

Introduction

Fonctionnement

- 1 heures de cours par semaine
- 1.5 heures de TD par semaine
- 2 heures de TP tous les 15 jours

- 3 devoirs surveillés (1 feuille A4 recto verso autorisée)
- 1 ou 2 projets à faire en TP

- Le tout pendant 17 semaines !

Présentation de JAVA

- Langage de Programmation développé chez Sun Microsystems
(www.sun.com)
- Première version : Début 96.
- 1998 : version 1.2 (java 2)
- fin 2004 : version 1.5, encore appelé Java 5
- <http://java.sun.com>
- <http://penserensjava.free.fr>

Java est

- Objet
- Simple
- efficace
 - ◆ du point de vue du développement
 - ◆ du point de vue de la rapidité (à voir !)
- complet
- gratuit
- portable

Java est orienté objet

- Java est fortement objet : Tout est objet excepté certains types primitifs
⇒ Plus proche de SmallTalk que de C++
- Pas de variables et de fonctions en dehors des objets
- Attention : C++ pour garder les utilisateurs de C

Java est Simple

- Mise à profit de 20 ans de programmation
- Allégé des sources d'erreurs de C/C++ (pointeur, gestion mémoire)
- OUF : Syntaxe TRES similaire à celle de C/C++ qui a fait ses preuves
 - ◆ Mêmes instructions, structures de contrôles
 - ◆ Mais pas de pré-processeur, ni de fichiers en-tête, ni de structures/union
- **Attention** ⇒ Langage de Programmation + Objet : Pas aussi simple que cela

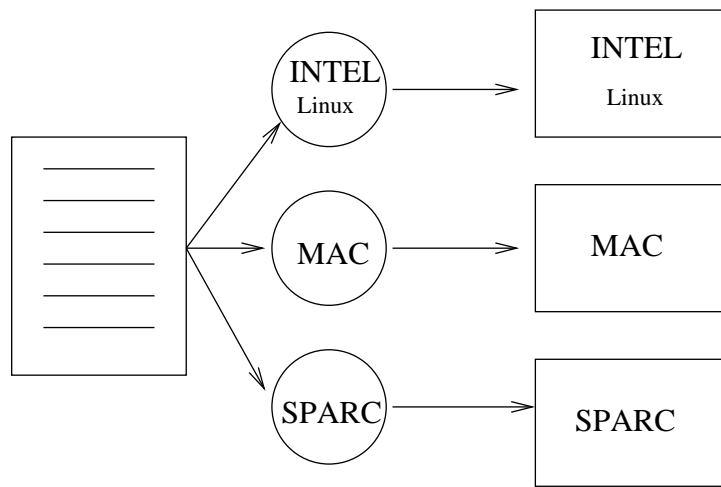
Java est Efficace (Développement)

- Développer en JAVA est 2 fois plus rapide qu'en C++
 - ◆ Beaucoup de vérifications sont faites à la compilation
 - ◆ Pas de de gestion mémoire \Rightarrow Pas de fuites
 - ◆ Beaucoup d'API : Gestion des threads, du fenêtrage...
- AIE : JAVA peut être 2 fois plus lent que C++
 - ◆ Pour de nombreuses applis la vitesse n'est pas importante
 - ◆ Java et Internet : Débit réseau/Vitesse processeur

Java est complet et gratuit

- Téléchargeable sur `java.sun.com`
- Java SDK : Outils de base permettant de créer/debugger/exécuter des applications
- IDE : Environnement de développement JAVA
- Tutorial, API...
- **Attention** Microsoft Java n'est pas standard

Java est portable

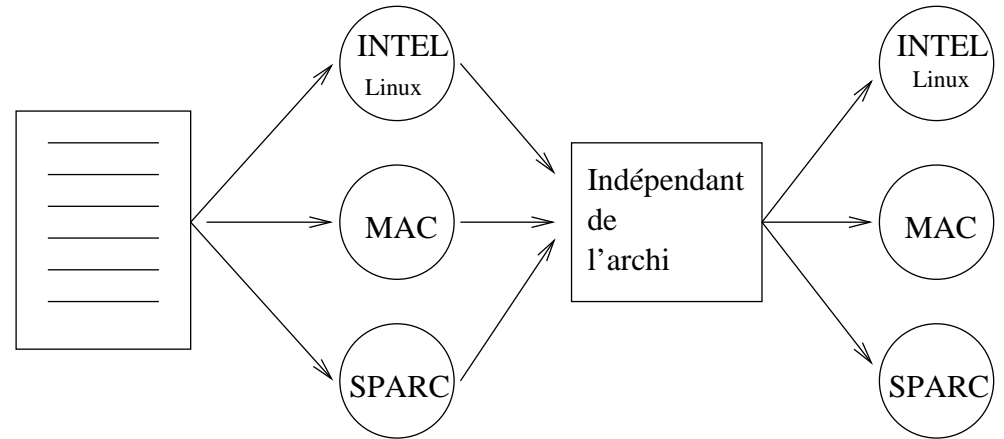


Source

Compilateur

Executable

Classique



Source

Compilateur
JAVAC

Pseudo Code
(Byte Code)

Interpréteur

JAVA

Machine Virtuelle Java (JVM)

Java est portable

- Tout fichier .class peut être exporté vers une autre machine
- La JVM effectue de nombreuses vérifications
- La JVM doit interpréter les instructions
- Java est donc un langage interprété.

Mais attention, il est plus proche de C/C++ que du Basic

Le futur de Java

JAVA futur standard ?

- Un réel engouement : de nombreux postes sur le marché du travail
- L'utilisation du réseau
- Les puces Java (téléphone portable....)

- Problème de compatibilité
- Java est jeune : 15 ans
- Inertie du monde de la programmation : La plus grande communauté de programmeurs est celle du COBOL !!

Rappels

Conventions de nommage

- Un nom de classe commence par une majuscule
- Un nom de variable (champ) ou de méthodes commence par une minuscule.
- Si le nom est composé de plusieurs mots, les suivants commencent par une majuscule
- Documentation disponibles :

`http://java.sun.com/docs/codeconv/`

non respect des conventions → points en moins aux DS

Pourquoi des conventions

- Amélioration de la lisibilité
- 80 % du temps passé sur un logiciel va vers la maintenance
- L'auteur original peut être substitué
- Portabilité : Grandes quantités d'API : sans conventions, point de salut

Types Primitifs

- Java est fortement objet
- Oui, mais....Pour simplifier la programmation, il existe des types primitifs
- Les types primitifs sont : boolean, char, byte, short, int, long, float, double
- On en déduit les variables primitives
- Tout type primitif est signé (pas de unsigned comme en C)
- Tout type primitif est invariant en taille quelque soit la machine

Type Primitifs en C/C++

Le type int est codé :

- sur 2 octets sur un 8086 (processeur 16 bits)
- sur 4 octets sur un Sparc
- sur 2 octets sur un Pentium sous Dos/Win3.1
- sur 4 octets sur un pentium sous Win95/NT...

⇒ **Obligation de compiler le code suivant l'architecture**

- Pas de portabilité
- Un peu plus de rapidité

Nombres flottants et Comparaison

- Il est nécessaire d'être très prudent avec les valeurs réelles lorsqu'elles sont utilisées au niveau de condition de structures de contrôles

```
double a = 1.0 ;  
double b = 1.0 ;  
while (((a + 1.0) - a) - 1.0 == 0.0) a = 2*a ;  
while (((a + b) - a) - b != 0.0) b = b + 1.0 ;  
return b ;
```

- Que vaut b ??
 - ◆ L'API java fournit une fonction pour comparer des doubles !!
 - ◆ La connaissance de l'API et la recherche d'information est essentielle

Le transtypage

`(typeCast) expression`

- Cette opération permet de convertir le type de l'expression en typeCast
- UpCasting : augmente la généralité du type : aucune risque
- DownCasting : diminue la généralité du type : comporte des risques
- Du moins général au plus général, on a :
 - ◆ `byte → short → int → long → float → double`

Le transtypage implicite

- Lors d'une affectation, une opération de upcasting implicite peut avoir lieu.

Exemple

```
double x = 15 ;
```

- La valeur 15 est transformée automatiquement en valeur double 15.0 avant d'être affectée à x

Le transtypage explicite

- Lors de certaines opérations, il est parfois nécessaire d'utiliser explicitement une opération de transtypage

Exemple

```
int x = 3, y = 2 ;  
double z = x/y ; /* z = 1.0 */
```

- Pour que z est la valeur 1.5, il faut faire

```
double z = (double)x/y ;
```

Opérateurs

Les opérateurs

- Arithmétiques classiques : +, -, *, /, % (modulo)
 - ◆ Attention la division de deux nombres entiers est un nombre entier
- incrémentation et décrémentation : ++,--
 - ◆ Opérateurs unaires
 - ◆ En position préfixe, ou postfixe
 - ◆ A utiliser avec précaution
- Opérateurs relationnels \leq , $<$, \geq , $>$, $==$, $!=$
 - ◆ On ne peut pas utiliser = à la place de == (OUF)

Les opérateurs

- Opérateurs logiques : $\&\&$, $\|\|$, $!$
 - ◆ Ils sont évalués de manière optimisée (comme en C)
- Opérateurs bits à bits : $\&$, $|$, \wedge , \sim
 - ◆ Ils s'utilisent avec des entiers
- Opérateurs de décalage : \gg , \ll
 - ◆ Ils s'utilisent avec des entiers
 - ◆ Décaler d'une position à gauche revient à multiplier le nombre par 2.

Les opérateurs

- Opérateurs d'affectation =
 - ◆ il peut être combiné avec des opérateurs arithmétiques
 - ◆ il peut être combiné avec des opérateurs bits à bits
- Opérateur ternaire (`cond ? opVrai : opFaux`)

Opérateurs et priorité : Mieux vaut mettre des parenthèses

Les instructions

Les instructions

- Une instruction peut être
 - Une affectation : `var = expr ;`
 - Une structure de contrôle
 - ◆ Conditionnelle : `if`, `switch`
 - ◆ Itérative : `while`, `for`, `do/while`
 - ◆ de saut : `break` `continue`
 - ◆ de retour de fonction : `return`
 - ◆ d'appel de méthodes d'objets
 - ◆ de création d'objets
- Un bloc d'instructions est une séquence d'instructions délimitée par des accolades
- Possibilité de déclarer des variables locales à un bloc

Portée de variables locales

```
{  
int x = 12;  
// Seul x est accessible  
    {  
    int q = 96;  
    // x et q sont accessibles  
    ...  
    }  
// seul x est accessible  
// q est hors de portée  
...  
}
```

Variables homonymes

Cacher une variable de portée plus étendue

- Programmation confuse, Source d'erreurs, Intérêt limité (nul)

```
{  
int x = 12;  
  {  
    int x = 96;  
    ...  
  }  
  ...  
}
```

- Autorisé en C/C++, **Interdit** en Java

Initialisation des variables locales

- Nombreuses erreurs de programmation sont dues à un oubli d'initialisation
- Erreurs très difficiles à trouver
- En Java : Obligation d'initialiser les variables locales

```
{  
int x;  
x = x + 1 ;  
// Erreur de compilation  
  
...  
}
```

Instruction non accessibles

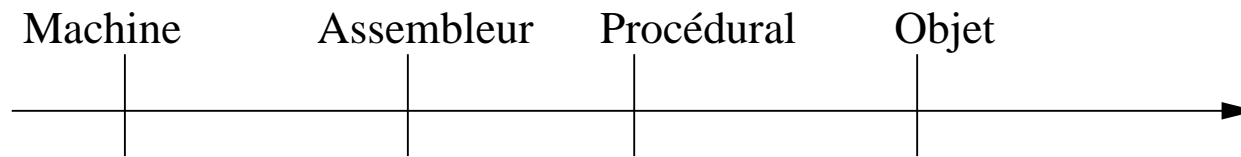
```
...  
if(1 ≤ 2) return a;  
a = a + 1;  
...
```

⇒ Erreur de compilation

Conception Procédurale VS Conception Objet

Les langages : Où en est t'on ?

But d'un langage : Fournir de l'abstraction pour simplifier les problèmes.



Programmation Procédurale

- Représentation d'une solution algorithmique en terme de procédures
- Permet de représenter et de structurer
 - ◆ les instructions (sous forme de procédures)
 - ◆ et les données (sous forme de types)
- Utilisable pour de petites applications
- Difficiles à utiliser pour de grosses applications : Problèmes de maintenance....

Programmation Objet

- Tout problème se ramène à des objets
- Modélisation plus naturelle et plus aisée
- Permet plus facilement de concevoir une architecture créateur/utilisateur
 - dans le sens serveur : créateur de classe
 - client : utilisateur de classe
- Adapté à la conception de GROSSES applications

Pourquoi ?

- Code robuste
- Code réutilisable (plus que dans la programmation procédurale)
- Façon de penser intuitive : que contient un objet, que fait un objet...

Un petit problème

- On doit créer un programme assurant que chaque élève connaît la salle de classe où se tiendra son prochain cours et qu'il sait comment s'y rendre

Programmation procédurale

- Voici les différentes étapes

1. Dresser La liste des élèves

2. Pour chaque élève

- (a) Identifier la salle où se déroule son prochain cours

- (b) Trouver l'emplacement de cette salle

- (c) Déterminer le chemin y conduisant

- (d) Indiquer à l'élève la façon de s'y rendre

```
allerEnClasseSuivante(Eleve e,Chemin c)
```

Une autre approche

- Afficher la liste des salles hébergeant les prochains cours
- Afficher les itinéraires conduisant aux diverses salles
- Demander aux élèves, censés connaître leur prochain cours, de se rendre dans leur salle

```
e.allerEnClasseSuivante()
```

Différence des deux approches

- Dans la première, en donnant des directives spécifiques, vous endossez toutes les responsabilités. Vous devez vous préoccuper de tous les détails.
- Dans la seconde, en ne fournissant que des instructions générales, vous responsabilisez les acteurs. Vous pouvez vous concentrer sur la globalité du problème

Délégation de responsabilité

Une modification

- Supposons que certains élèves (les 2emes années) doivent passer par le secrétariat. Quels changements cela impose t il ?
 - ◆ Dans le premier cas, il convient de distinguer les 2emes années des autres élèves, afin de les faire passer par le secrétariat. Ceci peut entraîner un gros bouleversement du programme
 - ◆ Dans le second cas, les élèves sont responsables. Vous n'avez qu'à écrire une procédure supplémentaire pour les 2emes années, et ceci, sans modifier le programme principal

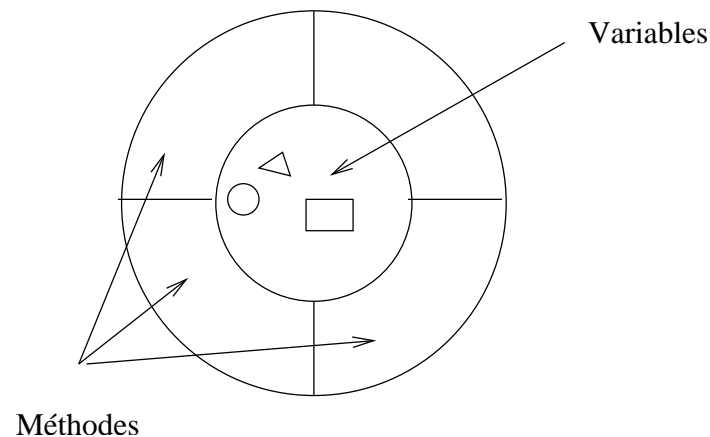
Points importants

- La réussite de cette approche dépend de trois facteurs :
 1. Les individus sont responsables d'eux mêmes
 2. Le programme moniteur doit communiquer de façon unique avec tous les individus (1ere, 2eme, et AS)
 3. Le programme moniteur n'a pas à se soucier des détails individuels concernant les élèves lors de leur déplacement

Les objets

Qu'est ce qu'un objet ? ?

- Un objet peut être vu comme un ensemble de données et de fonctions qui lui sont associées



- Un objet est responsable de ses données
- Un objet a connaissance de son type, et par ses données propres, de son état
- Le comportement d'un objet est dicté par le code de ses méthodes

Interface

- Afin d'assumer leurs responsabilités, les objets doivent disposer d'un moyen de communication
 - ◆ Les données dont ces objets disposent
 - ◆ Les méthodes qui réalisent leurs fonctionnalités
- Notre objet `Eleve` contient la méthode `allerEnClasseSuivante()`
- Pas de paramètre dans la méthode
 - ◆ L'objet sait ce qu'il lui faut pour se déplacer
 - ◆ Sait trouver les infos pour accomplir cette tâche

Un peu de terminologie

- On appelle Classe toute description d'objet. C'est le patron de l'objet. Elle définit ses données et ses méthodes.

Une classe peut être vue comme un type de donnée particulier

- On appelle instance, un objet particulier créé à partir d'une classe

Les différents étudiants seront des instances de la classe `Eleve`

Conception Objet

- Le choix des objets est primordial
- Nécessite une phase de réflexion !!
- Pour faciliter la tâche : le génie logiciel....
 - ◆ Diagrammes UML
 - ◆ Designs patterns

Aparté

- On distingue 3 niveaux dans le développement d'un logiciel
 1. *Conceptuel* : Seuls les concepts relatifs aux problèmes sont étudiés
L'objet représente un ensemble de responsabilités
 2. *Spécification* : Prise en compte du logiciel, définition des interfaces
L'objet devient un ensemble de méthodes
 3. *Implémentation* : Le code arrive

Aparté

- On distingue 3 niveaux dans le développement d'un logiciel
 1. *Conceptuel* : Seuls les concepts relatifs aux problèmes sont étudiés
L'objet représente un ensemble de responsabilités
 2. *Spécification* : Prise en compte du logiciel, définition des interfaces
L'objet devient un ensemble de méthodes
 3. *Implémentation* : Le code arrive

FAITES DES PETITES FONCTIONS !!!

- N'oubliez pas l'utilité des fonctions :
 - ◆ Éviter d'écrire plusieurs fois le même code
 - ◆ décomposer une action en plusieurs petites actions !!!

Java et les objets

- Les objets sont manipulés par des références
- Les identificateurs des objets sont en fait des références
- Les objets sont instanciés par allocation dynamique (opérateur new). On a alors créé une instance d'objet
- Il peut exister plusieurs instances d'un même objet (classe)

Java et les références

- Les instances d'objets sont manipulées à l'aide de référence

```
Eleve toto ;
```

- On a créé une référence de l'objet `Eleve`, mais on ne peut encore s'en servir

```
toto = new Eleve() ;
```

- On a associé cette référence à une instance de l'objet. Maintenant, on peut utiliser cette référence.

