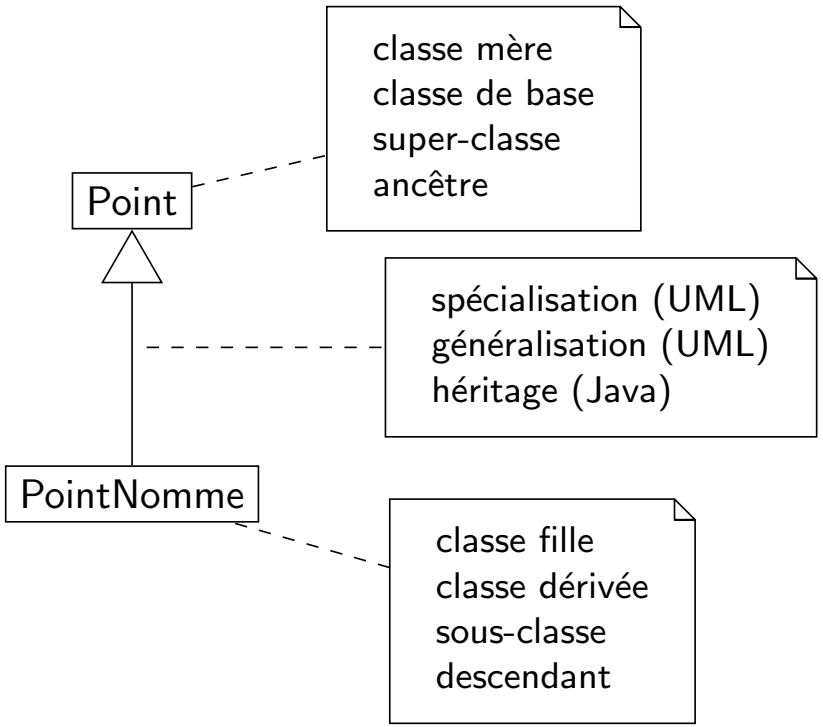
- la relation de **généralisation/spécialisation** en UML ;
- l'**héritage** en Java.

Plan de la partie 9 - Spécialisation et héritage

Notation et vocabulaire



Notation et vocabulaire




```
public class Point {
  ...
}

public class PointNomme
  extends Point {
  ...
}
```

Syntaxe (héritage)

```
public class A extends B { ... }
```

Sémantique

Quel est le **sens** d'une relation d'héritage ? On peut distinguer deux aspects (contenus tous les deux dans la notion d'héritage) :

1. héritage de contrat et de type

La sous-classe **acquiert le type** de la super-classe et donc en particulier sa structure et son comportement.

Principe (Substitution, Liskov)

Une instance d'une sous-classe peut être **utilisée** partout où une instance de sa super-classe est **déclarée**.

Remarque

C'est le seul héritage disponible en Java.

Sémantique

Quel est le **sens** d'une relation d'héritage ? On peut distinguer deux aspects (contenus tous les deux dans la notion d'héritage) :

2. héritage d'implantation

La sous classe acquiert l'implantation de la super-classe.
Dans ce cas, la sous-classe a accès aux champs et méthodes **accessibles** de la super-classe (donc pas aux caractéristiques **`private`**).

La suite du cours ne considérera que l'héritage de contrat et de type.

Héritage et sous-typage

Définition (sous-type)

Si une classe `A` hérite d'une classe `B`, alors `A` est un **sous-type** de `B`.

Assez intuitif : si `PointNomme` hérite de `Point`, alors tous les points nommés sont des points.

Remarque

On peut donc réutiliser tous les résultats vus sur les interfaces (principe de substitution, affectation).

Principe de substitution : rappel

Principe (Substitution, Liskov)

L'instance d'une classe descendante peut être utilisée partout où une classe ancêtre est utilisée pour une déclaration.

Intuition : tout ce qui est demandé à un objet de la classe mère peut l'être également à un objet de la classe fille d'après l'héritage.

Évidemment, l'inverse est faux : par exemple, tous les points ne sont pas forcément des points nommés.

Liskov, Barbara H. et Jeannette M. Wing (1994).
« A behavioral notion of subtyping ».
In : **ACM Trans. Program. Lang. Syst.** 16.6.
http://doi.acm.org/10.1145/197320.197383.

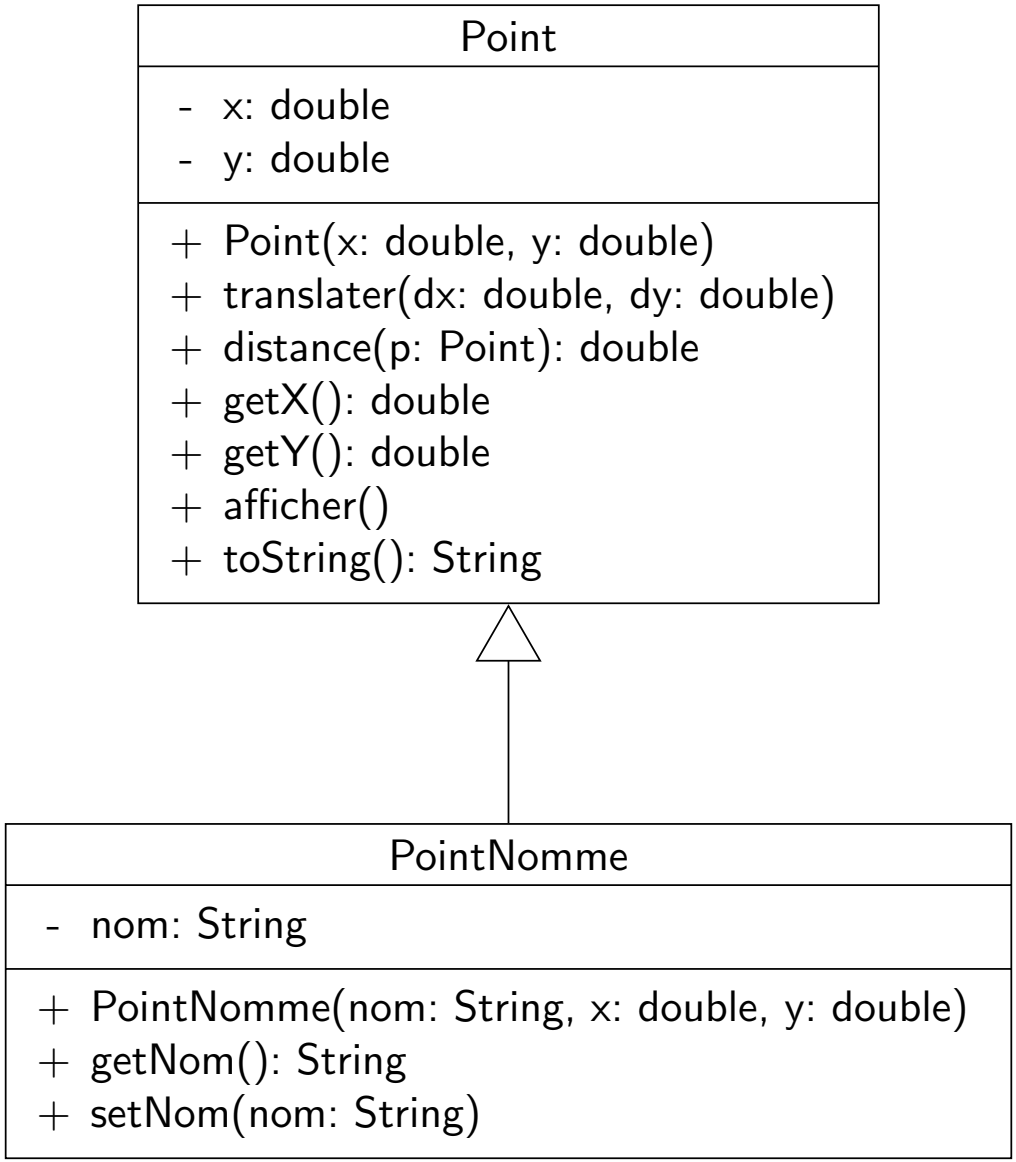
Affectation et héritage : rappel

Principe (vérification statique)

Lors d'une affectation d'un **objet** ou d'une **poignée** de type T_1 à une **poignée** de type T_2, le **compilateur** vérifie que T_1 est un sous-type de T_2.

Ce principe est également appliqué lors de la vérification du type des **paramètres** d'une méthode lors d'un appel.

Héritage : exemple avec UML



Héritage : exemple avec Java

PointNomme.java

```
/** Un point nomme est un point avec un nom. */

public class PointNomme extends Point {

    private String nom;

    /** Initialiser a partir de son nom et de ses coordonnees
     *  cartesiennes */
    public PointNomme(String nom, double x, double y) {
        super(x, y);
        this.nom = nom;
    }

    /** Le nom du point nomme */
    public String getNom() {
        return this.nom;
    }

    /** Changer le nom du point */
    public void setNom(String nom) {
        this.nom = nom;
    }
}
```

Héritage de type : exemple

exemple principe sub. avec une classe ExHerType

```
public class ExHerType {
    public static void main(String[] args) {
        PointNomme pn1 = new PointNomme("A", 0, 0);
        Point      p   = new PointNomme("B", 1, 1);
        PointNomme pn2 = new Point(2, 2);
    }
}
```

Compilation :

shell

```
[tof@suntof]~ $ javac ExHerType.java
ExHerType.java:5: error: incompatible types
        PointNomme pn2 = new Point(2, 2);
                         ^
  required: PointNomme
  found:    Point
1 error
```

Heritage d'implantation : exemple

exemple her. impl. avec une classe ExHerImp

```
public class ExHerImp {
    public static void main(String[] args) {
        PointNomme pn = new PointNomme("A", 0, 0);
        System.out.println("nom du point : " + pn.getNom());
        pn.translater(1,2);
        pn.afficher();
    }
}
```

Exécution :

shell

```
[tof@suntof]~ $ java ExHerImp
nom du point : A
(1.0,2.0)
```

Accès aux caractéristiques

En cas d'accès à une caractéristique **privée** de `Point` par exemple :

PointNomme.java

```
    public PointNomme(String nom, double x, double y) {
        super(x, y);
        this.nom = nom;
        this.x = x;
    }
```

compilation

```
[tof@suntof]~ $ javac PointNomme.java
PointNomme.java:11: error: x has private access in Point
        this.x = x;
            ^
1 error
```

Enrichir une classe

Dans une sous-classe, on peut ajouter des **nouveaux attributs**.
Il faut alors leur donner un nouveau nom (sinon il y a masquage, cf. plus loin).

PointNomme.java

```
public class PointNomme extends Point {

    private String nom;

}
```

Enrichir une classe

Dans une sous-classe, on peut ajouter des **nouvelles méthodes**.

PointNomme.java

```
    /** Le nom du point nomme */
    public String getNom() {
        return this.nom;
    }
```

Enrichir une classe

Remarque

Ajouter une nouvelle méthode s'entend au sens de la **surcharge**.

Par exemple, dans `PointNomme`, on peut ajouter :

PointNomme.java

```
public void afficher(String message) {
  System.out.print(message);
  this.afficher();      // utiliser le afficher de Point
}
```

qui **surcharge** la méthode `afficher()` de `Point`.

Plan de la partie 9 - Spécialisation et héritage

Constructeurs

Principe

Tout constructeur d'une sous-classe doit faire appel au constructeur de sa super-classe.

Justification : hériter d'une classe, c'est récupérer ses caractéristiques qui doivent être initialisées.

Constructeurs

En Java, on utilise le mot-clé **super** suivi des paramètres permettant au compilateur de choisir le constructeur de la classe parente.

Syntaxe (appel au constructeur de la classe parente)

```
super(par1, ..., parn);
```

Par exemple :

PointNomme.java

```
public PointNomme(String nom, double x, double y) {
  super(x, y);
  this.nom = nom;
}
```

Constructeur : appel de **super**

Principe

L'appel à **super doit être la première instruction** du constructeur.

Idée : la classe parent est toujours « initialisée » avant la sous-classe.

Principe

Si aucun appel à **super** n'est fait (omission ou **super** n'est pas la première instruction), le compilateur appelle automatiquement le constructeur par défaut de la classe parente, soit **super**`()`.

Dans ce cas, attention s'il y a un constructeur par défaut.

super mal placé : exemple

PoinNomme.java

```
 9      public PointNomme(String nom, double x, double y) {
10          this.nom = nom;
11          super(x, y);
12      }
```

À la compilation, **2 erreurs** :

shell

```
[tof@suntof]~ $ javac PointNomme.java
PointNomme.java:9: error: constructor Point in class Point cannot be
 applied to given types;
    public PointNomme(String nom, double x, double y) {
                                                      ^
  required: double,double
  found: no arguments
  reason: actual and formal argument lists differ in length
PointNomme.java:11: error: call to super must be
 first statement in constructor
        super(x, y);
             ^
2 errors
```

Plan de la partie 9 - Spécialisation et héritage

Plan de la partie 9 - Spécialisation et héritage

Redéfinition de méthodes

Une méthode peut avoir la **même signature** dans une classe et une de ses sous-classes.

Intuition : la sous-classe peut **adapter son comportement** et donc donner un nouveau corps à la méthode.

Exemple : dans `PointNomme`, on redéfinit `afficher` pour tenir compte du nom du point nommé

PointNomme.java

```
    public void afficher() {
        System.out.print(this.nom + ":");
        System.out.println("(" + this.getX() + "," +
                           this.getY() + ")");
    }
```

Redéfinition de méthodes

Dans le code de la méthode de la classe fille, on peut faire appel aux méthodes de la classe parente par l'intermédiaire de super.

Syntaxe (appel à une méthode de la classe parente)

```
super.methodeClasseParente()
```

Exemple :

PointNomme.java

```
    public void afficher() {
        System.out.print(this.nom + ":");
        super.afficher();
    }
```

Redéfinition de méthodes

Attention

- la méthode de la super-classe n'est pas masquée mais reçoit une nouvelle implantation.
- ne pas confondre **surcharge** et **redéfinition**.

Redéfinition de méthode : vérification

Comment être sûr qu'une méthode est bien la redéfinition d'une autre méthode ?

➥ on peut utiliser le **tag** `@Override` devant une méthode

ExempleOverride.java

```
1     @Override public void affiche() {
2         System.out.print(this.getNom() + ": ");
3         super.afficher();
4     }
```

Dans ce cas, le **compilateur** nous signale une erreur :

shell

```
[tof@suntof]~ $ javac PointNomme.java
PointNomme.java:24: error: method does not override or implement a method
from a supertype
    @Override public void affiche() {
    ^
1 error
```

Plan de la partie 9 - Spécialisation et héritage

Droits d'accès et redéfinition

Principe

On ne peut redéfinir que des méthodes **accessibles**.

A.java

```
package a;

public class A {
  void m() { }
}
```

B.java

```
package b;
import a.A;

public class B extends A {
  @Override void m() { }
}
```

shell

```
[tof@suntof]~ $ javac A.java B.java
B.java:5: error: method does not override or implement a method from
a supertype
  @Override void m() { }
  ^
1 error
```

Changer les droits d'accès

Les caractéristiques héritées conservent leur droit d'accès (une méthode **`protected`** de la classe mère reste **`protected`** dans la classe fille).

Principe (augmentation des droits d'accès)

La classe fille peut augmenter le droit d'accès des méthodes héritées. Il suffit de la **redéfinir** avec le nouveau droit d'accès.

Par contre, elle ne peut pas les réduire (contrainte du sous-typage).

Plan de la partie 9 - Spécialisation et héritage

Liaison tardive (*dynamic binding*)

```
Point p;                          // poignee sur un point
p = new PointNomme("A", 0, 0); // creer un PointNomme et l'attacher
...                               // a p
p.afficher();                     // appel de afficher() sur p
```

Il y a deux types ici :

- le **type apparent** qui est le type de la poignée (`Point` ici)
- le **type réel** qui est la classe de l'objet (`PointNomme` ici)

Remarque

- **la classe d'un objet ne change pas** (création avec **new**). Par contre, un même objet peut être attaché à des poignées de types différents.
- **le type apparent d'une poignée ne change pas**. Par contre, on peut réaffcter la poignée avec des objets de types compatibles avec son type apparent.

Liaison tardive (*dynamic binding*)

```
Point p;                          // poignee sur un point
p = new PointNomme("A", 0, 0); // creer un PointNomme et l'attacher
...                               // a p
p.afficher();                     // appel de afficher() sur p
```

Principe (liaison tardive)

Quand une méthode est appliquée sur une poignée, le corps de la méthode à exécuter est choisi **à l'exécution** (liaison tardive) en fonction de la classe de l'objet attaché à la poignée (donc du type réel).

Ici, c'est `afficher()` définie dans `PointNomme`.

Résolution d'un appel polymorphe

```
T p;                // declaration de la poignee (type apparent : T)
p = new X(...);     // affectation de la poignee (type reel : X)
...
p.m(a1, ..., an); // appel de la methode m() sur p
```

Principe (Résolution d'un appel polymorphe)

1. **résolution de la surcharge** (vérification statique)
   But : identification de la signature de la méthode à exécuter.
   La classe du type apparent de la poignée (classe `T`) doit avoir une méthode `m(T1, ..., Tn)` dont les paramètres correspondent en nombre et en type aux paramètres effectifs `a1, ..., an`.
   Si la signature n'est pas trouvée, il y a une erreur à la **compilation**.
2. **liaison dynamique** (à l'exécution généralement)
   Le système choisit la version de `m(T1, ..., Tn)` à exécuter : c'est la dernière (re)définition rencontrée partant du type `T` et descendant vers le type réel de l'objet attachée à la poignée, soit `X`.

Liaison tardive : exemple complet

```
Point p = new Point(0, 0);
PointNomme pn = new PointNomme("A", 1, 1);
Point p1;
p1 = p; p1.afficher();
p1 = pn; p1.afficher();

PointNomme pn1;
pn1 = p; pn1.afficher();
pn1 = pn; pn1.afficher();

pn1 = p1;
pn1.afficher();
```

Type apparent **Type réel**

Affichage console

Liaison tardive : exemple complet

```
Point p = new Point(0, 0);
PointNomme pn = new PointNomme("A", 1, 1);
Point p1;
p1 = p; p1.afficher();
p1 = pn; p1.afficher();

PointNomme pn1;
pn1 = p; pn1.afficher();
pn1 = pn; pn1.afficher();

pn1 = p1;
pn1.afficher();
```

Type apparent **Type réel**

Affichage console

Liaison tardive : exemple complet

```
Point p = new Point(0, 0);
PointNomme pn = new PointNomme("A", 1, 1);
Point p1;
p1 = p; p1.afficher();
p1 = pn; p1.afficher();

PointNomme pn1;
pn1 = p; pn1.afficher();
pn1 = pn; pn1.afficher();

pn1 = p1;
pn1.afficher();
```

	Type apparent	**Type réel**
p	Point	Point

Affichage console

Liaison tardive : exemple complet

```
Point p = new Point(0, 0);
PointNomme pn = new PointNomme("A", 1, 1);
Point p1;
p1 = p; p1.afficher();
p1 = pn; p1.afficher();

PointNomme pn1;
pn1 = p; pn1.afficher();
pn1 = pn; pn1.afficher();

pn1 = p1;
pn1.afficher();
```

	Type apparent	**Type réel**
p	Point	Point

Affichage console

Liaison tardive : exemple complet

```
Point p = new Point(0, 0);
PointNomme pn = new PointNomme("A", 1, 1);
Point p1;
p1 = p; p1.afficher();
p1 = pn; p1.afficher();

PointNomme pn1;
pn1 = p; pn1.afficher();
pn1 = pn; pn1.afficher();

pn1 = p1;
pn1.afficher();
```

	Type apparent	**Type réel**
p	Point	Point
pn	PointNomme	PointNomme

Affichage console

Liaison tardive : exemple complet

```
Point p = new Point(0, 0);
PointNomme pn = new PointNomme("A", 1, 1);
Point p1;
p1 = p; p1.afficher();
p1 = pn; p1.afficher();

PointNomme pn1;
pn1 = p; pn1.afficher();
pn1 = pn; pn1.afficher();

pn1 = p1;
pn1.afficher();
```

	Type apparent	**Type réel**
p	Point	Point
pn	PointNomme	PointNomme

Affichage console

Liaison tardive : exemple complet

```
Point p = new Point(0, 0);
PointNomme pn = new PointNomme("A", 1, 1);
Point p1;
p1 = p; p1.afficher();
p1 = pn; p1.afficher();

PointNomme pn1;
pn1 = p; pn1.afficher();
pn1 = pn; pn1.afficher();

pn1 = p1;
pn1.afficher();
```

	Type apparent	**Type réel**
p	Point	Point
pn	PointNomme	PointNomme
p1	Point	**null**

Affichage console

Liaison tardive : exemple complet

```
Point p = new Point(0, 0);
PointNomme pn = new PointNomme("A", 1, 1);
Point p1;
p1 = p; p1.afficher();
p1 = pn; p1.afficher();

PointNomme pn1;
pn1 = p; pn1.afficher();
pn1 = pn; pn1.afficher();

pn1 = p1;
pn1.afficher();
```

	Type apparent	**Type réel**
p	Point	Point
pn	PointNomme	PointNomme
p1	Point	**null**

Affichage console

Liaison tardive : exemple complet

```
Point p = new Point(0, 0);
PointNomme pn = new PointNomme("A", 1, 1);
Point p1;
p1 = p; p1.afficher();
p1 = pn; p1.afficher();

PointNomme pn1;
pn1 = p; pn1.afficher();
pn1 = pn; pn1.afficher();

pn1 = p1;
pn1.afficher();
```

	Type apparent	Type réel
p	Point	Point
pn	PointNomme	PointNomme
p1	Point	Point

Affichage console

```
(0,0)
```

Liaison tardive : exemple complet

```
Point p = new Point(0, 0);
PointNomme pn = new PointNomme("A", 1, 1);
Point p1;
p1 = p; p1.afficher();
p1 = pn; p1.afficher();

PointNomme pn1;
pn1 = p; pn1.afficher();
pn1 = pn; pn1.afficher();

pn1 = p1;
pn1.afficher();
```

	Type apparent	**Type réel**
p	Point	Point
pn	PointNomme	PointNomme
p1	Point	Point

Affichage console

```
(0,0)
```

Liaison tardive : exemple complet

```
Point p = new Point(0, 0);
PointNomme pn = new PointNomme("A", 1, 1);
Point p1;
p1 = p; p1.afficher();
p1 = pn; p1.afficher();

PointNomme pn1;
pn1 = p; pn1.afficher();
pn1 = pn; pn1.afficher();

pn1 = p1;
pn1.afficher();
```

	Type apparent	Type réel
p	Point	Point
pn	PointNomme	PointNomme
p1	Point	PointNomme

Affichage console

```
(0,0)
A:(1,1)
```

Liaison tardive : exemple complet

```
Point p = new Point(0, 0);
PointNomme pn = new PointNomme("A", 1, 1);
Point p1;
p1 = p; p1.afficher();
p1 = pn; p1.afficher();

PointNomme pn1;
pn1 = p; pn1.afficher();
pn1 = pn; pn1.afficher();

pn1 = p1;
pn1.afficher();
```

	Type apparent	**Type réel**
p	Point	Point
pn	PointNomme	PointNomme
p1	Point	PointNomme

Affichage console

```
(0,0)
A:(1,1)
```

Liaison tardive : exemple complet

```
Point p = new Point(0, 0);
PointNomme pn = new PointNomme("A", 1, 1);
Point p1;
p1 = p; p1.afficher();
p1 = pn; p1.afficher();

PointNomme pn1;
pn1 = p; pn1.afficher();
pn1 = pn; pn1.afficher();

pn1 = p1;
pn1.afficher();
```

	Type apparent	Type réel
p	Point	Point
pn	PointNomme	PointNomme
p1	Point	PointNomme
pn1	PointNomme	**null**

Affichage console

```
(0,0)
A:(1,1)
```

Liaison tardive : exemple complet

```
Point p = new Point(0, 0);
PointNomme pn = new PointNomme("A", 1, 1);
Point p1;
p1 = p; p1.afficher();
p1 = pn; p1.afficher();

PointNomme pn1;
pn1 = p; pn1.afficher();
pn1 = pn; pn1.afficher();

pn1 = p1;
pn1.afficher();
```

	Type apparent	Type réel
p	Point	Point
pn	PointNomme	PointNomme
p1	Point	PointNomme
pn1	PointNomme	**null**

Affichage console

```
(0,0)
A:(1,1)
```

Liaison tardive : exemple complet

```
Point p = new Point(0, 0);
PointNomme pn = new PointNomme("A", 1, 1);
Point p1;
p1 = p; p1.afficher();
p1 = pn; p1.afficher();

PointNomme pn1;
pn1 = p; pn1.afficher();                    interdit par le compilateur
pn1 = pn; pn1.afficher();

pn1 = p1;
pn1.afficher();
```

	Type apparent	Type réel
p	Point	Point
pn	PointNomme	PointNomme
p1	Point	PointNomme
pn1	PointNomme	Impossible !

Affichage console

```
(0,0)
A:(1,1)
```

Liaison tardive : exemple complet

```
Point p = new Point(0, 0);
PointNomme pn = new PointNomme("A", 1, 1);
Point p1;
p1 = p; p1.afficher();
p1 = pn; p1.afficher();

PointNomme pn1;
pn1 = p; pn1.afficher();
pn1 = pn; pn1.afficher();

pn1 = p1;
pn1.afficher();
```

	Type apparent	**Type réel**
p	Point	Point
pn	PointNomme	PointNomme
p1	Point	PointNomme
pn1	PointNomme	**null**

Affichage console

```
(0,0)
A:(1,1)
```

Liaison tardive : exemple complet

```
Point p = new Point(0, 0);
PointNomme pn = new PointNomme("A", 1, 1);
Point p1;
p1 = p; p1.afficher();
p1 = pn; p1.afficher();

PointNomme pn1;
pn1 = p; pn1.afficher();
pn1 = pn; pn1.afficher();

pn1 = p1;
pn1.afficher();
```

	Type apparent	Type réel
p	Point	Point
pn	PointNomme	PointNomme
p1	Point	PointNomme
pn1	PointNomme	PointNomme

Affichage console

```
(0,0)
A:(1,1)
A:(1,1)
```

Liaison tardive : exemple complet

```
Point p = new Point(0, 0);
PointNomme pn = new PointNomme("A", 1, 1);
Point p1;
p1 = p; p1.afficher();
p1 = pn; p1.afficher();

PointNomme pn1;
pn1 = p; pn1.afficher();
pn1 = pn; pn1.afficher();

pn1 = p1;
pn1.afficher();
```

	Type apparent	**Type réel**
p	Point	Point
pn	PointNomme	PointNomme
p1	Point	PointNomme
pn1	PointNomme	PointNomme

Affichage console

```
(0,0)
A:(1,1)
A:(1,1)
```

Liaison tardive : exemple complet

```
Point p = new Point(0, 0);
PointNomme pn = new PointNomme("A", 1, 1);
Point p1;
p1 = p; p1.afficher();
p1 = pn; p1.afficher();

PointNomme pn1;
pn1 = p; pn1.afficher();
pn1 = pn; pn1.afficher();

pn1 = p1;                                    interdit par le compilateur
pn1.afficher();
```

	Type apparent	Type réel
p	Point	Point
pn	PointNomme	PointNomme
p1	Point	PointNomme
pn1	PointNomme	PointNomme

Affichage console

```
(0,0)
A:(1,1)
A:(1,1)
```

Attention aux « fausses » redéfinitions !

```
public class A {
    private void m() {
        System.out.println("method in A");
    }

    public void aff() {
        this.m();
    }
}
```

```
public class B extends A {
    private void m() {
        System.out.println("method in B");
    }
}
```

Attention aux « fausses » redéfinitions !

```
public class ExPrivateCall {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        a.aff();
        b.aff();
    }
}
```

exécution

```
[tof@suntof]~ $ java ExPrivateCall
method in A
method in A
```

Remarque

D'où l'intérêt d'utiliser `@Override` !

Interrogation dynamique de type

```
Point p = new PointNomme("A", 1, 1);
PointNomme pn;
pn = p;           // Interdit par le compilateur !
```

Le compilateur **interdit** cette affectation car il s'appuie sur les types apparents. Or, un point nommé est attaché à p. L'affectation aurait donc un sens.

Interrogation dynamique de type

C'est le problème de l'**affectation renversée** résolu en Java par le **transtypage** :

```
Point p = new PointNomme("A", 1, 1);
PointNomme pn;
pn = (PointNomme) p;    // Autorise par le compilateur !
```

Remarque

Le transtypage ressemble au *cast* du C, mais cette conversion est vérifiée à l'exécution (levée d'une exception `ClassCastException`).

Opérateur **instanceof**

Pour vérifier le type d'un objet attaché à une poignée, on utilise l'opérateur **instanceof** :

```
if (p instanceof PointNomme) {
  PointNomme pn = (PointNomme)p;
  System.out.println(pn.getNom());
}
```

Syntaxe (instanceof)

```
poignee instanceof Type
```

Plan de la partie 9 - Spécialisation et héritage

Masquage d'attributs

Les attributs ne peuvent pas être redéfinis, ils sont **masqués**.

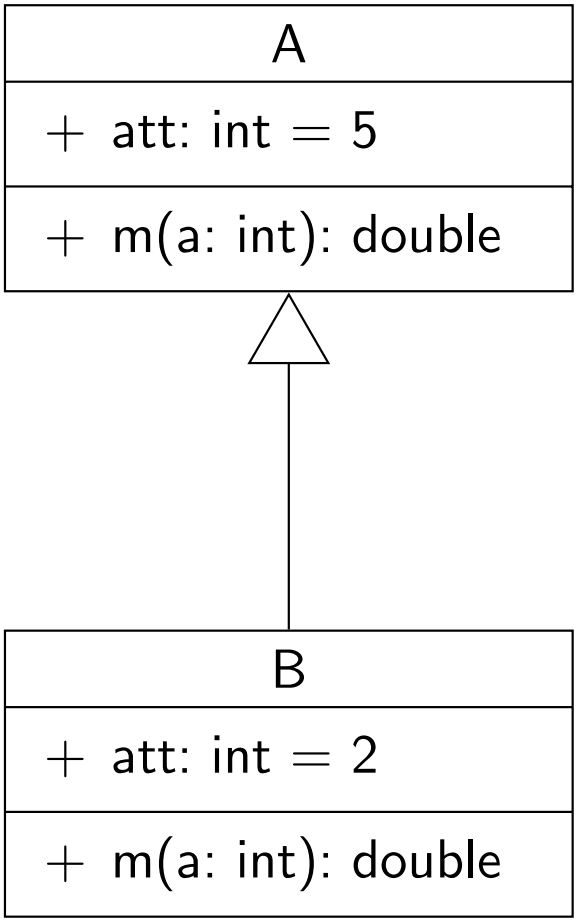
Il faut utiliser **`super`** ou une autre référence vers le type de la classe mère pour y accéder.