La référence null

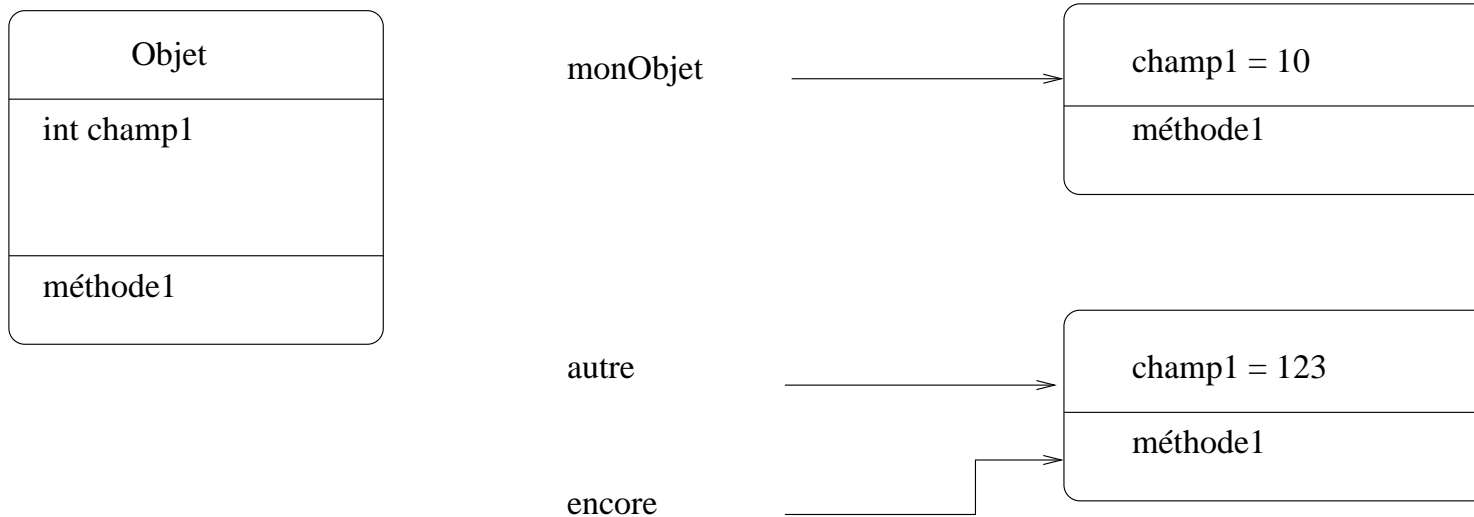
- La valeur initiale d'une référence est **null**

Elle ne pointe sur rien

```
Eleve toto ; /* toto pointe sur null*/
```

Représentation

```
class Test {  
    public static void main(String[] s) {  
        Objet monObjet; // La référence  
        monObjet = new Objet(); // On crée l'instance  
        Objet autre = new Objet(); // Autre façon  
        Objet encore = autre;  
    }  
}
```



Accès aux champs et aux méthodes

- A l'intérieur d'une classe, on les appelle directement par leur nom

```
Class Toto {  
    int champ1 ;  
    void incChamp1() {  
        champ1++ ;  
    }  
    ...  
}
```

Accès aux champs et aux méthodes

- Les autres classes accèdent généralement aux membres d'un objet par l'intermédiaire d'une référence

```
Toto monObjet ;  
monObjet = new Toto() ;  
monObjet.champ1 = 2 ;  
char c = monObjet.incChamp1() ;
```

L'encapsulation

Exemple

- On propose une implémentation toute simple de l'objet Date
- On se trouve ici du côté concepteur (programmeur de la classe Date)

```
class Date {  
    int j,m,a ;  
    void afficher() {  
        System.out.print(j+"."+m+"."+a");  
    }  
}
```

Exemple ... Suite

- On se trouve ici du côté client (utilisateur de la classe Date)

```
public static void main(String [] args) {  
    Date inst = new Date();  
    inst.j = 31;  
    inst.m = 2; /* Erreur de cohérence*/  
    inst.a = 2002;  
    inst.afficher();  
}
```

Exemple II - Le retour

- Pour des raisons obscures, on souhaite
 - ◆ coder le jour et le mois sur un seul entier
Utilisation de \gg et \ll
 - ◆ coder l'année sur un autre entier

```
class Date {  
    int jm ;  
    int a ;  
    void afficher() {  
        System.out.print((jm>>16)+"."  
            +((jm<<16)>> 16)+"."+"a");  
    }  
}
```

Exemple II - le retour ... Suite

```
public static void main(String [] args) {  
    Date inst = new Date();  
    inst.j = inst.j | (31 << 16);  
    inst.m = inst.j | 2; /* Toujours PB*/  
    inst.a = 2002;  
    inst.afficher();  
}  
}
```

Solution Catastrophique

Remarques

- internet : Une multitude de bibliothèques (API) prêtes à l'emploi
- Les utilisateurs d'une bibliothèque doivent être sur qu'ils n'auront pas à réécrire du code si une nouvelle version de celle-ci sort.
- On doit à tout prix préserver la cohérence
 - ◆ Ne pas supprimer des méthodes existantes
 - ◆ Rendre la partie interface (utilisateur) aussi simple que possible

Séparer la partie utilisateur (interface) et la partie implémentation

L'encapsulation

- Cacher les données d'une classe. Seules certaines méthodes sont accessibles
- Meilleure structuration
- Meilleure gestion de la cohérence
- Maintenance beaucoup plus facile

Les Spécificateurs Java : **public, private**

Les spécificateurs

- **public** : une méthode ou un champ déclaré public est accessible à partir de n'importe quelle classe
- **private** : une méthode ou un champ private n'est pas accessible en dehors de la classe
- En pratique : tous les champs seront déclarés **private**
- utilisation de méthodes **get** et **set** pour accéder aux différents champs (convention)
- Attention : Mettre les getters et setters uniquement si cela est **NÉCESSAIRE**

Visibilité

```
class Vision {  
    private int a;  
    private int b;  
  
    private int f() {  
        ...  
    }  
  
    public void visible1() {  
        ....  
    }  
  
    public int visible2(int c) {  
        ....  
    }  
}
```

Visibilité

```
class Vision {
```

```
private int a;  
private int b;  
private int f() {  
    ...  
}
```

```
public void visible1() {
```

```
    ....  
}
```

```
public int visible2(int c) {
```

```
    ....  
}
```

```
}
```

Exemple III - la dernière !

- Version avec 3 entiers + fonction main
- Version avec deux entiers + fonction main
- ces deux versions ont toujours un problème de cohérence !
- On crée une dernière version, qui change la date comme un triplet et vérifie la cohérence

Les mérites de l'encapsulation

- Les méthodes sont simples puisqu'il n'y a pas à se soucier des particularités de l'implémentation
- Les modifications de l'implémentation peuvent se faire en totale indépendance vis à vis de l'appelant

Les mérites de l'encapsulation

- Plus on responsabilise les objets dans leur comportement, plus le programme moniteur s'en trouve déchargé et plus il est simple
- L'encapsulation rend la modification du comportement interne d'un objet totalement transparente aux autres objets
- L'encapsulation permet de réduire les effets secondaires indésirables

La surcharge

Introduction à la surcharge

- Redondance dans le langage humain (le sens se déduit du contexte)

exemple : *conduire*

conduire une moto

conduire une voiture

- La plupart des langages de programmation imposent des noms uniques pour chaque fonction

```
conduireMoto(une_moto)
```

```
conduireVoiture(une_voiture) ;
```

- POURQUOI ??

La surcharge

- capacité de définir plusieurs méthodes de même nom dans une même classe.
- bien entendu, il vaut mieux qu'elles aient le même objectif !!
- caractéristique puissante et utile
- lorsque la méthode est appelée, le compilateur choisit la méthode qu'il convient d'exécuter d'après les arguments
- Chaque méthode surchargée doit prendre une liste **unique** de types de paramètres
 - ◆ comment le développeur lui-même pourrait-il choisir entre deux méthodes du même nom, autrement que par le type des paramètres ?

Exemple

```
class Surcharge {  
    void testons() {  
        System.out.print("Aucun");  
    }  
    void testons(int i) {  
        System.out.print("Un int" + i);  
    }  
    void testons(double i) {  
        System.out.print("Un double " + i);  
    }  
    void testons(int i, double a) {  
        System.out.print("Deux " + i + a);  
    }  
}
```

Exemple

```
static public void main(String[] args) {  
    int x = 1 ;  
    double a=3.0 ;  
    Surcharge s = new Surcharge() ;  
    s.testons() ;  
    s.testons(x) ;  
    s.testons(2) ;  
    s.testons(7.0) ;  
    s.testons(a) ;  
    s.testons(x,a) ;  
}
```

Exemple II - Le retour

```
class Surcharge {  
    void testons() {  
        System.out.print("Aucun");  
    }  
    void testons(int i) {  
        System.out.print("Un int" + i);  
    }  
    void testons(int i,double a) {  
        System.out.print("Deux " + i + a);  
    }  
}
```

Exemple II - Le retour

```
static public void main(String[] args) {  
    int x = 1 ;  
    double d=3.0 ;  
    Surcharge s = new Surcharge() ;  
    s.testons() ;  
    s.testons(x) ;  
    s.testons(2) ;  
    s.testons(7.0) ; /* Erreur de compilation*/  
    s.testons(d) ; /* Erreur de compilation*/  
    s.testons((int)d) ; /* ok */  
    s.testons(x,a) ;  
}
```

Encore un exemple

- Voir le programme `Point.java` donné en annexe

Surcharge de type de retour

```
class SurchargeType{    ...  
    int f() {...};  
    void f() {...};  
    ...
```

```
x = f(); // OK  
f(); //??
```

- Comment savoir quelle fonction utiliser ?
- Engendre des difficultés de compréhension
- **Interdit en JAVA**

Surcharge des opérateurs

- En C++, on peut surcharger les opérateurs +, - ...
- Intéressant par exemple avec la classe `NbComplexe`
- Le code est plus élégant
- Plus difficile à maintenir ?
- **Interdit** en Java

Initialisation et autres destruction

Initialisation

- La programmation sans garde-fou est la principale cause des coûts de développements excessifs
- De nombreux bugs viennent d'un oubli d'initialisation de variables.
- L'utilisation de bibliothèques augmente ce risque
faut il initialiser ? comment ?
- Il serait intéressant d'utiliser une méthode d'initialisation pour chaque classe

La solution : **Le constructeur**

Les constructeurs

- Objectif : initialise de façon cohérente l'état d'un objet
- Lors de toute création d'instances (`new`),
 - ◆ de l'espace mémoire est alloué
 - ◆ le constructeur est appelé
 - ◆ L'objet sera obligatoirement initialisé avant qu'il puisse être manipulé.
- Définition et initialisation sont des concepts unifiés. Il est impossible d'avoir l'un sans l'autre

Les constructeurs II

- Un constructeur est appelé automatiquement. Et c'est l'unique appel durant toute la vie de l'instance
- Il porte le même nom que la classe (majuscules-minuscules compris). Pas de conventions de nommage ici...
- N'a pas de type de retour (\Rightarrow pas de `return`)
- Si aucun constructeur ne figure dans une classe, un constructeur par défaut est appelé. Celui-ci ne contient pas d'arguments

Exemples

```
class Date {  
    private int j,m,a;  
    Date() {  
        setDate(1,1,2002);  
        System.out.println("Le constructeur Date()  
            est appelé");  
    }  
}
```

```
Date inst = new Date();
```

- Intéressant!!!
- Mais on ne peut pas choisir la date !!

Exemples

```
class Date {  
    private int j,m,a ;  
    Date(int cj,int cm,int ca) {  
        setDate(cj,cm,ca) ;  
        System.out.println("Le constructeur Date(...) est appelé");  
    }  
}
```

```
Test inst = new Test(16,9,2002) ;
```

- on utilise les fonctions déjà définies qui vérifient la cohérence
- Et si on veut plusieurs constructeurs : [La surcharge](#)

Surcharge et constructeur

- On peut surcharger des constructeurs
- On les personnalise suivant les besoins d'initialisation que l'on a

```
class Date {  
    private int j,m,a;  
    Date() {  
        setDate(1,1,2002);  
        System.out.println("Le constructeur Date()  
            est appelé");  
    }  
    Date(int cj,int cm,int ca) {  
        setDate(cj,cm,ca);  
        System.out.println("Le constructeur Date(...)  
            est appelé");  
    }  
}
```

Constructeur sans arguments

- Pour pouvoir utiliser un constructeur sans arguments, il faut que celui-ci figure dans la classe sauf si aucun constructeur n'y figure

```
class Date {  
    private int j,m,a ;  
    Date(int cj,int cm,int ca) {  
        setDate(cj,cm,ca) ;  
        System.out.println("Le constructeur Date(...) est appelé") ;  
    }  
}
```

- On ne peut pas utiliser `Date inst = new Date() ;`

Règle pour les constructeurs

- Si aucun constructeur n'existe, alors le compilateur crée un constructeur par défaut, celui ci ne contient pas d'arguments
- Si au moins un constructeur est spécifié alors le compilateur n'en crée pas de supplémentaires

- On remarquera que dans de nombreux cas, plusieurs constructeurs sont définis, et parmi eux, un constructeur sans arguments

Règle de programmation

- Les constructeurs existent !!!
- Servez vous en
- **TOUTES** les initialisations d'un objet doivent être faites dans un constructeur
- Comme dans toute règle, il y aura des exceptions.

Suppression d'objet

- En C, la libération de la mémoire est à la charge du programmeur
- En C++, il existe des destructeurs servant à libérer la mémoire allouée dynamiquement
- Oublier de nettoyer la mémoire peut induire un manque de ressources
- En Java, On ne s'occupe pas de la gestion de la mémoire
- Que se passe-t-il lorsque l'on a plus besoin d'un objet ?
 - ◆ Le ramasse miette : Garbage collector
 - ◆ GC est un charognard !!
 - ◆ Supprime les instances d'objets qui ne sont plus utiles

Exemple

```
public static void main(String[] args) {  
    Date inst = new Date();  
    {  
        Date inst2 = new Date();  
    }  
    Date inst3 = inst;  
    inst = null;  
    Date inst3 = new Date();  
}
```

- Etat du système après l'utilisation du GC ?

La documentation

Pourquoi commenter

- Un programme sans commentaire est un *programme-à-écrire-seulement*
 - 80% du temps est passé à la maintenance
 - Utilisation par d'autres personnes (bibliothèques)
 - Modification 6 mois plus tard
-
- La documentation de java

Commentaires

- Trois façons de procéder. Les deux classiques :
 - ◆ Sur une ou plusieurs lignes
`/* Ceci est un commentaire
qui continue ici */`
 - ◆ Sur une seule ligne (jusqu'à la fin)
`// Une seule ligne`
 - ◆ Les commentaires de documentation
`/** contiennent des tags
@author Gilles */`

Commentaires de documentation

- Grand nombre d'API
- Une uniformisation des documentaires est importante
- Création de documentation HTML avec le programme **javadoc**
- Les commentaires peuvent porter sur
 - ◆ Une classe
 - ◆ Un attribut
 - ◆ Une méthode. Dans la pratique, on commentera tous les champs et toutes les méthodes **publics**

Commentaires de documentation

- Les commentaires peuvent intégrer
 - ◆ des balises HTML
 - ◆ des tags

Liste des tags

@version

@author

@param

@return

@see <class-name>

@see <class-name>#<method-name>

Un exemple

- Code source de `Point.java`
- Documentation générée par `javadoc`
- Documentation du package `Math`

this

Le mot clé **this**

- **this** est employé dans deux contextes distincts
 - ◆ Pour référencer l'objet ayant reçu un message
 - ◆ Pour effectuer l'appel à un constructeur à partir d'un autre
- Les deux notions sont importantes et légèrement difficiles à comprendre

this référence

- **this** est une variable référence
- Elle désigne l'objet ayant reçu le message à traiter et donc pour lequel on doit exécuter la méthode
- Le plus souvent **this** est implicite
- Quelque fois, **this** doit être explicite

this implicite

```
class Cuisinier{
    void choisirMenu() { ... }
    void faireCourse() { ... }
    ...
    void travailler {
        choisirmenu(); /* this.choisirMenu(); */
        faireCourse(); /* this.faireCourse(); */
    }
    ...
}
```

Autre exemple

- On revient à la classe `Point`
- Ou peut on mettre des `this`
- Un peu de partout...
- Il faut l'utiliser uniquement si ca aide à la compréhension du code !

this explicite

```
class Nombre {  
    private double x;  
    Nombre(double x) {  
        this.x = x; Differencier les x */  
    ...  
    }  
}
```

Attention

- Le procédé précédent est souvent utilisé avec les constructeurs et les modificateurs (fonctions de type set..)
- Il peut être dangereux

```
class Nombre {  
    private double MonNombre;  
    Nombre(double MonNombre) {  
        this.MonNombre = MonNombre; //Problème !!  
    ...  
    }  
}
```

- Pensez à utiliser la complétion automatique

this constructeur

- Pourquoi ne pas pouvoir appeler un constructeur depuis un autre constructeur
 - ◆ La duplication de code rajoute des probabilités d'erreurs (satané copié-collé)
 - ◆ Eviter la duplication du code
- On utilise pour cela le mot clé **this**
- Restrictions
 - ◆ doit être la première instruction du constructeur
 - ◆ un seul appel possible par constructeur

this constructeur

```
class Date {
    private int j,m,a;
    Date() {
        this(1,1,2002); /* Appel constructeur avec args */
        System.out.println("Le constructeur Date()
            est appelé");
    }
    Date(int cj,int cm,int ca) {
        setDate(cj,cm,ca); /* On vérifie la cohérence */
        System.out.println("Le constructeur Date(...)
            est appelé");
    }
}
```

Retour à la case Point

- Ecrivons les constructeurs de la classe Point

Les tableaux

Les tableaux

- En java, les tableaux sont des objets particuliers
- Le type des éléments est appelé type de base du tableau
- Le nombre d'éléments gérés est un attribut fixé, appelé longueur du tableau.
- Ressemblance avec les tableaux en C/C++, mais plus facile !!
- On accède aux différents éléments avec l'opérateur d'index []

Création de tableaux

- Un tableau à une dimension : `int t1[]` ou `String[] t1`
- Un tableau à deux dimensions : `double t2[][]` ;
- et ainsi de suite...
- Les variables `t1` et `t2` représentent des variables références
- A ce stade, elles ne peuvent pas encore être utilisées.

Instanciación

- Une opération d'instanciation est nécessaire pour créer le tableau

```
t1 = new int[3];
```

```
t1[0] = 9;
```

```
t1[1] = 12;
```

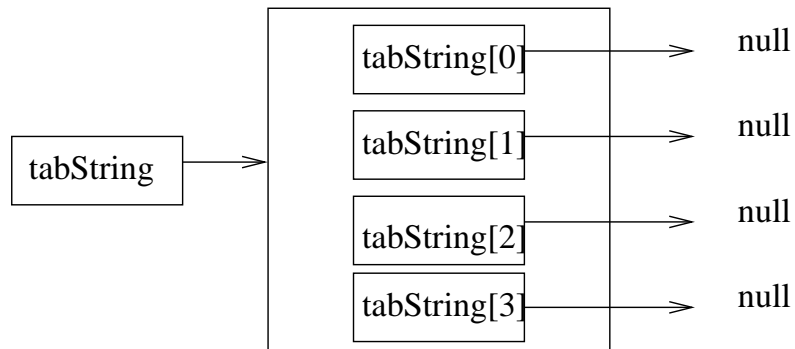
```
/* t1[2] = 0; */
```

- Comme en C/C++, le premier élément d'un tableau à l'indice 0

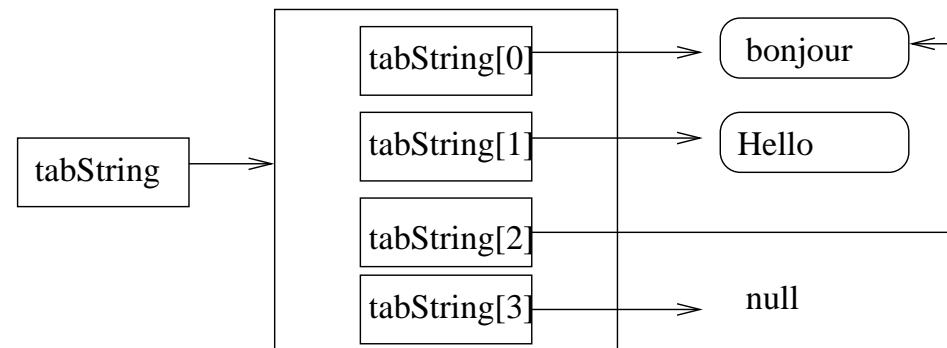
Instanciation et types objets

- Lorsque l'on crée des tableaux avec des objets non primitifs, il faut instancier les références

```
String tabString = new String[4];
```

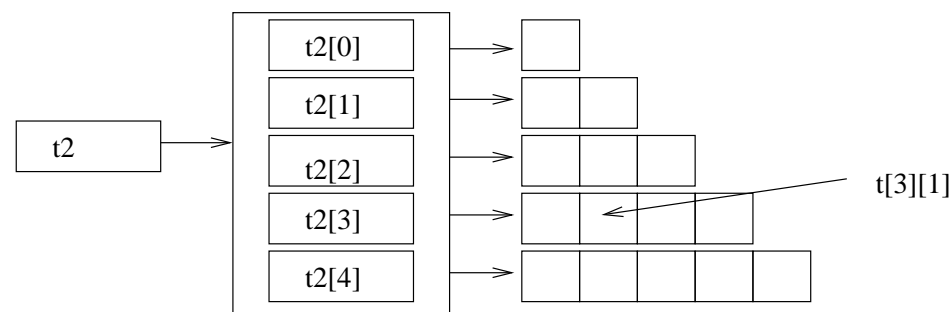


```
tabString[0] = new String("Bonjour");  
tabString[1] = new String("Hello");  
tabString[2] = tabString[0];
```



Plusieurs dimensions

```
Random r = new Random();  
int t2[][];  
t2 = new int[5][];  
for(int i = 0; i < t2.length; i++)  
    t2[i] = new int[i+1];  
for(int i = 0; i < t2.length; i++) {  
    for(int j = 0; j < t2[i].length; j++)  
        t2[i][j] = r.nextInt(100);  
}
```



Statique

Exemple

- Considérons une classe `Cercle`. On a alors besoin du nombre Π
 - ◆ Est ce que Π est spécifique à chaque instance de la classe `Cercle`
 - ◆ NON : Π est une constante générale

Exemple

```
public class Cercle {  
    double PI= 3.14116 ;  
    double rayon ;  
    public Cercle(double r) { this.rayon = r ;}  
  
    public double aire() {  
        return PI*rayon*rayon ;  
    }  
    ...  
}
```

Champs et méthodes statiques

- Normalement, rien n'existe réellement avant de créer un objet avec **new**
- il peut être intéressant d'avoir une zone de stockage pour les données spécifiques qui soit indépendant du nombre d'instances créées (de 0 à ...)

le mot-clé **static**

Exemple

```
public class Cercle {  
    double static PI= 3.14116 ;  
    double rayon ;  
    public Cercle(double r) { this.rayon = r ;}  
  
    public double aire() {  
        return PI*rayon*rayon ;  
    }  
    ...  
}
```

Remarque

- Etant donné qu'une variable statique existe même sans instance de la classe où elle est définie
- On peut utiliser cette constante à tout moment
- Comment faire ??
- On utilise le nom de la classe comme référence !!!