C'est le même principe pour les attributs et méthodes statiques.

Principe

`this` $\equiv$ référence vers l'objet vu comme instance de la classe
`super` $\equiv$ référence vers l'objet vu comme instance de la super-classe

Exemple de redéfinition et de masquage




```
B objB = new B();
```

`objB.att`	vaut 2
`((A)objB).att`	vaut 5
`objB.m(2)`	appelle la methode de `B`
`((A)objB).m(2)`	appelle la methode de `B`

```
A objA = new A();
```

`objA = objB`	autorise
`objB = objA`	ne compile pas
`objB = (B)objA`	compile, mais. . .

```
A objA = new B();
```

`objA = objB`	autorise
`objA.att`	vaut 5
`objA.m(2)`	appelle la methode de `B`
`objB = objA`	ne compile pas
`objB = (B)objA`	compile et « marche »

Liaison tardive : intérêts

Elle permet d'éviter des structures de type choix multiples :

- les alternatives sont traitées par le compilateur (sûreté : pas d'oubli) ;
- l'ajout d'une nouvelle alternative est facilité (extensibilité).

En absence de liaison tardive, il faudrait **tester explicitement** le type de l'objet associé à une poignée pour **choisir quelle méthode lui appliquer**.

La liaison tardive permet de réaliser le **principe de choix unique** (B. Meyer) :

Principe (choix unique)

Chaque fois qu'un système logiciel doit prendre en compte un ensemble d'alternatives, un et un seul module doit en connaître la liste exhaustive.

Plan de la partie 9 - Spécialisation et héritage

Contraintes sur la redéfinition

Principe (respect de la sémantique)

La redéfinition d'une méthode doit préserver la sémantique de la version précédente : la nouvelle version doit fonctionner au moins dans les mêmes cas et faire au moins ce qui était fait.

Exemple : une méthode `f(A a,...)` utilise une classe polymorphe `A` qui contient une méthode polymorphe g.

- l'auteur de `f` ne connaît que `A` *a priori*. Il va donc utiliser la spécification de `A` pour utiliser g ;
- la méthode `f` est appelée avec un objet de la classe `B` qui dérive de `A` (possible grâce au principe de substitution) et redéfinissant g ;
- avec la liaison dynamique, c'est la version de g de la classe `B` qui est appelée ;
- **conclusion** : la version de g dans `B` doit fonctionner dans les cas prévus dans `A` et faire au moins ce qui était prévu dans `A`.

Contraintes sur la redéfinition

Principe

Une classe fille doit réussir les tests de ses classes mères.

Il faut également faire attention à la sémantique des paramètres de la classe : elle doit être la même que celle des paramètres de la classe mère.

Principe (Méthodes polymorphes et constructeurs)

Un constructeur ne devrait pas appeler de méthode polymorphe.

Exemple : le constructeur de la classe `A` appelle une méthode polymorphe `m`. Cette méthode est redéfinie dans `B`, classe fille de `A` et elle utilise un attribut `att` propre à `B`. Que se passe-t-il à la création d'un objet de type `B` ?

« Checklist » pour spécialiser une classe

Lorsque l'on veut spécialiser une classe `A` en une classe `B`, on doit vérifier les éléments suivants :

1. **le principe de substitution est-il vérifié ?**
   C'est la propriété la plus importante à vérifier (Java par exemple ne donne aucune construction syntaxique permettant de l'imposer). Il faut vérifier qu'une instance de `B` se comportera au moins « aussi bien » qu'une instance de `A`.
2. **comment écrire le ou les constructeurs ?**
   Il faut vérifier que l'on définit des constructeurs de `B` permettant d'initialiser correctement les objets, en particulier permettant d'appeler correctement le ou les constructeurs de `A`.
3. **y-a-t'il des méthodes à redéfinir ?**
   Les méthodes à redéfinir représentent des comportements que l'on souhaite adapter pour les instances de la classe `B`. Là encore, il faut bien vérifier le principe de substitution : ces méthodes doivent fonctionner au moins dans les cas prévus par les méthodes correspondantes de `A` et se comporter « au moins » de la même façon qu'elles !

Tester la classe `PointNomme`

Exercice

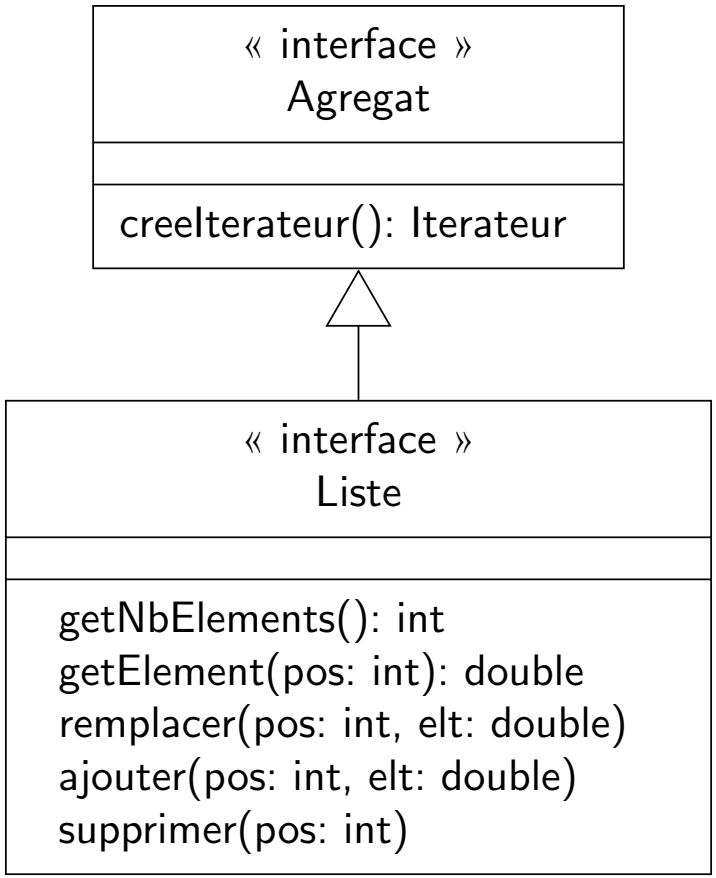
On suppose que l'on dispose d'une classe de test `PointTest` pour JUnit. On cherche maintenant à développer une classe de test pour `PointNomme`.

- qu'impose le principe de substitution ?
- comment développer une classe de test `PointNommeTest` sans faire de copier/coller ?

Plan de la partie 9 - Spécialisation et héritage

Héritage entre interfaces

Une interface peut hériter d'une autre interface : l'interface fille hérite du contrat de l'interface mère.



Héritage entre interfaces

Si une classe réalise l'interface `Liste`, elle devra implanter la méthode `creeIterateur`.

Remarque

Dans le TP précédent, si on avait eu cette relation d'héritage entre `Liste` et `Agregat`, on aurait pu écrire directement un itérateur pour tous les types de listes.

Comme une interface définit un type, le **principe de substitution** s'applique également dans ce cas.

Héritage entre interfaces

En Java, on utilise également le mot-clé **extends** :

Syntaxe (héritage entre interfaces)

```
public interface I extends I1, I2, ... ...
```

Par exemple :

Liste.java

```
public interface Liste extends Agregat {
  ...
}
```

Remarque (importante)

Une interface peut hériter de **plusieurs** interfaces.

Plan de la partie 9 - Spécialisation et héritage

Le modifieur `final`

Le modifieur **`final`** donne le sens d'immuable, de non modifiable à une caractéristique d'une classe.

Il est utilisé :

- pour une **variable locale** : c'est une constante (qui doit donc être initialisée lors de sa déclaration) ;
- pour un **attribut d'instance ou de classe** (qui doit donc être nécessairement initialisé par une valeur par défaut) ;
- une **méthode** : la méthode ne peut pas être redéfinie par une sous-classe. Elle n'est pas polymorphe ;
- une **classe** : la classe ne peut pas être spécialisée. Elle n'aura aucun descendant.

La classe `Object`

En Java, une classe qui n'a pas de classe parente hérite implicitement la classe `Object`. C'est l'ancêtre commun à toutes les classes.

Elle contient en particulier les méthodes :

- **`public boolean`** `equals(Object obj)` : égalité de **`this`** et de `obj` (par défaut, égalité des adresses). Elle doit (normalement...) être redéfinie
- **`protected`** `Object clone()` : elle est normalement telle que `x != x.clone()`, mais `x.equals(x.clone())`
- **`public`** `String toString()` : chaîne de caractères décrivant l'objet. Elle est utilisé dans `print`, `println` et l'opérateur de concaténation +
- **`protected void`** `finalize()` : méthode appelée lorsque le ramasse-miettes récupère la mémoire d'un objet
- **`public final`** `Class getClass()` : renvoie un objet représentant la **classe** de l'objet

assertEquals dans JUnit

Exercice

Nous avons développé une méthode `equals(Orbite o)` dans la classe `Orbite`.
Lorsque l'on développait des tests unitaires avec JUnit, on pouvait vérifier que deux orbites avec des caractéristiques identiques étaient égales, via la méthode `assertEquals` de JUnit.
Or cela ne fonctionnait pas. Pourquoi ?

On donne une version expurgée de la méthode `assertEquals` :

```
public static void assertEquals(Object expected, Object actual) {
  if (actual.equals(expected)) {
    // test OK!
  } else {
    // test failed!
  }
}
```

Héritage ou composition ?

Pour l'héritage...

- la sous-classe a le **type** de sa super-classe
- la sous-classe **hérite** de l'implantation d'une partie de sa super-classe

Pour la composition...

- beaucoup plus simple
- on n'a parfois pas besoin du type de la super-classe
- les changements sur l'interface d'une super-classe ou d'une sous-classe peuvent être problématiques et compliqués à gérer
- on peut changer dynamiquement l'objet composé

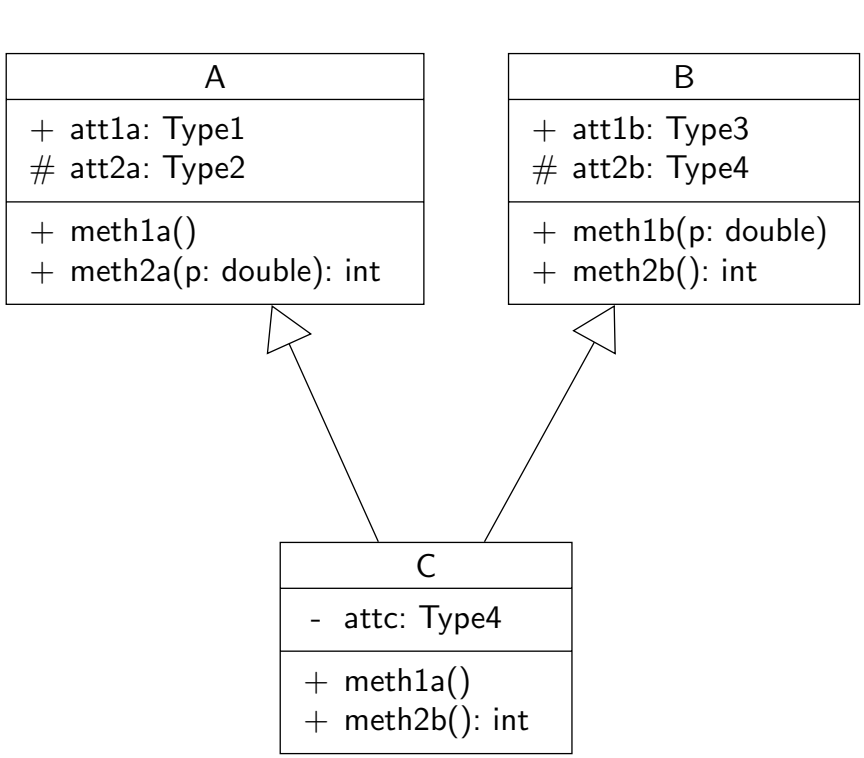
Héritage ou composition ?

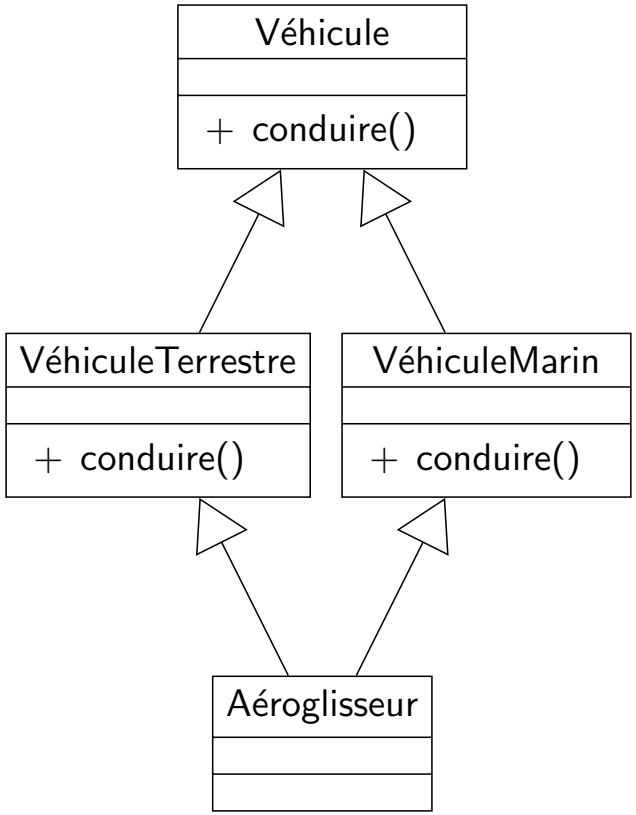
Conseil

N'utiliser l'héritage que lorsque l'on en a **réellement** besoin ! La réutilisation de code peut être faite via la composition par exemple.

Héritage multiple

Il est autorisé en UML

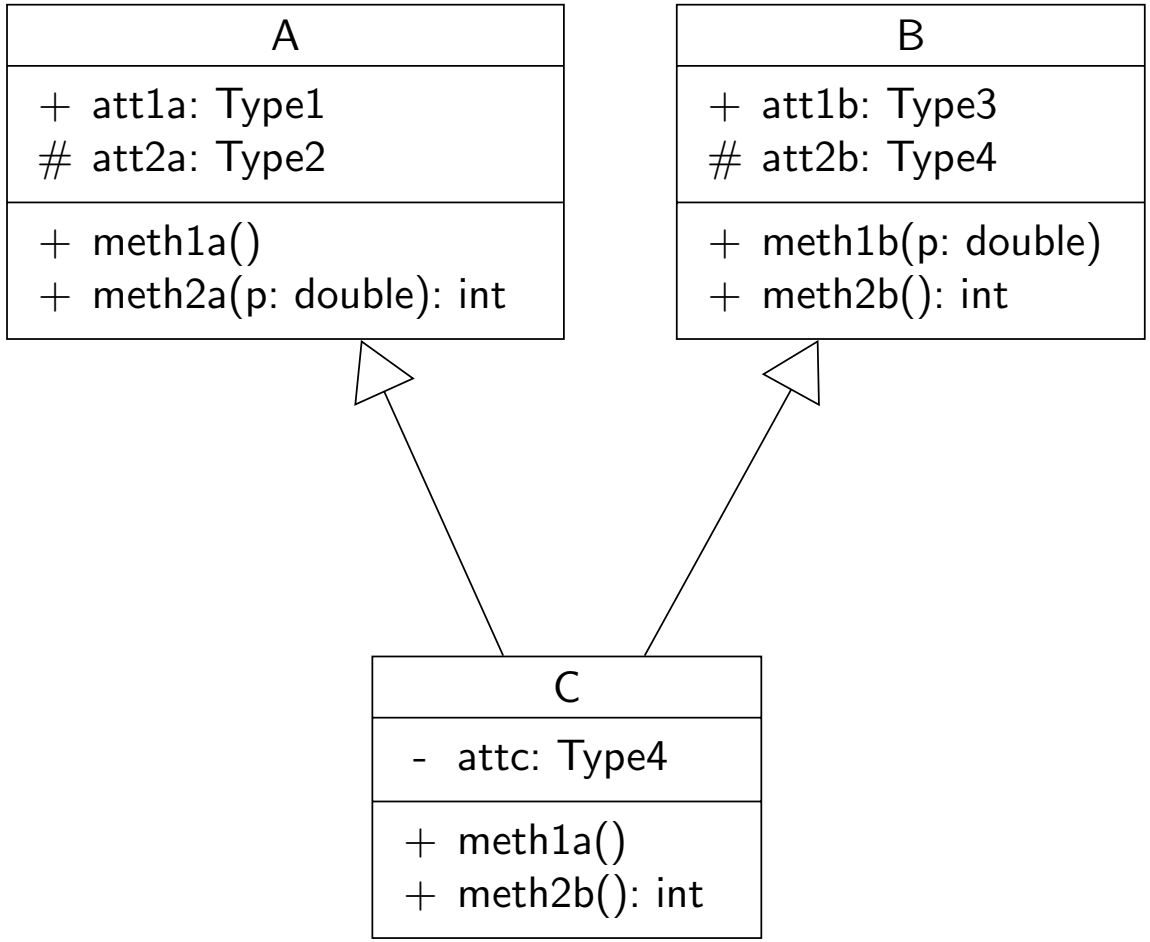




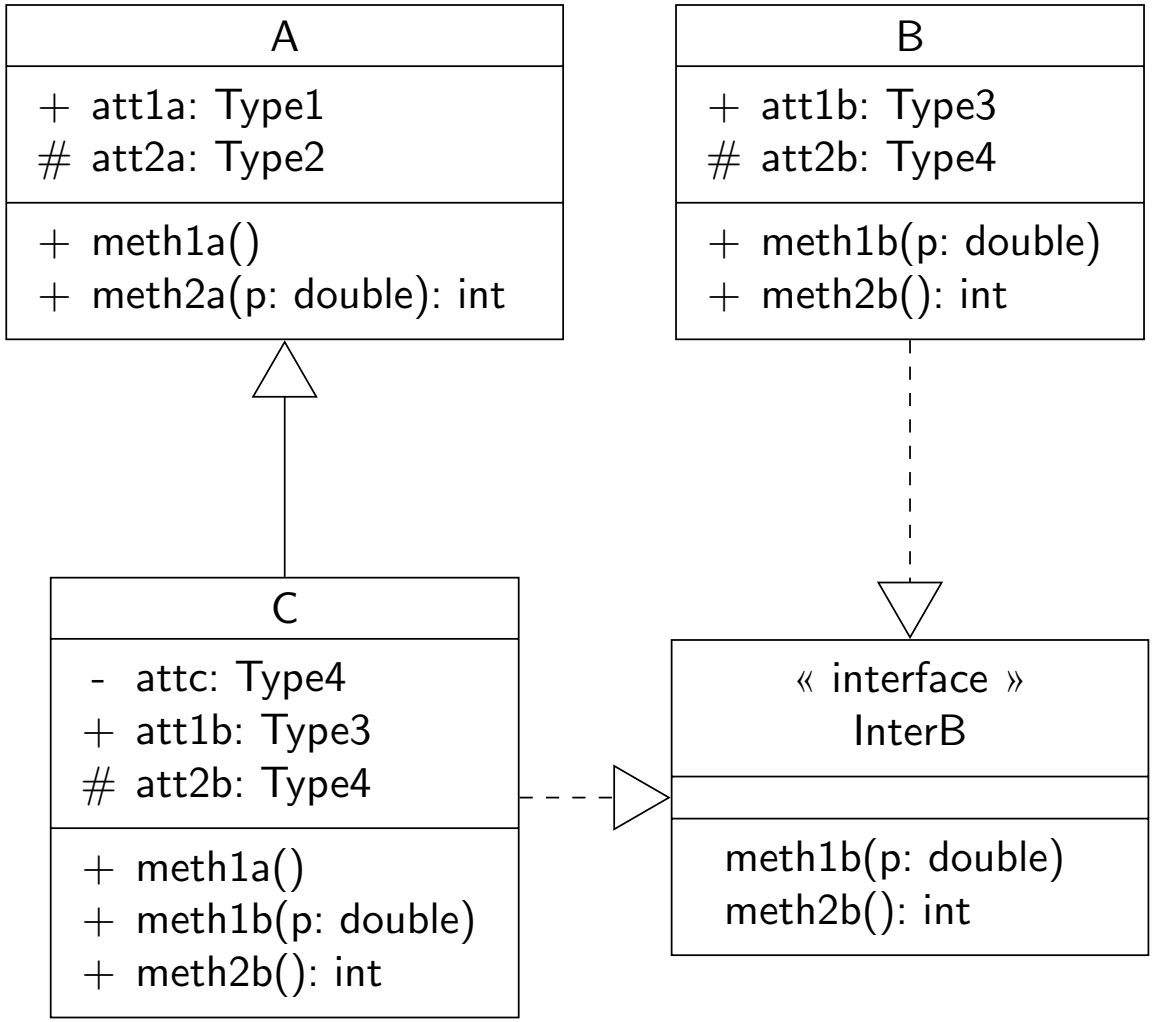
Il n'est pas autorisé en Java, mais est autorisé en C++ (problème des structures en « diamant »).

Héritage multiple

Les interfaces permettent de simuler un héritage multiple :



Héritage multiple



10 - Généralisation – classes abstraites

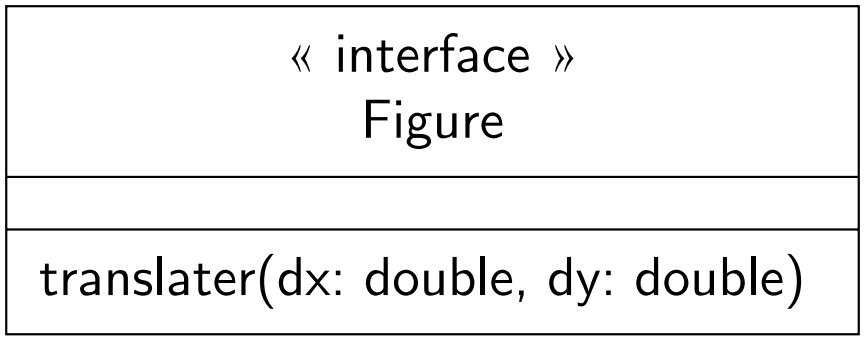
Introduction

Supposons que l'on cherche à regrouper dans une entité `Figure` les caractéristiques communes aux points, aux polygones, aux segments etc.

Les points, polygones et segments ont un **comportement commun** : on peut tous les translater.

Or on ne sait pas écrire de façon générique le code de la méthode `translater` : il dépend de la figure concernée.

On pourrait donc concevoir `Figure` comme une interface :



Introduction

Mais puisque l'on a défini une méthode `toString` dans toutes les classes, on peut écrire une méthode `afficher` générique :

```
public void afficher() {
  System.out.println(this);
}
```

On peut également supposer que l'on veuille introduire un **nouvel attribut pour toutes les figures**, la couleur par exemple.

`Figure` n'est donc

- ni une classe (on ne sait pas écrire `translater`)
- ni une interface (on peut écrire `afficher` et ajouter des attributs)

On va dire que `Figure` est une classe **abstraite** et que `translater` est une méthode **abstraite** de `Figure`.

Remarque

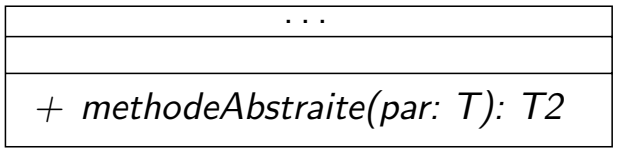
Le processus qui consiste à regrouper des caractéristiques communes à plusieurs classes existantes dans une super-classe s'appelle la **généralisation** (par opposition à la spécialisation).

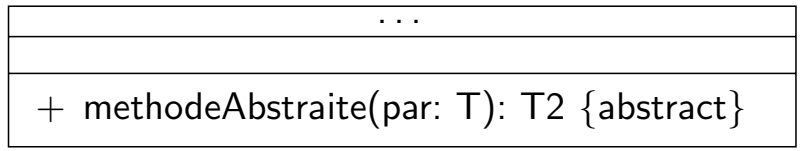
Méthode abstraite

Définition (méthode abstraite)

Une méthode **abstraite** est une méthode dont on ne sait pas écrire le code.

Notation UML :





Méthode abstraite

Traduction en Java :

Syntaxe (méthode abstraite)

```
abstract public T2 methodeAbstraite(T par) ;
```

On ne met pas d'accolades après une méthode abstraite.

Remarque

abstract est incompatible avec les modifieurs **final** et **static**, car une méthode retardée est nécessairement une méthode d'instance polymorphe.

Classe abstraite

Définition (classe abstraite)

Une classe abstraite est une classe que l'on ne peut pas instancier.

Principe

Une classe possédant une méthode abstraite est nécessairement une classe abstraite.

C'est normal, car le comportement représenté par les méthodes abstraites n'est pas défini.

Les méthodes abstraites devront donc être redéfinies dans les descendants de la classe.

On peut évidemment définir un constructeur pour une classe abstraite si celui-ci a un sens.

Classe abstraite

Notation UML :

Notation Java :

Syntaxe (classe abstraite)

```
abstract public class { ... }
```

Redéfinition des méthodes abstraites

Attention lors de la redéfinition de méthodes abstraites :

MaClasseAbstraite.java

```
abstract public class MaClasseAbstraite {
    abstract public void methode();
}
```

MaClasseConcrete.java

```
public class MaClasseConcrete extends MaClasseAbstraite {
    public void method() { }
}
```

shell

```
[tof@suntof]~ $ javac MaClasseConcrete.java
MaClasseConcrete.java:1: error: MaClasseConcrete is not abstract and
does not override abstract method methode() in MaClasseAbstraite
public class MaClasseConcrete extends MaClasseAbstraite {
       ^
1 error
```

Redéfinition des méthodes abstraites

Attention lors de la redéfinition de méthodes abstraites :

MaClasseAbstraite.java

```
abstract public class MaClasseAbstraite {
    abstract public void methode();
}
```

MaClasseConcrete.java

```
public class MaClasseConcrete extends MaClasseAbstraite {
    public void method() { }
}
```

D'où l'intérêt d'utiliser `@Override`...

Intérêt des classes abstraites

Intérêts

- **factoriser** des comportements communs même si on n'est pas capable de les coder. Ce comportement peut être utilisé ;
- **classifier** les objets : figures générales, figures fermées, figures ouvertes ;
- permettre au compilateur de **vérifier** que la définition des méthodes abstraites héritées est bien donnée dans les sous-classes ;
- **remarque importante** : une sous-classe d'une classe abstraite ne redéfinissant pas une des méthodes abstraites de sa classe mère doit être déclarée comme abstraite (sinon, il y a une erreur à la compilation).

Interfaces et classes abstraites

Classe abstraite

- une classe abstraite peut posséder des attributs ;
- une classe abstraite peut avoir des méthodes possédant une implantation ;
- par contre, on est obligé d'utiliser une relation d'héritage (attention en Java).

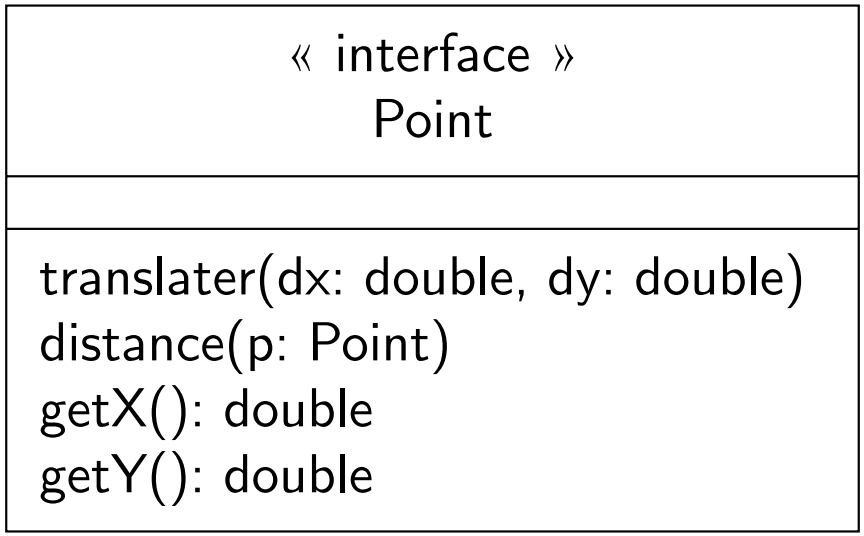
Interface

- on peut faire de l'héritage multiple entre interfaces ;
- l'interface permet de définir un type au sens des services rendus par une classe ;
- par contre, elle ne possède ni attributs (sauf les constantes) ni de méthodes implantées.

Le choix dépend du problème !

`Point` : interface ou classe abstraite ?

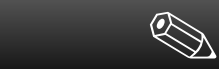
Nous avons défini `Point` comme une interface avec les méthodes suivantes :



Exercice

Peut-on faire de `Point` une classe abstraite ? Quel en est l'intérêt ?

Modélisation d'une constellation de satellites

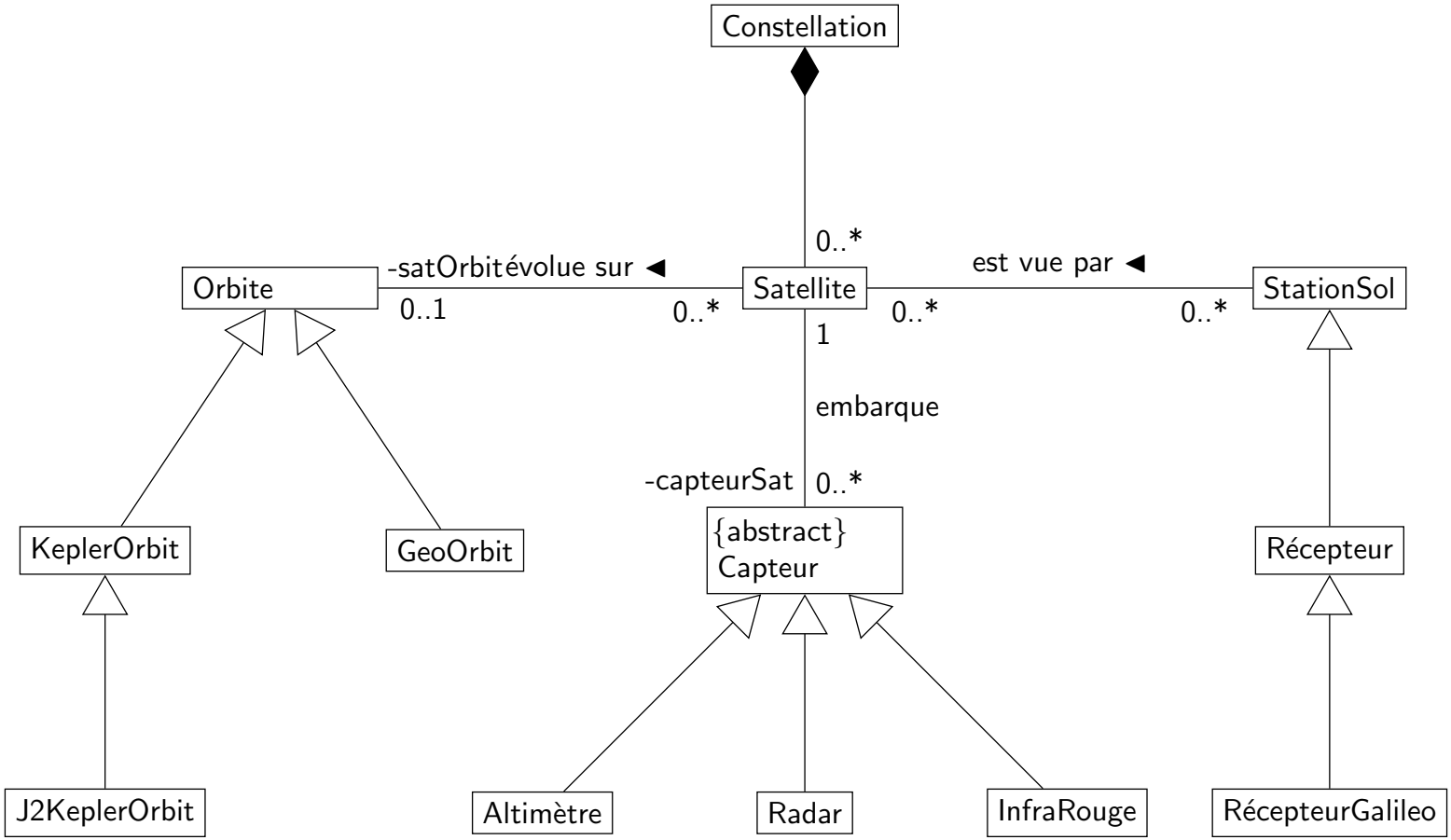


Exercice (modélisation du système Galileo)

Les systèmes de navigation comme GPS ou Galileo utilisent une constellation de satellites. Une constellation est constituée d'un ensemble de satellites. On peut considérer qu'une constellation ne possède pas de satellites si ceux-ci ne sont pas encore lancés. Une constellation est caractérisée par sa taille, et on peut ajouter un satellite dans une constellation si celle-ci n'est pas encore complète.
Chaque satellite évolue sur une orbite. Il existe plusieurs types d'orbites différentes : par exemple, une orbite géostationnaire ou une orbite de Kepler sont deux orbites particulières. Une orbite de Kepler de type J2 est elle même une orbite de Kepler particulière.
Un satellite peut embarquer des capteurs comme par exemple un radar, un altimètre ou un capteur infra-rouge.
Un satellite est caractérisé par une position courante, une vitesse courante et une date courante. On peut demander à un satellite de renvoyer toutes les informations concernant sa position.
Sur la Terre, les satellites d'une constellation peuvent être vus par des stations sol. Celles-ci ont une position représentée par des coordonnées cartésiennes. Une station sol peut être par exemple un récepteur, qui peut lui même être un récepteur Galileo ou GPS.

Proposer un diagramme UML d'analyse du problème.

Modélisation d'une constellation de satellites



11 - Interfaces graphiques

Introduction

Une interface graphique est une façade pour une application.

Elle permet à un utilisateur externe de demander des services à l'application via une vue plus intuitive.

Nous ne nous intéresserons pas ici aux aspects « ergonomie » des interfaces qui sont toutefois très importants (poste de commande d'une centrale, cockpit d'avion, . . .).

Plan de la partie 11 - Interfaces graphiques

Un exemple d'application

CalculRacinesCarrees.java

```
import java.util.SortedMap;
import java.util.TreeMap;

public class CalculRacinesCarrees {

    private SortedMap<Double, Double> racines;

    public void calculRacines(double inf, double sup, double pas) {
        this.racines = new TreeMap<Double, Double>();

        double d = inf;

        while (d < sup) {
            this.racines.put(d, Math.sqrt(d));
            d = d + pas;
        }
    }

    public SortedMap<Double, Double> getRacines() {
        return this.racines;
    }
}
```

Première interface utilisant la console

InterfaceConsoleRacinesCarrees.java

```
import java.util.SortedMap;

public class InterfaceConsoleRacinesCarrees {
    private CalculRacinesCarrees calculRacines;

    public void calculerRacinesCarrees() {
        double inf = console.Console.readDouble("Borne inferieure  :");
        double sup = console.Console.readDouble("Borne superieure  :");
        double pas = console.Console.readDouble("Pas               :");

        CalculRacinesCarrees rc = new CalculRacinesCarrees();
        rc.calculRacines(inf, sup, pas);

        SortedMap<Double, Double> tab = rc.getRacines();

        afficheEntete(tab); // to correctly print headers

        for (double d : tab.keySet()) {
            System.out.printf("| %1$23.16f | %2$23.16f |\n",
                              d, tab.get(d));
        }
    }
```

Première interface utilisant la console

InterfaceConsoleRacinesCarrees.java

```
    public static void main(String[] args) {
        new InterfaceConsoleRacinesCarrees().calculerRacinesCarrees();
    }

    private static void afficheEntete(SortedMap<Double, Double> tab) {


        ...
    }
}
```

Première interface : exécution

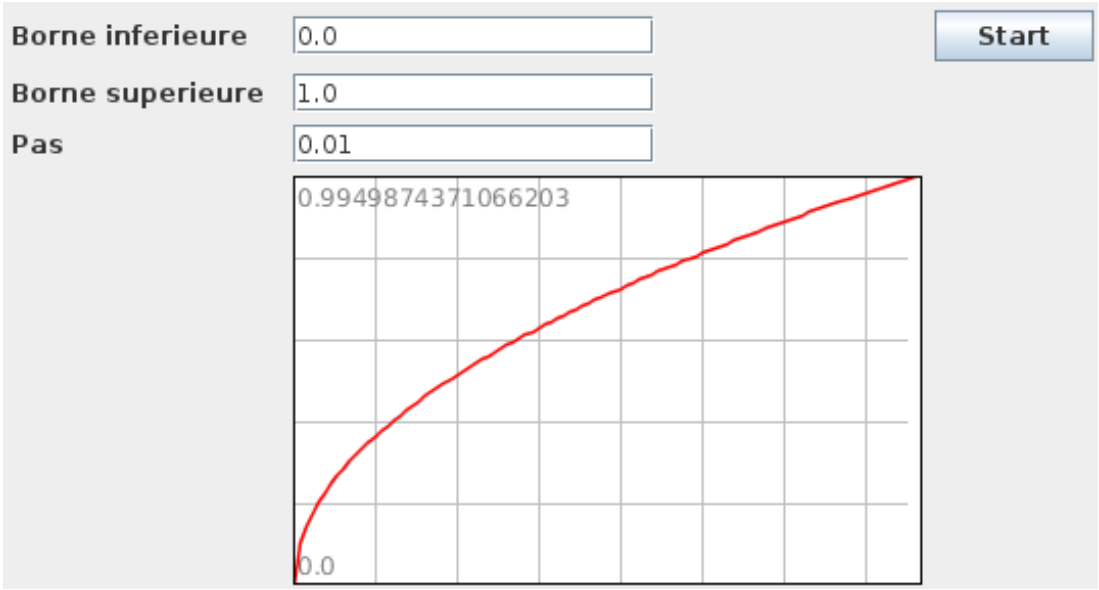
On peut utiliser cette interface pour calculer et afficher des racines carrées :

shell

```
[tof@suntof]~ $ java InterfaceConsoleRacinesCarrees
Borne inferieure  : 0
Borne superieure  : 1
Pas               : 0.1

| nombre                  | racine                  |
|-------------------------+-------------------------|
|      0.0000000000000000 |      0.0000000000000000 |
|      0.1000000000000000 |      0.3162277660168379 |
|      0.2000000000000000 |      0.4472135954999579 |
|      0.3000000000000000 |      0.5477225575051662 |
|      0.4000000000000000 |      0.6324555320336759 |
|      0.5000000000000000 |      0.7071067811865476 |
|      0.6000000000000001 |      0.7745966692414834 |
|      0.7000000000000001 |      0.8366600265340756 |
|      0.8000000000000000 |      0.8944271909999159 |
|      0.9000000000000000 |      0.9486832980505138 |
```

Seconde interface : une interface graphique

On peut également construire une interface « graphique » pour le même service :



Interface graphique : le « code » de la fenêtre

InterfaceGraphiqueRacinesCarrees.java

```
public class InterfaceGraphiqueRacineCarree extends JFrame {
    // pour les calculs
    private CalculRacinesCarrees racineCalcul;

    // composants graphiques standards
    private JTextField fieldInf;
    private JTextField fieldSup;
    private JTextField fieldPas;

    private JButton start;

    // un composant graphique capable d'afficher les points
    // contenu dans le tableau renvoye par racineCalcul
    private MyJPanel affichageCourbe;
```

Interface graphique : le « code » de la fenêtre

InterfaceGraphiqueRacinesCarrees.java

```
    public InterfaceGraphiqueRacineCarree(String titre) {
        super(titre);

        // creation d'une instance de RacineCarree
        // creation et placement des composants
        this.fieldInf = new JTextField(15);
        this.add(this.fieldInf);

        ...

        // on ecoute le bouton Start : si on appuie dessus
        // on execute le listener associe
        this.start.addActionListener(new Visualiser(this.racineCalcul,
                                                    this.affichageCourbe));
    }
}
```

Interface graphique : le « code » de l'écouteur

Visualiser.java

```
public class Visualiser implements ActionListener {
    // ce dont on a besoin pour lancer le calcul

    public Visualiser(CalculRacinesCarrees racineCalcul,
                      MyJPanel affichageCourbe) {
        // affectation des objets aux attributs
    }

    public void actionPerformed(actionEvent e) {
        // on recalcule les racines
        this.racineCalcul.calculRacines(inf, sup, pas);

        // on demande a l'affichage de se mettre a jour
        this.affichageCourbe.repaint();
    }
}
```

Et c'est tout ! En particulier, on ne gère pas directement les événements générés par l'utilisateur (changement d'apparence du bouton, quand appeler la méthode lançant les calculs etc).

Constat

On remarque que l'**application métier** permettant de calculer les racines carrées est décorrélée des différentes **présentations** des résultats et des **interactions** (contrôle) avec l'utilisateur :

- utilisation de la console ;
- utilisation d'une interface graphique.

Cette séparation nous permet de **minimiser les impacts d'une modification** d'une composante par rapport aux autres composantes. Par exemple :

- on peut changer la façon de calculer la racine carrée dans `RacineCarree` sans impacter les différentes présentations ;
- on peut ajouter facilement une nouvelle façon de présenter les résultats (serait-ce facile si le code métier était embarqué dans l'interface graphique par exemple ?).

Constat

On remarque que l'**application métier** permettant de calculer les racines carrées est décorrélée des différentes **présentations** des résultats et des **interactions** (contrôle) avec l'utilisateur :

- utilisation de la console ;
- utilisation d'une interface graphique.

Principe

Ce constat devrait être le même pour n'importe quelle application : on **doit** séparer ces trois composantes !

Modèle MVC

Nous allons utiliser un patron, appelé **Modèle-Vue-Contrôleur**, pour séparer les trois composantes :

- **Modèle** : maintient la représentation du domaine et la logique métier et fournit un accès à celles-ci. Il ne connaît pas les vues et les contrôleurs qui lui sont associés ;
- **Vue** : représentation visuelle d'une partie des données (ou de toutes les données) ;
- **Contrôleur** : gère les événements et change le modèle et éventuellement la vue en réponse à ces événements.

Burbeck, Steve (1987, 1992).
Applications Programming in Smalltalk-80(TM) : How to use Model-View-Controller (MVC).
`http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html`.

Exemple de fonctionnement d'un MVC

- l'utilisateur appuie sur `Start` ;
- le contrôleur intercepte cet événement et met à jour le modèle en conséquence via `calculeRacineCarree` ;
- le modèle met à jour ses vues.

Quelques remarques sur le MVC

Mise à jour de la vue

- si le modèle est passif, le contrôleur s'occupe de mettre à jour la vue ;
- si le modèle est actif, la vue est mise à jour via le patron Observateur.

Quelques remarques sur le MVC

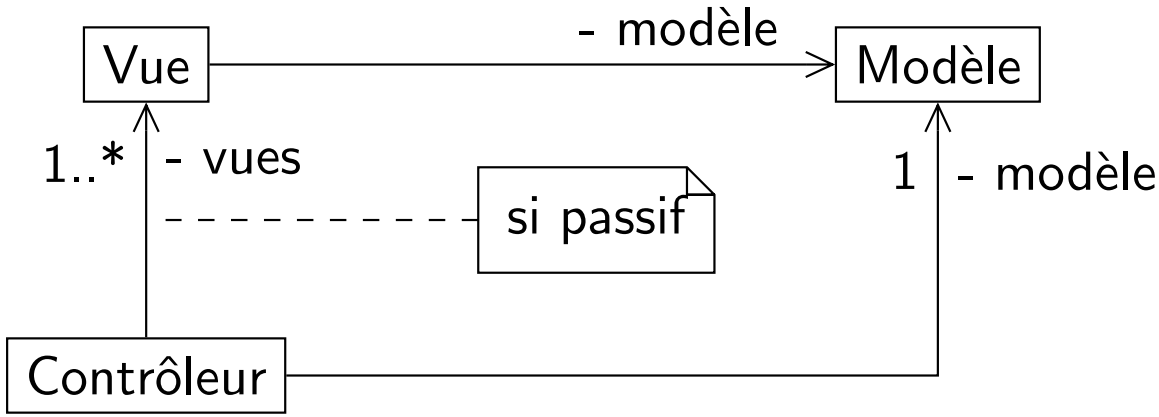
Avantages du MVC

- modèle très stable : on n'a pas tendance à le reconstruire lorsque l'on veut reconstruire la vue ;
- plusieurs vues pour un même modèle peuvent coexister en même temps ;
- test facilité avec la séparation modèle/vue.

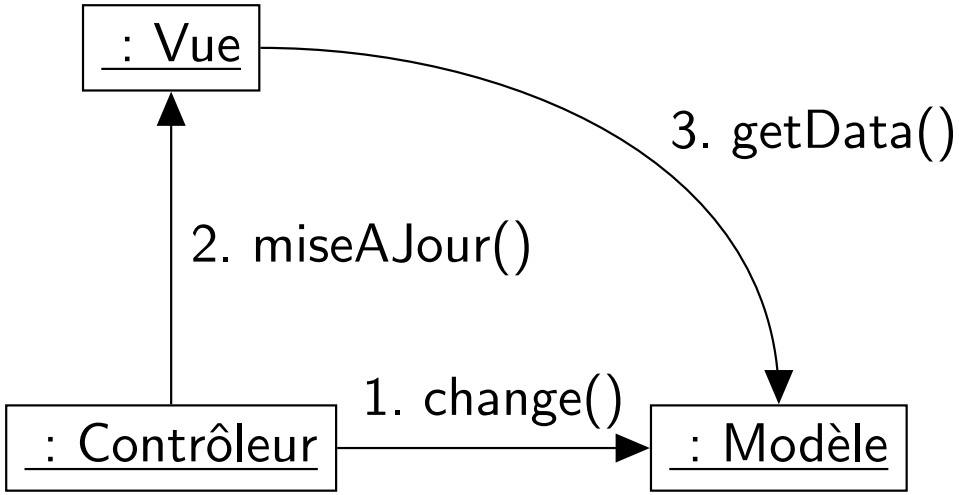
Inconvénients du MVC

- complexité de l'application (un grand nombre de patrons sont utilisés en réalité).
- la séparation entre contrôleur et vue est difficile.

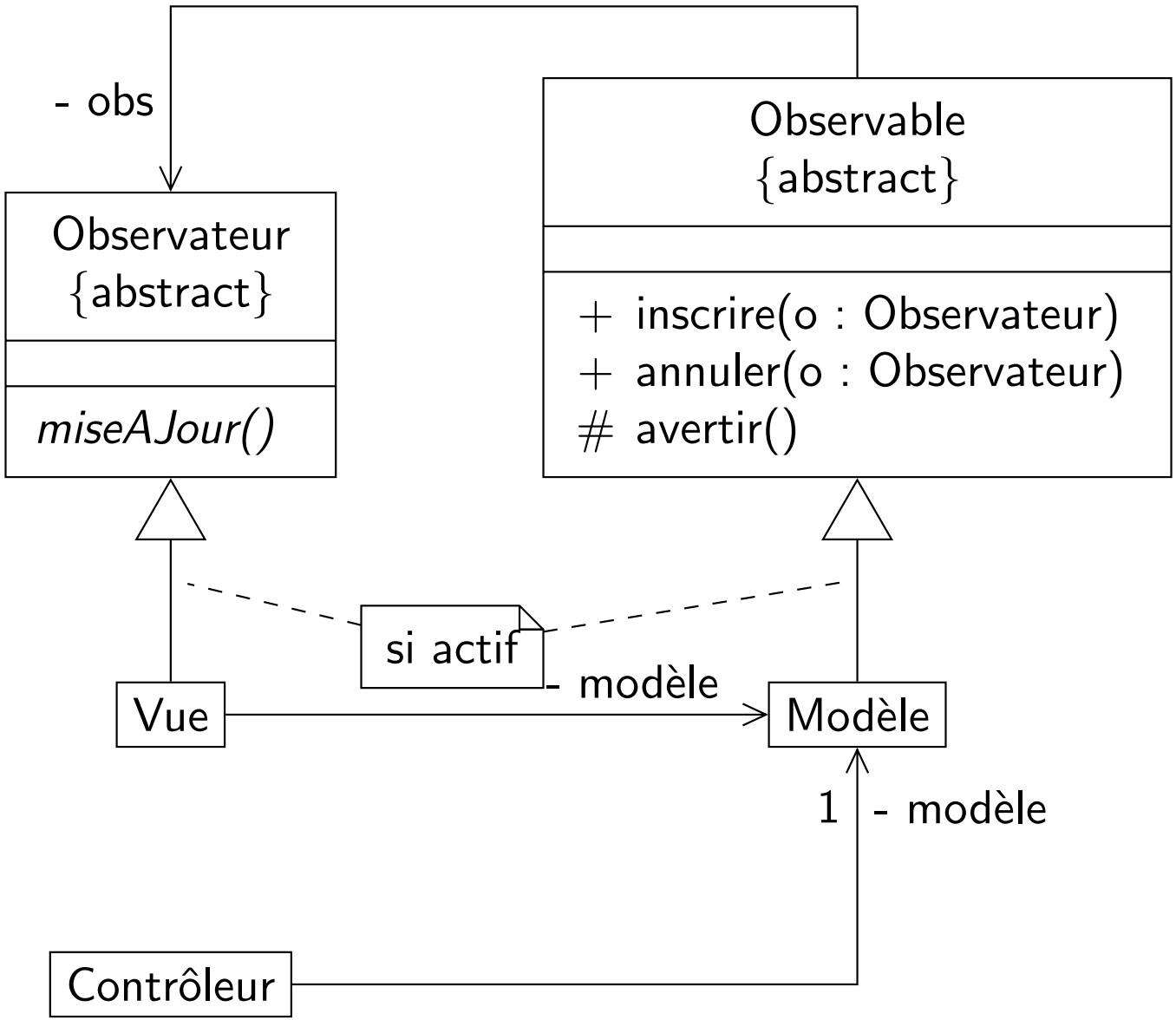
MVC : un diagramme de classe avec un modèle passif



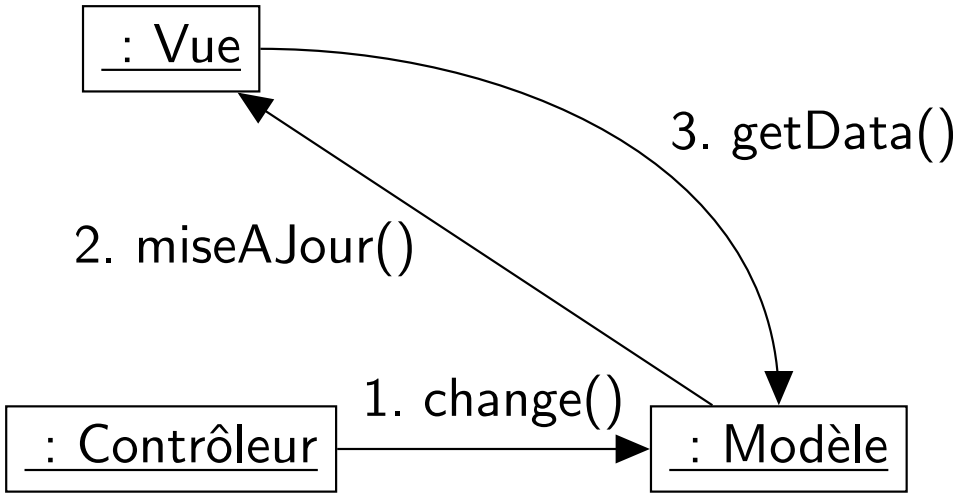
MVC : un diag. de collaboration avec un modèle passif



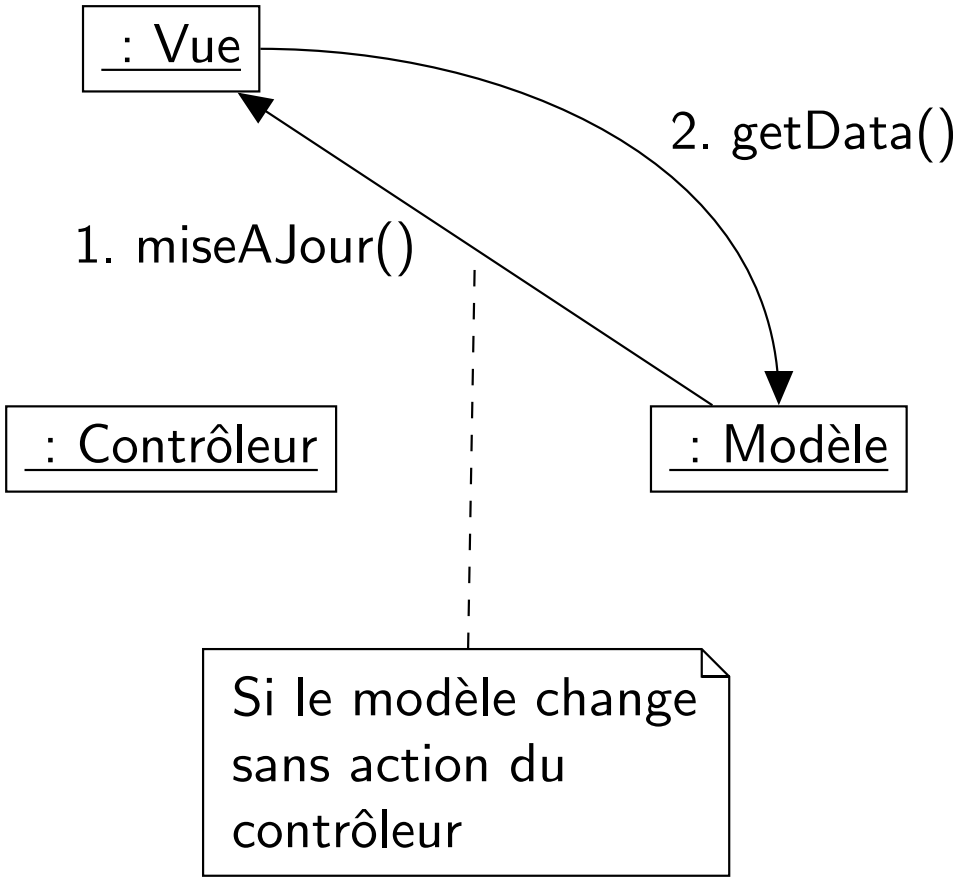
MVC : un diagramme de classe avec un modèle actif



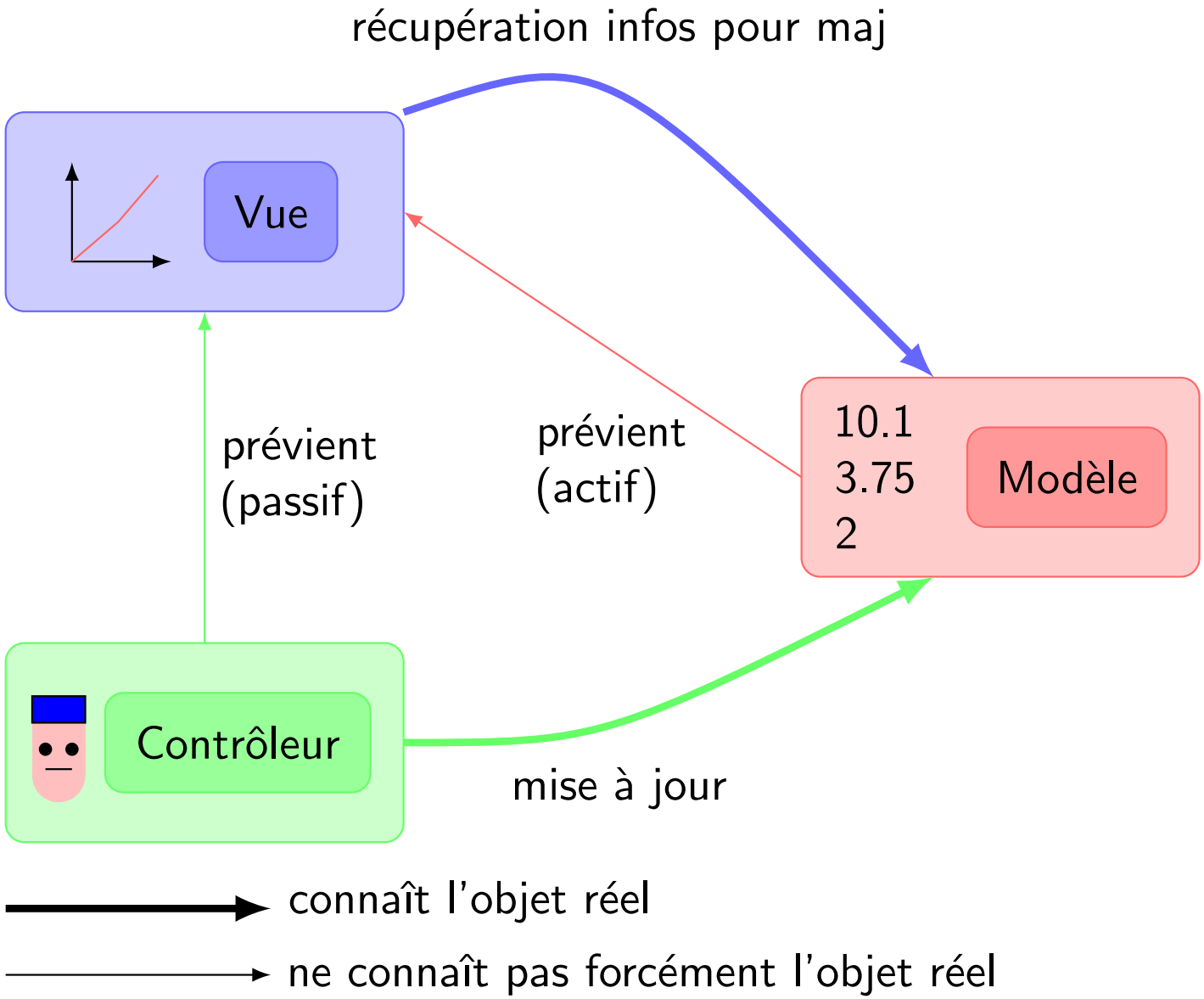
MVC : un diag. de collaboration avec un modèle actif



MVC : un diag. de collaboration avec un modèle actif



MVC : résumé


récupération infos pour maj
Vue
prévient
(passif)
prévient
(actif)
10.1
3.75
2
Modèle
Contrôleur
mise à jour
connaît l'objet réel
ne connaît pas forcément l'objet réel

Plan de la partie 11 - Interfaces graphiques

Historique de Swing

Premier ensemble de classes développées par Sun Microsystems : AWT (*Abstract Window Toolkit*)
Ces classes ont été développées pour construire de petites interfaces graphiques, pour les *applets* par exemple.

Il y a des fonctionnalités manquantes (*clipboard*, support de l'impression etc.), des composants manquants (*popup menus*, *scrollpanes* etc.) et le modèle événementiel est « mal » construit.

Historique de Swing

AWT fonctionne avec des **composants lourds** (*heavyweight components*) fondés sur le principe de *peer* :

- un *peer* est un composant natif d'interface graphique qui effectue un certain nombre de tâches comme se dessiner, réagir aux événements ;
- les classes de AWT n'avaient donc plus grand chose à faire (ex : `java.awt.Panel` doit faire douze lignes de code) ;

Problèmes

- chaque composant dessiné utilise une fenêtre graphique native, ce qui devient vite très pénalisant en terme de performances ;
- il est très difficile d'obtenir un comportement cohérent des composants sur toutes les plateformes disponibles ;
- les développeurs ont passé beaucoup de temps sur des composants natifs buggués.