Cercle.PI

- C'est une convention
- De cette manière, on sait que l'on est en présence d'une variable statique

Méthodes statiques

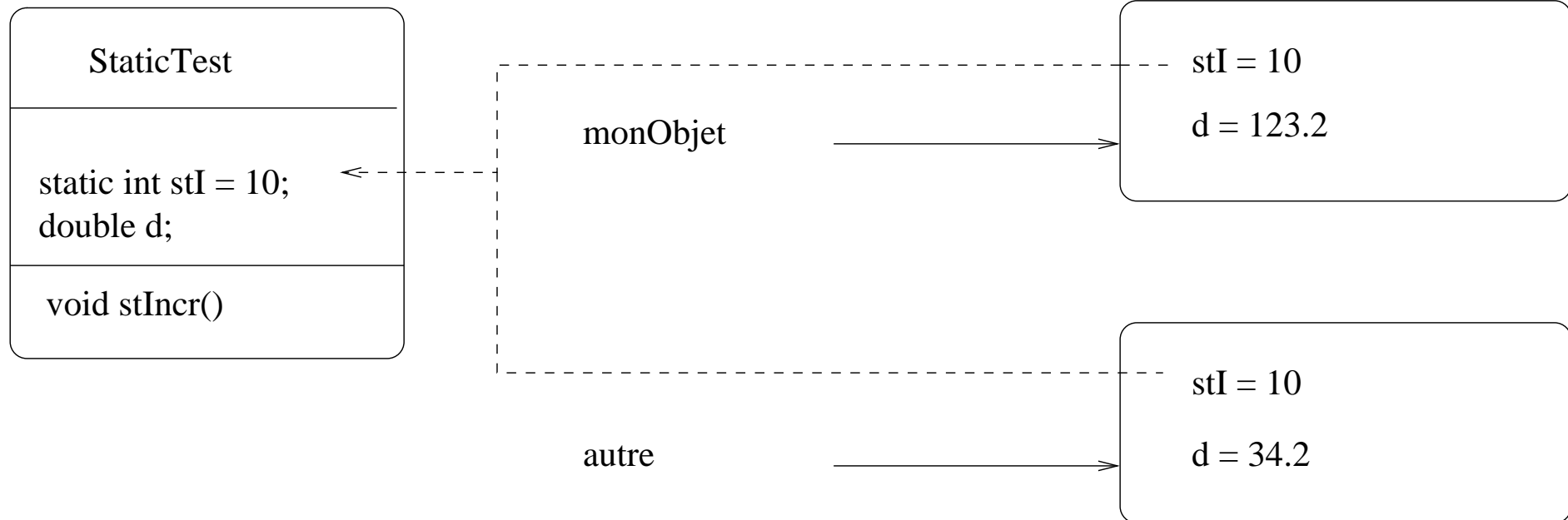
- De la même manière que l'on peut vouloir une variable statique, on peut avoir besoin d'une méthode statique
- Celle ci est indépendante du nombre d'instances de l'objet créé
- Elle sert souvent pour des méthodes générales (calcul de racine carré d'un double...)

Exemple II

```
class StaticTest {  
    static int stl= 8 ;  
    double d = 12 ;  
    static void stlIncr() {stl++ ;}  
  
    ...  
}
```

```
public static void main(String arg[]) {  
    monObjet = new StaticTest() ;  
    autre = new StaticTest() ;  
    autre.stlIncr() ;  
}
```

static : Exemple



Méthodes et static

- Une méthode de classe (static) ne peut appeler une méthode d'instance (non static)
 - ◆ Normal : elle existe même si aucune instance de la classe n'existe
- Une méthode d'instance peut appeler une méthode de classe

Pourquoi ?

- Bibliothèque de fonctions : Regrouper des fonctionnalités générales au sein d'une même entité : les classes `Math`, `Mediator`
- Voyons voir la classe `Math`
- Vis à vis de certaines classes, il n'est pas possible de créer directement des objets (avec `new`)
`Image img = toolkit.getImage("image.jpg")`
- Données globales à une classe : Compter le nombre d'instances créées, constantes...

La méthode main

```
public static void main(String[] a)
```

- publique : car sinon comment entrer dans le programme
- statique : aucune instance n'est encore créée lorsque `main` commence son execution
- void : elle ne renvoie rien
- `String[]` : les arguments passés en ligne de commande sont contenus dans un tableau de chaînes de caractères

package

Composants Fondamentaux en Java

- Les Classes
- Les Fichiers Sources
 - ◆ Plusieurs classes possibles dans un même fichier
 - ◆ MAIS, Une seule classe publique par fichier
- Packages (Unité de bibliothèque)
 - ◆ Plusieurs fichiers sources
- API (Application Program Interface)
 - ◆ Plusieurs packages

Les Packages

- Un package est un regroupement logique de plusieurs classes
- Beaucoup de packages en Java
- Un package correspond à la notion de bibliothèques de langages comme le C
- Un package peut contenir des sous packages

Quelques Packages de Java 2

- `java.lang` : classes de base de java
- `java.util` : classes utilitaires
- `java.math` : fonctions mathématiques
- `java.io` : Entrées-Sorties
- `java.net` : Applications réseau

Utilisation d'un package

- Pour utiliser un package, il faut préfixer le nom de la classe avec le nom du package

Exemple : `java.util.Date`

```
java.util.Date d = new java.util.Date()
```

Utilisation d'un package ... 2

- Cela peut s'avérer très lourd à écrire
- Les informaticiens sont des feinénants

Importation

- Il est possible d'inclure une directive d'importation en **début** de fichier (avant toute définition de classe)
- On importe une classe d'un package

```
import java.util.Date
Date d = new Date()
```

Utilisation d'un package ... 3

- Il est possible d'importer toutes les classes d'un package
`import java.util.*`
- la package `java.lang` est automatiquement importé
- L'utilisation sans importation est surtout intéressante pour lever des ambiguïtés entre deux classes de même nom dans des packages différents.

Nom d'un package

- Le nom d'un package est entièrement déterminé par le chemin indiquant l'emplacement des classes de celui-ci
- La première partie du chemin peut-être omise si elle appartient à la variable d'environnement `CLASSPATH`
- Cette variable est définie dans le fichier d'initialisation (`.bashrc`, `.cshrc`, `autoexec.bat`)
 - ◆ en bash : `export CLASSPATH=chemin1 :chemin2`
 - ◆ sous windows `CLASSPATH=chemin1 ;chemin2...`

Appartenance à un package

- On peut créer des packages personnels
- On définit l'appartenance d'une classe à un package par le mot clé :
package
 package nomPackage
- Une telle directive précède les directives d'importation
- Un package par défaut regroupe toutes les unités de compilation qui apparaissent dans le même répertoire et qui n'explicitent pas d'appartenance à un package

Création d'un package - Exemple

- Je dispose d'une bibliothèque générale sauvée dans le répertoire

```
/home/audemard/java/mabilbio
```

- La variable d'environnement est défini (dans `.bashrc`)

```
CLASSPATH=. :/usr/local/java :/home/audemard/java/
```

- La package que je définis va alors s'appeler `mabiblio`

- Dans chaque unités de compilation de `/home/audemard/java/mabiblio`, je dois définir l'appartenance au package :

```
package mabiblio
```

sous bibliothèque

- Que faire si je rajoute une sous bibliothèque
- Je crée le répertoire `/home/audemard/java/mabiblio/base`
- Les classes qui sont dans le répertoire `base` appartiennent alors au package `mabiblio.base`
- Voyons voir un exemple
- Eclipse simplifie tout cela....

Visibilité

- Plusieurs niveaux de visibilité (du plus restrictif au moins restrictif)
 - ◆ private (déjà vu)
 - ◆ package (pas de mot clé)
 - ◆ protected
 - ◆ public (déjà vu)

| | Classe | Package | Sous Classe | monde |
|-----------|--------|---------|-------------|-------|
| private | x | | | |
| package | x | x | | |
| protected | x | x | x | |
| public | x | x | x | x |

Devoir surveillé

- La semaine 40 (dans 15 jours) durant le cours.

Réutilisation des classes

Redondance de code

- Traitement similaire à des endroits différents du code
- copier/coller avec adaptation
- Fléau pour le développement et la maintenance d'applications
 - ◆ Perte de temps durant l'écriture du code
 - ◆ Perte de temps durant le débogage
 - ◆ Perte de temps durant la maintenance
- Handicap de la programmation procédurale

Réutilisation de code

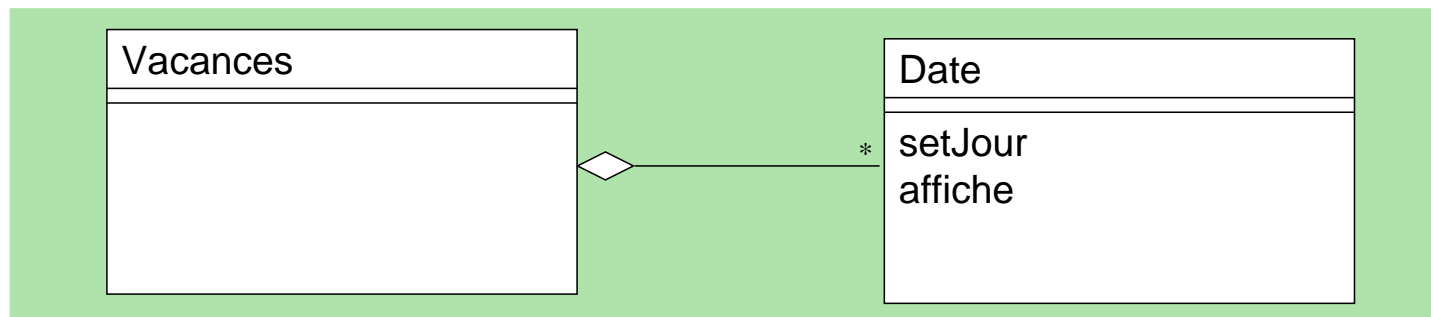
- La réutilisation de code est l'une des idées directrices de la programmation
- Elle est rendue possible grâce :
 - ◆ au passage de paramètres
 - ◆ à la composition
 - ◆ à l'héritage
- le paradigme objet met en oeuvre le mécanisme d'héritage
- Renforce la lutte contre la redondance de code

Passage de paramètres

- Possibilité de définir des actions paramétrées
- Fondamental en programmation
- Tous les langages (évolués) offrent ce mécanisme

Composition

- La composition consiste à utiliser des variables références comme champs d'une nouvelle classe
- La nouvelle classe est cliente des classes des variables références
- La composition représente un lien "*possède un*"
- Le diagramme UML associé



Exemple

```
public class Vacances {  
    private Date départ ;  
    private Date arrivée ;  
    private String lieu ;  
    public Vacances(Date arrivée) {  
        this.arrivee=arrivee ;  
        this.depart = new Date(22,09,2003) ;  
    } ...  
}
```

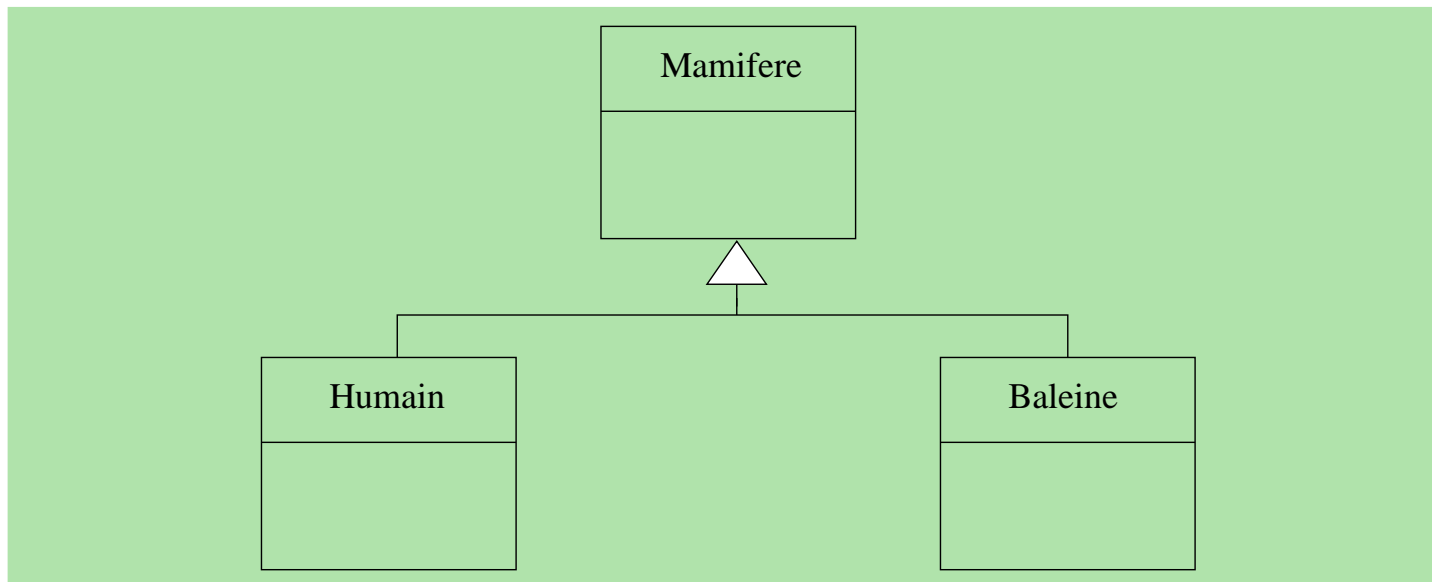
- départ et arrivée sont initialisés à null.
- **NE PAS OUBLIER** d'initialiser les instances lors de l'appel d'un constructeur
Vacances

Introduction à l'héritage

- Le langage humain permet de spécialiser des concepts
 - ◆ Les mammifères
 - ◆ L'homme est un mammifère
 - ◆ La baleine est un mammifère
- Disant cela, je rends implicite certains critères concernant les hommes et les baleines
- Une conversation entre personnes nécessite ces non-dits

Introduction à l'héritage II

- L'héritage représente un lien "*est un*" ou "*est comme un*"
- Permet la création d'une nouvelle classe en spécialisant une classe existante.
- Le diagramme UML associé

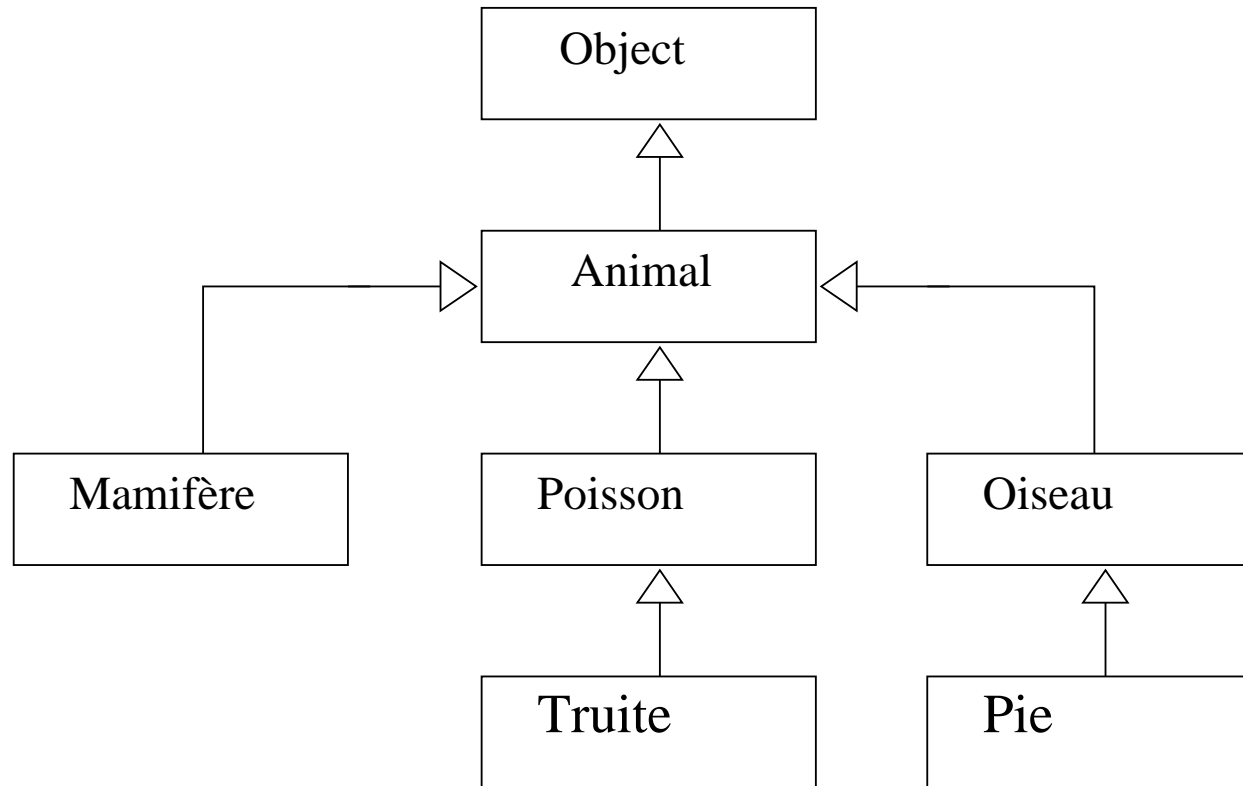


Intérêt de l'héritage

- Réutilisation du code
 - ◆ Sans toucher aux classes existantes (Pas de détérioration)
 - ◆ En indiquant simplement les nouvelles caractéristiques propres à la sous classe
 - ◆ En redéfinissant éventuellement certaines méthodes de la super classe
- Développement incrémental

Très intéressant

Exemple



Exemple... Suite

```
public class Animal {  
    ...}
```

```
public class Mammifère extends Animal {  
    ...}
```

```
public class Poisson extends Animal {  
    ...}
```

```
public class Oiseau extends Animal {  
    ...}
```

```
public class Truite extends Poisson {  
    ...}
```

D'autres définitions

- La nouvelle classe est dite sous classe, classe dérivée ou classe enfant. On dit aussi qu'elle hérite ou étend ou qu'elle est une spécialisation de la classe de départ
- La classe de départ est appelée super-classe, classe de base ou classe parent. On dit aussi qu'elle est une généralisation de celle-ci
- Hiérarchie de classe ou hiérarchie d'héritage
- Si il existe un chemin dans la hiérarchie qui mène de la classe D à la classe A, alors
 - ◆ D est dite classe descendante de A
 - Truite est descendant de animal
 - ◆ A est dite classe ancêtre de D
 - Object est ancêtre de Pie (Object est ancêtre de tout le monde !)