Historique de Swing

Swing a été conçu en suivant plusieurs buts :

- entièrement en Java
- une seule API avec plusieurs look-and-feels
- programmation par les modèles
- principes des JavaBeans pour l'intégration aux IDE
- compatibilité avec AWT

On ne manipule que quelques composants lourds (les plus importants).

La plupart des composants sont *lightweight* : ils n'ont pas de *peer* et sont contenus dans des composants légers ou lourds.

Les composants légers ne dépendent pas d'une fenêtre graphique native. Ils peuvent donc avoir un fond transparent par exemple.

Le principe d'interaction avec l'interface graphique est fondé sur le modèle MVC.

Contenu de Swing

Il existe un grand nombre de composants dans Swing : fenêtres, boîtes de dialogue, boutons, menus, *scrollpanes* etc.

On peut choisir un *look-and-feel* (Windows, Motif, Java etc.).

Swing n'est disponible qu'à partir de la version 1.2 du JDK (attention aux applets).

Remarque (paquetage)

Tous les composants de Swing sont contenus dans `javax.swing`.

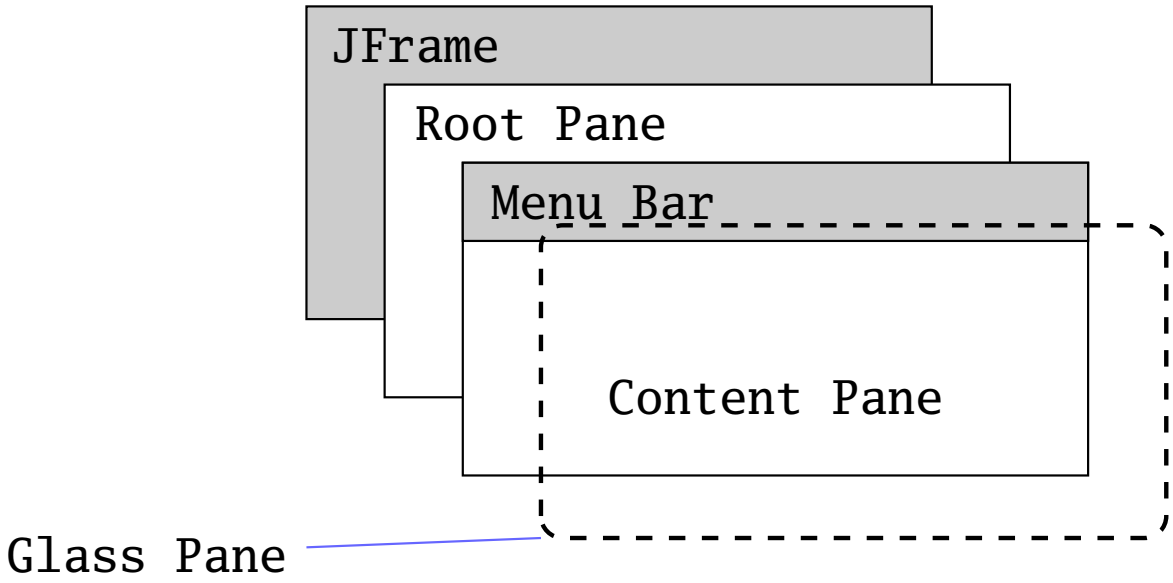
Plan de la partie 11 - Interfaces graphiques

Composants lourds

Dans Swing, quatre composants lourds (ou *top-level containers*) sont disponibles :

- `JFrame` pour une application « classique » ;
- `JApplet` pour une applet avec une interface graphique ;
- `JDialog` pour une boîte de dialogue ;
- `JWindow` pour une fenêtre sans décoration.

Pour une `JFrame` :



Le JRootPane et le ContentPane

Un conteneur lourd n'est jamais accessible directement. Par contre, il possède des objets qui permettent d'interagir avec lui.

En particulier, il possède un objet de type `JRootPane` à qui il délègue ses opérations. Le `JRootPane` contient plusieurs objets :

- un objet de type `java.awt.Container` appelé `contentPane` qui contient les composants du conteneur. C'est lui que nous allons utiliser pour ajouter des boutons, des zones de texte etc.
- un objet de type `Component` appelé `glassPane` qui « flotte » au dessus des autres composants, peut intercepter des événements et être transparent. Il peut être utilisé pour faire apparaître des commentaires par exemple sur le contenu de la fenêtre.
  Nous ne l'utiliserons pas ici.

Système natif et Swing

Remarque

L'aspect graphique « extérieur » d'une `JFrame` (bordure, boutons de fermeture et d'agrandissement de la fenêtre etc.) est délégué au système sous-jacent (composant lourd).

Créer une JFrame

MonApplicationGraphique.java

```
import javax.swing.JFrame;

public class MonApplicationGraphique {
    public static void main(String[] args) {
        JFrame fen = new JFrame("Essai de JFrame");

        // on demande a l'application de s'arreter lorsque l'on
        // ferme la fenetre
        fen.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // on place correctement les eventuels composants
        fen.pack();

        // on rend la fenetre visible
        fen.setVisible(true);
    }
}
```

Créer une `JFrame`

Résultat de l'exécution :

Évidemment, il y a normalement les décorations du système d'exploitation (boutons de fermeture et autres, titre de la fenêtre, etc.).

Créer « proprement » une application graphique

La classe précédente n'est pas satisfaisante : on crée l'instance de `JFrame` directement dans le `main` !

Une solution plus « propre » :

- créer une classe (par exemple `MyFrame`) héritant de `JFrame`
- déclarer comme attributs les composants graphiques que l'on veut utiliser dans la fenêtre
- initialiser les composants via le constructeur de `MyFrame` :
  - directement
  - en appelant une méthode privée. Cette solution est souvent utilisée par les IDE, la méthode en question s'appelant `initComponents`
- créer une instance de `MyFrame` dans un programme

Créer « proprement » une GUI : exemple

MyFrame.java

```
import javax.swing.JFrame;

public class MyFrame extends JFrame {
    // declarer les composants de la fenetre comme attributs

    public MyFrame() {
        this.initComponents();

        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.pack();
        this.setVisible(true);
    }

    private void initComponents() {
        // creer et placer les composants
    }
}
```

Lancer l'application

Pour lancer l'application, il suffit de créer une instance de `MyFrame` :

TestMyFrame.java

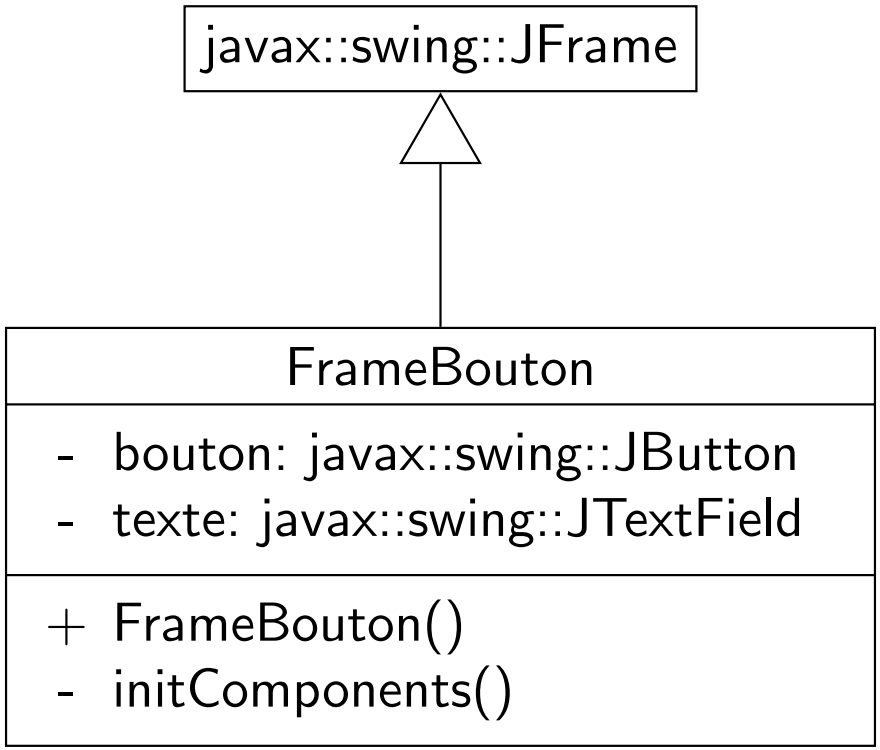
```
public class TestMyFrame {
    public static void main(String[] args) {
        new MyFrame();
    }
}
```

Un premier exemple de vue

On souhaite construire une vue contenant :

- un bouton
- un champ dans lequel on peut introduire du texte

La classe correspondante que nous allons construire s'appelera `FrameBouton` :



Récupérer le contentPane

La méthode getContentPane permet de récupérer l'objet de type ContentPane associé à la fenêtre. C'est sur objet que l'on positionne le *layout manager* et que l'on place des composants (cf. plus loin).

FrameBouton.java

```
private void initComponents() {
    Container c = this.getContentPane();
}
```

Placer les composants

Pour placer les composants dans la fenêtre, il faut utiliser un `LayoutManager` qui va permettre de préciser comment ajouter les composants dans le `contentPane`.

Nous choisissons ici d'utiliser un `FlowLayout` qui nous permet de « ranger » les composants en lignes :

FrameBouton.java

```
private void initComponents() {
    Container c = this.getContentPane();

    c.setLayout(new FlowLayout());
}
```

Ajouter les composants

On ajoute ensuite un `JButton` et un `JTextField` dans la fenêtre :

FrameBouton.java

```
private void initComponents() {
    Container c = this.getContentPane();

    c.setLayout(new FlowLayout());

    c.add(new JButton("Go!"));
    c.add(new JTextField(20));
}
```

Ajouter les composants

On ajoute ensuite un `JButton` et un `JTextField` dans la fenêtre :

FrameBouton.java

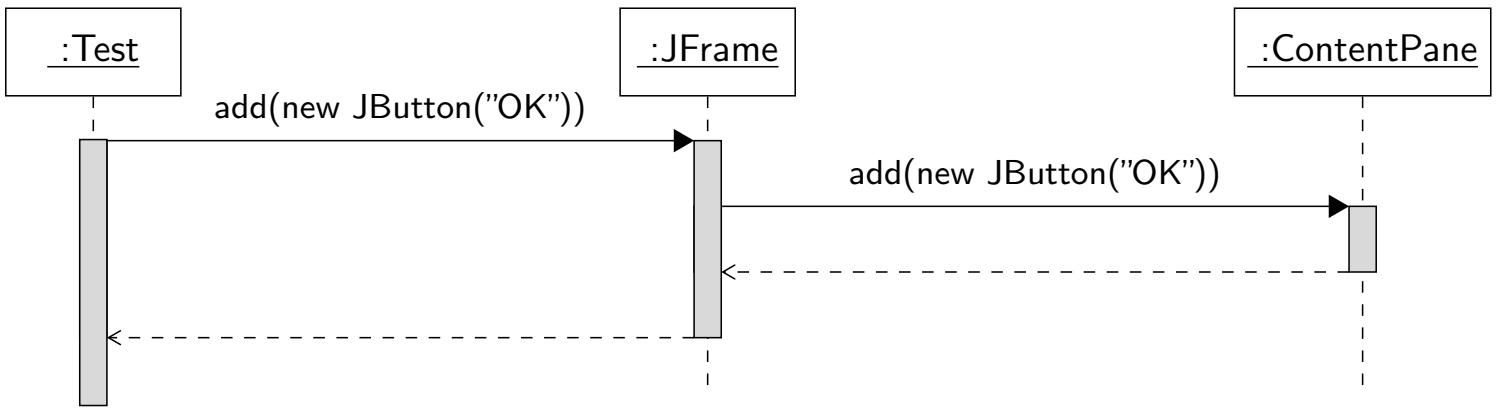
```
private void initComponents() {
    Container c = this.getContentPane();

    c.setLayout(new FlowLayout());

    c.add(new JButton("Go!"));
    c.add(new JTextField(20));
}
```

Remarque

Attention, dans le diagramme de classes, les composants sont des attributs de la classe. Il faut alors les initialiser (cf. version finale de la classe).

ContentPane et JFrame

Depuis la version 5 du JDK, le services offerts par le `ContentPane` d'une `JFrame` (`add`, `setLayout` etc.) sont accessibles via des méthodes de même nom dans `JFrame` (délégation de service) :



On aurait donc pu par exemple coder `initComponents` comme suit :

```
private void initComponents() {
    this.setLayout(new FlowLayout());

    this.add(new JButton("Go!"));
    this.add(new JTextField(20));
}
```

Écrire un contrôleur pour l'application

On souhaiterait maintenant que lorsque l'on appuie sur le bouton, on récupère le texte introduit dans le champ et qu'on l'affiche dans la console.

Pour cela, on va utiliser un **listener** qui va attendre que l'on appuie sur le bouton `Go` pour effectuer l'opération.

Le type de *listener* correspondant à cette action est un objet de type `java.awt.event.ActionListener`.

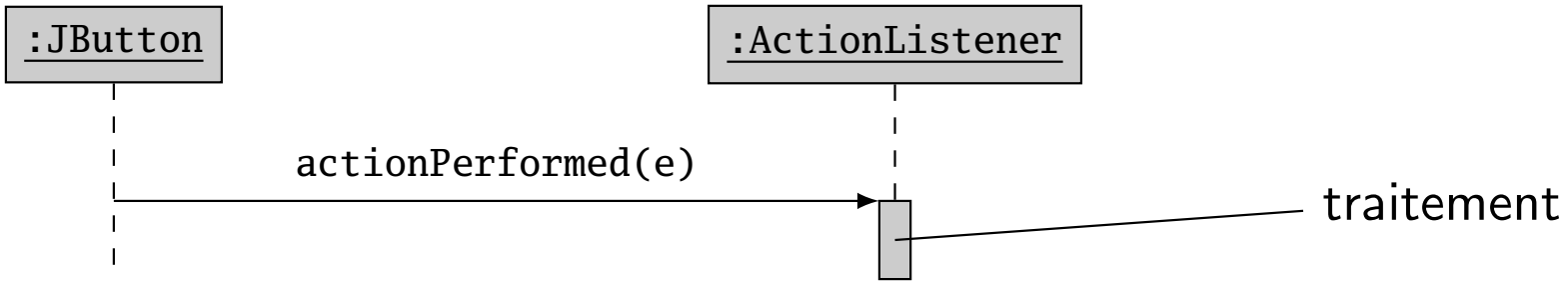
Écrire un contrôleur pour l'application

`java.awt.event.ActionListener` possède une seule méthode :

`java.awt.event.ActionListener`

```
public interface ActionListener {
  public void actionPerformed(java.awt.event.ActionEvent e);
}
```

C'est cette méthode qui sera appelée lorsque l'on appuie sur le bouton.



Écrire un *listener* pour le bouton

Le contrôleur qui nous intéresse serait donc une instance de la classe suivante :

ListenerBouton.java

```
import java.awt.event.*;
import javax.swing.*;

public class ListenerBouton implements ActionListener {

    private JTextField texte;

    public ListenerBouton(JTextField texte) {
        this.texte = texte;
    }

    public void actionPerformed(ActionEvent e) {
        System.out.println(texte.getText());
    }
}
```

Inscrire le *listener* sur le bouton

FrameBouton.java

```
import javax.swing.*;
import java.awt.*;

public class FrameBouton extends JFrame {

    private JTextField texte;
    private JButton bouton;

    public FrameBouton() {
        super("Essai de JFrame");

        this.initComponents();

        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.pack();
        this.setVisible(true);
    }

    private void initComponents() {
        this.setLayout(new FlowLayout());

        this.add((this.bouton = new JButton("Go!")));
        this.add((this.texte = new JTextField(20)));

        this.bouton.addActionListener(new ListenerBouton(this.texte));
    }
}
```

Plan de la partie 11 - Interfaces graphiques

Composants pouvant servir de conteneurs intermédiaires

Un certain nombre de composants légers peuvent servir de conteneurs intermédiaires :

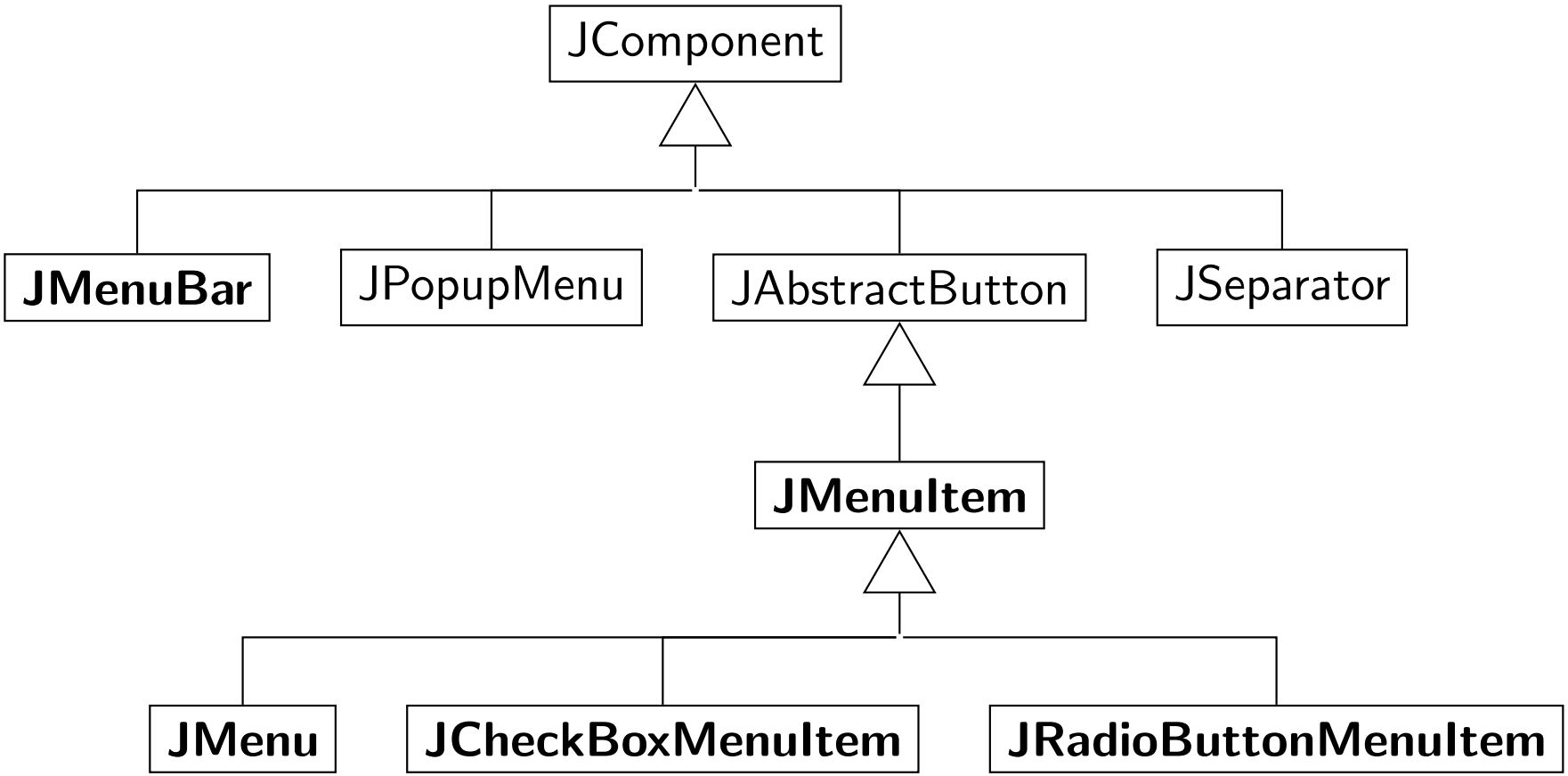
`JPanel`	le composant le plus flexible
`JScrollPane`	« ascenseur »
`JSplitPane`	affiche deux composants dont on peut changer la taille
`JTabbedPane`	« intercalaires »
`JToolBar`	groupe plusieurs composants dans une colonne ou une ligne
`JInternalFrame`	composant léger possédant les propriétés d'une `JFrame` : *dragging*, menu etc.

Composants atomiques

`JButton`	et ses dérivés...
`JColorChooser`	choisir une couleur dans une palette
`JComboBox`	« menu déroulant »
`JLabel`	affichage de texte et d'images
`JList`	choisir dans une liste d'objets
`JProgressBar`	barre de progression
`JSlider`	choix d'une valeur numérique sur une barre
`JTable`	tableau de données
`JTextComponent`	manipulation de texte (éditeur...)
`JTree`	données sous forme hiérarchique
`JFileChooser`	fenêtre de sélection d'un fichier

Les menus

Certains composants comme les instances de `JFrame` peuvent avoir une barre de menu.



Création et gestion des menus

Création d'une barre de menu et ajout d'un menu :

```
JMenuBar barre = new JMenuBar();
JMenu menu = new JMenu("File");
barre.add(menu);
```

Ajout de deux items et d'un séparateur :

```
menu.add(new JMenuItem("Quitter"));
menu.addSeparator();
menu.add(new JCheckBoxMenuItem("Ready"));
```

Ajout du menu à une instance `fen` de `JFrame` :

```
fen.setJMenuBar(barre);
```

Plan de la partie 11 - Interfaces graphiques

LayoutManager fournis

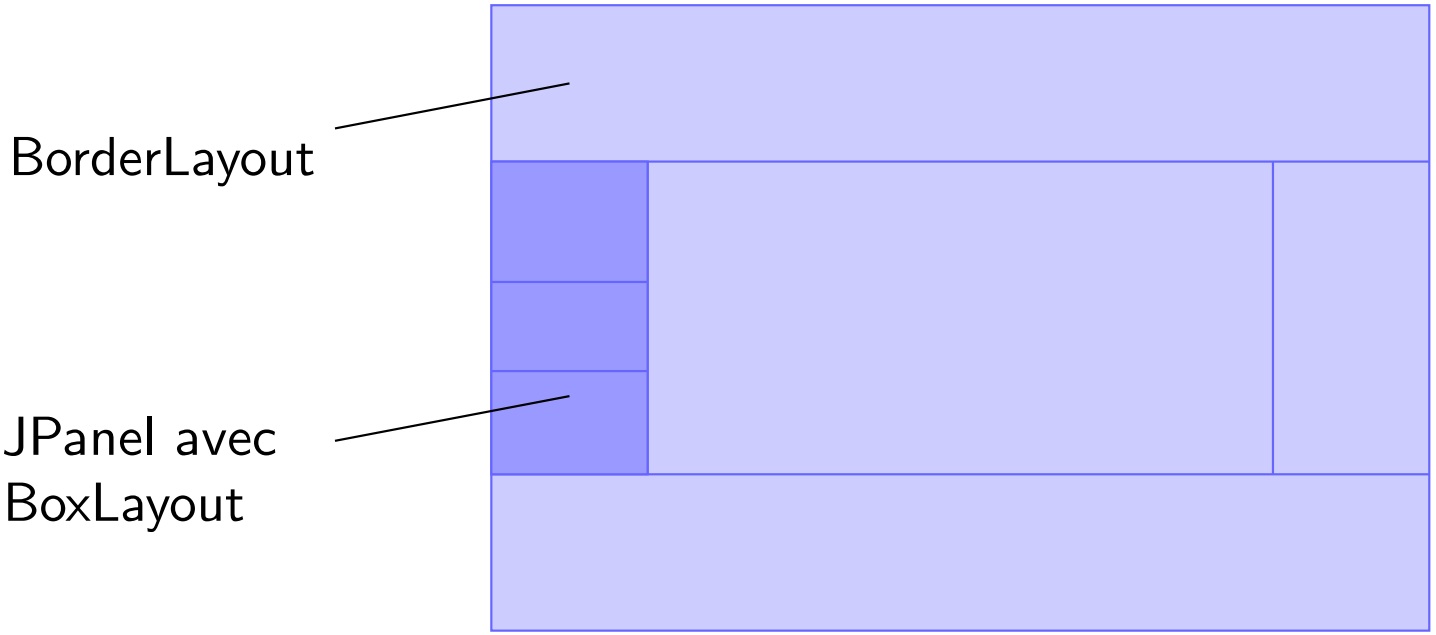
Un certain nombre de *LayoutManager* permettant de positionner des composants sont fournis. En voici une présentation rapide :

`FlowLayout`	tous les composants sont mis sur une seule ligne
`GridLayout`	les composants sont dans un tableau. On utilise la taille préférée du plus gros composant pour calculer la taille des cellules.
`BorderLayout`	quatre positions (N, S, E, W) et un emplacement central
`CardLayout`	les composants sont « empilés » les uns sur les autres et on peut choisir quel est le composant affiché
`BoxLayout`	une seule colonne ou une seule ligne
`GridBagLayout`	complétement personnalisable

Il faut regarder la documentation Javadoc de ces classes pour en comprendre le fonctionnement. En particulier, `GridBagLayout` est un *LayoutManager* puissant, mais la lecture de sa documentation est nécessaire pour en comprendre le fonctionnement.

Créer un layout compliqué

Pour créer un *layout* compliqué, on peut utiliser des *containers* intermédiaires comme les `JPanel` pour « composer » les layouts.



Comment fonctionne le calcul de la taille d'une fenêtre ?

Supposons que nous ayons une instance de `JFrame` possédant un certain nombre de composants. Voici ce qui se passe :

1. un appel à `pack` sur l'instance de `JFrame` demande à la fenêtre d'être à sa taille préférée ;
2. pour cela, le `contentPane` doit calculer sa taille. Le `LayoutManager` lui fournit ;
3. pour cela, le `LayoutManager` demande à chacun des composants sa taille préférée. Suivant sa nature, il calcule ensuite sa taille ;
4. pour chaque composant, soit l'utilisateur a précisé une taille préférée via la méthode `setPreferredSize(Dimension d)`, soit on utilise la taille fournie par le *look and feel* utilisé.

Lorsqu'un composant change de taille, le `LayoutManager` adapte sa taille en conséquence.

Plan de la partie 11 - Interfaces graphiques

Gestion d'événements

Pour pouvoir réaliser un contrôleur, nous allons utiliser la **programmation événementielle** : des actions vont être effectuées lorsque certains événements sont déclenchés (on sort alors du modèle de programmation « classique »). On parle alors de fonctions ou de méthodes de **callback**.

Avec Swing, quand l'utilisateur presse un bouton ou bouge la souris, un **événement** se produit.

Un objet peut être notifié de l'événement : il doit être enregistré comme un **event listener** auprès de la source de l'événement (un composant Swing comme un `JButton` par exemple).

Les listeners disponibles

Les différents *listeners* disponibles sont :

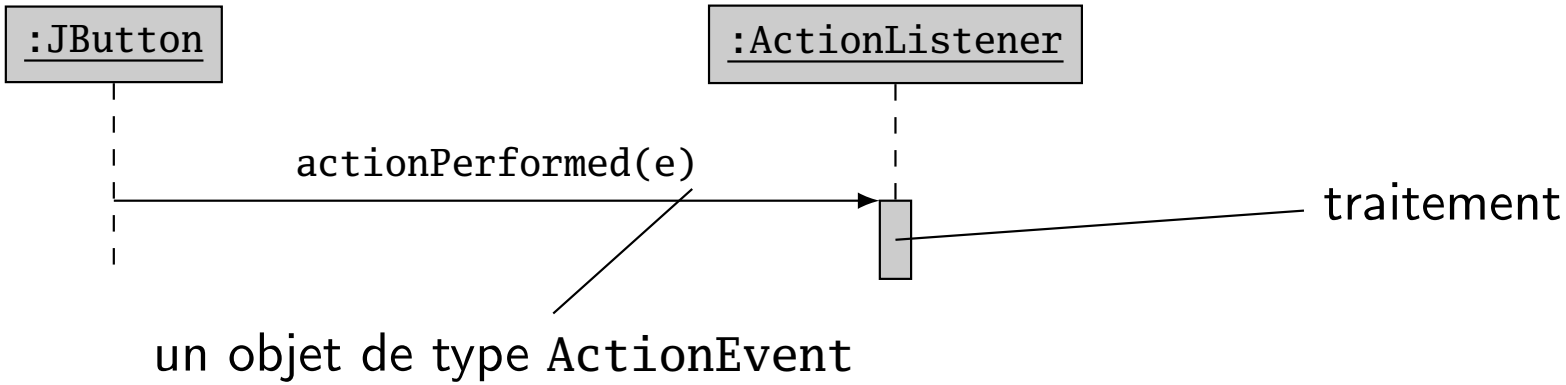
`ActionListener`	appui sur un bouton, `Return`, choix dans un menu
`WindowListener`	fermeture d'une `Frame`
`MouseListener`	appui sur le bouton de la souris
`MouseMotionListener`	l'utilisateur bouge la souris
`ComponentListener`	le composant devient visible
`FocusListener`	le composant a le focus (clavier)
`ListSelectionListener`	la sélection d'une liste change

Ils appartiennent tous au paquetage `java.awt.event`.

Gestion d'événements : exemple

Reprenons l'exemple vu précédemment concernant l'appui sur un bouton.

1. l'appui sur le bouton génère un événement de type `ActionEvent` ;
2. si un objet de type `ActionListener` est enregistré auprès du bouton, la méthode `actionPerformed` du listener est invoquée ;



L'objet de type `ActionEvent` permet de **récupérer des informations** et possède des méthodes intéressantes :

- `getModifiers()` permet de savoir si `Shift` était pressée en même temps ;
- `getSource()` permet de récupérer l'objet à l'origine de l'événement (le bouton ici).

Réaliser un *listener*

Pour pouvoir réaliser un *listener*, il faut **implanter une classe réalisant l'interface** correspondant à l'événement que l'on veut intercepter. Par exemple :

ListenerBouton

```
import java.awt.event.*;
import javax.swing.*;

public class ListenerBouton implements ActionListener {

    private JTextField texte;

    public ListenerBouton(JTextField texte) {
        this.texte = texte;
    }

    public void actionPerformed(ActionEvent e) {
        System.out.println(texte.getText());
    }
}
```

Enregistrer le *listener* auprès d'un composant

On l'**enregistre ensuite auprès de l'objet** que l'on veut « écouter » (attention au nom de la méthode d'enregistrement) :

FrameBouton.java

```
public class FrameBouton extends JFrame {

    private JTextField texte;
    private JButton bouton;

    private void initComponents() {

        ...

        this.bouton.addActionListener(new ListenerBouton(this.texte));
    }
}
```

Bien sûr, il est quasiment certain que le *listener* devra connaître le modèle et éventuellement la vue associée, donc il faudra prévoir des attributs en conséquence.