Possibilité de l'héritage

- Les fonctions et champs existant dans la classe de base sont accessibles dans la classe dérivée
- Possibilité d'étendre la définition de la classe de base en ajoutant :
 - ◆ des champs
 - ◆ des méthodes
 - ◆ des constructeurs
- Possibilité de **redéfinir** des méthodes d'une super classe

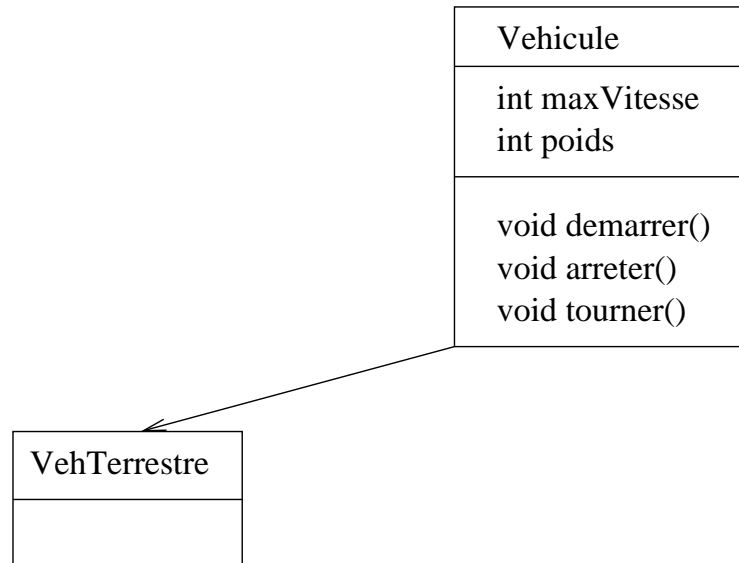
Redéfinition

- Lorsque l'on définit une nouvelle classe par héritage, il est possible de **redéfinir** des méthodes
- Il s'agit de placer dans cette classe une méthode de même en-tête qu'une méthode existante dans la super classe.
- La nouvelle méthode cache alors la méthode redéfinie
- La redéfinition, comme son nom l'indique, permet de redéfinir le comportement d'une méthode par rapport à la spécialisation induite par l'héritage

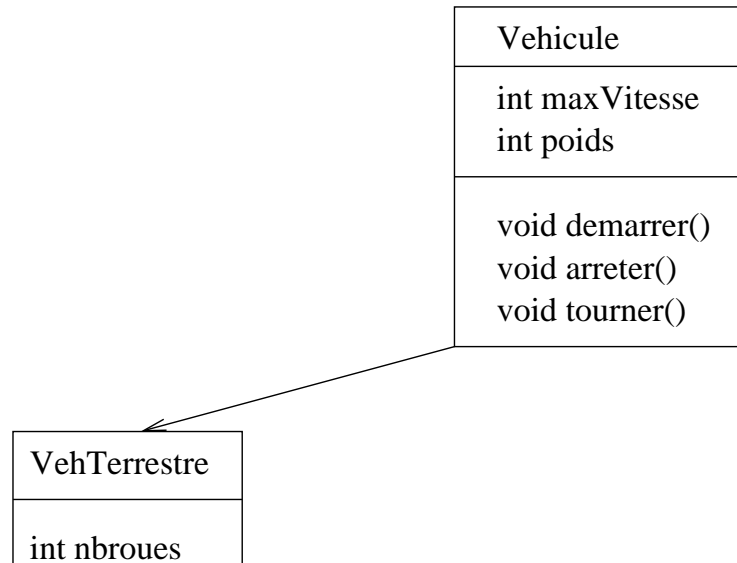
Exemple

| |
|---|
| Vehicle |
| int maxVitesse int poids |
| void demarrer() void arreter() void tourner() |