Classes internes

On voit que l'on doit écrire une classe pour chaque *listener* que l'on va vouloir utiliser sur une GUI. Cela peut devenir fastidieux, surtout en considérant que les *listeners* n'ont pas un « travail » important à fournir. De plus, les *listeners* sont fortement liés à la vue.

On peut en Java utiliser des classes **internes**, déclarées à l'intérieur d'une autre classe.

Classes internes : un exemple

FrameBoutonInterne.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class FrameBoutonInterne extends JFrame {

    private JTextField texte;
    private JButton bouton;

    public class ListenerBouton implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            System.out.println(texte.getText());
        }
    }

    public FrameBoutonInterne() {
        // idem FrameBouton


    }

    private void initComponents() {


        this.bouton.addActionListener(new ListenerBouton());
    }
}
```

Classes internes : caractéristiques

Les classes internes sont des **membres à part entière** de leur classe englobante comme les attributs et les méthodes.
Elles ont donc un accès **direct** à tous les éléments de la classe, y compris les caractéristiques privées !

Une instance de `ListenerBouton` est **toujours** liée à une instance de `FrameBoutonInterne`. Par exemple :

TestVueInterne.java

```
public class TestVueInterne {
    public static void main(String[] args) {
        FrameBoutonInterne vue = new FrameBoutonInterne();

        FrameBoutonInterne.ListenerBouton b = vue.new ListenerBouton();
    }
}
```

Classes internes : caractéristiques

Il existe également des classes internes qui sont statiques et sont donc attachées à une classe particulière.

À la compilation, on obtient un *bytecode* `FrameBoutonInterne.ListenerBouton.class`.

Remarque

Comme tous les éléments d'une classe, on peut donner une visibilité, même **`private`**, à une classe interne. Par exemple, est-il nécessaire d'avoir la classe `ListenerBouton` publique ?

Classes anonymes

On peut également créer des classes qui n'ont pas de noms, mais qui étendent une classe ou réalisent une interface. Par exemple :

dans FrameBouton.java

```
JButton bouton = new JButton("OK!");
bouton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    System.out.println(text.getText());
  }
 }
);
```

Ces classes n'ont pas de constructeurs, ne peuvent pas utiliser **`extends`** ou **`implements`** et ont accès aux champs de la classe englobante.

Les fichiers contenant les bytecodes des classes ont alors un nom anonyme : `FrameBoutonInterne$1.class` par exemple.

Classes anonymes

On peut également créer des classes qui n'ont pas de noms, mais qui étendent une classe ou réalisent une interface. Par exemple :

dans FrameBouton.java

```
JButton bouton = new JButton("OK!");
bouton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    System.out.println(text.getText());
  }
 }
);
```

Remarque

Attention à la lisibilité du code !

Classes Adapter

Il se peut qu'une interface `Listener` contienne plus d'une méthode.

Exemple : `MouseListener`

```
public void mousePressed(MouseEvent e);
public void mouseReleased(MouseEvent e);
public void mouseEntered(MouseEvent e);
public void mouseExited(MouseEvent e);
public void mouseClicked(MouseEvent e);
```

Si on réalise l'interface, on doit implanter les cinq méthodes.

Si une seule méthode est intéressante, on utilise la **classe** `MouseAdapter` que l'on étend.

Plan de la partie 11 - Interfaces graphiques

Plan de la partie 11 - Interfaces graphiques

Un MVC interne pour les composants ?

Est-ce que l'on peut modéliser les composants de Swing par un MVC ?
Par exemple :

1. un `JButton`

2. un `JSlider`

MVC interne de Swing

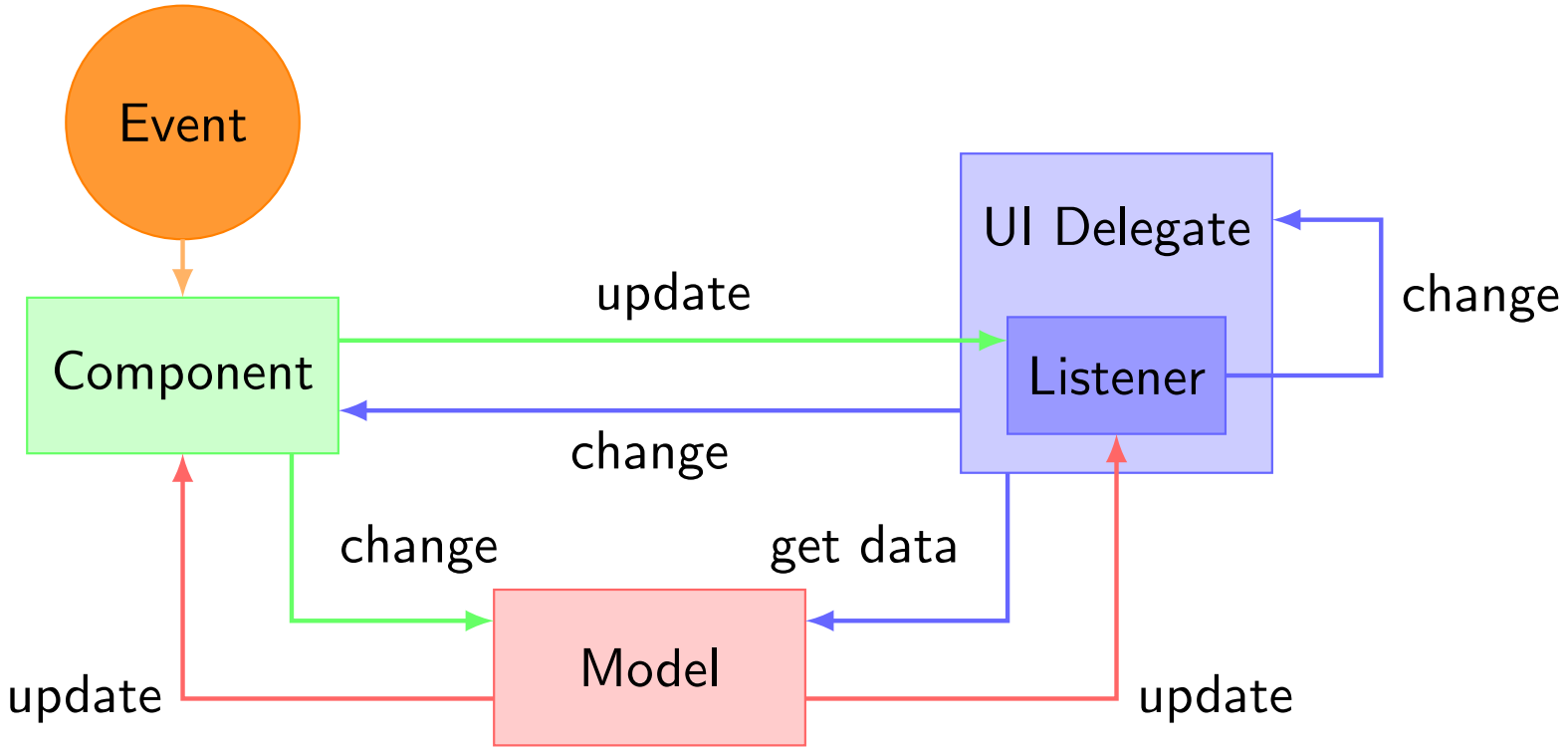
Tous les composants Swing sont gérés par un MVC « interne » :

- un **model** qui contient les données associées au composant (taille, valeurs minimales et maximales pour un `JSlider` par exemple) ;
- un **UI delegate** qui combine une vue et un contrôleur (appelé ici *listener*) : par exemple, lorsque l'on bouge le curseur d'un `JSlider`, la vue du composant change ;
- un **component** qui étend `JComponent`.

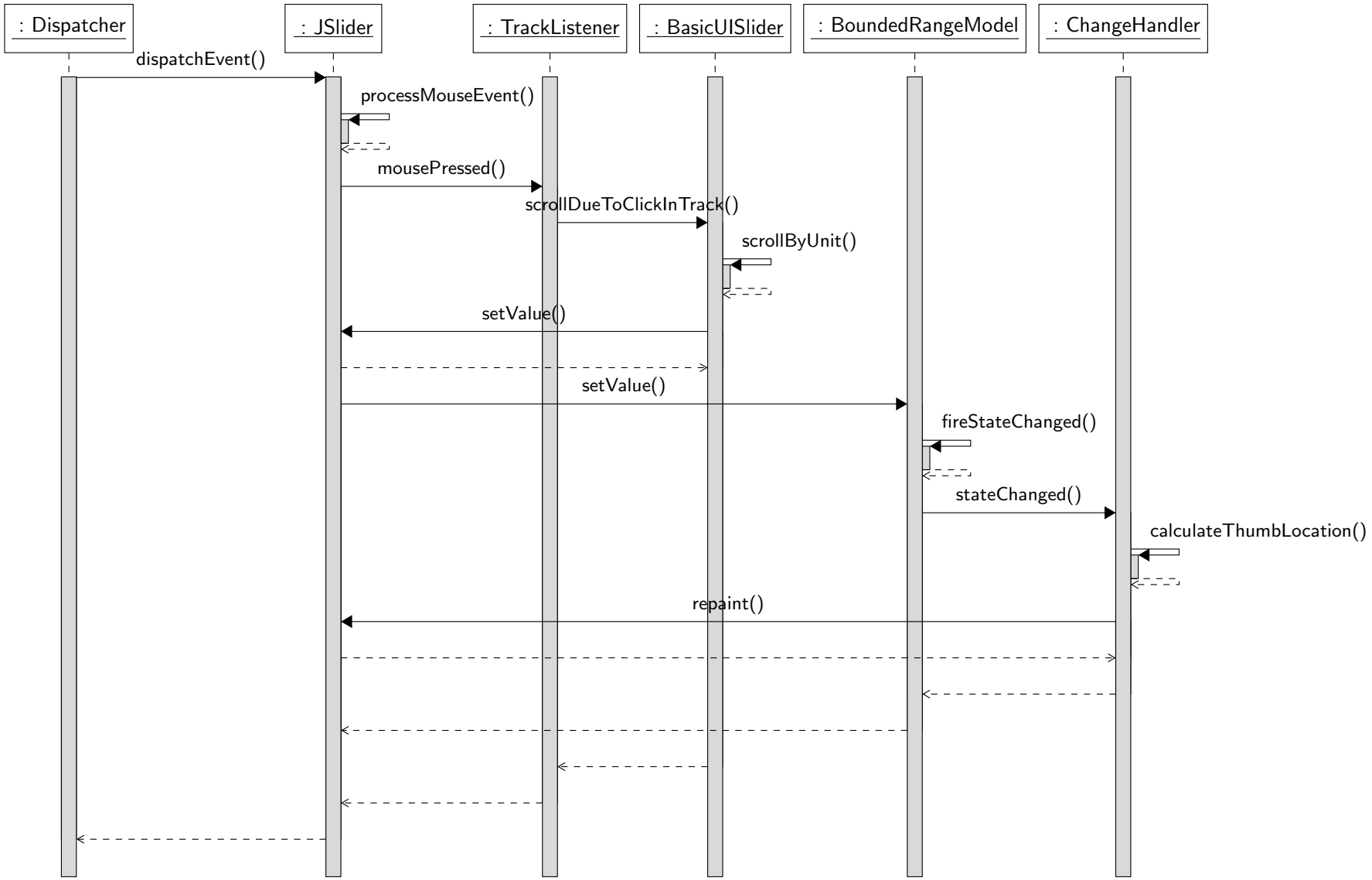
Le *component* est une API qui facilite le travail :

- création du *UI delegate* ;
- gestion du modèle ;
- changement des propriétés du modèle et mise à jour de la vue.

MVC interne de Swing : vue dynamique



MVC interne : exemple d'un JSlider



MVC interne de Swing : UI Delegate

On peut par exemple changer le UI Delegate d'un bouton :

```
import javax.swing.*;
import javax.swing.plaf.metal.MetalButtonUI;

public class TestButtonUI {
    public static final void main(final String[] args)
        throws InterruptedException {
        JFrame fen = new JFrame("Button UI");
        JButton b = new JButton("OK!");
        fen.add(b);
        fen.pack();
        fen.setVisible(true);

        Thread.sleep(5000);
        b.setUI(new BasicButtonUI());
    }
}
```

MVC interne : deux types de modèles

On trouve dans le MVC interne de Swing deux types de modèles :

1. des modèles dits d'**état d'interface graphique** qui définissent le statut du composant.
   Par exemple :
   - `ButtonModel` représente l'état d'un bouton (appuyé/relâché)
   - `BoundedRangeModel` représente l'état d'un `JSlider` (min/max/position du curseur)
2. des modèles dits de **données d'application** qui représente des données liées à l'application
   Par exemple :
   - `TreeModel` représente les données dans un `JTree`
   - `TableModel` représente les données dans une `JTable`

Certains modèles servent à la fois de modèles d'état d'IHM et de données, comme par exemple `BoundedRangeModel`.

Modèles internes : listeners

Les modèles du MVC interne peuvent notifier d'autres éléments de leurs changements en utilisant des *listeners*.

Par exemple, un objet peut être prévenu des changements d'un modèle d'un `JSlider` en utilisant un `ChangeListener` :

`ChangeListener`

```
public interface ChangeListener {
    void stateChanged(ChangeEvent e);
}
```

Il existe des *listeners* plus complexes pour d'autres modèles.

Fowler, Amy (2013).
A Swing architecture overview.
http://www.oracle.com/technetwork/java/architecture-142923.html.

Utilisation d'un modèle d'IHM : exemple

On souhaite utiliser un `JSlider` pour redimensionner une image :

Utilisation d'un modèle d'IHM : exemple

On va pour cela utiliser le modèle interne du `JSlider` :

- c'est une instance de `BoundedRangeModel`, donc on connaît la valeur min, la valeur max, la valeur actuelle
- le composant affichant l'image réalisera `ChangeListener`
- il sera donc prévenu lorsque l'utilisateur fait bouger le *slider*
- on récupérera la valeur actuelle du *slider* via le modèle
- on calculera la nouvelle taille de l'image
- on fera de même pour le `JLabel` affichant le pourcentage

Remarque

`BoundedRangeModel` est également un modèle d'application (valeurs min et max dépendant de l'application).

Utilisation d'un modèle d'IHM : code

SliderEtImage.java

```
 1 import javax.swing.*;
 2 import javax.swing.event.*;
 3 import java.awt.*;
 4
 5 public class SliderEtImage extends JFrame {
 6
 7     JSlider slider;
 8     JLabel valeurZoom;
 9
10     ImageIcon image;
11     ImageView imageView;
12
13     public SliderEtImage() {
14         this.initComponents();
15
16         this.pack();
17         this.setVisible(true);
18     }
```

Utilisation d'un modèle d'IHM : code

SliderEtImage.java

```
20      private void initComponents() {
21          this.slider = new JSlider(new
22                                    DefaultBoundedRangeModel(100, 0,
23                                                             0, 100));
24          this.valeurZoom = new JLabel("100%");
25
26          this.image = new ImageIcon("miaou.jpg");
27          this.imageView = new ImageView(image, this.slider.getModel());
28
29          JPanel panel = new JPanel();
30
31          panel.add(new JLabel("Taille de l'image :"));
32          panel.add(this.slider);
33          panel.add(this.valeurZoom);
34
35          this.add(panel, BorderLayout.NORTH);
36          this.add(imageView, BorderLayout.CENTER);
37
38          this.slider.addChangeListener(new ValeurZoomListener());
39      }
```

Utilisation d'un modèle d'IHM : code

SliderEtImage.java

```
41      class ValeurZoomListener implements ChangeListener {
42          public void stateChanged(ChangeEvent e) {
43              String s = Integer.toString(slider.getModel().getValue());
44              valeurZoom.setText(s + "%");
45              valeurZoom.revalidate();
46          }
47      }
48  }
```

Utilisation d'un modèle d'IHM : code

ImageView.java

```
 1 import javax.swing.*;
 2 import javax.swing.event.*;
 3 import java.awt.*;
 4
 5 public class ImageView extends JScrollPane implements ChangeListener {
 6     private JPanel panel = new JPanel();
 7     private Dimension tailleOrig = new Dimension();
 8     private Image image;
 9     private ImageIcon icone;
10
11     public ImageView(ImageIcon icone, BoundedRangeModel model) {
12         panel.setLayout(new BorderLayout());
13         panel.add(new JLabel(icone));
14
15         this.icone = icone;
16         this.image = icone.getImage();
17
18         tailleOrig.width = icone.getIconWidth();
19         tailleOrig.height = icone.getIconHeight();
20
21         setViewportView(panel);
22         model.addChangeListener(this);
23     }
```

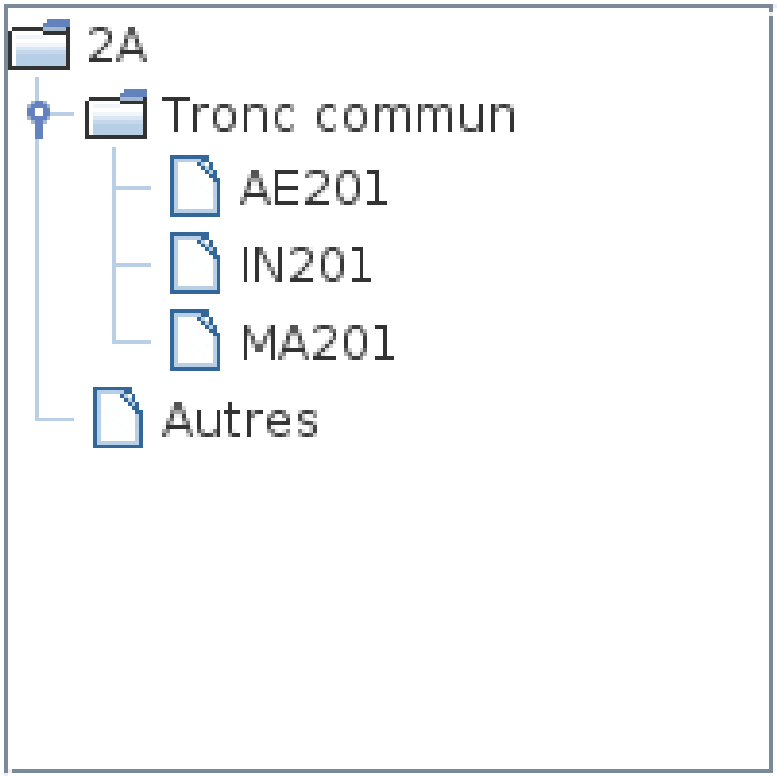
Utilisation d'un modèle d'IHM : code

ImageView.java

```
25      public void stateChanged(ChangeEvent e) {
26          BoundedRangeModel model = (BoundedRangeModel) e.getSource();
27
28          if (!model.getValueIsAdjusting()) {
29              int min = model.getMinimum();
30              int max = model.getMaximum();
31              int span = max - min;
32              int valeur = model.getValue();
33
34              double mult = (double) valeur/(double) span;
35              mult = mult == 0.0 ? 0.01 : mult;
36
37              Image nouvelleImage = image.getScaledInstance(
38                                      (int) (tailleOrig.width * mult),
39                                      (int) (tailleOrig.height * mult),
40                                      Image.SCALE_FAST);
41
42              icone.setImage(nouvelleImage);
43              panel.revalidate();
44          }
45      }
46  }
```

Utilisation d'un modèle de données

Supposons que l'on veuille développer une IHM avec l'arbre suivant :



Utilisation d'un modèle de données

Il faut définir un modèle de l'arbre qui contienne les données **de l'application**.

Une instance de `JTree` est liée par défaut à une instance de `DefaultTreeModel`.

`DefaultTreeModel` est un modèle d'arbre contenant des instances de `TreeNode`.
Pour simplifier, on choisit ici d'utiliser des instances de `DefaultMutableTreeNode`.
Une instance de `DefaultMutableTreeNode` contient un objet quelconque.

Enfin, on peut utiliser le constructeur de `JTree` pour lui passer le nœud racine de l'arbre à utiliser comme modèle.

Utilisation d'un modèle de données : code

Arbre.java

```
public class Arbre extends JFrame {
    public Arbre(String title) {
        super(title);

        DefaultMutableTreeNode top = new DefaultMutableTreeNode("2A");
        DefaultMutableTreeNode tc = new DefaultMutableTreeNode("Tronc commun");
        DefaultMutableTreeNode others = new DefaultMutableTreeNode("Autres");
        top.add(tc);
        top.add(others);

        DefaultMutableTreeNode ae201 = new DefaultMutableTreeNode("AE201");
        DefaultMutableTreeNode in201 = new DefaultMutableTreeNode("IN201");
        DefaultMutableTreeNode ma201 = new DefaultMutableTreeNode("MA201");
        tc.add(ae201);
        tc.add(in201);
        tc.add(ma201);

        JTree arbre = new JTree(top);
        this.add(new JScrollPane(arbre));

        this.pack();
        this.setVisible(true);
    }
}
```

Plan de la partie 11 - Interfaces graphiques

Comment sont affichés les composants ?

Lorsqu'un composant Swing est affiché (première « apparition », demande d'un réaffichage via `repaint()`), les composants sont dessinés du plus haut vers le plus bas (on descend la hiérarchie des conteneurs).

Ce processus peut être très gourmand en temps (cf. section sur la concurrence), même si :

- Swing utilise un système de *double-buffering* pour l'affichage ;
- on peut préciser que certains composants sont opaques (via un appel à `setOpaque(`**`true`**`)`) pour éviter d'essayer de peindre les composants situés derrière.

Comment sont affichés les composants ?

Lorsque l'on repeint une instance de `JComponent`, on fait appel à trois méthodes :

- `paintComponent(Graphics g)` : peint l'arrière plan (si nécessaire) et le composant ;
- `paintBorder(Graphics g)` : peint la bordure du composant ;
- `paintChildren(Graphics g)` : demande aux composants inclus dedans de se peindre.

La classe Graphics

Chacune de ces méthodes utilise une instance de `java.awt.Graphics`. Il s'agit d'un objet représentant le contexte graphique du composant sur lequel on peut dessiner.

Il possède plusieurs méthodes dont :

- `setColor(Color c)`
- `setFont(Font f)`
- `drawLine(int x1, int y1, int x2, int y2)`
- `fillRect(int x, int y, int w, int h)`
- `...`

On peut donc utiliser cet objet pour redéfinir un composant et personnaliser l'affichage.

Il existe également une méthode `getGraphics` dans `JComponent` qui renvoie l'objet `Graphics` du composant.

La classe Graphics : exemple d'une barre de progression

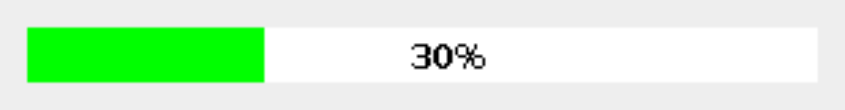
MyProgressBar.java

```
public class MyProgressBar extends JLabel {

    private double percentage;

    public MyProgressBar() {
        this.percentage = 0;
        this.setPreferredSize(new Dimension(300, 30));
    }

    public void setPercentage(double p) {
        if ((p >= 0) && (p <= 100)) {
            this.percentage = p;
        }
    }
```

La classe Graphics : exemple d'une barre de progression

MyProgressBar.java

```
    public void paintComponent(Graphics g) {
        g.setColor(Color.WHITE);
        g.fillRect(5, 5, 290, 20);

        g.setColor(Color.GREEN);
        g.fillRect(5, 5, (int) (this.percentage * 290 / 100), 20);

        g.setColor(Color.BLACK);
        g.drawString("" + ((int) this.percentage) + "%", 145, 20);
    }
}
```



Les classes `Graphics2D` et de manipulation d'images

Depuis Java 2 (i.e. la version 1.2 du JDK), on utilise des objets de type `java.awt.Graphics2D`, sous classe de `Graphics`, qui offrent plus de fonctionnalités. On les « obtient » en **transtypant** les objets de type `Graphics` retournés par `getGraphics` ou passés en paramètre de `paintXXX`.

En utilisant un objet de type `Graphics2D`, on peut :

- définir le style de trait ;
- utiliser des formes complexes pour le dessin (voir la classe `GeneralPath`) ;
- translater une figure, lui appliquer une rotation ;
- utiliser des textures, des gradients de couleurs ;
- ...

Il existe également des classes de manipulations avancées d'images (filtrage etc.) : `BufferedImage`...

Voir les liens dans le transparent « Références » pour plus de détails.

Remarques sur l'utilisation de Graphics et Graphics2D

Attention, lorsque l'on utilise un objet de type Graphics, celui-ci est initialisé à l'appel de la méthode paintComponent et passé ensuite en paramètre de paintBorder et de paintChildren.

En conséquence, il faut remettre l'objet Graphics dans le même état qu'en entrée de paintComponent pour éviter les surprises.

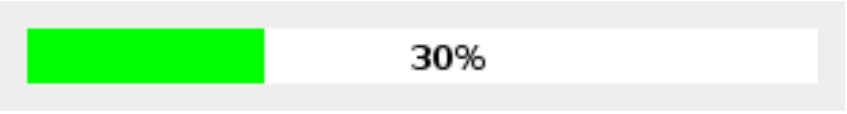
Pour cela, une solution est d'utiliser le code suivant :

```
public void paintComponent(Graphics g) {
  Graphics2D g2d = (Graphics2D) g.create();
  ...
  g2d.dispose(); // liberation des ressources
}
```

On peut également rétablir l'état de l'objet Graphics « à la main » après l'avoir utilisé.

Graphics2D : exemple d'antialiasing pour le texte

```
public void paintComponent(Graphics g) {
    Graphics2D g2d = (Graphics2D) g.create();
    RenderingHints rh = new RenderingHints(
        RenderingHints.KEY_ANTIALIASING,
        RenderingHints.VALUE_ANTIALIAS_ON);

    g2d.setRenderingHints(rh);

    g2d.setColor(Color.WHITE);
    g2d.fillRect(5, 5, 290, 20);

    g2d.setColor(Color.GREEN);
    g2d.fillRect(5, 5, (int) (this.percentage * 290 / 100), 20);

    g2d.setColor(Color.BLACK);
    g2d.drawString("" + ((int) this.percentage) + "%", 145, 20);

    g2d.dispose();
}
```



Plan de la partie 11 - Interfaces graphiques

L'event-dispatch thread de Swing

Que se passe-t-il si on utilise un listener sur une instance de `JButton` dont la méthode `actionPerformed` est définie comme suit :

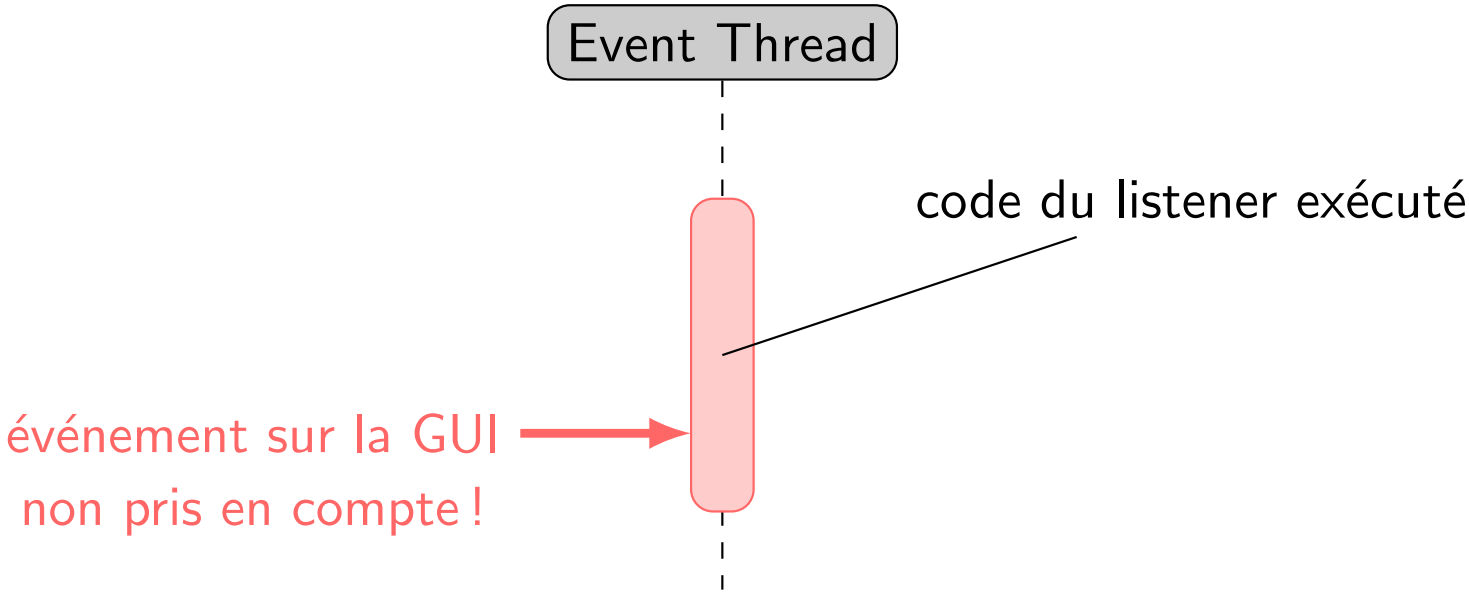
```
public void actionPerformed(ActionEvent e) {
  for (int i = 0; i < 1E6; i++) {
    System.out.println(i);
  }
}
```

Dans ce cas, si l'on appuie sur le bouton, la méthode `actionPerformed` est exécutée. Pendant cette durée d'exécution, l'interface graphique est gelée, ne répond plus aux événements et les composants ne sont pas repeints.

En effet, la gestion des événements, le code contenu dans les *listeners*, l'affichage et le rafraîchissement des composants sont tous gérés dans un *thread* (fil de contrôle) particulier, l'**event-dispatch thread** (EDT). On ne peut rafraîchir l'interface que depuis ce *thread*.

L'event-dispatch thread de Swing

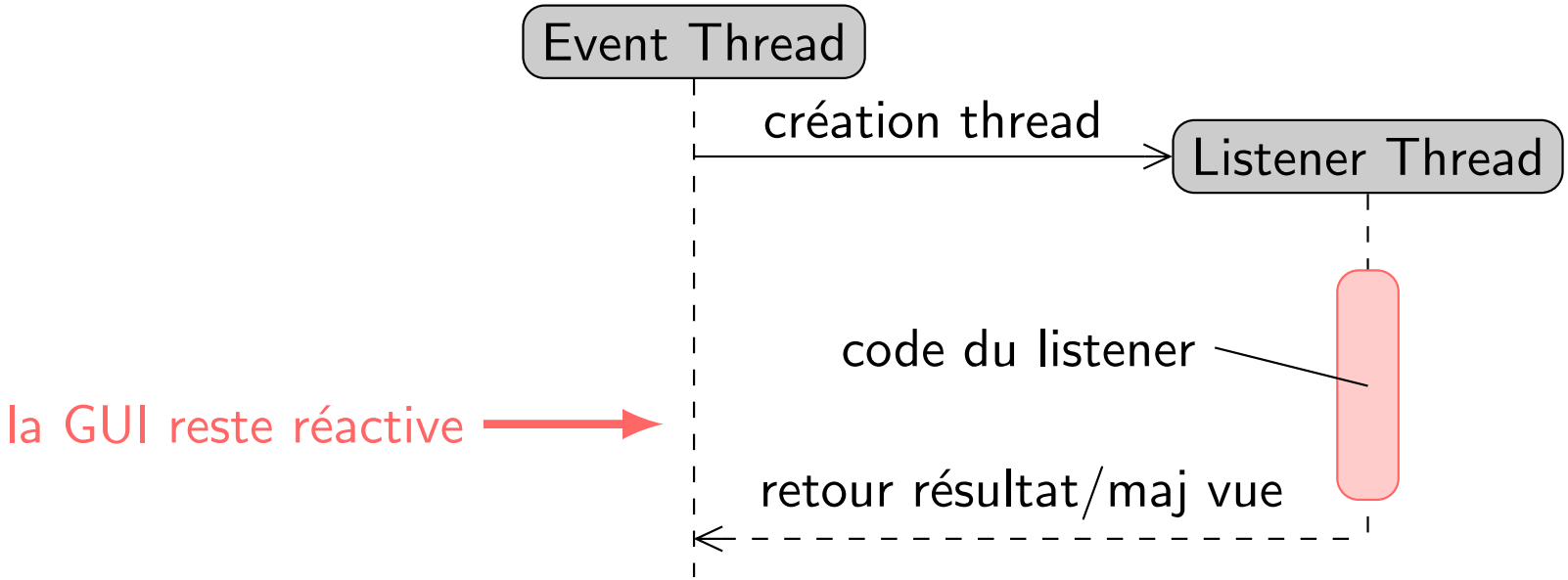
Si du code contenu dans un *listener* est exécuté dans l'*event-dispatch thread*, alors pendant toute cette exécution, le système ne pourra pas rafraîchir l'interface graphique.



L'event thread de Swing : solution

Il faudrait pouvoir exécuter le code contenu dans le *listener* **parallèlement** à l'*event thread*.

➥ c'est le rôle des *threads*



Utilisation de la classe `SwingWorker`

Le tutorial de Sun nous fournit une classe, `SwingWorker`, qui permet d'exécuter du code parallèlement à l'*event-dispatch thread*.

Cette classe est générique et ses deux paramètres formels de type représentent respectivement :

- le type de retour du traitement effectué par le *worker* (on peut utiliser `Void` si on n'en a pas besoin)
- le type de retour des résultats intermédiaires que l'on pourrait demander (même remarque que précédemment)

Utilisation de la classe SwingWorker

Pour l'utiliser, il faut la spécialiser et redéfinir la méthode doInBackground qui va contenir le code pouvant bloquer l'*event thread*.

Par exemple :

```
public void actionPerformed(ActionEvent e) {
  final SwingWorker<Void, Void> worker = new SwingWorker<Object,
                                                         Void> () {
    public Object doInBackground() {
      for(int i = 0; i < 10000000; i++) {}
      return null;
    }
  };

  worker.execute();
}
```

Utilisation de la classe SwingWorker

Pour pouvoir mettre à jour les composants de l'interface lorsque le SwingWorker a fini son travail, on peut redéfinir la méthode **public void** done() de SwingWorker. Cette méthode est exécutée dans l'*event thread*.

Pour récupérer le résultat du traitement, on utilise la méthode get sur le *worker*.

```
public void done() {
  // demander aux composants de changer

  label.setText(this.get().toString());
}
```

SwingWorker : un petit exemple

Considérons l'IHM suivante :

- les deux boutons `Hello !` et `Goodbye !` permettent de changer le texte de la zone de texte située à gauche
- le bouton `Go !` lance un calcul prenant du temps (simulé ici par des affichages sur la console) et affiche ensuite « Job finished ! » dans la zone de texte

Un premier listener sans SwingWorker

Si l'on considère le *listener* suivant pour le bouton Go ! :

SimpleListenerWithoutWorker

```
public class SimpleListenerWithoutWorker implements ActionListener {
    private JTextArea text;

    public SimpleListenerWithoutWorker(JTextArea text) {
        this.text = text;
    }

    public void actionPerformed(ActionEvent e) {
        for (int i = 0; i < 1E6 ; i++) {
            System.out.println(i);
        }

        this.text.setText("Job finished!");
    }
}
```

alors l'IHM ne répond plus aux événements durant le « calcul ».

Un listener avec SwingWorker

Si l'on considère maintenant le *listener* suivant pour le bouton Go ! :

SimpleListenerWithWorker

```
public class SimpleListenerWithoutWorker implements ActionListener {
    private JTextArea text;

    public SimpleListenerWithoutWorker(JTextArea text) {
        this.text = text;
    }

    public void actionPerformed(ActionEvent e) {
        for (int i = 0; i < 1E6 ; i++) {
            System.out.println(i);
        }
```

Un listener avec SwingWorker

avec le *worker* suivant, l'IHM est alors réactive :

SimpleWorker

```
class SimpleWorker extends SwingWorker<Void, Void> {

    private JTextArea text;

    public SimpleWorker(JTextArea text) {
        this.text = text;
    }

    @Override public Void doInBackground() {
        for (int i = 0; i < 1E6 ; i++) {
            System.out.println(i);
        }

        return null;
    }

    @Override public void done() {
        this.text.setText("Job finished!");
    }
}
```

`SwingWorker` : un exemple plus compliqué

Supposons que l'on veuille réutiliser la barre de progression que l'on a développée précédemment et qu'on veuille la mettre à jour au fur et à mesure de l'avancée du calcul.
Dans ce cas, on peut utiliser les méthodes `publish` et `process` de `SwingWorker` qui permettent de publier des résultats intermédiaires et de les utiliser dans l'IHM.

SwingWorker : un exemple plus compliqué

WorkerProgressBar.java

```
public class WorkerProgressBar extends SwingWorker<Void, Double> {

    private MyProgressBarAA pb;

    public WorkerProgressBar(MyProgressBarAA pb) {
        this.pb = pb;
    }

    @Override public Void doInBackground() {
        for (int i = 1; i <= 1E6 ; i++) {
            System.out.println(i);

            if (i % 2000 == 0) {
                publish((double) i / 10000);
            }
        }

        return null;
    }
```

SwingWorker : un exemple plus compliqué

WorkerProgressBar.java

```
    @Override protected void process(List<Double> percentages) {
        for (double p : percentages) {
            this.pb.setPercentage(p);
        }
    }
}
```

La liste de valeurs publiée par `publish` est « récupérée » automatiquement dans `process` et on peut mettre à jour la vue sans risque depuis `process`.

Si on ne veut pas utiliser la liste de valeurs intermédiaires, on peut utiliser la méthode `setProgress` de `SwingWorker` avec une `JProgressBar` et un `PropertyChangeListener` (cf. javadoc de `SwingWorker`).

Démarrage d'une application Swing

Au démarrage d'une application Swing (typiquement lors de l'appel à `pack` sur une `JFrame`), le *thread* d'événements de Swing est créé. Cela peut être problématique, car les composants sont en train d'être créés et on peut avoir des événements notifiés sur les composants.

Pour éviter cela, on peut créer l'application de la manière suivante :

```
public static void main(String[] args) {
  javax.swing.SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new MaFrame(...);
        }
    });
}
```

Nous n'expliquerons pas dans ce cours l'utilisation de `invokeLater`, mais vous pouvez utiliser les références fournies à la fin des transparents.

Utilisation d'horloges

Swing propose une classe, `javax.swing.Timer`, dont les instances sont des horloges qui créent des événements à des intervalles de temps réguliers.

À la création de l'horloge, on précise un `ActionListener` qui va être prévenu lors du déclenchement des événements gérés par le *timer*. On utilise ensuite les méthodes `start` et `stop` de `Timer` pour démarrer ou arrêter le *timer*.

Exemple :

```
Timer t = new Timer(1000, new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.out.println("Je viens d'etre prevenu !");
      }
});

t.start();
```

On peut préciser que le *timer* s'arrêtera au premier événement avec la méthode `setRepeats(boolean b)`.

Références de base


Topley, K. (2000).
Core Swing advanced programming.
Prentice Hall.


Geary, D. M. (1999).
Graphic Java 2, Swing.
3e éd. T. 2.
Prentice Hall.


The Swing tutorial .
`http://docs.oracle.com/javase/tutorial/uiswing/`.


The 2D Graphics tutorial .
`http://docs.oracle.com/javase/tutorial/2d/index.html`.

Tester une IHM

FEST Fixtures for Easy Software Testing .
http://code.google.com/p/fest/.

SwingUnit .
http://java.net/projects/swingunit.

Références pour faire de jolies IHM...

Haase, Chet et Romain Guy (2008).
Filthy Rich Clients.
Addison-Wesley.
http://filthyrichclients.org/.

JavaDesktop .
Lots of useful projects like JGoodies, Swinglabs, Substance etc.
http://java.net/projects/javadesktop.

Glazed Lists .
http://java.net/projects/glazedlists/.

Better Swing application framework .
http://kenai.com/projects/bsaf/pages/Home.

DesignGridLayout : an easy and powerful Swing layout manager .
http://java.net/projects/designgridlayout/pages/Home.

12 - Exceptions

Plan de la partie 12 - Exceptions

Exemple : affichage des moyennes

Moyenne.java

```
1  import java.io.*;     // pour les entrees/sorties
2
3  public class Moyenne {
4
5      /** Afficher la moyenne des valeurs contenues dans le
6       *  fichier args[0]
7       */
8      public static void main(String[] args) {


             ...


29     }
30 }
```

Exemple : affichage des moyennes

Moyenne.java (corps de main)

```
 9          try {
10              BufferedReader in = new BufferedReader(new FileReader(args[0]));
11
12              double somme = 0;          // somme
13              int nb = 0;                // nb de valeurs lues
14              String ligne;              // une ligne du fichier
15
16              while ((ligne = in.readLine()) != null) {
17                  somme += Double.parseDouble(ligne);
18                  nb++;
19              }
20
21              in.close();
22
23              System.out.println("Moyenne : " + (somme / nb));
24          } catch (IOException e) {
25              System.out.println("Probleme d'entree/sortie : " + e);
26          } catch (NumberFormatException e) {
27              System.out.println("Donnee non numerique : " + e);
28          }
```

Exécutions de Moyenne

donnees1

```
10
15
20
```

shell

```
[tof@suntof]~ $ java Moyenne donnees1
Moyenne : 15.0
```

Exécutions de Moyenne

donnees2

```
10
15
20
quinze
```

shell

```
[tof@suntof]~ $ java Moyenne donnees2
Donnee non numerique : java.lang.NumberFormatException:
 For input string: "quinze"
```

Exécutions de Moyenne

donnees3

shell

```
[tof@suntof]~ $ java Moyenne donnees3
Moyenne : NaN
```

Exécutions de Moyenne

donnees4 (contient une chaîne vide)

shell

```
[tof@suntof]~ $ java Moyenne donnees4
Donnee non numerique : java.lang.NumberFormatException: empty String
```

Exécutions de Moyenne

donnees5 n'existe pas...

shell

```
[tof@suntof]~ $ java Moyenne donnees5
Probleme d'entree/sortie : java.io.FileNotFoundException:
 donnees5 (No such file or directory)
```

Exécutions de Moyenne

Pas de fichier en argument...

shell

```
[tof@suntof]~ $ java Moyenne
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at Moyenne.main(Moyenne.java:10)
```

Exceptions : intérêts

Définition (exception)

Une exception est un événement exceptionnel interrompant le flot d'instructions d'un programme.

Les exceptions permettent :

- de transférer le flot de contrôle de l'instruction qui lève l'exception (qui détecte l'anomalie) vers la partie du programme capable de la traiter ;
- d'éviter de surcharger le code d'une méthode avec de nombreux tests concernant ces cas anormaux ;
- de regrouper le traitement des cas anormaux et erreurs ;
- de classifier les anomalies (différents types d'exceptions).

Remarque

Il ne faut pas en abuser. Il faut les réserver aux cas réellement anormaux ou exceptionnels.

Principe

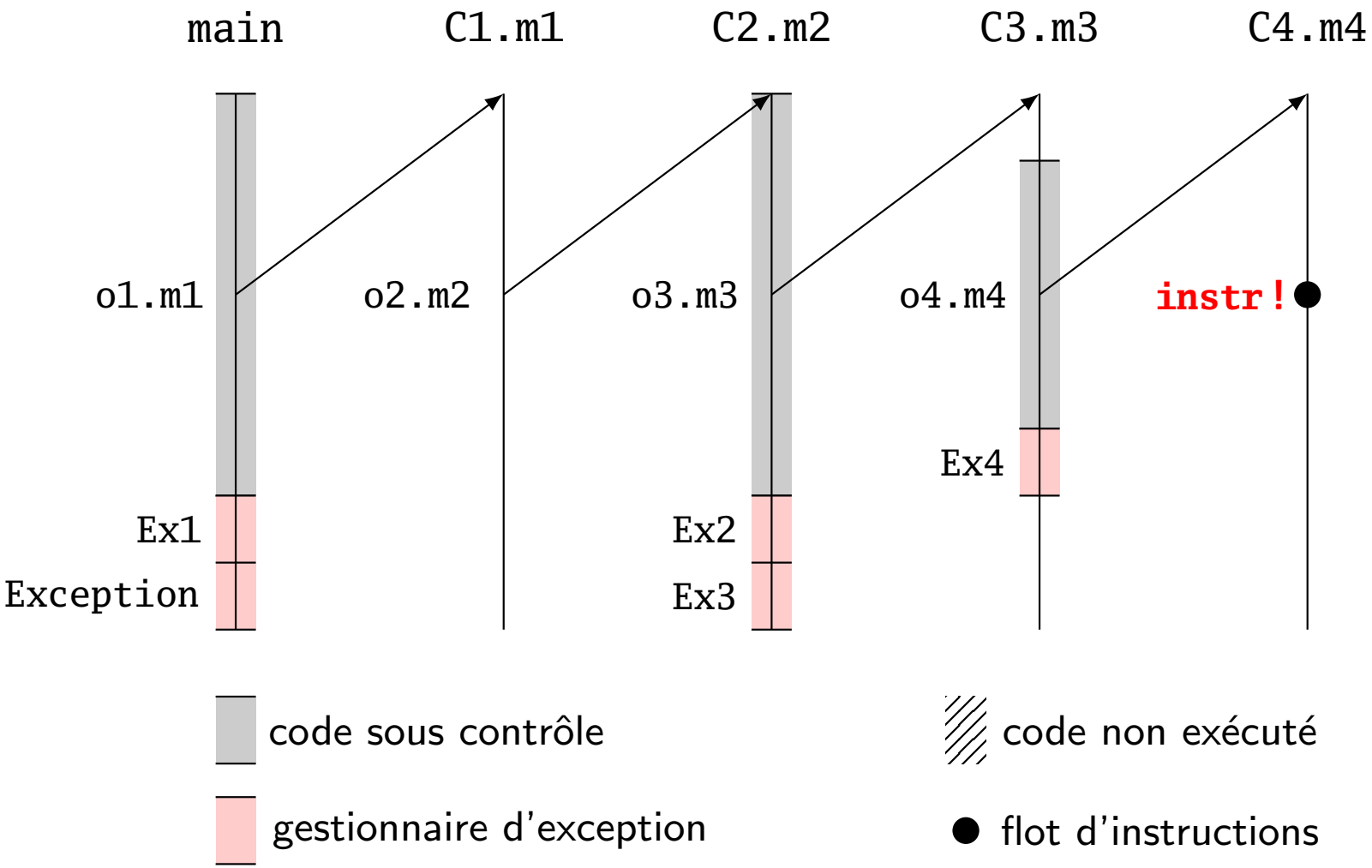
La gestion des exceptions repose sur trois phases :

- une exception est **levée** quand une erreur ou une anomalie est détectée ;
- l'exception est propagée : l'exécution séquentielle du programme est **interrompue** et le flot de contrôle est transféré aux gestionnaires d'exceptions ;
- l'exception est (éventuellement) **récupérée** par un gestionnaire d'exceptions. Elle est traitée et l'exécution reprend avec les instructions qui suivent le gestionnaire d'exceptions.

Attention

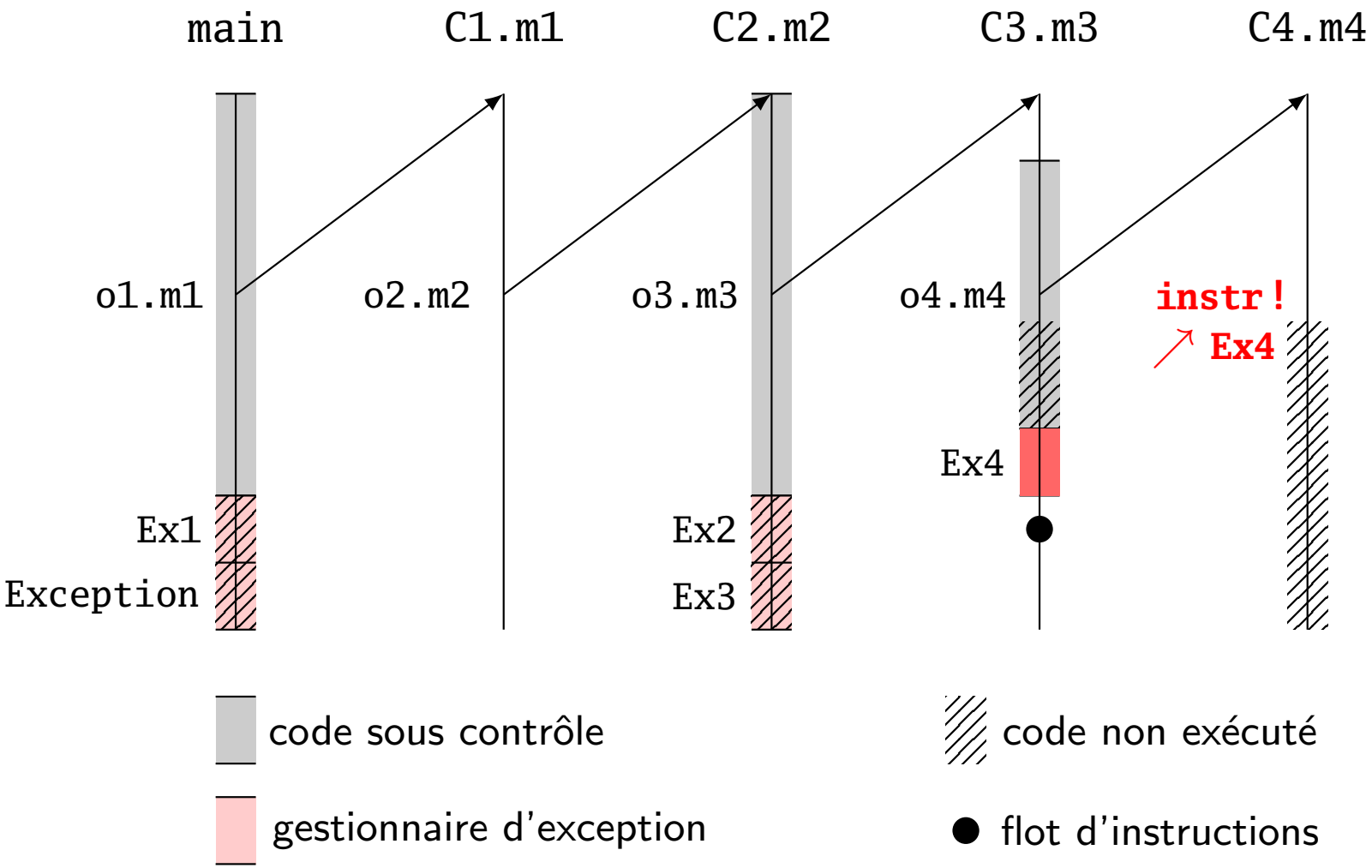
Une exception non récupérée provoque l'arrêt du programme (avec affichage de la trace des appels de méthodes depuis l'instruction qui a levé l'exception jusqu'à l'instruction appelante de la méthode principale). C'est ce qu'il se passait jusqu'à présent quand le programme en levait une.

Mécanisme de propagation des exceptions



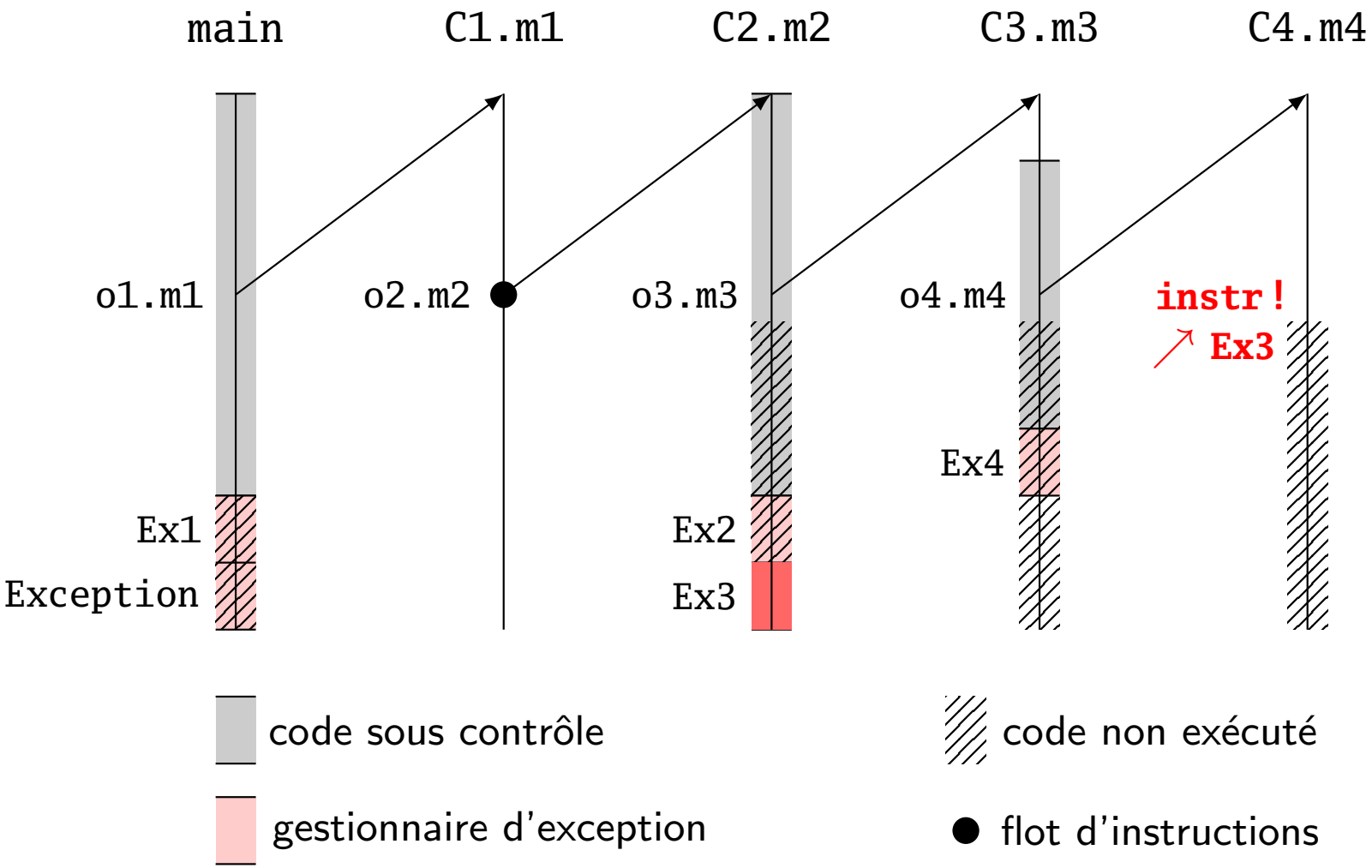
Que se passe-t-il lorsque `instr !` lève `Ex4`, `Ex3`, `Ex1`, `Ex5`, `Err` ?

Mécanisme de propagation des exceptions



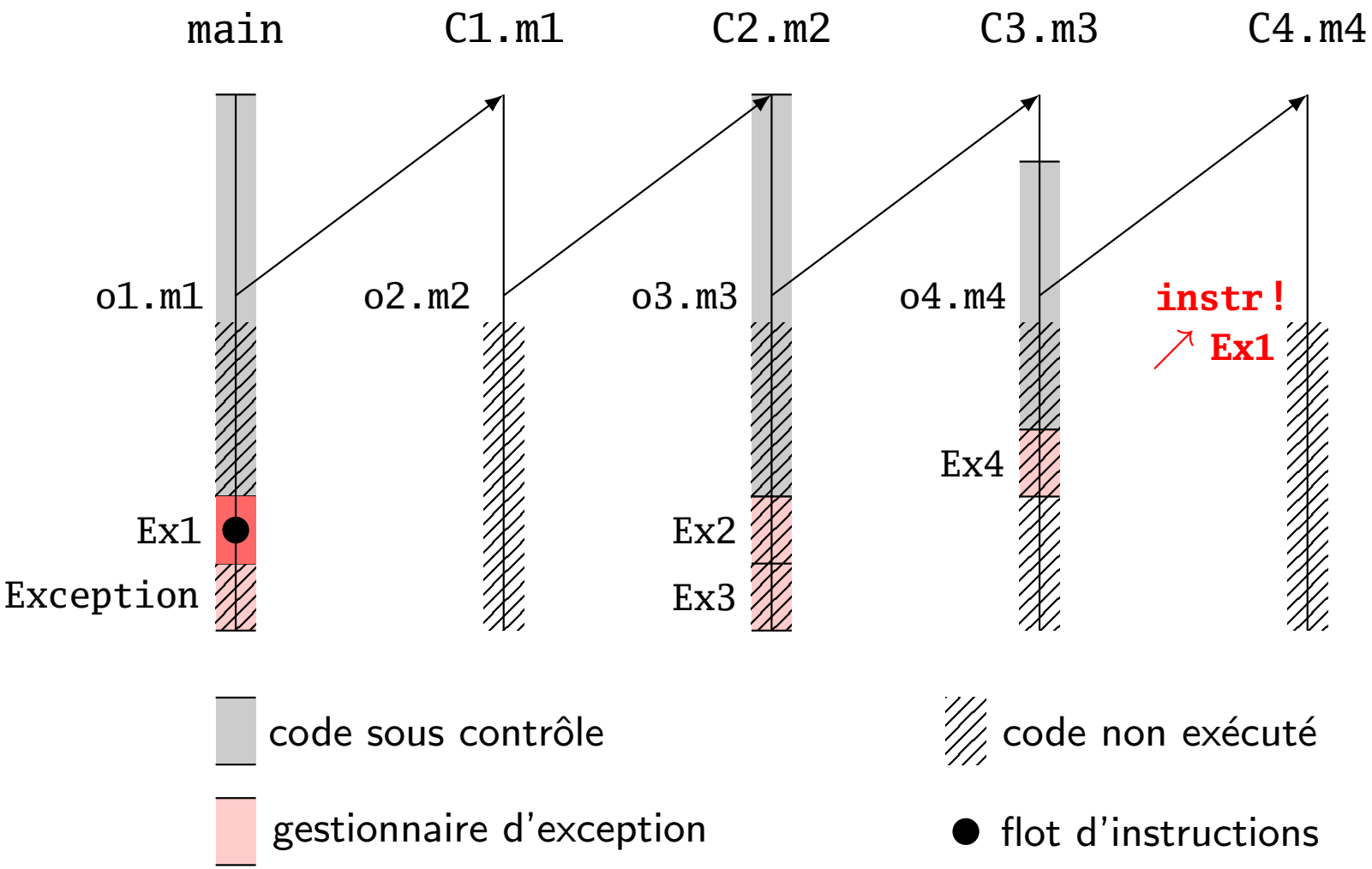
Que se passe-t-il lorsque `instr !` lève **Ex4**, Ex3, Ex1, Ex5, Err ?

Mécanisme de propagation des exceptions



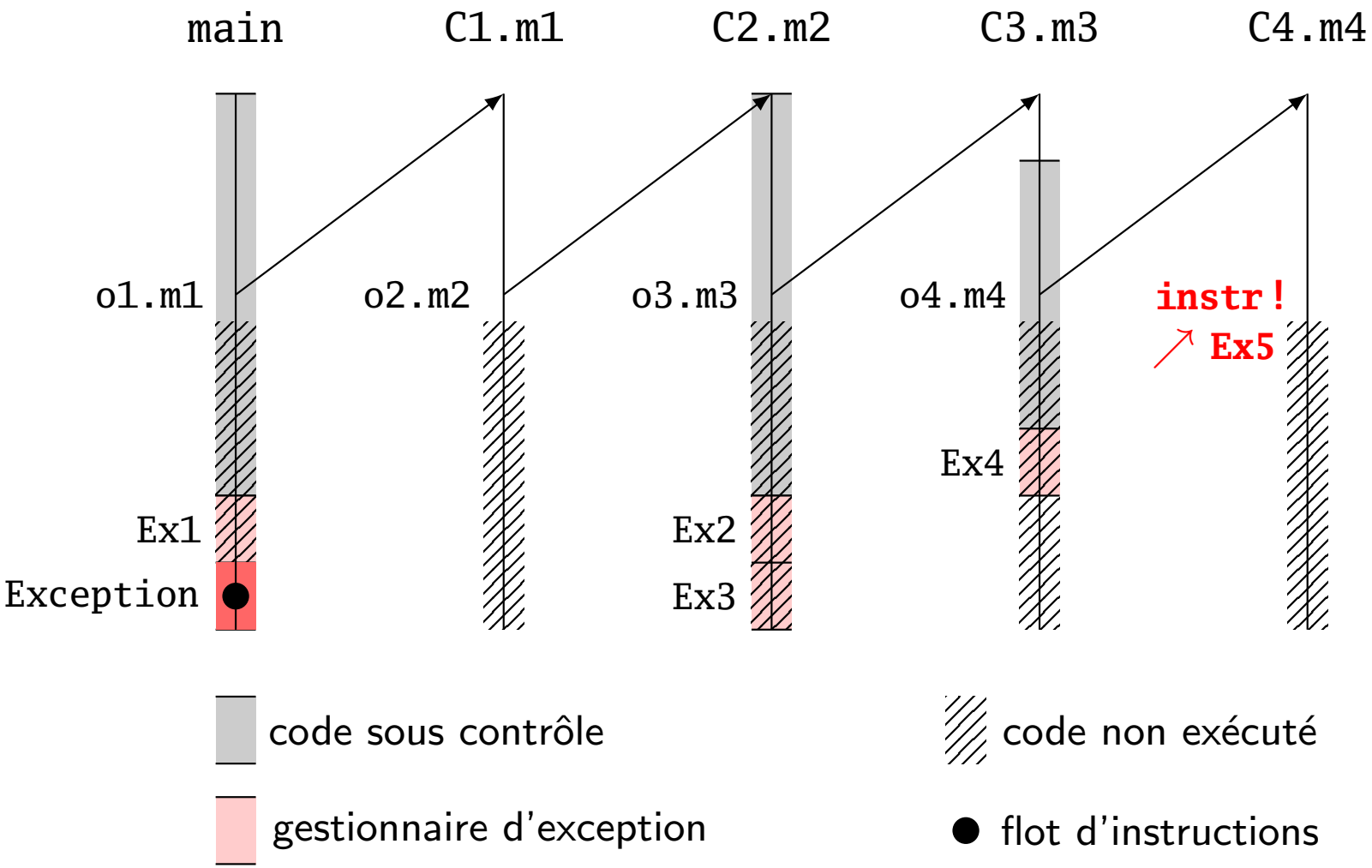
Que se passe-t-il lorsque `instr !` lève `Ex4`, **`Ex3`**, `Ex1`, `Ex5`, `Err` ?

Mécanisme de propagation des exceptions



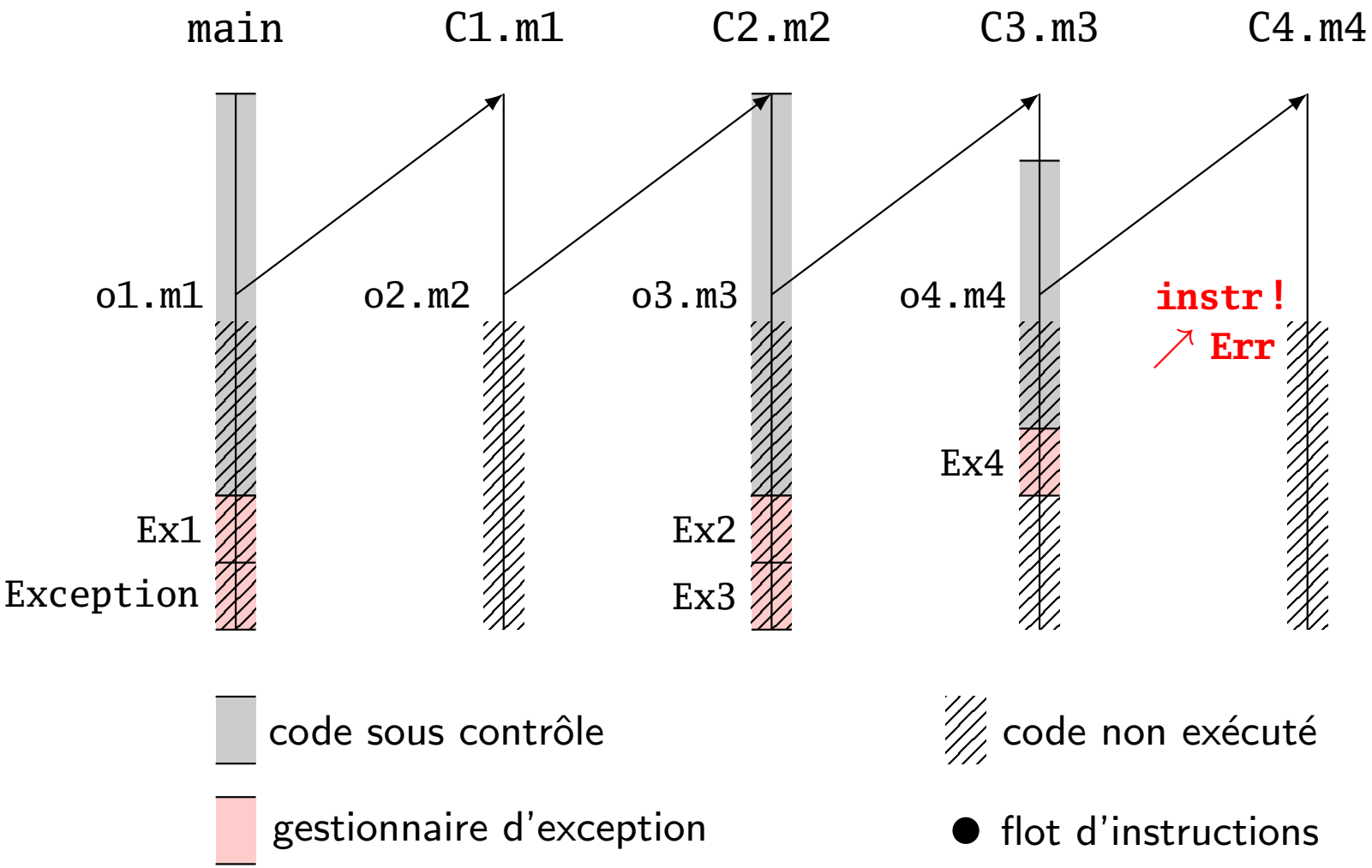
Que se passe-t-il lorsque `instr !` lève `Ex4`, `Ex3`, **`Ex1`**, `Ex5`, `Err` ?

Mécanisme de propagation des exceptions



Que se passe-t-il lorsque `instr !` lève `Ex4`, `Ex3`, `Ex1`, **`Ex5`**, `Err` ?

Mécanisme de propagation des exceptions



Que se passe-t-il lorsque `instr !` lève `Ex4`, `Ex3`, `Ex1`, `Ex5`, **`Err`** ?

Illustration du mécanisme de propagation

Propagation.java