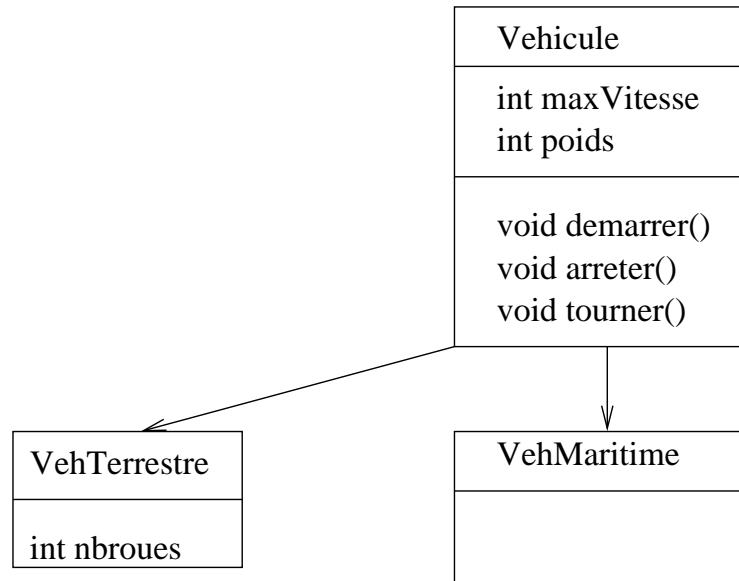
Exemple



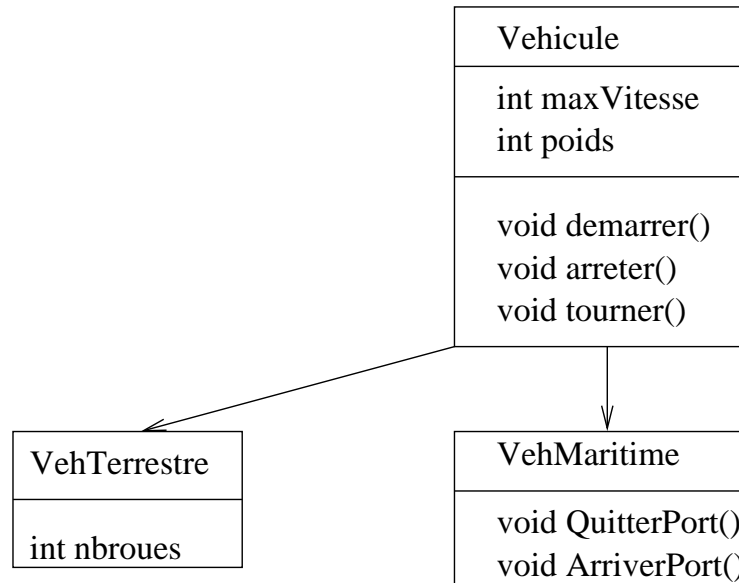
Exemple



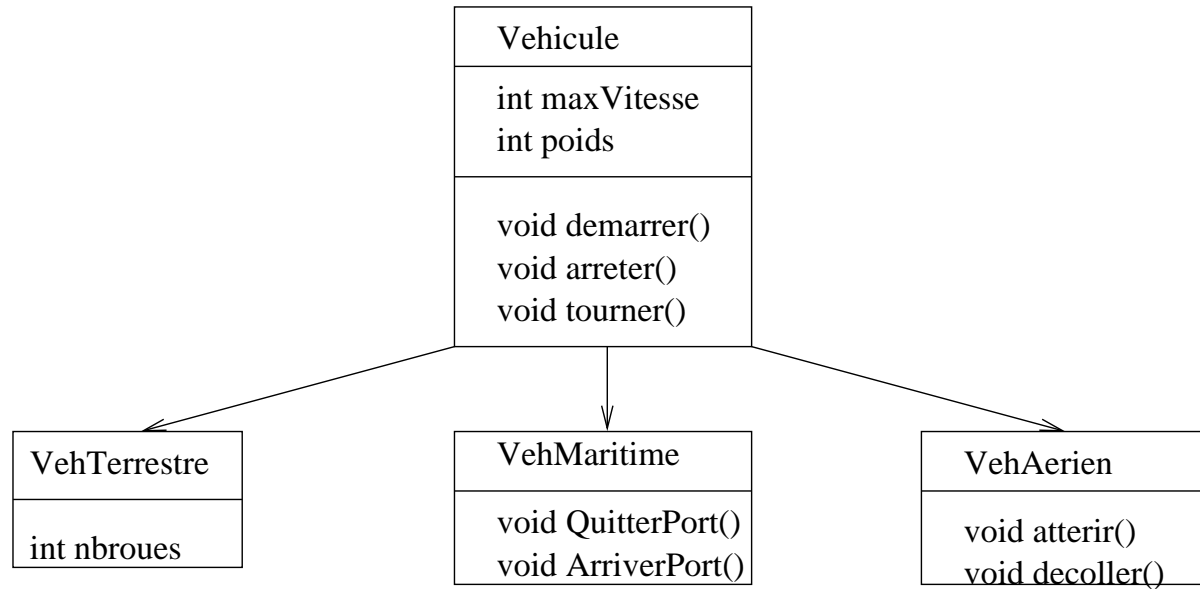
Exemple



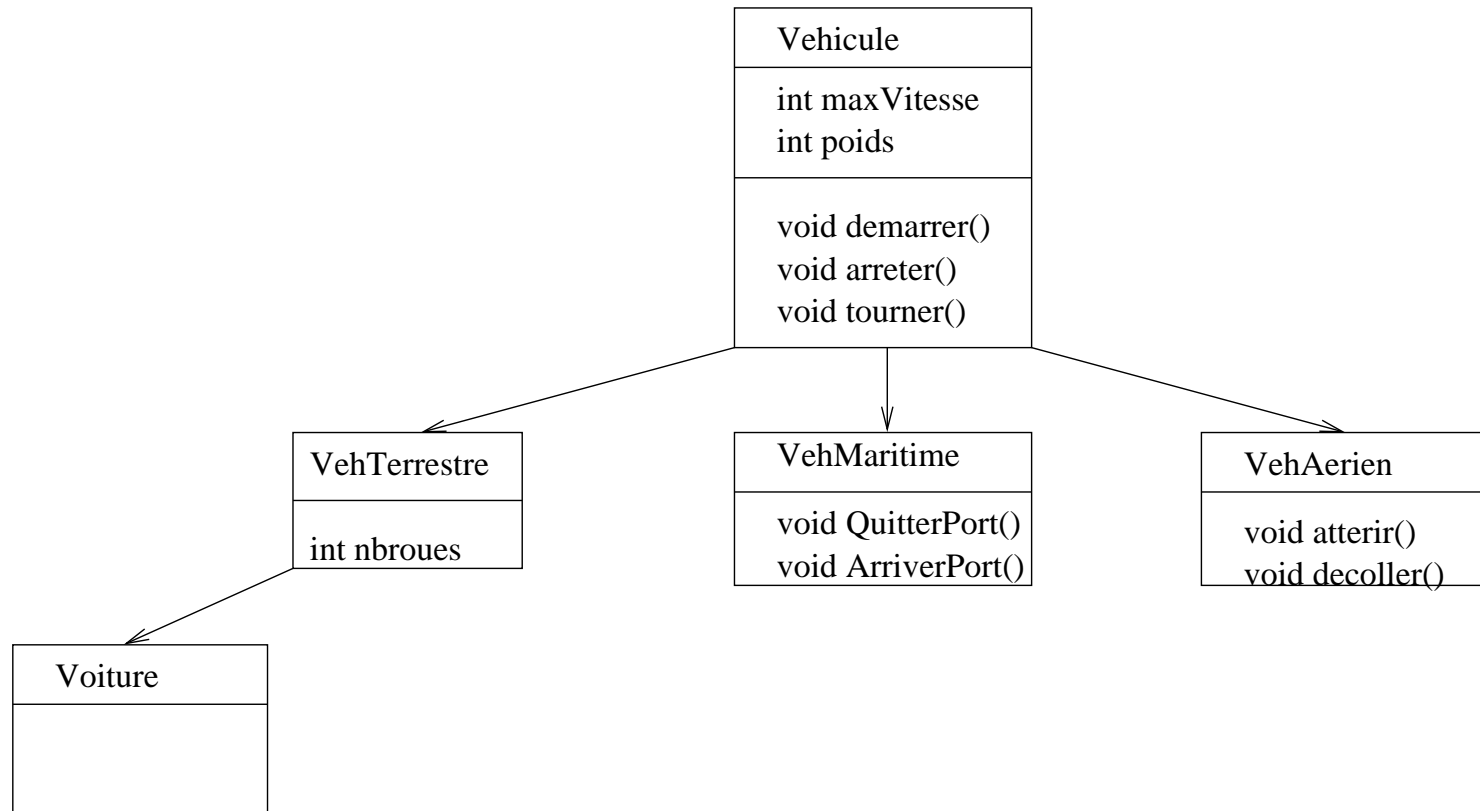
Exemple



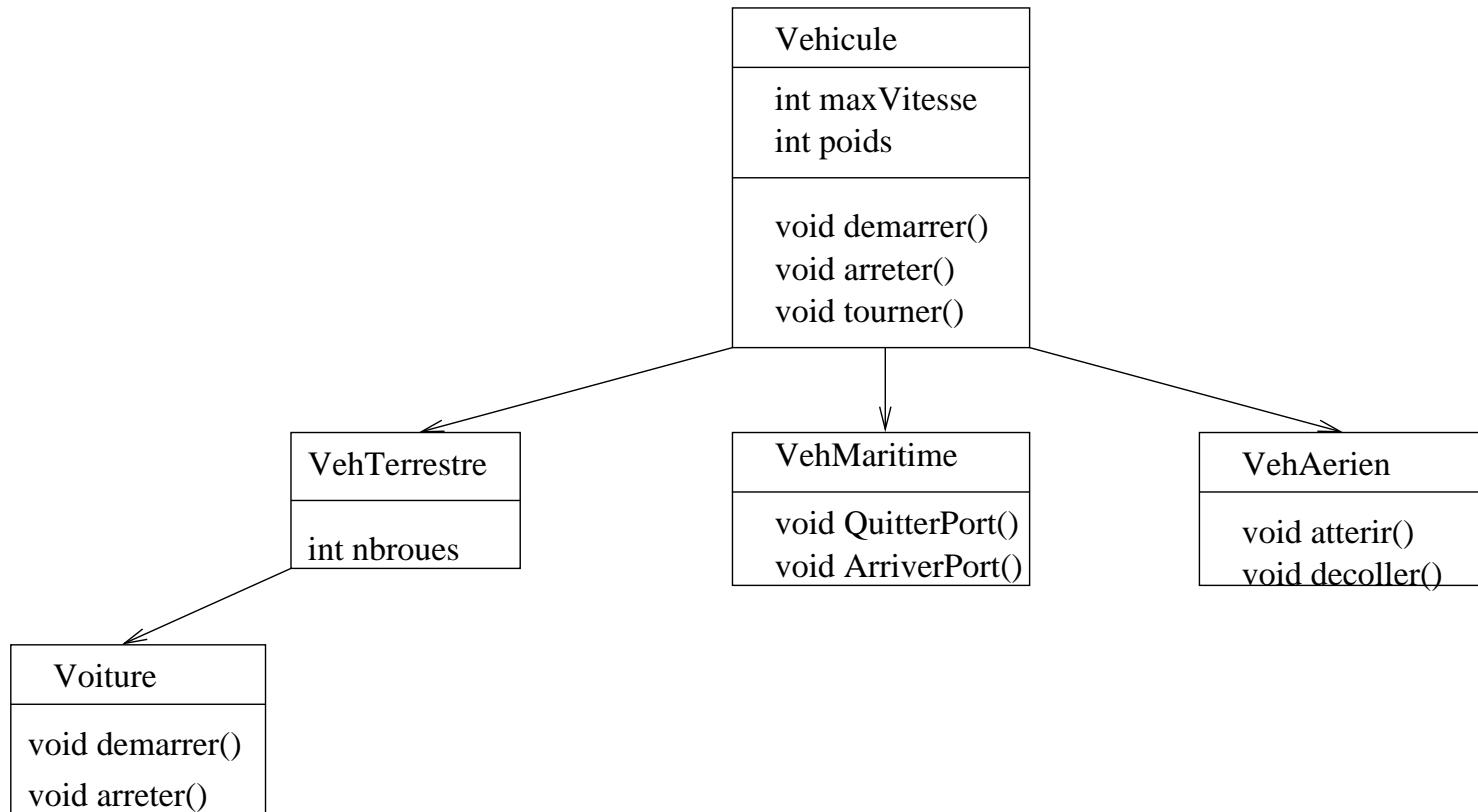
Exemple



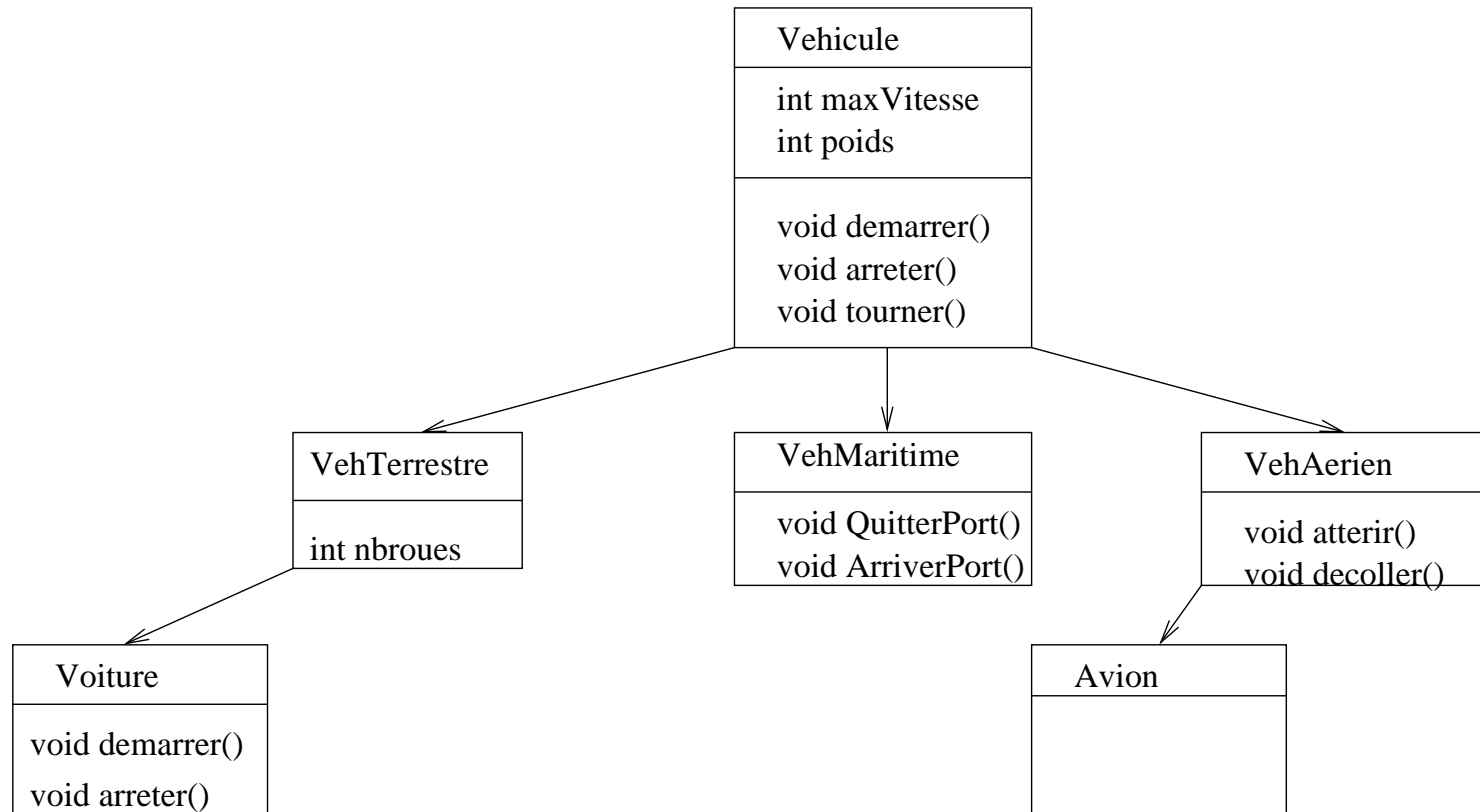
Exemple



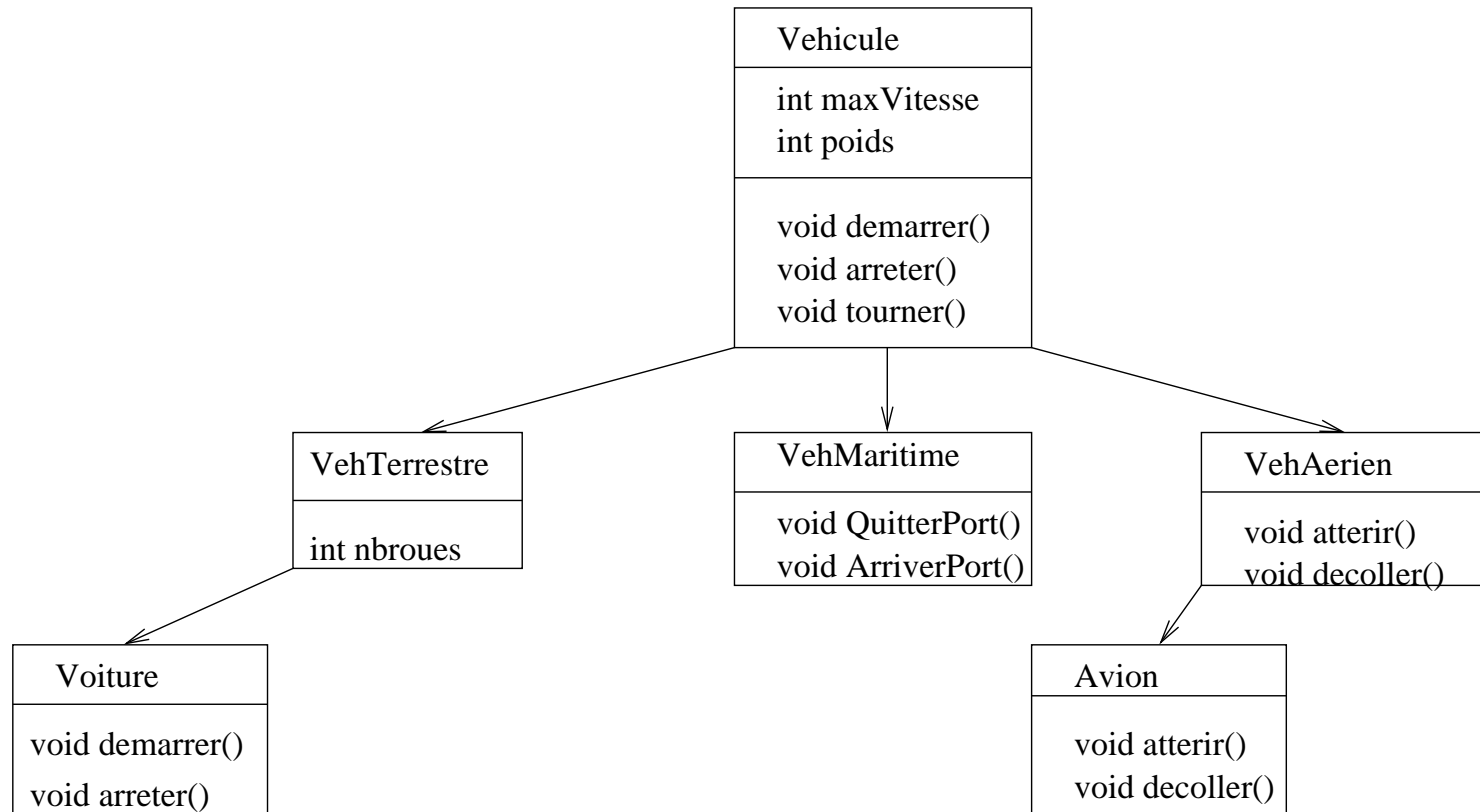
Exemple



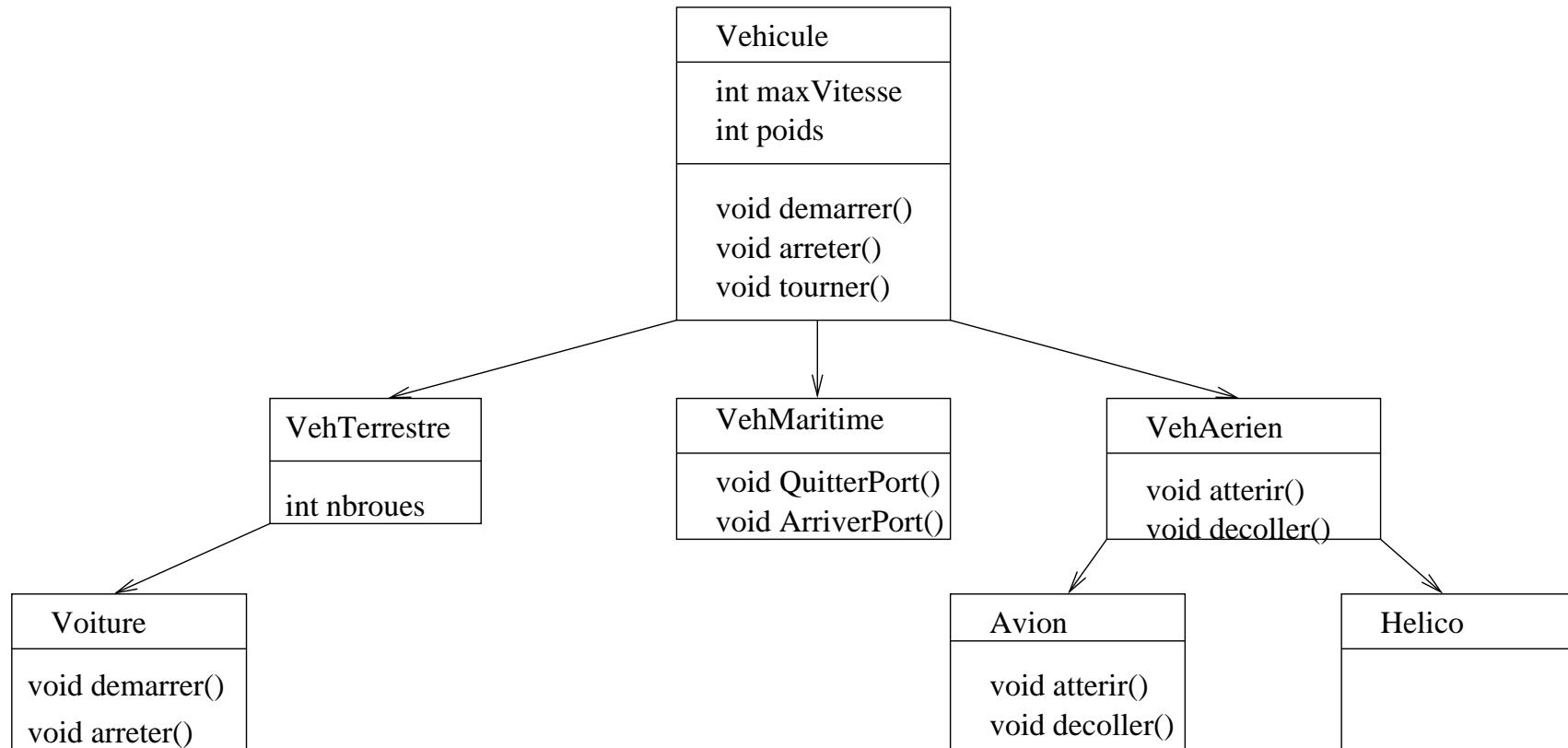
Exemple



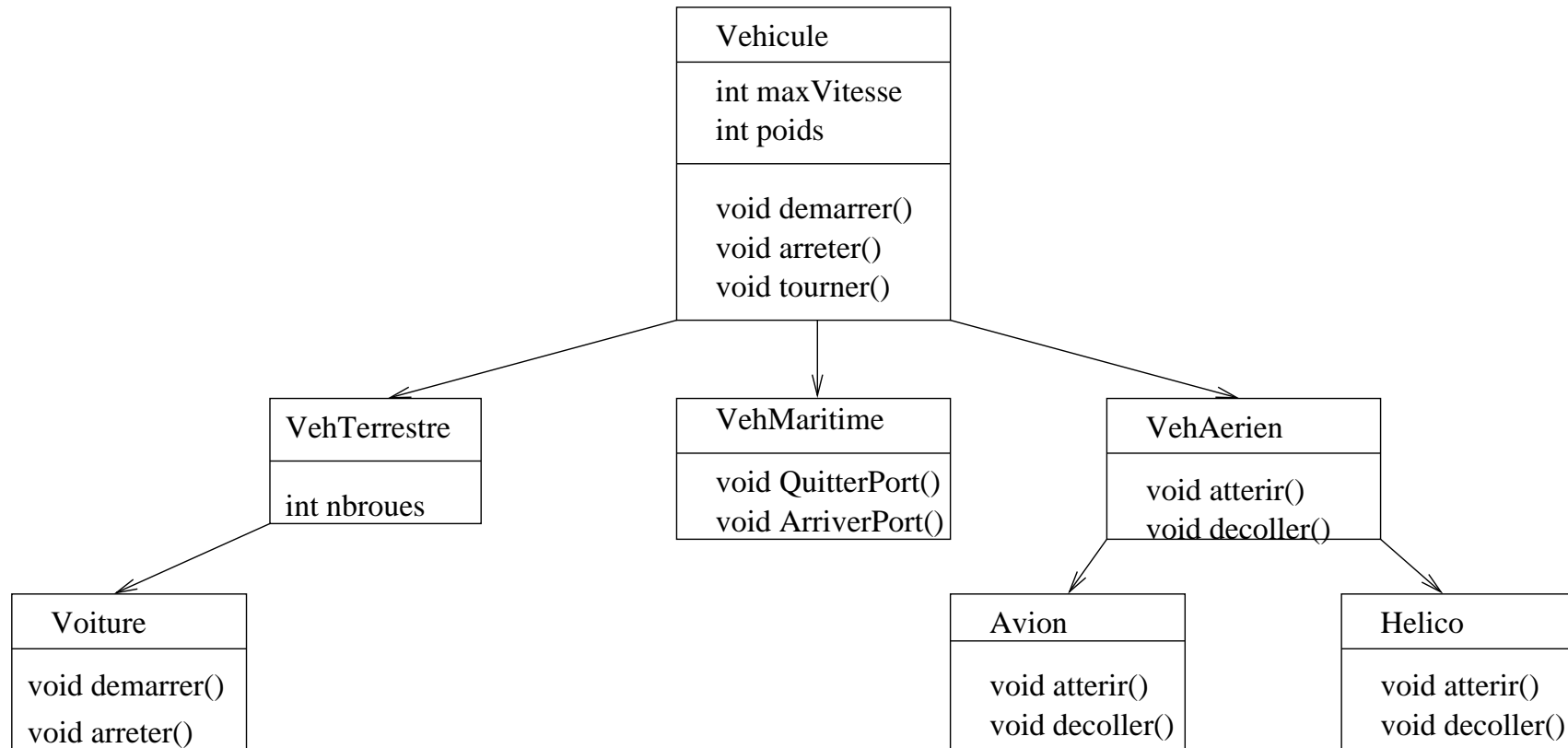
Exemple



Exemple



Exemple



Exemple

```
class TestA {  
    public void f() {System.out.println("f de testA");}  
    public void g() {System.out.println("g de testA");}  
}  
class TestB extends TestA{  
    public void f() {System.out.println("f de testB");}  
    public void h() {System.out.println("h de testB");}  
}
```

```
TestA ta = new TestA();  
TestB tb = new TestB();  
ta.f(); /* OK */  
ta.g(); /* OK */  
ta.h(); /* ERREUR */  
tb.f(); /* OK */  
tb.g(); /* OK */  
tb.h(); /* OK */
```

f de testA
g de testA

f de testB
g de testA
h de testB

Contraintes

- La redéfinition implique certaines contraintes
 - ◆ Sur la signature
 - ◆ Sur l'accessibilité : La méthode redéfinie ne doit jamais être moins accessible que la méthode présente dans la super classe
 - ◆ Sur les exceptions (PLUS TARD)
- Si celles ci ne sont pas respectées, une erreur de compilation est signalée

Surcharge vs Redéfinition

- **surcharge** : Il s'agit d'une méthode qui possède le même nom mais pas la même signature qu'une méthode de classe ou de super classe
 - ◆ Donne plusieurs possibilités d'appels pour une fonction donnée
- **Redéfinition** Il s'agit d'une méthode qui possède la même signature qu'une méthode de la classe mère
 - ◆ modifie le comportement d'une fonction par rapport à la spécialisation

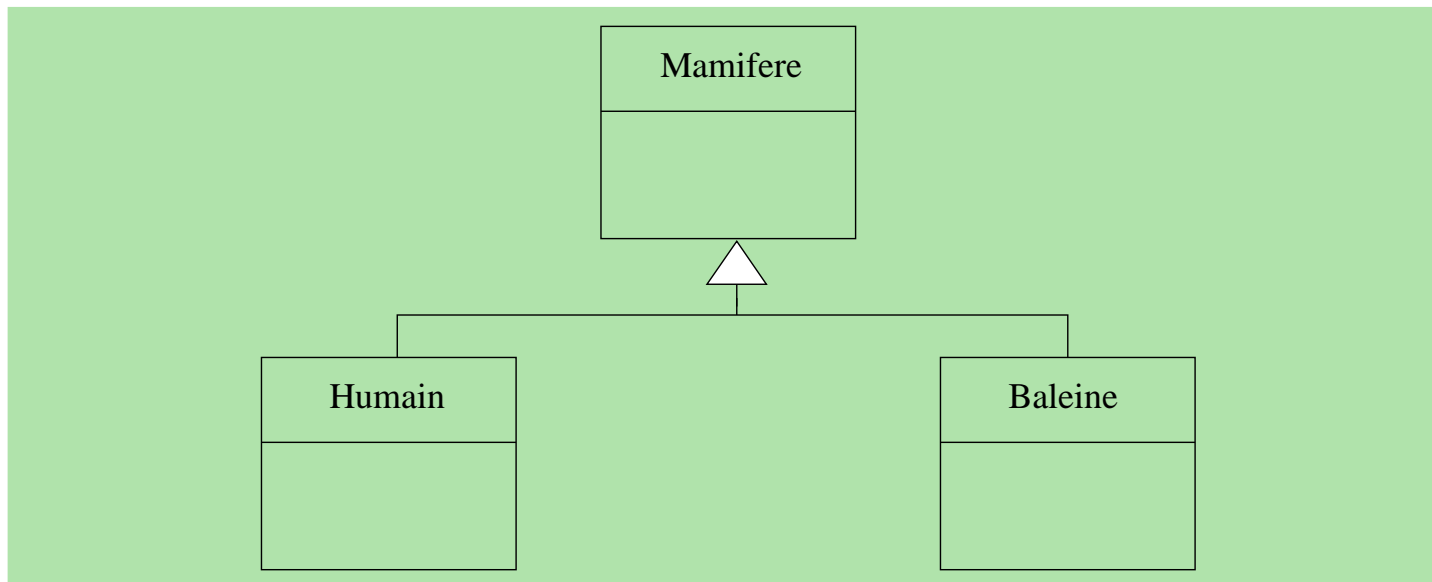
Exemple

```
class TestA {
    public void f() {System.out.println("f de testA");}
    public void g() {System.out.println("g de testA");}
}
class TestB extends TestA{
    public void f(int a) {System.out.println("f de testB" + a);}
    public void h() {System.out.println("h de testB");}
}
```

```
TestA ta = new TestA();
TestB tb = new TestB();
ta.f(); /* OK */           f de testA
ta.g(); /* OK */           g de testA
tb.f(); /* OK */           f de testA
tb.f(1) /* OK */          f de testB 1
tb.g(); /* OK */           g de testA
tb.h(); /* OK */           h de testB
```

Introduction à l'héritage II

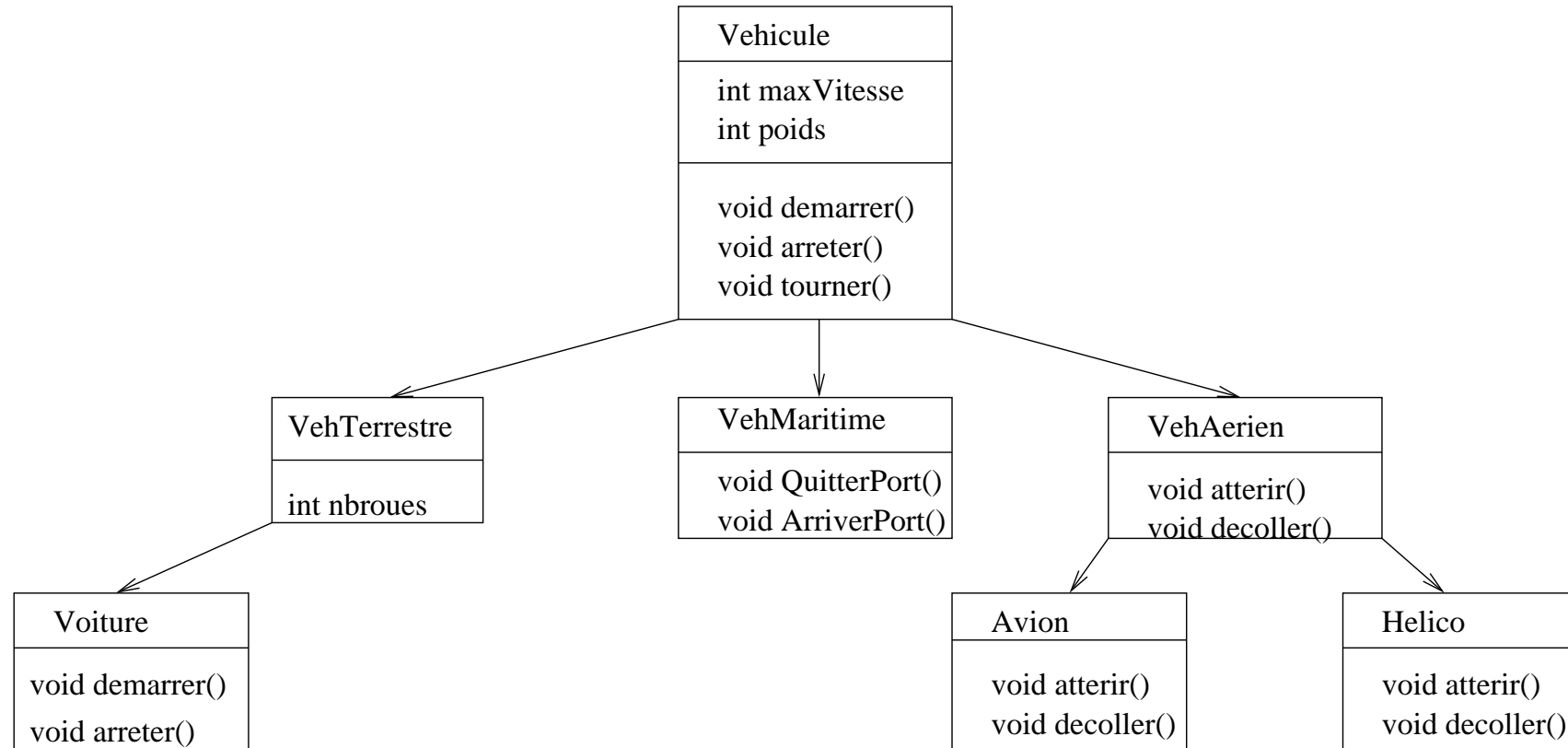
- L'héritage représente un lien "*est un*" ou "*est comme un*"
- Permet la création d'une nouvelle classe en spécialisant une classe existante.
- Le diagramme UML associé



Possibilité de l'héritage

- Les fonctions et champs existant dans la classe de base sont accessibles dans la classe dérivée
- Possibilité d'étendre la définition de la classe de base en ajoutant :
 - ◆ des champs
 - ◆ des méthodes
 - ◆ des constructeurs
- Possibilité de **redéfinir** des méthodes d'une super classe