```
 1 public class Propagation {
 2     public static void m1() {
 3         m2();
 4     }
 5     public static void m2() {
 6         m3();
 7     }
 8     public static void m3() {
 9         throw new RuntimeException();
10     }
11     public static void main(String[] args) {
12         m1();
13     } // end of main()
14 }
```

shell

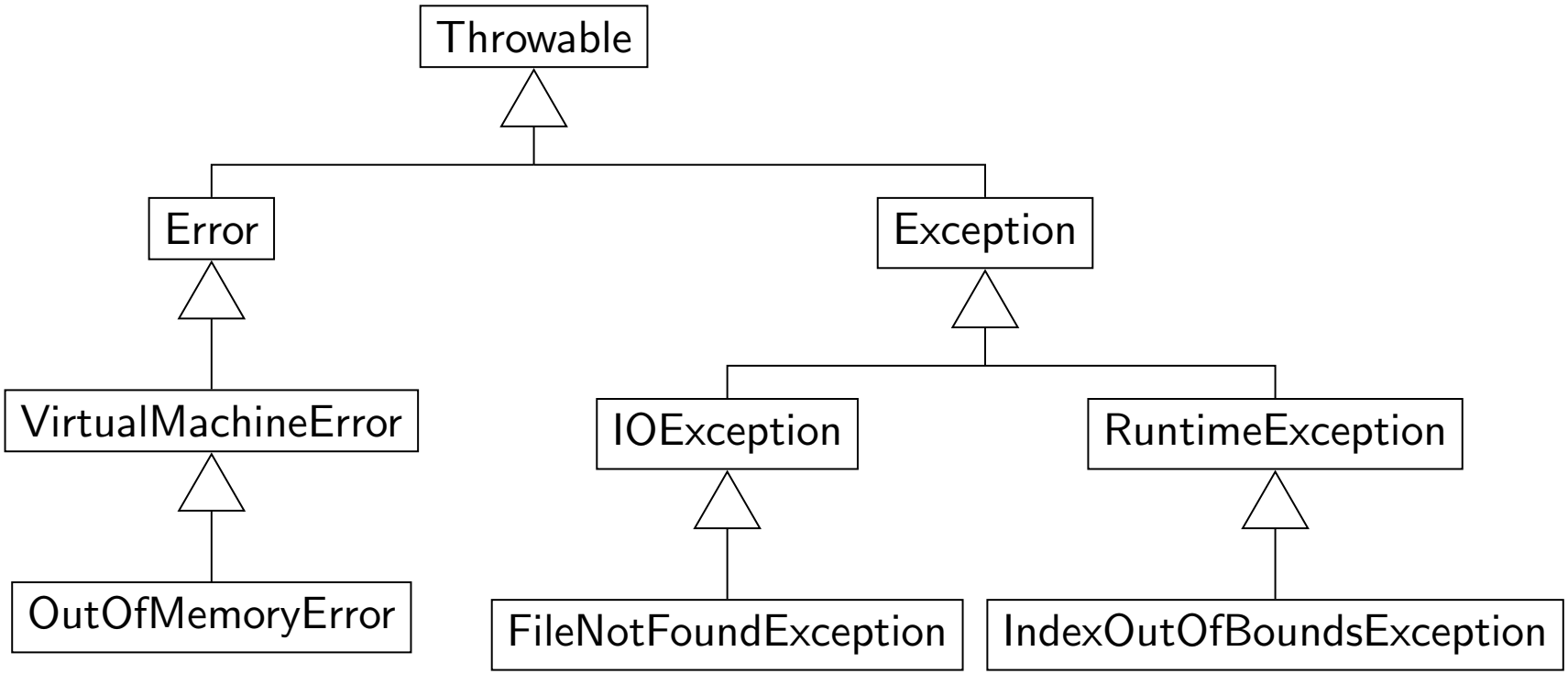
```
[tof@suntof]~ $ java Propagation
Exception in thread "main" java.lang.RuntimeException
        at Propagation.m3(Propagation.java:9)
        at Propagation.m2(Propagation.java:6)
        at Propagation.m1(Propagation.java:3)
        at Propagation.main(Propagation.java:12)
```

Plan de la partie 12 - Exceptions

Hiérarchie des exceptions

En Java, une exception est un objet particulier.

Il existe une hiérarchie des exceptions :



Classification des exceptions

Les exceptions sont classés en deux catégories :

- les exceptions **hors contrôle** :
  - classes dérivées de `java.lang.Error` : ce sont des erreurs non accessibles qui ne peuvent généralement pas être récupérées (plus de mémoire, classe non trouvée etc.) ;
  - classes dérivées de `java.lang.RuntimeException` : ce sont des erreurs de programmation, elles ne devraient pas se produire (`NullPointerException`, `IndexOutOfBoundsException`).
- les exceptions **sous contrôle**. Ce sont les classes dérivées de `Exception` (mais pas de `RuntimeException`). Elles peuvent (doivent !) être récupérées et traitées. Elles se produisent dans des circonstances définies que l'on connaît.
  Elles correspondent à la notion de **robustesse**.

`Exception` et `Error` n'ajoutent aucune nouvelle caractéristique à `Throwable`. Elles permettent juste de classifier les anomalies.

Définir sa propre exception

On peut définir sa propre exception en spécialisant la classe `Exception`.

On peut également spécialiser `Throwable` ou `Error`.

Par exemple :

MonException.java

```
public class MonException extends Exception {

  public MonException(String message) {
    super(message);
  }
}
```

Le constructeur prenant un objet de type `String` en paramètre permet de spécifier un message pour l'exception.
Ce message est récupérable par `getMessage()`.

Plan de la partie 12 - Exceptions

Plan de la partie 12 - Exceptions

Lever une exception

L'opérateur **throw** permet de lever une exception. Rappel : une exception est une **instance** d'une classe descendant de `Throwable`.

Forme générale :

Syntaxe (levée d'une exception)

```
if (<condition anormale>) {
  throw new TypeException(<parametres effectifs>);
}
```

Lever une exception : constructeur

`Throwable` dispose d'un constructeur prenant en paramètre une chaîne de caractères décrivant l'exception. C'est le constructeur le plus souvent utilisé et on le retrouve dans les classes descendantes.

Exemple sur une classe `Fraction` :

Fraction.java

```
public Fraction(int num, int den) {
  if(den == 0) {
    throw new ArithmeticException("Division par zero");
  }
  ...
}
```

Catch or Specify Requirement

Comment utiliser du code pouvant lever une exception ?

Java impose l'utilisation du *Catch or Specify Requirement* qui précise que l'appel à un tel code doit être encapsulé :

- soit dans un bloc **`try`** et traité ensuite par un gestionnaire d'exceptions approprié ;
- soit dans une méthode dont la spécification précise que l'exception en question peut être propagée.

Ce principe ne concerne que les exceptions **sous contrôle**.

Principe (Catch or Specify Requirement)

Le compilateur vérifie que toutes les exceptions sous contrôle produites par les instructions du corps d'une méthode sont :

- soit récupérées et traitées par la méthode ;
- soit déclarées comme étant propagées.

Plan de la partie 12 - Exceptions

Spécifications des exceptions

Une exception correspond à un résultat transmis par une méthode. Java impose donc de spécifier les exceptions propagées dans la spécification d'une méthode.

On ne spécifie que les exceptions **sous contrôle**.

Toutes les exceptions sous contrôle qui sont **levées ou propagées** par une méthode doivent être déclarées en utilisant le mot-clé **throws**.

Syntaxe (spécification d'une exception)

```
<modifieurs> Type maMethode(<parametres>) throws TypeExc1, TypeExc2, ...
```

RacineCarre.java

```
public static double racineCarree(double x) throws MathException {
  ...
}
```

Spécification des exceptions : documentation

Pour documenter une méthode pouvant lever une exception, on doit utiliser l'étiquette `@exception` dans la documentation javadoc.

parseDouble

```
public static double parseDouble(String s)
                          throws NumberFormatException
```

Returns a new double initialized to the value represented by the specified `String`, as performed by the `valueOf` method of class `Double`.

Parameters:
`s` - the string to be parsed.
Returns:
the double value represented by the string argument.
Throws:
`NumberFormatException` - if the string does not contain a parsable double.
Since:
1.2
See Also:
`valueOf(String)`

---

isNaN

```
public static boolean isNaN(double v)
```

Returns `true` if the specified number is a Not-a-Number (NaN) value, `false` otherwise.

Parameters:
`v` - the value to be tested.
Returns:
`true` if the value of the argument is NaN; `false` otherwise.

---

isInfinite

```
public static boolean isInfinite(double v)
```

Returns `true` if the specified number is infinitely large in magnitude, `false` otherwise.

Parameters:
`v` - the value to be tested.
Returns:
`true` if the value of the argument is positive infinity or negative infinity; `false` otherwise.

Plan de la partie 12 - Exceptions

Le bloc **try**

Lorsque l'on veut utiliser une instruction pouvant lever une exception (typiquement un appel à une méthode), on utilise un bloc **try** pour encapsuler cet appel si l'on veut pouvoir le traiter localement.

Exemple :

```
public double calculRacine(String reel) {
  try {
    double d = Double.parseDouble(reel);
    ...
  }
  ...
}
```

Récupérer une exception

Le bloc **try** peut être suivi par plusieurs gestionnaires d'exceptions :

```
catch (TypeExc1 e) {      // gestionnaire de l'exception TypeExc1
                          // instructions a executer quand
                          // l'exception TypeExc1 se produit
} catch (TypeExc2 e) {    // e est un parametre formel
                          // instructions a executer quand
                          // l'exception TypeExc2 se produit
} catch (Exception e) {   // toutes les exceptions
                          // instructions a executer quand une
                          // exception se produit
}
```

L'ordre des **catch** est très important (principe de substitution).
Après l'exécution des instructions d'un **catch**, l'exécution continue après le dernier **catch** (sauf si une exception est levée dans le **catch**).

Principe

Ne récupérer une exception que si l'on sait la traiter.

Récupérer une exception : exemple

Catch.java

```
public class Catch {

    public static void exempleCatch(Exception exc) {
        try {
            System.out.println("start of try");

            if (exc != null) { throw(exc); }

            System.out.println("end of try");
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException caught");
        } catch (Exception e) {
            System.out.println("Exception caught");
        }

        System.out.println("outside try/catch block");
    }
```

Récupérer une exception : exemple

Catch.java

```
    public static void main(String[] args) throws Exception {
        if (args.length == 1) {
            exempleCatch((Exception) Class.forName(args[0]).newInstance());
        } else {
            exempleCatch(null);
        }
    }
}
```

Récupérer une exception : exemple

Catch.java

```
        try {
            System.out.println("start of try");

            if (exc != null) { throw(exc); }

            System.out.println("end of try");
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException caught");
        } catch (Exception e) {
            System.out.println("Exception caught");
        }

        System.out.println("outside try/catch block");
```

shell

```
[tof@suntof]~ $ java Catch
start of try
end of try
outside try/catch block
```

Récupérer une exception : exemple

Catch.java

```
        try {
            System.out.println("start of try");

            if (exc != null) { throw(exc); }

            System.out.println("end of try");
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException caught");
        } catch (Exception e) {
            System.out.println("Exception caught");
        }

        System.out.println("outside try/catch block");
```

shell

```
[tof@suntof]~ $ java Catch java.lang.ArithmeticException
start of try
ArithmeticException caught
outside try/catch block
```

Récupérer une exception : exemple

Catch.java

```
        try {
            System.out.println("start of try");

            if (exc != null) { throw(exc); }

            System.out.println("end of try");
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException caught");
        } catch (Exception e) {
            System.out.println("Exception caught");
        }

        System.out.println("outside try/catch block");
```

shell

```
[tof@suntof]~ $ java Catch java.lang.Exception
start of try
Exception caught
outside try/catch block
```

Traiter une exception

Si une exception est récupérée, c'est que l'on est capable de la traiter (au moins partiellement). Le traitement est donc fait dans les instructions du **catch**. Il peut consister à :

- réparer le problème et exécuter de nouveau l'opération ;
- rétablir un état cohérent et continuer l'exécution sans recommencer ;
- calculer un autre résultat remplaçant celui de la méthode ;
- réparer localement le problème et propager l'exception :

```
catch (TypeException e) {
  reparer localement // par exemple retablir la coherence de l'etat
  throw e;           // propager l'exception
  throw new ExcQuiVaBien(e); // chainer l'exception
                             // (a partir de Java 1.4)
}
```

- réparer localement le problème et lancer une nouvelle exception ;
- terminer le programme.

Traiter une exception : exemple du réessai

```
boolean reussi = false;
do {
  try {
    instructions_qui_peuvent_echouer();
    reussi = true;  // n'est executee que si aucune exception n'est
                    // levee
  }
  catch (TypeExc1 e) {
    traiterTypeExc1(); // redemander des parametres a l'utilisateur
                       // par exemple
  } catch (TypeExc2 e) {
    traiterTypeExc2();
  }
} while (! reussi);
// les instructions ont ete executees sans erreurs !
```

La clause **finally**

Un bloc **finally** peut être mis en place après le dernier bloc **catch**. Ce bloc sera exécuté qu'il y ait une exception levée, qu'elle soit récupérée ou non.

```
try {
  instructions_qui_peuvent_echouer();
}
catch (TypeExc1 e) {
  traiterTypeExc1();
} catch (TypeExc2 e) {
  traiterTypeExc2();
} finally {
  instructions_toujours_executees();
}
```

Intérêt : être sûr de libérer une ressource (un fichier ouvert en écriture par exemple).

Récupérer une exception : exemple

Finally.java

```
public class Finally {

    public static void exempleFinally(Exception exc) throws Exception {
        try {
            System.out.println("start of try");

            if (exc != null) { throw(exc); }

            System.out.println("end of try");
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException caught");
        } finally {
            System.out.println("finally executed");
        }

        System.out.println("outside try/catch block");
    }
```

Récupérer une exception : exemple

Finally.java

```
    public static void main(String[] args) throws Exception {
        if (args.length == 1) {
            exempleFinally((Exception) Class.forName(args[0]).newInstance());
        } else {
            exempleFinally(null);
        }
    }
}
```

Récupérer une exception : exemple

Finally.java

```
        try {
            System.out.println("start of try");

            if (exc != null) { throw(exc); }

            System.out.println("end of try");
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException caught");
        } finally {
            System.out.println("finally executed");
        }

        System.out.println("outside try/catch block");
```

shell

```
[tof@suntof]~ $ java Finally
start of try
end of try
finally executed
outside try/catch block
```

Récupérer une exception : exemple

Finally.java

```
        try {
            System.out.println("start of try");

            if (exc != null) { throw(exc); }

            System.out.println("end of try");
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException caught");
        } finally {
            System.out.println("finally executed");
        }

        System.out.println("outside try/catch block");
```

shell

```
[tof@suntof]~ $ java Finally java.lang.ArithmeticException
start of try
ArithmeticException caught
finally executed
outside try/catch block
```

Récupérer une exception : exemple

Finally.java

```
        try {
            System.out.println("start of try");

            if (exc != null) { throw(exc); }

            System.out.println("end of try");
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException caught");
        } finally {
            System.out.println("finally executed");
        }

        System.out.println("outside try/catch block");
```

shell

```
[tof@suntof]~ $ java Finally java.lang.Exception
start of try
finally executed
Exception in thread main java.lang.Exception
        at Finally.exempleFinally(Finally.java:7)
        at Finally.main(Finally.java:21)
```

Plan de la partie 12 - Exceptions

Exceptions, héritage et redéfinition de méthode

Une méthode peut propager (et donc lever) toute exception qui est :

- une descendante de `RuntimeException` ou `Error` (**`throws`** implicite) ;
- une descendante d'une des exceptions listées dans la clause **`throws`**.

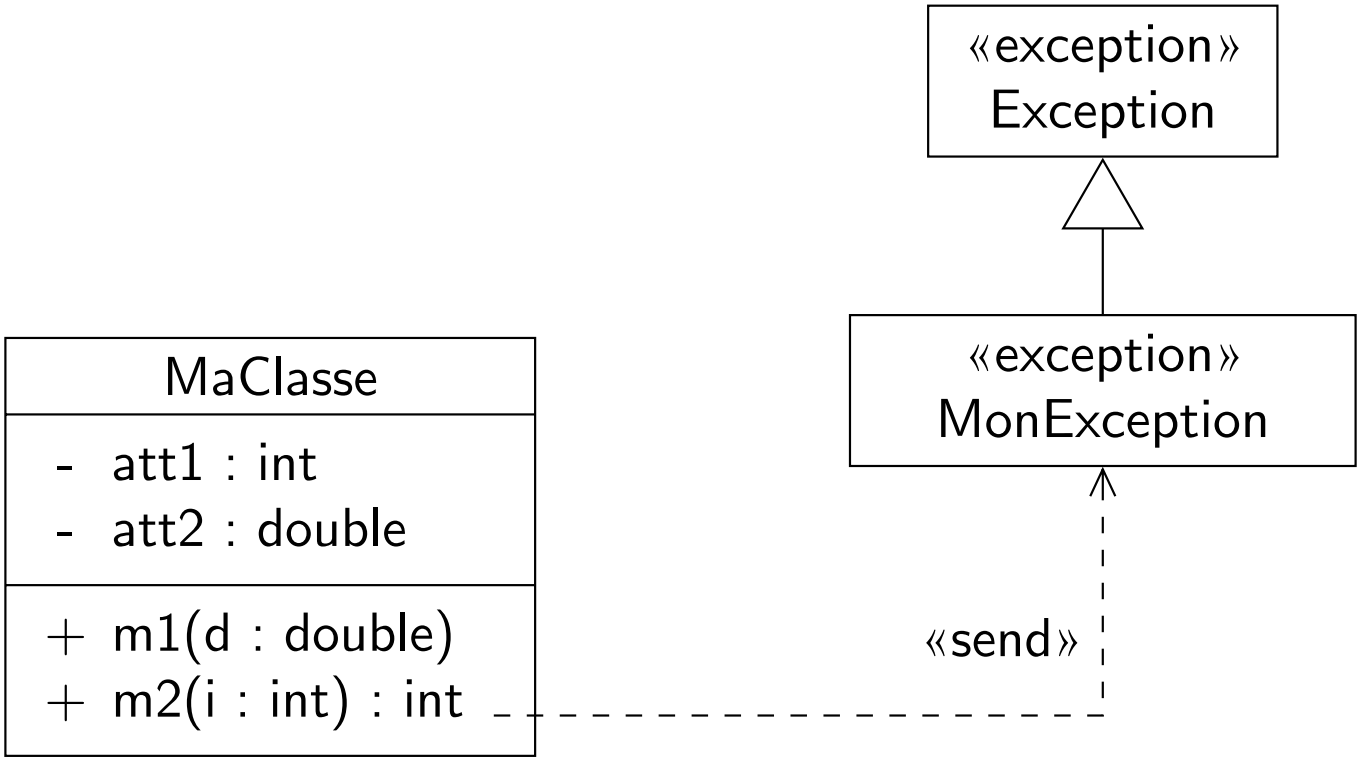
Comme les exceptions sont des objets, l'héritage entre exceptions permet d'après le principe de substitution de limiter le nombre d'exceptions à déclarer dans une clause **`throws`** ou de récupérer dans un même **`catch`** plusieurs exceptions.

Principe (redéfinition de méthodes et exceptions)

Une méthode redéfinie ne peut lever que des exceptions qui ont été spécifiées par sa déclaration dans la classe parente. Elle ne peut pas lever de nouvelles exceptions.

Plan de la partie 12 - Exceptions

Représentation des exceptions en UML



Plan de la partie 12 - Exceptions

Cas où préférer les exceptions

- l'évaluation de la précondition est coûteuse et redondante avec le traitement ;
- impossibilité de donner une précondition (ex : écriture dans un fichier etc.) ;
- cas anormal mais que l'on considère comme pouvant se produire (pas une erreur de programmation) : saisie utilisateur etc.

Cas où préférer la programmation par contrat

- les parties liées par contrat sont maîtrisées ;
- erreur manifeste de programmation (non récupérable) ;
- efficacité du programme (éviter les tests dans les méthodes appelées) ;
- mécanisme supporté par l'environnement de développement.

Tester une méthode pouvant lever une exception

Peut-on tester avec JUnit une méthode pouvant lever une exception ?

➥ oui, avec le tag **@Test**

Exemple :

```
@Test(expected=monPackage.MonException.class)
  public void testMethode() throws MonException {
    ...
    methode(); // instruction devant lever une exception
               // de type monPackage.MonException
}
```

Attention

- on utilise le nom complet de la classe d'exception dont on veut vérifier la levée dans le tag **@Test** (même si on a importé cette classe) ;
- on est obligé de préciser quand même la clause **throws** de la méthode de test.

Quelques conseils. . .

- les exceptions ne sont pas censées remplacer un test simple.
- multiplier les blocs **`try`** (et ainsi gérer de façon très fine les exceptions) pénalise les performances du programme.
- ne pas museler les exceptions :

```
try { beaucoup de code }
catch (Exception e) {}
```

  Si le compilateur signale des exceptions sous contrôle non gérées, c'est pour aider.
- éviter d'imbriquer des blocs **`try`** (faire des méthodes auxiliaires).
- ne pas hésiter à propager une exception.
- documenter les exceptions dans la javadoc grâce au tag `@exception`.
- pour les préconditions, utiliser `IllegalArgumentException`.

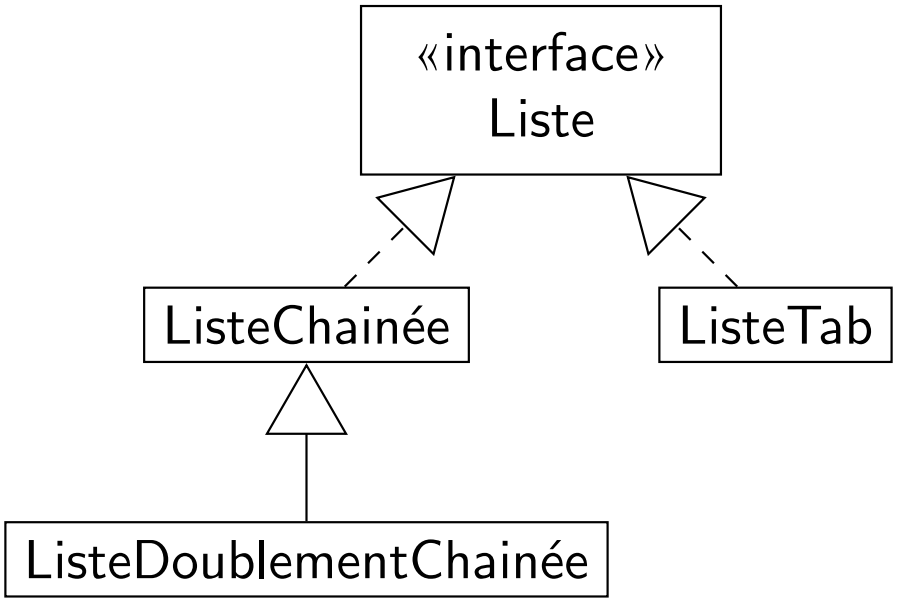
13 - Généricité

Plan de la partie 13 - Généricité

Introduction

On s'intéresse à la modélisation d'ensembles de données classiques : listes, listes chaînées etc.

Nous avons vu que nous pouvons construire une hiérarchie de types entre `Liste`, `ListeTab`, `ListeChainee` et `ListeDoublementChainee` :



On peut ainsi factoriser des comportements communs et en ajouter d'autres suivant les structures (trouver l'élément précédent, l'élément suivant etc).

Rendre la structure de liste générique ?

L'héritage nous permet de factoriser des comportements communs aux différents types de listes et de représenter la hiérarchie de types.

Par contre, que l'on utilise une liste d'entiers, de points ou d'itérateurs, l'algorithmique du fonctionnement de la liste est la même : seuls les **types** des attributs, les **types de retours** et les **types des paramètres** des méthodes changent.

Rendre la structure de liste générique ?

Comment écrire alors la classe `ListeTab` par exemple ?

ListeTab.java

```
public class ListeTab {

  private ?[] tableau;

  ...

  public ? get(int index) {
    return this.tableau[index];
  }

  public add(? element, int index) {
    this.tableau[i] = element;
  }
}
```

Une première approche de la généricité

Une première approche de la généricité : on utilise la classe `Object` (dont héritent toutes les classes).

ListeTabObject.java

```
public class ListeTabObject {

    private Object[] tableau;

    public ListeTabObject(int capacite) {
        this.tableau = new Object[capacite];
    }

    public Object get(int index) {
        return this.tableau[index];
    }

    public void add(Object element, int index) {
        this.tableau[index] = element;
    }
}
```

Problèmes dûs à l'utilisation d'Object

Ajouter des objets de type String dans une liste et les récupérer :

```
ListeTabObject l = new ListeTabObject(10);
l.add("Coucou", 0);            // cela fonctionne grace au
l.add("Blabla", 1);            // sous-typage

String s1 = l.get(0);          // erreur a la compilation
String s2 = (String) l.get(0);
```

C'est au **programmeur** de transtyper le résultat de l'appel à get.

Problèmes dûs à l'utilisation d'Object

Problème « classique » : comme on utilise Object, le compilateur accepte n'« importe quoi ».