Exemple

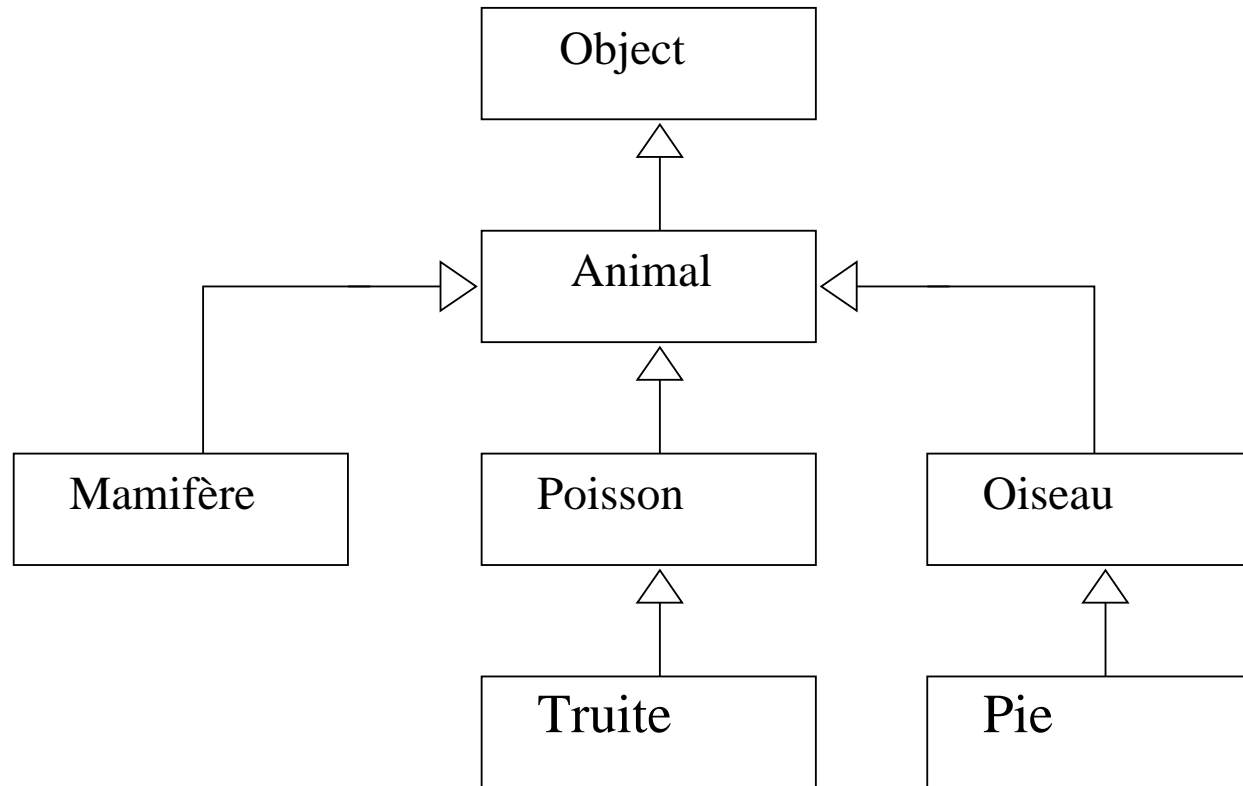


La classe Object

Introduction

- La classe `Object` est une classe particulière
- C'est la classe Racine
- toute classe est soit
 - ◆ Directement descendante de `Object` si aucun héritage n'est précisé
 - ◆ Possède `Object` comme ancêtre sinon (ce cas englobe l'autre)

Arborescence et classe objet



Arborescence et classe objet

```
public class Animal {  
    ...}
```

```
public class Mammifère extends Animal {  
    ...}
```

```
public class Poisson extends Animal {  
    ...}
```

```
public class Oiseau extends Animal {  
    ...}
```

```
public class Truite extends Poisson {  
    ...}
```

Particularité

- La classe `Object` ne possède aucun champ (variable)
- Par contre, elle possède des méthodes intéressantes qu'il est possible de redéfinir
- Allons voir la javadoc de cette classe
- Et résumons les maintenant

La méthode toString

- La méthode `toString()` est définie dans la classe `Object`. Elle renvoie une description de l'objet sous forme de chaîne de caractères.
- Cette méthode est utilisée par exemple lors de l'appel de la fonction `System.out.print`
- Initialement, la fonction `toString()` renvoie l'adresse en mémoire de l'instance créée. Assez peu intéressant.
- Essayons avec la classe `Point`
- C'est pour cela que la méthode `toString` est presque toujours redéfinie.
- Essayons de nouveau avec la classe `Point`

Un autre Exemple

```
class Date {  
    private int j,m,a;  
    public String toString() { return j + "." + m + "." + a; }  
}  
...  
d = new Date(12,5,2001);  
System.out.println("la date est " + d);
```

Comparaison d'objets

- On souhaite comparer 2 `Points`. Comment Faire

```
Point A = new Point(3,2);  
Point B = new Point(3,2);  
...  
if(A==B) {  
}
```

- Utiliser l'opérateur `==` ?? **NON, NON et encore NON**
- Il faut utiliser une fonction de comparaison

La méthode equals

- Elle renvoie **true** si l'objet recevant l'appel est identique à celui passé en paramètre de la méthode
- La version initiale renvoie **true** si les deux objets ont la même adresse. Elle doit donc être redéfinie
- Attention, la méthode doit respecter certaines contraintes :
 - ◆ Réflexivité : `x.equals(x)` est **true**
 - ◆ Transitivité : si `x.equals(y)` et `y.equals(z)` sont **true** alors `x.equals(z)` doit m'être aussi
 - ◆ Symétrique : si `x.equals(y)` est **true** alors `y.equals(x)` doit l'être aussi
 - ◆ Pour toute référence non nulle, `x.equals(null)` doit être **false**

La méthode equals 2 ... Le retour

- Essayons avec la classe String
- Et maintenant avec la classe Point

Autres méthodes

- Il existe une dizaine de fonctions dans la classe `Object`
 - ◆ Clonage : La méthode `clone()`
 - ◆ Tables de hachage : `hashCode()`
 - ◆ Threads : `wait()`, `notify()`, `notifyAll()`
 - ◆ destruction de l'objet : `finalize()`
- On reviendra sur certaines d'entre eux

Héritage multiple

Introduction

- Supposons une classe `Rectangle` et une classe `Losange`
- Nous devons créer une classe `Carre`
- De qui héritons nous ??
- Certains langages comme le C++, permettent d'hériter de plusieurs classes
- C'est ce que l'on appelle l'héritage multiple

Héritage multiple

- En Java, on ne dispose que de l'héritage simple. Il n'y a pas d'héritage multiple
- Chaque classe hérite d'une seule classe :
 - ◆ Soit de la classe `Object`
 - ◆ Soit d'une classe dont le nom est précisé après le mot-clé `extends`

Héritage et constructeurs

Introduction

- Lors de l'héritage, il y a au moins deux classes concernées
 - ◆ La classe mère
 - ◆ La classe enfant
- En pratique, la classe enfant contient un sous objet de la classe mère
- Il est essentiel que cette sous classe soit correctement initialisée
- Un petit exemple...

Exemple

```
class Vehicule {  
    Vehicule() {System.out.println("Constructeur de Vehicule"); }  
}  
class VehTerrestre extends Vehicule{  
    VehTerrestre() {System.out.println("Constructeur de Terrestre"); }  
}  
class Voiture extends VehTerrestre{  
    Voiture() {System.out.println("Constructeur de Voiture"); }  
}
```

- Que se passe t il lors de l'instruction
 - ◆ Vehicule v = new Vehicule() ; ??
 - ◆ Voiture Deuch = new Voiture() ; ??

Constructeur

- La classe de base doit **et va** être correctement initialisée
- Que se passe t il si on supprime le constructeur de `Voiture`
 - ◆ Pas de constructeur : Un constructeur par défaut est créé
- Que faire si la classe mère possède plusieurs constructeurs
- Il faut lui spécifier celui que l'on veut utiliser
Le mot clé **super**

Le mot clé super

- `super` peut être vu comme le complémentaire de `this`
- Il va servir à appeler un constructeur de la classe mère
- Il va également servir à appeler des fonctions de la classe mère
- Modifions le code des classes
 - ◆ Vehicule et compagnie
 - ◆ Quart Supérieur droit

Un petit exemple pour répéter

```
class VehTerrestre extends Vehicule {  
    Vehicule(int n) { System.out.println("Veh 1 arguments"); }  
}
```

```
class Moto extends VehTerrestre {  
    Moto() {  
        super(2);  
    }  
}
```

Un peu de théorie : Les règles

- Si la première instruction d'un constructeur n'est ni `this(. .)` ni `super(. . .)` alors java insère, comme première instruction, un appel implicite à `super()`
- Si la première instruction d'un constructeur est `super(. . .)`, alors java appelle le constructeur choisi et exécute, au retour, les instructions du constructeur choisi.
- Si la première instruction est `this`, alors java invoque le constructeur sélectionné et l'appel au constructeur de la classe mère est fait dans le constructeur surchargé (`this`).
- et ainsi de suite : La toute première instruction exécutée est donc le constructeur de la classe `Object`

Exemple

```
class VehTerrestre {
    VehTerrestre() { System.out.println("Ter 0 arguments"); }
    VehTerrestre(int n) { System.out.println("Ter 1 arguments"); }
}

class Moto extends VehTerrestre {
    Moto() {
        super(2);
        System.out.println("Moto 0 arguments");
    }
    Moto(String nom) {
        System.out.println("Moto 1 arguments");
    }
    Moto(String nom,int n) {
        this(nom);
        System.out.println("Moto 2 arguments");
    }
}
```

Autre utilisation de super

- Théoriquement la redéfinition cache la fonction de la classe mère
- Et la redondance alors !!
- `super` peut être utilisé pour appeler une fonction de la classe mère

```
class Animal {  
    void manger(Nourriture f) { ... }  
}  
class Herbivore extends Animal {  
    void manger(Nourriture f) {  
        if(...) /*Verification si f est vert !*/  
            super.mange(f)  
    }  
}
```

- Sans cela, redondance de code !!!

Limites

- On ne peut pas remonter plus haut que la classe mère pour appeler une fonction redéfinie
 - ◆ pas de `(classeAncêtre.m())`
 - ◆ pas de `super.super.m()`

Héritage et Tanstypage

Le transtypage implicite

- Lors d'une affectation, une opération de upcasting implicite peut avoir lieu.

Exemple

```
double x = 15 ;
```

- La valeur 15 est transformée automatiquement en valeur double 15.0 avant d'être affectée à x

Le transtypage explicite

- Lors de certaines opérations, il est parfois nécessaire d'utiliser explicitement une opération de transtypage

Exemple

```
int x = 3, y = 2 ;  
double z = x/y ; /* z = 1.0 */
```

- Pour que z est la valeur 1.5, il faut faire
double z = (double)x/y ;

Et pour les classes

- Il est possible d'effectuer des opérations de transtypage de classes
- Une variable référence déclarée d'une certaine classe peut parfois contenir un lien vers un objet d'une classe différente
- la syntaxe est la même que pour les types primitifs
- on peut faire du transtypage sur les classes uniquement si l'une est ancêtre de l'autre et vice versa.

Upcasting

- Cette opération consiste à considérer une référence comme étant plus générale que ce qu'elle est
- Elle peut être fait de façon implicite
- Les champs et les méthodes d'une classe sont en effet valides pour la classe dérivée.

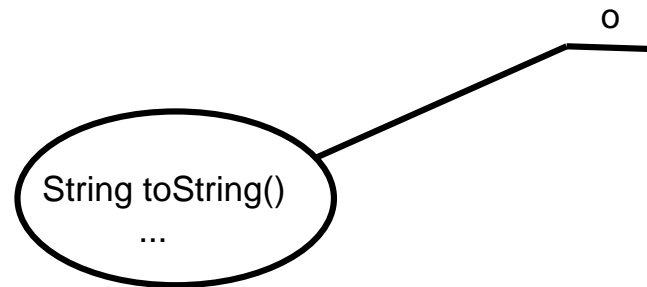
- `Vehicule v = new Moto("Fazer 600")` est un upcasting implicite
- `Vehicule v = (Vehicule) new Moto("Fazer 600")` est un upcasting explicite équivalent

DownCasting

- Cette opération consiste à considérer une référence comme étant moins générale que ce qu'elle semble être
- Nécessairement explicite
- Vehicule d = new Avion("Airbus A300");
- d.decoller(); /* Erreur */
- ((Avion)d).decoller(); /* OK */

Visibilité

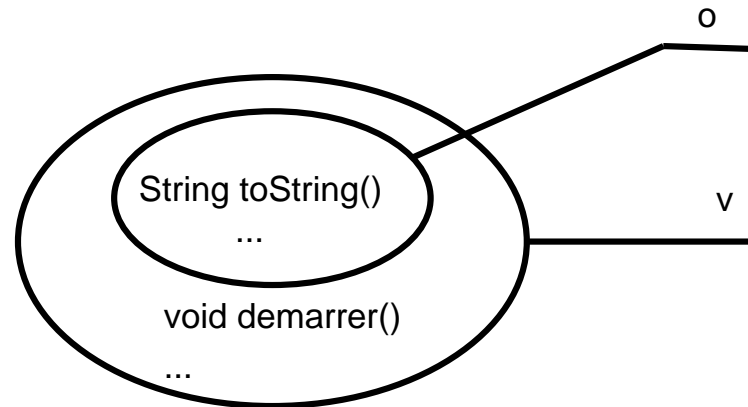
Object o = new Avion();



Visibilité

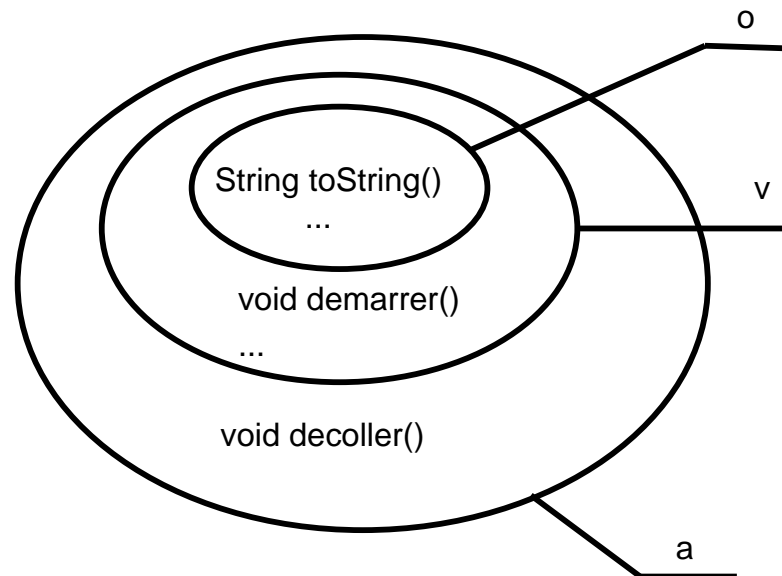
Object o = new Avion();

Vehicule v = (Vehicule) o;



Visibilité

```
Object o = new Avion();  
Vehicule v = (Vehicule) o;  
Avion a = (Avion)o;
```



Intérêt

- Sans upcasting, pas besoin de downcasting
- Mais, le upcasting est fondamental pour utiliser le polymorphisme

- Il est toujours possible de tester la classe réelle d'une référence avec le mot-clé `instanceof`
- `(reference instanceof Classe)` retourne `true` si la référence est une instance de la classe

Casting et méthode equals

- Revenons à la fonction `equals`
- Maintenant tout (j'espère) s'éclaircie..

```
public class Date {  
    int j,m,a ;  
    public boolean equals(Object o) {  
        if(! o instanceof Date) return false ;  
        return (((Date)o).j==j) && (((Date)o).m==m)  
            && (((Date)o).a==a) ;  
    }  
}
```

```
Date d1 = new Date(12,12,2002) ;  
Date d2 = new Date(3,3,2002) ;  
String s = "bonjour " ;  
d1.equals(d2) ;  
d1.equals(d1) ;  
d1.equals(s) ;
```

Les classes abstraites

Introduction

- Plus on remonte dans la hiérarchie des classes, plus celles ci deviennent générales et abstraites
- A un moment donné, la classe ancêtre devient tellement générale qu'on la considère surtout comme un moule pour des classes dérivées et non plus comme une véritable classe dotée d'instances
- Pourquoi ne pas spécifier ce statut de moule (d'abstraction)

Les classes et les méthodes abstraites

Méthodes abstraites

- Une méthode abstraite spécifie uniquement le prototype d'une fonction.
- le corps de la méthode n'est pas fourni
- Modèle pour créer des classes d'applications spécifiques

Le mot-clé **abstract**

- `abstract void function(...)`

Méthodes abstraites

- Toute méthode abstraite doit être implémenté dans au moins une classe descendante
- Une méthode static ne peut pas être abstraite

Classes abstraites

- Toute classe comportant au moins une méthode abstraite doit être déclarée abstraite
- Toute classe ne comportant aucune méthode abstraite peut être déclarée abstraite
- Il n'est pas possible de créer des instances d'une classe abstraite

But

- Les classes abstraites sont utilisées pour
 - ◆ Structurer l'ensemble des classes
 - ◆ Faciliter la conception des classes en améliorant leur clarté
 - ◆ Créer des modèles rigoureux pour créer des applications spécifiques
 - ◆ Factoriser le code
 - ◆ Permettre le polymorphisme

Exemple

- Notre classe Véhicule est le type même de classe que l'on doit déclarer abstraite

```
public abstract class Vehicule {  
    private int maxVitesse ;  
    private int poinds ;  
    public abstract void démarrer() ;  
    public abstract void arreter() ;  
    public void tourner() { /* Non abstraite : corps ... */ }  
}
```

- On ne peut pas créer d'instances de la classe Vehicule

Exemple 2

- De même, VehTerrestre doit aussi être déclarée abstraite (qu'est ce qu'un véhicule terrestre)

```
public abstract class VehTerrestre {  
    private int nbRoues ;  
}
```

Exemple 3

- Et maintenant que l'on crée la classe Voiture, on doit implémenter les méthodes abstraites

```
public class Voiture extends VehTerrestre {  
    public void demarrer() { /* Corps */}  
    public void arreter() { /* Corps */}  
    public void tourner() { /* Corps si redéfinition */}  
}
```

Remarque

- Une classe abstraite ne contient pas nécessairement que des méthodes abstraites
- La classe `java.io.InputStream` possède une seule méthode abstraite appelée `read()`.
- Le reste de la classe `java.io.InputStream` fournit des fonctionnalités basé sur la méthode `read` (comme les fonctions `read` surchargées)
- Une classe fille de `java.io.InputStream` (comme la classe `java.io.FileInputStream` hérite de ces méthodes non abstraites basées sur la méthode `read` qu'elle implémente

Le mot clé final

- Il permet de rendre final une classe, un champ...
- Variable finale
 - ◆ Les variables finales ne peuvent pas être modifiées : ce sont des constantes. Souvent utilisées avec static

```
static final double PI = 3.14
```
 - ◆ la valeur d'une variable finale ne peut pas être changée après initialisation

```
final int i = (int)(Math.random());  
final int j = i/2;
```
- Référence finale
 - ◆ Ne peut changer de référence d'objet
 - ◆ L'objet peut être modifié

Le mot clé final II ... Le retour

■ Méthode finale

- ◆ Une méthode finale ne peut pas être redéfinie

■ Classe finale

- ◆ Une classe déclarée `final` ne peut pas être dérivée.

```
final class toto ...  
class titi extend toto /* IMPOSSIBLE */
```

- ◆ Toutes les méthodes d'une classe finale sont implicitement finales
- ◆ `String` et `Vector` sont des classes finales

Le Polymorphisme

Introduction

- Le polymorphisme est la capacité qu'offre le langage à exécuter une méthode en fonction de la classe de l'objet en question et non en fonction du type de la référence
- Evite la redondance de code
- Permet l'extension des fonctionnalités sans toucher au code existant

Introduction 2

- La notion de polymorphisme est assez difficile à comprendre
- Elle est simple à mettre en oeuvre
- Elle est une des caractéristiques essentielles d'un langage orienté objet

La lutte des classes

- Nous souhaitons écrire le code permettant de gérer les employés d'une entreprise
- Les employés peuvent être des ouvriers, des cadres, le patron...
- L'usine possède la liste des employés et doit les faire travailler

Première méthode

- Voir `Employe1` et `Usine1`
- Le code correspondant aux différents employés est mélangé :-)
- Le code est lourd :-)
- L'ajout d'une catégorie d'employé est particulièrement lourd :-)

TRES MAUVAISE SOLUTION

Deuxième méthode

- Voir `Usine2` et compagnie
- Le code correspondant aux différents employés est séparé :-)
- Le code est lourd :-)
- Les employés doivent être considéré comme des `Object`
- Il est nécessaire de tester le type d'appartenance :-)
- L'ajout d'une catégorie d'employé est toujours lourd :-)

MAUVAISE SOLUTION

Troisième tentative

- Voir `Usine3` et compagnie
- Le code correspondant aux différents employés est séparé :-)
- Le code est toujours lourd, mais ca va mieux : Héritage !!
- Il est toujours nécessaire de tester le type d'appartenance :-)
- L'ajout d'une catégorie d'employé est toujours lourd :-)

MAUVAISE SOLUTION

Quatrième tentative

- Voir `Usine4`
- Le code correspondant aux différents employés est séparé :-)
- Le code est simple :-)
- L'ajout d'une catégorie d'employé est simple :-)
- Il est possible de construire un `Employe` :-)

BONNE SOLUTION

Cinquième tentative

- Voir `Usine5`
- Le code correspondant aux différents employés est séparé :-)
- Le code est simple :-)
- L'ajout d'une catégorie d'employé est simple :-)
- Il est impossible de construire un `Employe` :-)
- `Ouvrier` ne peut pas hériter d'autres classes (pas d'héritage multiple)

EXCELLENTE SOLUTION

Retour au polymorphisme

- Le polymorphisme est le fait qu'une même écriture puisse correspondre à différents appels de méthodes
- Ce concept est une notion fondamentale de la programmation objet, indispensable pour une utilisation efficace de l'héritage

Liaison tardive

- Si on regarde bien le code de la version 5, on ne sait pas à la compilation quelle fonction va être exécuté
- Le polymorphisme est obtenu grâce au mécanisme de liaison dynamique (ou retardée)
- Lorsqu'un objet reçoit un message, la méthode qui est exécutée est déterminée :
 - ◆ au moment de l'exécution (et non celui de la compilation)
 - ◆ par la classe de cet objet (et non celle de la variable référence désignant cet objet)

Liaison tardive II ... Le retour

- Soit l'appel suivant : `o.m()` ;
 - ◆ Si la classe `C` de l'objet référencé par `o` (et non la classe de la variable référence `o`) contient la définition de la méthode `m()` alors celle-ci est exécutée,
 - ◆ sinon, la méthode `m()` est recherchée dans la classe mère de `C`, puis, si elle ne s'y trouve pas, dans la classe mère de cette classe mère, et ainsi de suite ...