```
3         ListeTabObject l = new ListeTabObject(10);
4         l.add("Coucou", 0);            // cela fonctionne grace au
5         l.add(new Integer(3), 1);      // sous-typage
6
7         String s1 = (String) l.get(1); // erreur a l'execution
```

L'erreur se produit à l'**exécution** du programme (ClassCastException).

shell

```
[tof@suntof]~ $ java ListeTabObjectCast
Exception in thread "main" java.lang.ClassCastException:
 java.lang.Integer cannot be cast to java.lang.String
        at ListeTabObjectCast.main(ListeTabObjectCast.java:7)
```

Les types génériques en Java

On cherche donc à pouvoir avoir des types **génériques** (i.e. des « variables » de types) qui puissent être vérifiés **par le compilateur et pas par l'utilisateur**.

En Java, il existe trois types de paramétrisation :

- les types paramétrés (classes et interfaces)
- les méthodes paramétrées
- les classes internes paramétrées (non abordé ici)

Plan de la partie 13 - Généricité

Plan de la partie 13 - Généricité

Définition d'un type paramétré : exemple

Reprenons l'exemple de `Liste`. On peut utiliser une **variable de type** pour déclarer l'interface :

Liste.java

```
import java.util.Iterator;

public interface Liste<T> extends Iterable<T> {

    T get(int index);

    void add(T element, int index);

    Iterator<T> iterator();
}
```

Définition d'un type paramétré

Remarques

- cela fonctionne de la même façon pour les classes :

```
public class ListeTab<T> implements Liste<T> { ... }
```

- `T` est utilisé dans `Liste` comme une **variable**
  - attention, variable est entendue ici comme pouvant accepter différents types (choisis à la création d'une instance du type paramétré)
  - en particulier, on ne peut l'utiliser dans les expressions (`T = Liste` n'a pas de sens par exemple) !
- on dit que `T` est un **paramètre de type formel** de `Liste`

Utilisation d'un type générique : exemple

Création d'une liste d'objets de type `String` :

```
ListeTab<String> l = new ListeTab<String>();
```

Utilisation de la liste :

```
5         l.add("Coucou", 0);    // OK
6         l.add(3, 1);           // erreur a la compilation
7
8         Integer n = l.get(0);  // erreur a la compilation
9         String s = l.get(0);   // OK
```

Utilisation d'un type générique : exemple

Utilisation de la liste :

```
5         l.add("Coucou", 0);    // OK
6         l.add(3, 1);           // erreur a la compilation
7
8         Integer n = l.get(0);  // erreur a la compilation
9         String s = l.get(0);   // OK
```

Les éventuelles erreurs de type sont détectées **à la compilation** :

shell

```
[tof@suntof]~ $ javac ListeCompilErreur.java
ListeCompilErreur.java:6: error: method add in
class ListeTab<T> cannot be applied to given types;
        l.add(3, 1);           // erreur a la compilation
         ^
  required: String,int
  found: int,int
  reason: actual argument int cannot be converted to String by method invocation c
  where T is a type-variable:
    T extends Object declared in
```

Déclaration d'un type paramétré

De façon générale, un type paramétré se déclare de la façon suivante :

Syntaxe (déclaration d'un type paramétré)

```
visibilite [class|interface] nom<T, U, ...> ...
```

Par convention (et pour ne pas confondre les paramètres formels avec les noms de classes et d'interfaces), on utilise les noms `T`, `U`, `S` pour les types paramétrés.

Déclaration d'un type paramétré

On peut utiliser plusieurs paramètres formels de type :

Paire.java

```
public class Paire<T, U> {

    private T element1;
    private U element2;

    public Paire(T element1, U element2) {
        this.element1 = element1;
        this.element2 = element2;
    }
}
```

Utilisation d'un type paramétré

Syntaxe (utilisation d'un type paramétré)

```
TypeParametre<Type1_reel, ...>
```

Par exemple :

```
ListeTab<String> l1 = new ListeTab<String>(); // ne pas oublier les ()

void maMethode(Liste<Integer> l) { ... }
```

Nous verrons plus loin que :

- on peut parfois se passer de l'instanciation des paramètres de type grâce à l'inférence de type
- on peut utiliser des *wildcards*

Plan de la partie 13 - Généricité

Définition d'une méthode paramétrée

On peut également utiliser un paramètre formel de type dans une méthode :

- pour typer un paramètre
- comme type de retour

Pour cela, on déclare les éventuels paramètres de type entre les modifieurs et le type de retour de la méthode :

Syntaxe (méthode paramétrée)

```
public <T, U, ...> T methode(U parametre) ...
```

Définition d'une méthode paramétrée

Exemple :

```
public class AddToList {

    public static <T> void addToAll(T element, ListeTab<ListeTab<T>> l) {
        for (Liste<T> liste : l) {
            liste.add(element, 0);
        }
    }

}
```

Utilisation d'une méthode paramétrée

De la même façon qu'on instancie un type paramétré, on précise à l'appel d'une méthode paramétrée le type instancié des paramètres de type formels :

```
ListeTab<Integer> l1 = new ListeTab<Integer>();
ListeTab<Integer> l2 = new ListeTab<Integer>();

ListeTab<ListeTab<Integer>> l = new ListeTab<ListeTab<Integer>>();
l.add(l1, 0);
l.add(l2, 1);

AddToList.<Integer>addToAll(new Integer(3), l);
```

Le compilateur est également capable de faire de l'**inférence de type quand il n'y a pas d'ambiguïté**. On peut remplacer l'appel précédent par :

```
AddToList.addToAll(new Integer(4), l);
```

Classes paramétrées : on les utilise déjà !

L'API de collections de Java utilise des classes paramétrées. Par exemple nous utilisons depuis longtemps `ArrayList` qui est paramétrée par un type `E`.

On trouve alors dans l'API de `ArrayList` les méthodes

- `void add(int index, E element)`
- `E get(int index)`
- `...`

Plan de la partie 13 - Généricité

Plan de la partie 13 - Généricité

Type erasure

Que se passe-t-il à la déclaration d'un objet de type `Liste<String>` par exemple ?

➥ il n'y a pas de fichier `Liste<String>.class` à charger par la JVM

Java utilise une technique appelée *type erasure* (effacement de type) : le compilateur enlève les informations concernant les paramètres formels de type.
Nous verrons plus tard quels sont les types qui sont alors utilisés dans le code.

Le compilateur ajoute donc au code :

- des **instructions de transtypage** où c'est nécessaire
- de **nouvelles méthodes**, dites méthodes *bridge* (nous n'en parlerons pas ici)

Type erasure : exemple et propriété

Par exemple, le code suivant :

```
Liste<String> l = new ListeTab<String>();
...
String s = l.get(0);
```

est « transformé » en fait par le compilateur en :

```
Liste l = new ListeTab();
...
String s = (String) l.get(0);
```

Principe

Le transtypage implicite ajouté par le compilateur n'échoue (normalement) pas.

Type erasure : conséquences

Lorsque l'on utilise `Liste<String>`, le principe d'effacement de type implique que :

- la JVM ne connaît que le type brut `Liste` (*raw type*)
- les paramètres formels de type sont en fait remplacés par `Object` (ou une autre classe, cf. plus loin).

Conséquence de l'effacement de type

La JVM ne connaît pas les paramètres formels de type !

On dit également que les paramètres de type formels ne sont pas **réifiés**.

Type erasure : conséquences

Un certain nombre d'opérations ne sont pas disponibles car elles sont « effectuées » au *runtime*.

Par exemple, dans une classe utilisant un paramètre formel `T`, les instructions suivantes provoquent une erreur de **compilation** (*unexpected type*) :

- `obj` **`instanceof`** `T`
- **`new`** `T()`
- `T[] tab =` **`new`** `T[15]`
- **`extends`** `T` ou **`implements`** `T`

Type erasure et cast

Que se passe-t-il si on essaye de transtyper une référence en utilisant un type générique ?

Exemple :

```
3         ListeTab<String> l = new ListeTab<String>();
4         Object o = l;
5         ListeTab<String> l2 = (ListeTab<String>) o;
```

On obtient alors un *warning* à la compilation :

```
[tof@suntof]~ $ javac ListeCast.java
Note: ListeCast.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Type erasure et cast

Que se passe-t-il si on essaye de transtyper une référence en utilisant un type générique ?

Exemple :

```
3         ListeTab<String> l = new ListeTab<String>();
4         Object o = l;
5         ListeTab<String> l2 = (ListeTab<String>) o;
```

En recompilant avec l'option `-Xlint:unchecked` :

```
[tof@suntof]~ $ javac -Xlint:unchecked ListeCast.java
ListeCast.java:5: warning: [unchecked] unchecked cast
        ListeTab<String> l2 = (ListeTab<String>) o;
                                                 ^
  required: ListeTab<String>
  found:    Object
1 warning
```

Type erasure et cast

L'avertissement *unchecked cast* est **normal** : comme le type formel n'existe plus à l'exécution, le compilateur ne peut pas garantir qu'il n'y aura pas d'erreurs de transtypage.

ListeCastRun.java

```
3         ListeTab<String> l = new ListeTab<String>();
4         l.add("Coucou", 0);
5
6         Object o = l;
7         ListeTab<Integer> l2 = (ListeTab<Integer>) o; // unchecked cast
8         Integer i = l2.get(0);
```

Attention

L'exception ne se produira pas lors du transtypage, mais lors de l'**utilisation** de l'objet à l'**exécution** (appel de méthode par exemple).

Type erasure et cast

L'avertissement *unchecked cast* est **normal** : comme le type formel n'existe plus à l'exécution, le compilateur ne peut pas garantir qu'il n'y aura pas d'erreurs de transtypage.

ListeCastRun.java

```
3         ListeTab<String> l = new ListeTab<String>();
4         l.add("Coucou", 0);
5
6         Object o = l;
7         ListeTab<Integer> l2 = (ListeTab<Integer>) o; // unchecked cast
8         Integer i = l2.get(0);
```

shell

```
[tof@suntof]~ $ java ListeCastRun
Exception in thread "main" java.lang.ClassCastException:
 java.lang.String cannot be cast to java.lang.Integer
        at ListeCastRun.main(ListeCastRun.java:8)
```

Type erasure et raw types

Un *raw type* est un type paramétré utilisé sans paramètre.

➥ il sert à la compatibilité avec les anciennes versions

Règles

- `Liste l = new Liste<String>(); //OK`
- `Liste<String> l = new Liste(); //unchecked conversion`

Les mêmes principes que pour le transtypage s'appliquent ici.

Plan de la partie 13 - Généricité

Typer les paramètres formels ?

Supposons que l'on ne veuille construire que des listes de nombres. La classe `Number` est la super-classe des classes `Integer`, `Double` etc.

On peut alors utiliser cette classe comme **borne** de notre paramètre de type formel :

ListeNumber.java

```
public class ListeNumber<T extends Number> {


}
```

Remarque

C'est cette borne qui sert lors de l'effacement de type.

Utilisation d'une borne : vérification

Lors de la déclaration ou de la création d'un objet de type `ListeNumber<T>`, `T` doit respecter la contrainte imposée.

Par exemple, la déclaration et l'affectation suivantes :

```
ListeNumber<String> l = new ListeNumber<String>();
```

provoquent les erreurs suivantes à la **compilation** :

shell

```
[tof@suntof]~ $ javac ListeNumberCreateErreur.java
ListeNumberCreateErreur.java:3:
 error: type argument String is not within bounds of type-variable T
       ListeNumber<String> l = new ListeNumber<String>();
                    ^
  where T is a type-variable:
    T extends Number declared in class ListeNumber
ListeNumberCreateErreur.java:3:
 error: type argument String is not within bounds of type-variable T
       ListeNumber<String> l = new ListeNumber<String>();
                                               ^
  where T is a type-variable:
    T extends Number declared in class ListeNumber
```

Des bornes plus compliquées...

On peut utiliser une interface pour contraindre un paramètre formel de type :

```
public class MaClasse<T extends I1> { }
```

Si l'on veut à la fois utiliser une classe et des interfaces dans une borne, on utilise le symbole & :

```
public class MaClasse<T extends C & I1 & I2> { }
```

Dans ce cas, `C` doit **obligatoirement être une classe** (première borne).

Des bornes plus compliquées...

On peut également utiliser des paramètres formels de type dans les bornes, mais dans ce cas, ils ne peuvent pas être utilisés avec d'autres bornes

MaClasseBoundsErreur.java

```
class MaClasseBoundsErreur<U, T extends U & Cloneable> { }
```

shell

```
[tof@suntof]~ $ javac MaClasseBoundsErreur.java
MaClasseBoundsErreur.java:1: error: a type variable may not be followed
 by other bounds
class MaClasseBoundsErreur<U, T extends U & Cloneable> { }
                                          ^
1 error
```

Type d'un paramètre formel de type

Quel est le type d'un paramètre formel de type dans du code générique ?

Définition (type d'un paramètre de type formel)

Le type d'un paramètre de type formel dans du code générique est celui de sa borne.

Attention, cette propriété « empêche » l'utilisation de la surcharge par exemple.
En effet, la surcharge est résolue à la **compilation**, c'est donc le type de la borne qui sera utilisé pour la résolution de la surcharge.

Type d'un paramètre formel et surcharge

TypeDynamicBinding.java

```
public class TypeDynamicBinding<T> {

    private T att;

    public TypeDynamicBinding(T att) { this.att = att; }

    public void mdyn(Object o) {
        System.out.println("C'est un Object");
    }

    public void mdyn(Integer o) {
        System.out.println("C'est un Integer");
    }

    public void appelDyn() {
        this.mdyn(this.att);
    }

    public static void main(String[] args) {
      new TypeDynamicBinding<Object>(new Object()).appelDyn();
      new TypeDynamicBinding<Integer>(new Integer(2)).appelDyn();
    }
}
```

Type d'un paramètre formel et surcharge

À l'exécution :

shell

```
[tof@suntof]~ $ java TypeDynamicBinding
C'est un Object
C'est un Object
```

Plan de la partie 13 - Généricité

Héritage et type paramétré

Une classe peut hériter/réaliser une classe/interface paramétrée instanciée ou pas :

```
public class ListeEntier implements Liste<Integer> { }

public class ListeChainee<T> implements Liste<T> { }
```

Héritage et type paramétré

Une classe ne peut pas réaliser la même interface paramétrée instanciée avec des types différents :

MaComp.java

```
public class MaComp implements Comparable<String>, Comparable<Integer> {
    public int compareTo(String s) { return 0; }
}
```

shell

```
[tof@suntof]~ $ javac MaComp.java
MaComp.java:1: error: repeated interface
public class MaComp implements Comparable<String>, Comparable<Integer> {
                                                                ^
MaComp.java:1: error: Comparable cannot be inherited with
 different arguments: <java.lang.String> and <java.lang.Integer>
public class MaComp implements Comparable<String>, Comparable<Integer> {
       ^
2 errors
```

Héritage entre types paramétrés

Deux types paramétrés **instanciés avec le même type** peuvent être liés par une relation de sous-typage.

Par exemple, si on a :

ListeChainee.java

```
public class ListeChainee<T> implements Liste<T> { ... }
```

alors `ListeChainee<String>` est bien un sous-type de `Liste<String>`.

Héritage entre types paramétrés

On sait que la classe `Integer` hérite de la classe `Number`. Est-ce que pour autant `Liste<Integer>` hérite de `Liste<Number>` ?

```
        ListeTab<Integer> li = new ListeTab<Integer>();
        ListeTab<Number> ln = li;
```

Non ! On a une erreur *incompatible types* à la compilation :

shell

```
[tof@suntof]~ $ javac ListeHeritageErreur.java
ListeHeritageErreur.java:4: error: incompatible types
        ListeTab<Number> ln = li;
                              ^
  required: ListeTab<Number>
  found:    ListeTab<Integer>
1 error
```

C'est tout à fait normal : une liste d'entiers ne peut pas se **substituer** à une liste de nombres.

ListeTab<Number> et ListeTab<Integer>

Si une liste d'entiers pouvait se substituer à une liste de nombres, alors le code suivant serait licite :

```
ListeTab<Integer> lInt = new ListeTab<Integer>();
lInt.add(1, 0);
lInt.add(5, 1);

ListeTab<Number> lNum = lInt;
lNum.add(3.14, 2);        // oups
```

Wildcards

Supposons que l'on veuille écrire une méthode `m` prenant en paramètre une liste de nombres :

```
public double somme(Liste<Number> l) {
    double somme = 0;

    for (Number n : l) {
        somme += n.doubleValue();
    }

    return somme;
}
```

Dans ce cas, l'appel suivant ne fonctionnera pas (cf. transparent précédent) :

```
Liste<Integer> l = new ListeTab<Integer>();
...
System.out.println(somme(l));
```

Wildcards

Pour pouvoir représenter le fait que la méthode `m` prend en paramètre une liste d'objets qui doivent être des sous-types de `Number`, on utilise le symbole ? appelé **wildcard** :

```
public void somme(Liste<? extends Number> l) { ... }
```

Le *wildcard* permet de représenter une **famille** de types concrets.

Un exemple dans l'API de `ArrayList<E>` :

```
addAll(Collection<? extends E> c)
```

Wildcard et borne supérieure

On peut utiliser un *wildcard* avec une borne supérieure comme précédemment :

```
public void m(Liste<? extends Number> l) { ... }
```

Que peut-on utiliser dans cette méthode ?

```
public void m(Liste<? extends Number> l) {
    Number n = l.get(0);   // ok
    String s = l.get(0);   // ne compile pas !
    Integer i = l.get(0);  // ne compile pas !

    l.add("Coucou", 0);       // ne compile pas !
    l.add(new Integer(3), 0); // ne compile pas !
    l.add(null, 0);           // c'est la seule solution
}
```

Wildcard et borne inférieure

On peut également utiliser un *wildcard* avec une borne inférieure :

```
public void m(Liste<? super Integer> l) { ... }
```

Dans ce cas, les problèmes d'utilisation du paramètre sont inversés par rapport au cas de la borne supérieure :

```
public void m2(Liste<? super Integer> l) {
    l.add(new Integer(3), 0); // ok
    l.add(new Object(), 0);   // ne compile pas !

    Integer i = l.get(0);  // ne compile pas !
    Object o = l.get(0);   // c'est la seule solution
}
```

Plan de la partie 13 - Généricité

Redéfinition de méthode : type de retour

On peut redéfinir une méthode paramétrée :

```
class MaClasse {
    <T> T methode() {
        return null;
    }
}

class MaClasseExt extends MaClasse {
    @Override <U> U methode() {
        return null;
    }
}
```

Redéfinition de méthode : type de retour

Attention, U doit avoir une borne qui est la même que celle utilisée pour T :

```
 1 class MaClasseBornee {
 2     <T extends Number> T methode() {
 3         return null;
 4     }
 5 }
 6
 7 class MaClasseBorneeExt extends MaClasseBornee {
 8     @Override <U extends Integer> U methode() {
 9         return null;
10     }
11 }
```

shell

```
[tof@suntof]~ $ javac RedefinitionMethErreurSsType.java
RedefinitionMethErreurSsType.java:8:
 error: method does not override or implement a method from a supertype
    @Override <U extends Integer> U methode() {
    ^
1 error
```

Redéfinition de méthode : arguments

On peut également redéfinir une méthode ayant des arguments typés par un paramètre formel de type :

```
class MaClasseParam {
    <T extends Number> void methode(T element) { }
}

class MaClasseParamExt extends MaClasseParam {
    @Override <U extends Number> void methode(U element) { }
}
```

Redéfinition de méthode : arguments

Attention, les deux paramètres formels de type doivent avoir la même borne. En particulier, le sous-typage ne fonctionne pas !

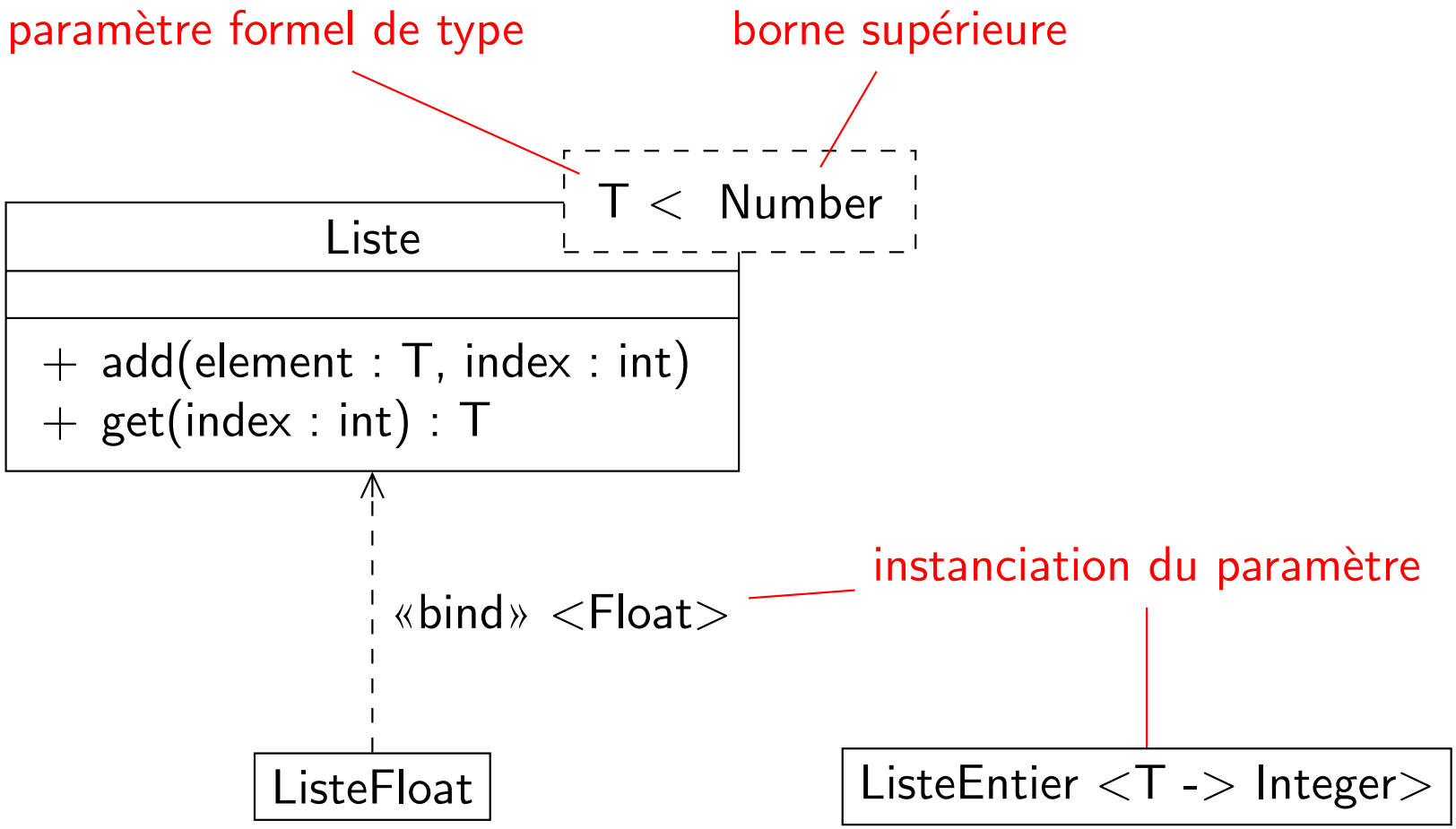
```
1 class MaClasseParamErreur {
2     <T extends Number> void methode(T element) { }
3 }
4
5 class MaClasseParamExtErreur extends MaClasseParamErreur {
6     @Override <U extends Integer> void methode(U element) {  }
7 }
```

shell

```
[tof@suntof]~ $ javac RedefinitionMethErreurBorne.java
RedefinitionMethErreurBorne.java:6:
 error: method does not override or implement a method from a supertype
    @Override <U extends Integer> void methode(U element) {  }
    ^
1 error
```

Plan de la partie 13 - Généricité

Représentation UML

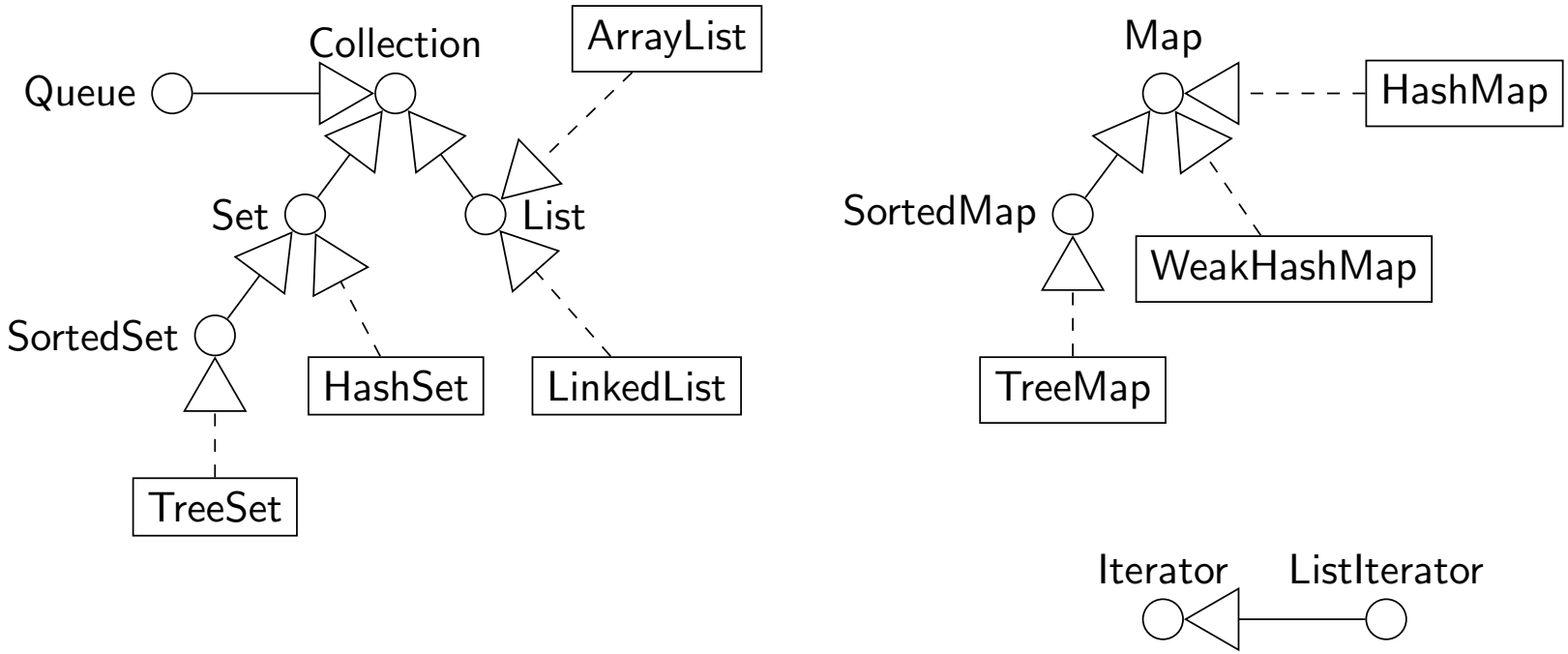


Plan de la partie 13 - Généricité

Les collections : classes et interfaces de base

La notion de **collection** est un élément important de tout langage orienté objet : elle permet de représenter un ensemble d'objets.

Le SDK de Java propose plusieurs interfaces et classes représentant différentes implantations d'une collection. **Ces types sont tous des types génériques**.



Pourquoi différentes implantations ?

Il existe plusieurs interfaces et classes pour les collections :

- pour différentes représentations des ensembles : séquence, arbre, etc.
- pour des questions de complexité algorithmique sur des opérations d'insertion, de recherche etc.

On notera que toutes les collections utilisent le mécanisme d'**itérateur** vu précédemment en TP.

On peut également utiliser la boucle **`for`** particulière déjà vue lors de l'utilisation de la classe `ArrayList`.

Remarque

La boucle **`for`** est utilisable sur tout sous-type de l'interface `Iterable`.

On pourra se référer à la bibliographie donnée pour plus de détails.

Références


Langer, Angelika (2013).
Java Generics FAQ.
http : / / www . angelikalanger . com / GenericsFAQ / JavaGenericsFAQ.html.


Oracle (2013).
The Collections framework.
http : / / docs . oracle . com / javase / 7 / docs / technotes / guides/collections/index.html.


Cormen, T.H. et al. (2004).
Introduction à l'algorithmique.
2e éd.
Dunod.


Naftalin, M. et P. Wadler (2006).
Java Generics and Collections.
O'Reilly.