- `Argl` : Ce mécanisme peut ralentir sensiblement l'exécution des programmes

Intérêt du polymorphisme

- Code plus clair : Plus de `switch`
- L'extension est facilitée : On peut rajouter des sous classes sans toucher au code existant
- Héritage et polymorphisme sont indissociables

Une dernière Version

- L'usine peut croître
- On va stocker les employés dans une liste chaînée (LinkedList)

Un autre exemple

- Je crée une instance de Object, je l'affiche avec toString
- Je crée un Ouvrier, je le caste vers Object, je l'affiche !

Interfaces

Introduction

- Pousse le concept d'abstraction un cran plus loin
- Une interface représente en quelque sorte une classe purement abstraite
- Elle déclare : "*Voilà à quoi ressembleront toutes les classes qui implémenteront mon interface*"
- Une interface peut donc être utilisée pour établir des protocoles entre les classes
- Une interface ne peut contenir que
 - ◆ des méthodes qui seront toujours par défaut : `public abstract`
 - ◆ des champs qui seront toujours par défaut `public static final`

Exemple

```
interface Figure {  
    public abstract void dessiner();  
    public abstract void rotation(double angle);  
}
```

Exemple

```
interface ConstantesPhysiques {  
    public static final double AVOGADRO = 6.02214e23 ;  
    public static final double MASSE_ELECTRON = 9.109e-31 ;  
}
```

Implémenter une Interface

- Le mot clé `implements` permet de rendre une classe conforme à une interface particulière.
- Il dit en gros : *"L'interface spécifie ce à quoi ressemble la classe, maintenant on va spécifier comment cela fonctionne"*

```
class Rectangle implements Figure {  
    double x1,y1,x2,y2;  
    void dessiner() { /* Corps de la méthode */  
    void rotation(double angle) { /* Corps de la méthode */  
  
}
```

- Dans la classe Rectangle, on doit trouver le corps de toutes les méthodes de l'interface Figure, sauf si Rectangle est déclarée abstraite

Implémentation et Héritage

- Bien entendu, une classe peut hériter d'une autre classe et en même temps implémenter une interface

```
class carre extends Rectangle {  
/* ici, les méthodes dessiner et rotation existent, elles peuvent être redéfi-  
nies*/  
}
```

```
class B extends A implements C {  
}
```

Remarques

- Il est possible d'implémenter plusieurs interfaces
`class B implements I1, I2, I3...`
- Il est possible d'implémenter plusieurs interfaces et de dériver une classe
`class C extends A implements I1, I2, I3...`
- Si A implémente I4 alors C implémente I4 aussi et n'a pas besoin de le spécifier

Interface et Transtypage

- Il suffit de penser qu'une interface est une classe particulière (et abstraite) pour gérer le transtypage

```
Figure f = new Rectangle(); /* Upcasting */
```

```
Rectangle r = (Rectangle)f; /* Downcasting */
```

Exemple d'utilisation

- On veut créer un algorithme de tri qui puisse être utilisé par tout type d'objet
- Un algorithme de tri se compose
 - ◆ d'une partie invariante : le mécanisme de tri
 - ◆ d'une partie variable : la comparaison de deux éléments

Exemple d'utilisation 2

- La partie variable va être codée sous la forme d'une interface déclarant une méthode de comparaison d'objet

```
interface Comparable {  
    int compareTo(Object o);  
}
```

- `obj1.compareTo(obj2)`
- retourne
 - ◆ 0 si même valeur
 - ◆ -1 si `obj1` est plus petit que `obj2`
 - ◆ +1 si `obj1` est plus grand que `obj2`

Exemple d'utilisation 3

- La partie invariante va être une classe offrant diverses méthodes de tri (bulle, quicksort...)

```
public class Sort {
    public static void triBulles(Comparable[] t) {
        for(int l = t.length-1 ; l > 0 ; l--) {
            for(int i = 0 ; i < l ; i++) {
                if(t[i].compareTo(t[i+1]) > 0) {
                    Comparable tmp = t[i];
                    t[i] = t[i+1];
                    t[i+1] = tmp;
                }
            }
        }
    }
}
```

Exemple d'utilisation 4

- Maintenant, on va utiliser cette classe et cette interface

```
class Date implements Comparable {
    private int j,m,a ;
    ...
    public int compareTo(Object o) {
        Date d = (Date)(o); /* donwcasting */
        if(geta()>d.geta()) return 1 ;
        if(geta()<d.geta()) return -1 ;
        if(getm()>d.getm()) return 1 ;
        if(getm()<d.getm()) return -1 ;
        if(getj()>d.getj()) return 1 ;
        if(getj()<d.getj()) return -1 ;
        return 0 ;
    }
}
```

Exemple d'utilisation 5

- Maintenant, on peut trier un tableau de date

```
Date d[] = new Date[15];  
/* Remplir le tableau */  
Sort.triBulle(d);
```

- plus besoin de réécrire des algos de tris
- L'interface Comparable existe dans l'API de java
- La méthode triBulle existe dans l'api de java : voir classe Arrays et les diverses méthodes qu'elle contient(sort....)
- Il est important de jeter un (GROS) coup d'oeil a l'api de java

Interface ou classe abstraite ?

- Une interface représente un ensemble de fonctionnalités qu'une classe implémentant celle ci doit fournir
- Une classe abstraite représente la description d'un objet qu'il est possible de créer
- Une classe abstraite permet de spécifier certaines fonctionnalités.
- Les interfaces ne peuvent pas contenir des variables d'instances (uniquement des variables de classes)
- Il est préférable d'utiliser les interfaces que les classes abstraites.
- Cependant, les classes abstraites sont quelques fois utiles !!

Interface et constantes

- On évitera d'utiliser les interfaces pour coder les types

```
interface ConstantesPhysiques {  
    public static final double AVOGADRO = 6.02214e23;  
    public static final double MASSE_ELECTRON = 9.109e-31;  
}
```

- On préférera

```
public class ConstantesPhysiques {  
    public static final double AVOGADRO = 6.02214e23;  
    public static final double MASSE_ELECTRON = 9.109e-31;  
    private ConstantesPhysiques(); /* Empêche toute instantiation */  
}
```

Stockage des objets

Introduction

- C'est un programme relativement simple que celui qui ne manipule que des objets dont le nombre et la durée de vie sont connues à l'avance
- Les collections : un concept fondamental de la programmation
- On va voir :
 - ◆ Les tableaux (déjà vu)
 - ◆ La classe Arrays
 - ◆ Les collections

Classe Arrays

- Question : Doit on écrire un algorithme de tri pour chaque type de tableau que l'on manipule ? ?
- La classe Arrays contient un ensemble de méthodes **statiques** réalisant des opérations utiles sur les tableaux
 - ◆ Remplissage
 - ◆ comparaison
 - ◆ tri
 - ◆ recherche
- Pratique, mais montre vite ses limites

Remplissage de tableaux

- La méthode `fill` autorise le remplissage d'un tableau par le même élément
- 4 types de signatures différents pour cette méthode
 - ◆ `static void fill(<TypePrimitif>[] a, int fromIndex, int toIndex, <TypePrimitif> val)`
 - ◆ `static void fill(Object[] a, int fromIndex, int toIndex, Object val)`
 - ◆ `static void fill(<TypePrimitif>[] a, <TypePrimitif> val)`
 - ◆ `static void fill(Object[] a, Object val)`

Comparaison de 2 tableaux

- La méthode statique `equals` de la classe `Arrays` compare les deux tableaux passé en paramètres
- Cette méthode possède diverses signatures suivant que le tableau est basé sur des types primitifs ou sur des objets
- Les deux tableaux seront identiques si
 - ◆ Il ont la même taille
 - ◆ Les éléments sont égaux (au sens de la méthode `equals` de leur classe)

Comparaison de 2 tableaux

- Tableau basé sur des types primitifs

```
int t1[] = {1,2,3,4,5}
int t2[] = {1,2,3,4,5}
System.out.println(Arrays.equals(t1,t2));           True
System.out.println(t1==t2);                         False
t1[0] = 3;
System.out.println(Arrays.equals(t1,t2));           False
```

Comparaison de 2 tableaux

- Tableau basé sur des objets

```
Point a1 = new Point(1,2);  
Point a2 = new Point(1,2);  
Point t1[] = new Point[4];  
Point t2[] = new Point[4];  
Arrays.fill(t1,a1);  
Arrays.fill(t2,a2);  
System.out.println(Arrays.equals(t1,t2));
```

- Dépend de la fonction `equals` de la méthode `Point`

Trier un tableau

- La méthode `sort` (proposant diverses signatures) permet de trier des tableaux
- Si le tableau est basé sur des types primitifs alors le tri est basé sur l'ordre naturel
- Si le tableau est constitué d'objets, alors ceux ci doivent implémenter l'interface `Comparable`

```
interface Comparable {  
    int compareTo(Object o);  
}
```

Exemple

```
class Date implements Comparable {  
    private int j,m,a ;  
    public int compareTo(Object o) {  
        Date d = (Date)(o) ; /* donwcasting */  
        if(geta()>d.geta()) return 1 ;  
        if(geta()<d.geta()) return -1 ;  
        if(getm()>d.getm()) return 1 ;  
        if(getm()<d.getm()) return -1 ;  
        if(getj()>d.getj()) return 1 ;  
        if(getj()<d.getj()) return -1 ;  
        return 0 ;  
    }  
}
```

■ Arrays.sort(d1);

Rechercher dans un tableau trié

- La méthode `binarySearch` permet de rechercher des items sur un tableau trié
- Retourne l'index de l'item si celui-ci existe dans le tableau, -1 sinon

Copie de tableau

- La classe `System` possède la méthode `arraycopy` qui permet la copie de tableaux bien plus rapidement que par une boucle `for`
- `static void`
`arraycopy(source, debut, destination, debut, longueur)`
- On a un tableau `noms` et on veut doubler sa taille

```
String tmp[] = new String[noms.length * 2];  
System.arraycopy(noms, 0, tmp, 0, noms.length);  
noms = tmp;
```

Les conteneurs

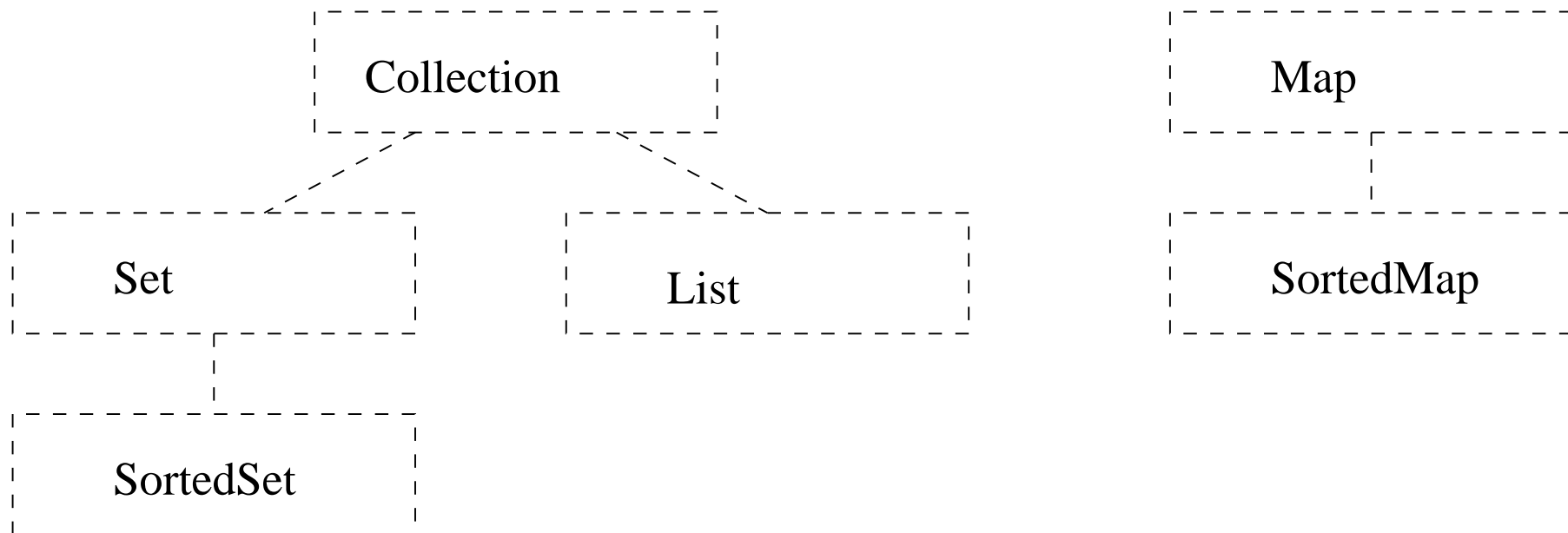
Conteneurs : Introduction

- Question : Doit on implémenter une file à chaque fois que l'on en a besoin ?
- Une file de `Points`, une File de `Date` restent des files

- Ensemble d'interfaces et de classes permettant de travailler sur
 - ◆ des ensembles
 - ◆ des listes chaînées
 - ◆ de tables de hachage
- Le mot clé : Réutilisation

Conteneurs et interfaces

- Les conteneurs sont basés sur plusieurs interfaces
- L'interface `Collection` est la mère des interfaces associés aux listes.
- L'interface `Map` est la mère des interfaces associés aux tables de hachage



Interface Collection

- Elle ne précise pas si les objets peuvent apparaître plusieurs fois
- Elle ne précise pas si les éléments sont insérés suivant un ordre
- Elle contient les fonctions suivantes

| | |
|-------------------------------------|---|
| <code>boolean add(Object)</code> | Ajoute un objet. Renvoie true si l'ajout a été possible |
| <code>void clear()</code> | Supprime tous les éléments du conteneur |
| <code>void contains(Object)</code> | renvoie true si l'objet appartient au conteneur |
| <code>boolean isEmpty()</code> | renvoie true si le conteneur est vide |
| <code>Iterator iterator()</code> | Renvoie un itérateur, utilisé pour parcourir les éléments |
| <code>boolean remove(Object)</code> | supprime l'objet de la collection. Renvoie true si ok |
| <code>int size()</code> | renvoie le nombre d'éléments de la collection |
| <code>Object[] toArray()</code> | renvoie un tableau contenant tous les éléments |

Interface Set

- Représente une collection dans laquelle la duplication d'éléments est interdite
- Elle ne possède pas d'autres méthodes que celles hérités de `Collection`
- Elle rajoute une règle interdisant la duplication d'objets. `add` renvoie `false` si on essaie de mettre deux fois le même objet

Interface SortedSet

- Interface associé aux ensembles (duplication d'éléments interdites)
- Les éléments de la collection sont maintenus dans l'ordre.
- Il faut donc implémenter l'interface `Comparable` sur les objets de la collection

Interface List

- Représente une collection dont les éléments sont rangés dans un ordre particulier
- La méthode `add(Object o)` ajoute l'objet `o` en fin de liste
- Plusieurs fonctions sont ajoutés :
 - ◆ `public void add(int index, Object o)` ajoute l'objet à un certain endroit de la liste
 - ◆ `public void remove(int index)` supprime l'objet situé en ieme position
 - ◆ `public void get(int index i)` retourne l'objet de la liste situé en ieme position

Interface Map

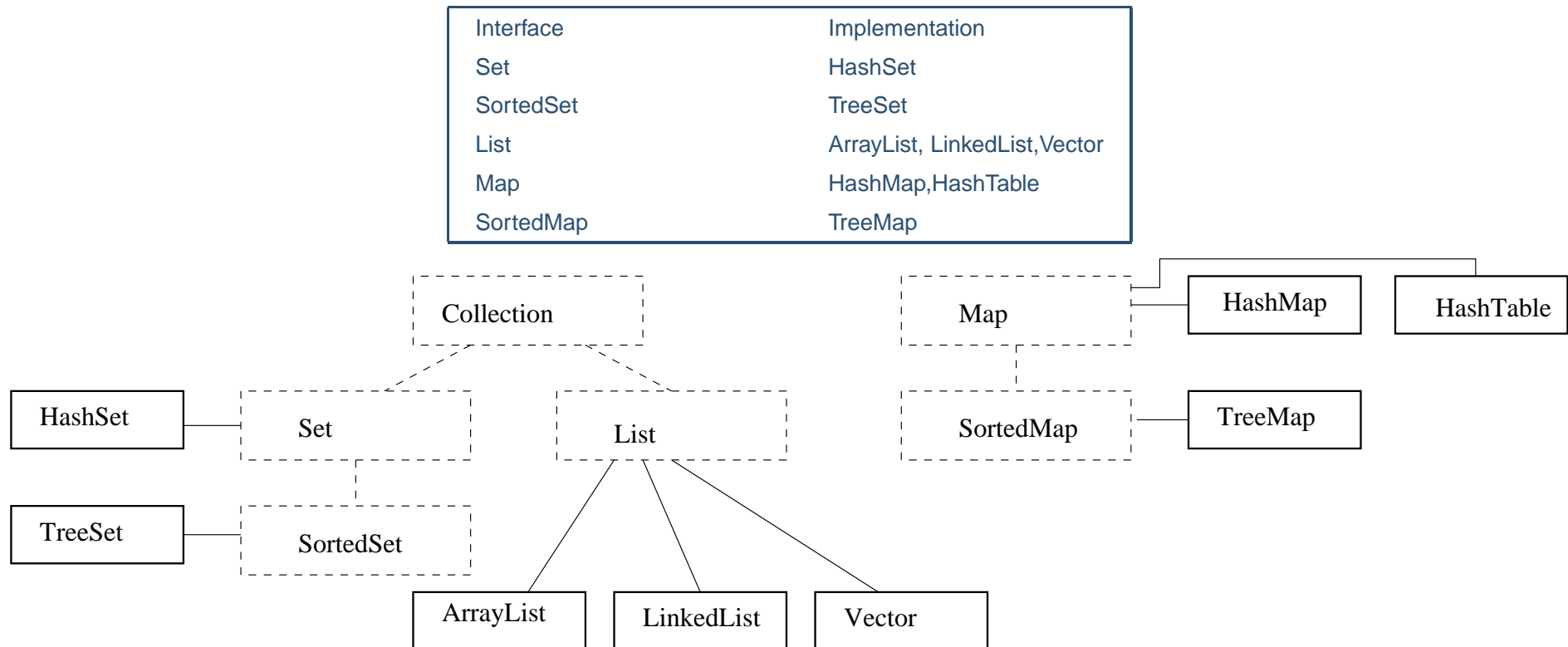
- Permet de gérer des collections de paires clé/valeur
- Elle stocke un élément de type objet et l'associe à une clé (de type Object)
- Les opérations de bases sont les suivantes
 - ◆ `public Object put(Object cle, Object val)` Ajoute la paire
 - ◆ `public Object get(Object cle)` Récupère la valeur associé à la clé
 - ◆ `public int size()` Nombre de paires Clé/valeurs
 - ◆ `public Set keySet()` renvoie une Set contenant toutes les clés de la carte
 - ◆ `public Collection()` renvoie une collection contenant toutes les valeurs de la carte

Interface SortedMap

- Cette interface entretient les paires clé/valeurs en ordre trié sur les clés
- Les objets associés au clés doivent implémenter l'interface `Comparable`
- Certaines méthodes sont ajoutés.

Implémentations

- Ces différentes interfaces ont été implémenté dans l'API



Implémentations ... 2

- ArrayList offre de bonnes performances lors d'ajouts fréquents en fin de liste
- LinkedList offre de bonnes performances lors de nombreux ajouts et suppressions en milieu de listes
- HashMap et Vector existent depuis le SDK1.0 : on les garde

Les itérateurs

- Les itérateurs `Iterator` servent à parcourir les collections
- Comment faire ?
 - ◆ Récupérer un itérateur : méthode `iterator` de `Collection`
 - ◆ utiliser les méthodes `next()` et `hasNext()` de `Iterator`

```
public void printElements(List listeDate) {  
    Iterator it = c.iterator();  
    while(it.hasNext()) {  
        Date d = (Date)it.next();  
        System.out.println(d);  
    }  
}
```

Et maintenant

- Depuis Java5, la gestion des listes a été améliorée grâce à la généricité
- On reviendra sur ce concept plus tard
- On peut créer des listes ne contenant que des Dates :

```
public void printElements(List<Date> listeDate) {  
    Iterator<Date> it = c.iterator();  
    while(it.hasNext()) {  
        Date d = it.next();  
        System.out.println(d);  
    }  
}
```

Parcours de listes...

- Et en plus on a le parcours simplifié depuis java 5.

```
public void printElements(List<Date> listeDate) {  
    for(Date d : listeDate) {  
        System.out.println(d);  
    }  
}
```

- Ça marche aussi pour les tableaux

Petit Problème

- Lorsque l'on insère des objets dans des collections on les transtype vers un type plus général (Object)
- Lorsque l'on voudra récupérer l'objet de la collection il faudra surement prévoir un cast explicite pour récupérer le vrai type de l'objet

Petit problème... 2

- On ne peut pas mettre directement dans un conteneur des types primitifs (entier, double...)
- Il faut les empaquetés dans les classes associés (Integer, Double...)
- Très désagréable
- Un exemple
- La nouvelle version de java 5.0 permet de stocker des entiers.... :-)

Les exceptions

Introduction

- L'instant idéal pour détecter une erreur est la compilation
- Tout ne peut pas être réglé à cette étape
- Les problèmes restants doivent être gérés à l'exécution avec un certain formalisme
- permettre à l'initiateur de l'erreur de passer une information suffisante à un récepteur qui va savoir comment traiter proprement cette anomalie.

Origine des erreurs

- Les erreurs d'entrées sorties
 - ◆ Fautes de saisies de l'utilisateur
 - ◆ Fichier corrompu / inexistant
- Les erreurs de matériels
 - ◆ Imprimante déconnectée
 - ◆ Page web temporairement inaccessible
- Les contraintes physiques
 - ◆ disque dur saturé
 - ◆ Mémoire insuffisante
- Les erreurs de programmation
 - ◆ index de tableau
 - ◆ emploi d'une référence nulle

Traitement des erreurs

- En présence d'une erreur, un programme peut avoir un comportement
 - ◆ Idéal : Revenir à un état défini et poursuivre l'exécution
 - ◆ Honnête : Prévenir l'utilisateur, sortir correctement après une sauvegarde
 - ◆ lamentable : Ecran bleu + erreur à l'adresse 0x7743!!

Traitement des erreurs en C

- Convention de programmation : un programme retourne un code qui permet de connaître son comportement durant l'exécution
- Est on sûr que tout le monde programme de cette façon : **NON**
- Les programmes C ne sont pas les plus sûrs du monde informatique

Traitement des erreurs en Java/C++

- Gestion d'exceptions
 - ◆ Si un problème survient, une exception est levée,
 - ◆ une exception est un objet spécialement créé pour gérer les erreurs
 - ◆ Toute exception peut/doit être levée
 - ◆ Les comportements anormaux sont gérés avec les exceptions
 - ◆ il est important de distinguer une condition exceptionnelle d'un problème normal
 - ◆ Avec une condition exceptionnelle vous ne pouvez pas continuer l'exécution car vous n'avez pas suffisamment d'informations pour la traiter dans le contexte courant
- Le code de gestion des exceptions est à part
- Les exceptions sont des classes de Java
- Les programmes Java sont assez sûrs

Les exceptions

- Certaines méthodes peuvent lancer des exceptions
- Lorsqu'on utilise une telle méthode , on peut
 - ◆ Attraper les exceptions levées
 - ◆ Ignorer les exceptions levées et les relancer

Exemple

- Dans la classe `InputStream` (buffer d'entrée) se trouve la méthode `read` définie comme suit :

```
public int read() throws IOException
```

- ainsi toute instance de `InputStream` peut éventuellement lancer une `IOException`
- Toute fonction utilisant cette méthode a la possibilité de lever l'exception `IOException`

Attraper une exception

- Une méthode qui utilise une méthode levant une exception peut attraper celle ci
- Un bloc **try{ }** délimite le code à surveiller
- Au moins un bloc **catch{ }** suit : Il gère les exceptions attrapées
 - ◆ Chaque bloc **catch** permet de récupérer la main pour traiter un type d'exception
 - ◆ Ils sont considérés dans l'ordre jusqu'à ce que l'exception levée corresponde au type ou à un sous type de celui indiqué en argument
- un bloc **finally{ }** apparaît éventuellement

Exemple

```
int lireAscii() {  
    int x = 0;  
    try { x=System.in.read(); }  
    catch(IOException e) {  
        System.out.println("Erreur : " + e);  
    }  
    System.out.println("toto" + x);  
    return x;  
}
```

En cas d'erreur

Erreur : java.io.IOException : ...

toto 0

Comportement normal

toto ?

Exemple

```
try {
    readFromFile("nomFile");
    ...
}
catch(FileNotFoundException) {
    /* Gestion du cas : fichier introuvable */
}
catch(IOException) {
    /* Gestion du cas : erreur de lecture */
}
catch(Exception) {
    /* Gestion du cas : Toute autre erreur */
}
```

Exemple

```
Scanner sc = new Scanner(System.in) ;
int nb = 0 ; try {
    nb = sc.nextInt() ;
} catch(InputMismatchException e) {
    S.o.p("un entier !!!") ;
}
```

Laisser passer une exception

- Dans certains cas, il est intéressant de laisser passer l'exception
- Il faut alors ajouter à l'en-tête le type de l'exception devant être relancée avec le mot-clé **throws**
- l'erreur « remonte » alors vers la méthode appelante qui peut elle-même l'attraper ou la laisser remonter
- Si tout le monde laisse passer l'exception, y compris la méthode main, alors le programme s'arrête

Exemple

```
int lireAscii() throws IOException {  
    int x = System.in.read();  
    return x;  
}
```

et finally

- Le bloc **finally** { } contient un traitement exécuté systématiquement
 - ◆ que le bloc try est levé ou pas l'exception
 - ◆ que l'exception est été attrapée ou non
- utilisé pour libérer certaines ressources

Mécanisme de traitement

Exception levé en dehors du bloc try

1. La méthode retourne immédiatement ; l'exception remonte vers la méthode appelante
2. La main est donnée à la méthode appelante
3. L'exception peut alors éventuellement être attrapée par cette méthode appelante ou par une des méthodes actuellement dans la pile d'exécution

Exception levée dans un bloc try

1. Si une des instructions du bloc try provoque une exception, les instructions suivantes du bloc try ne sont pas exécutées et,
 - si au moins une des clauses catch correspond au type de l'exception,
 - (a) la première clause catch appropriée est exécutée
 - (b) l'exécution se poursuit juste après le bloc trycatch
 - sinon
 - (a) la méthode retourne immédiatement
 - (b) l'exception remonte vers la méthode appelante

Pas d'exception générée dans le bloc try

- Dans les cas où l'exécution des instructions de la clause try ne provoque pas d'erreur/exception,
 1. le déroulement du bloc de la clause try se déroule comme si il n'y avait pas de bloc trycatch
 2. le programme se poursuit après le bloc trycatch

Exemples de traitement

- Fixer le problème et réessayer le traitement qui a provoqué le passage au bloc catch
- Faire un traitement alternatif
- Retourner (return) une valeur particulière
- Sortir de l'application avec System.exit()
- Faire un traitement partiel du problème et relancer (throw) la même exception (ou une autre exception)

Lancer une exception

- On lance une exception pour signaler une situation **anormale**
- Dans l'en-tête d'une méthode lançant une exception se trouve le mot-clé **throws** puis la liste des exceptions pouvant être lancées
- Dans le corps d'une méthode lançant une exception se trouvent une ou plusieurs instruction `throw` permettant de lancer des exceptions

Exemple

```
int lireDenominateur() throws ArithmeticException {  
    int x = 12;  
    x = Mediator.readInt();  
    if(x==0) throw new ArithmeticException();  
    System.out.println("x = " + x);  
    return x;  
}
```

Si $x = 0$

n'affiche rien

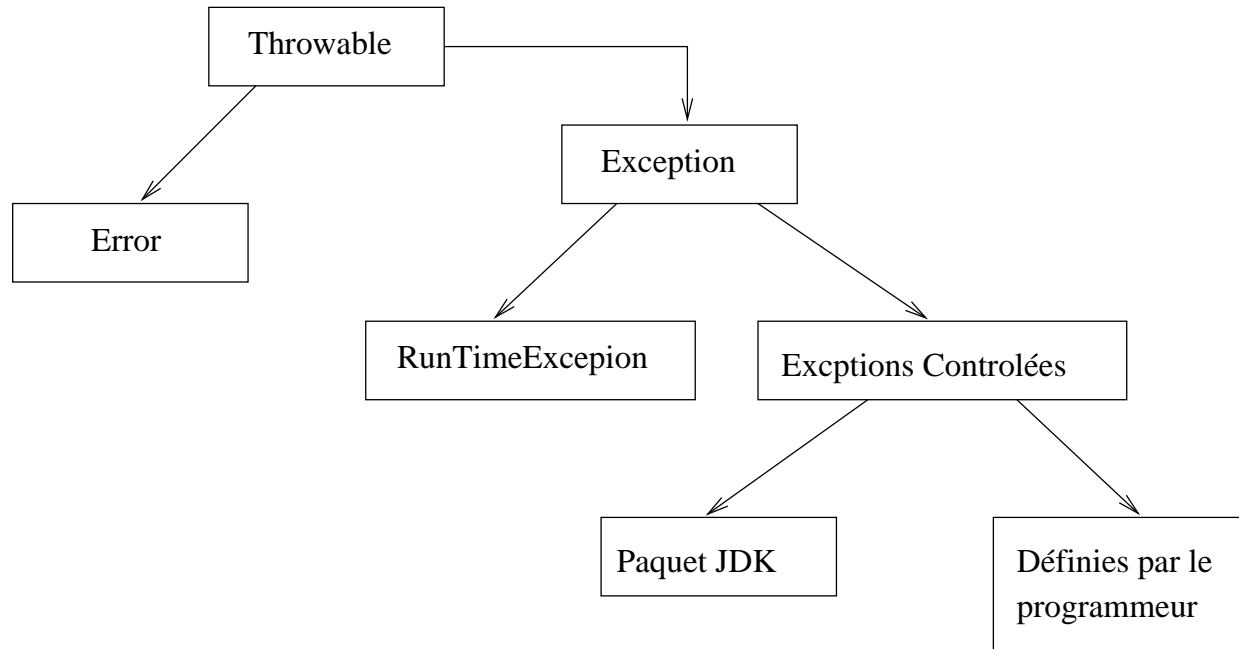
Si $x \neq 0$

$x = ? ?$

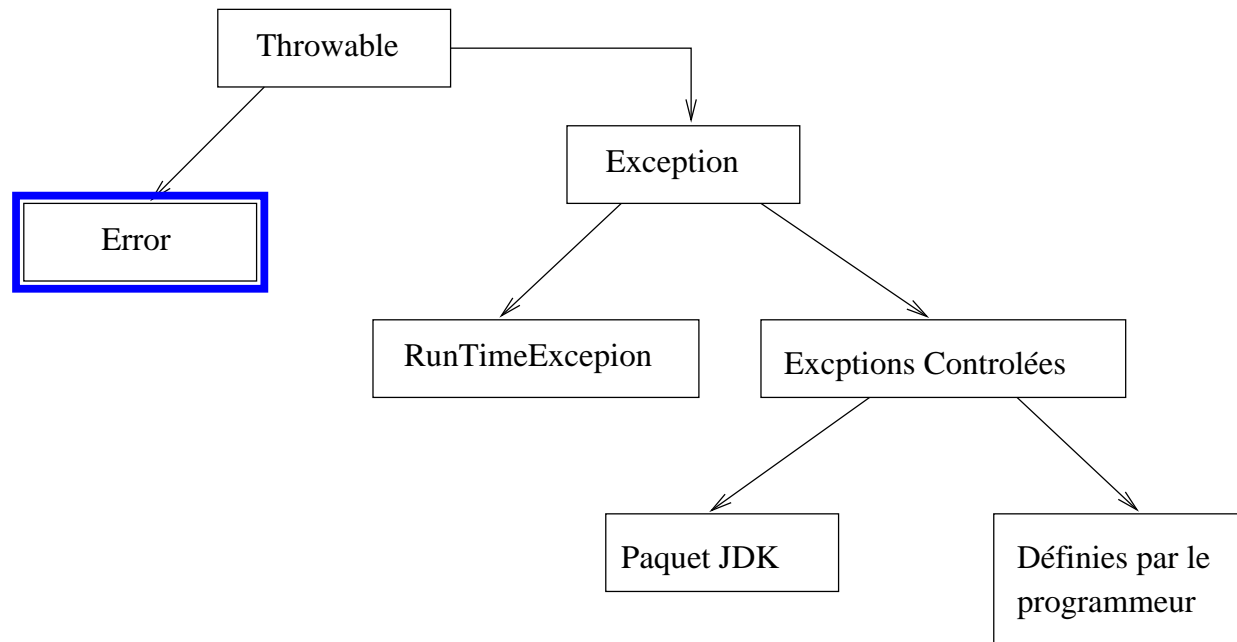
Hiérarchie des exceptions

- Les classes liées aux erreurs/exceptions sont des classes placées dans l'arborescence d'héritage de la classe Throwable
- Le JDK fournit un certain nombre de telles classes
- Le programmeur peut en ajouter de nouvelles

Arbre d'héritage des exceptions

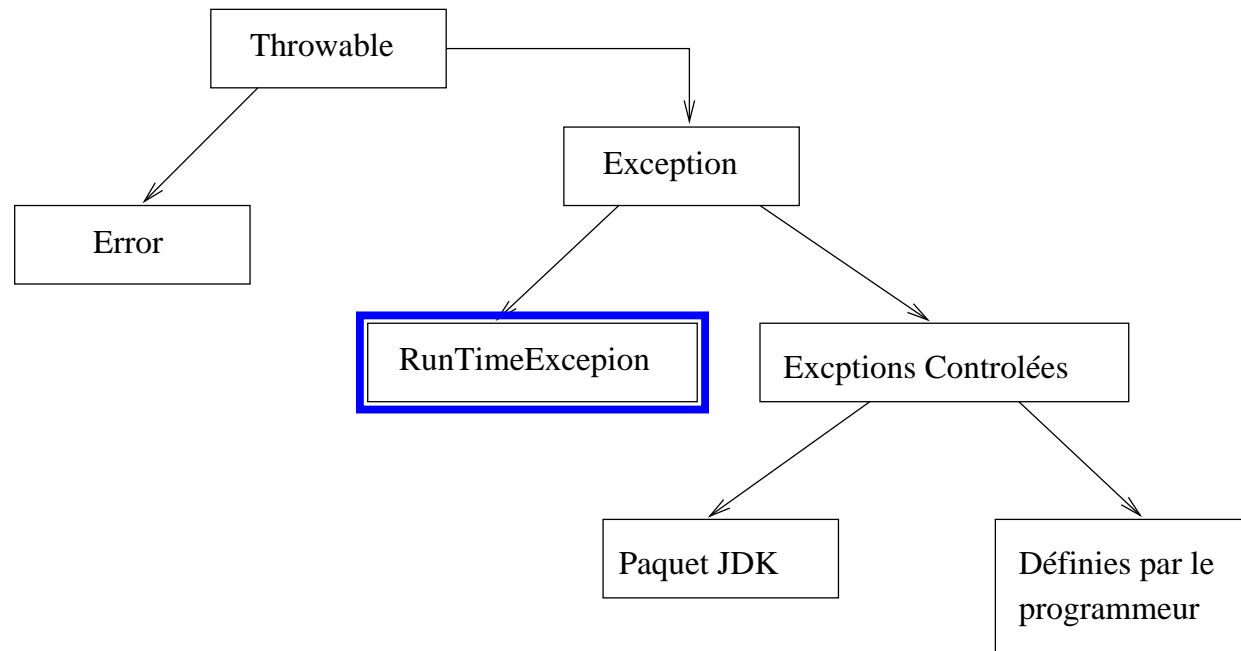


Arbre d'héritage des exceptions



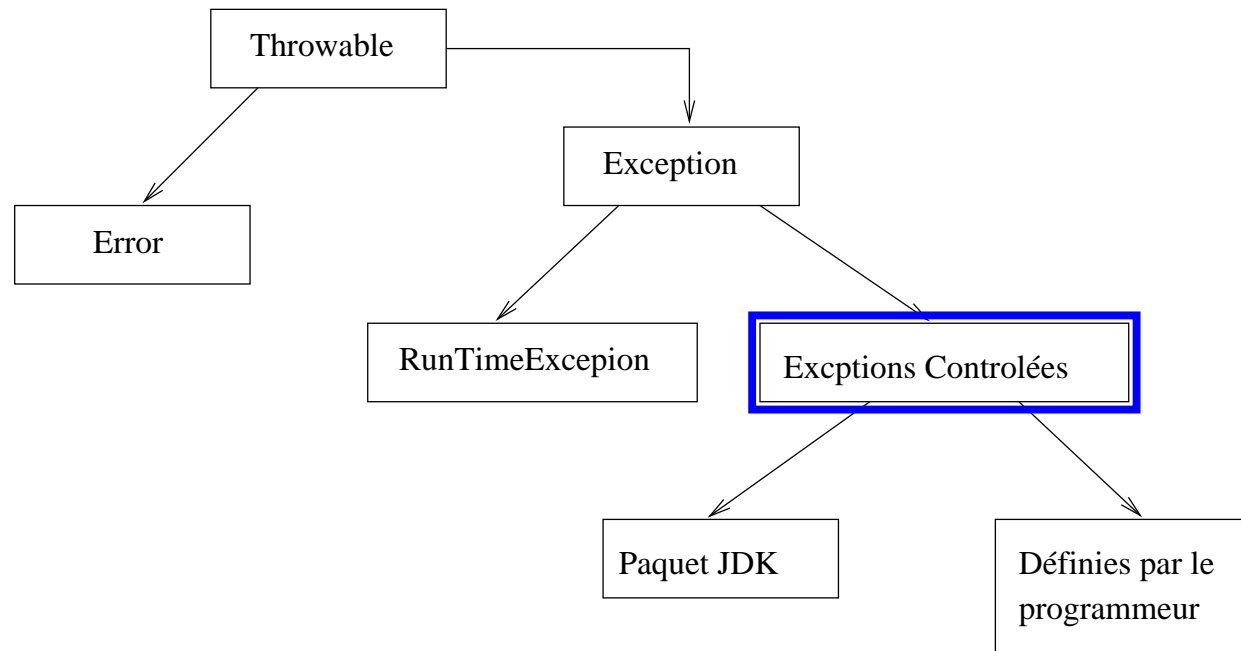
- Les erreurs graves

Arbre d'héritage des exceptions



- Les erreurs graves
- Les exceptions non contrôlées par le compilateur

Arbre d'héritage des exceptions

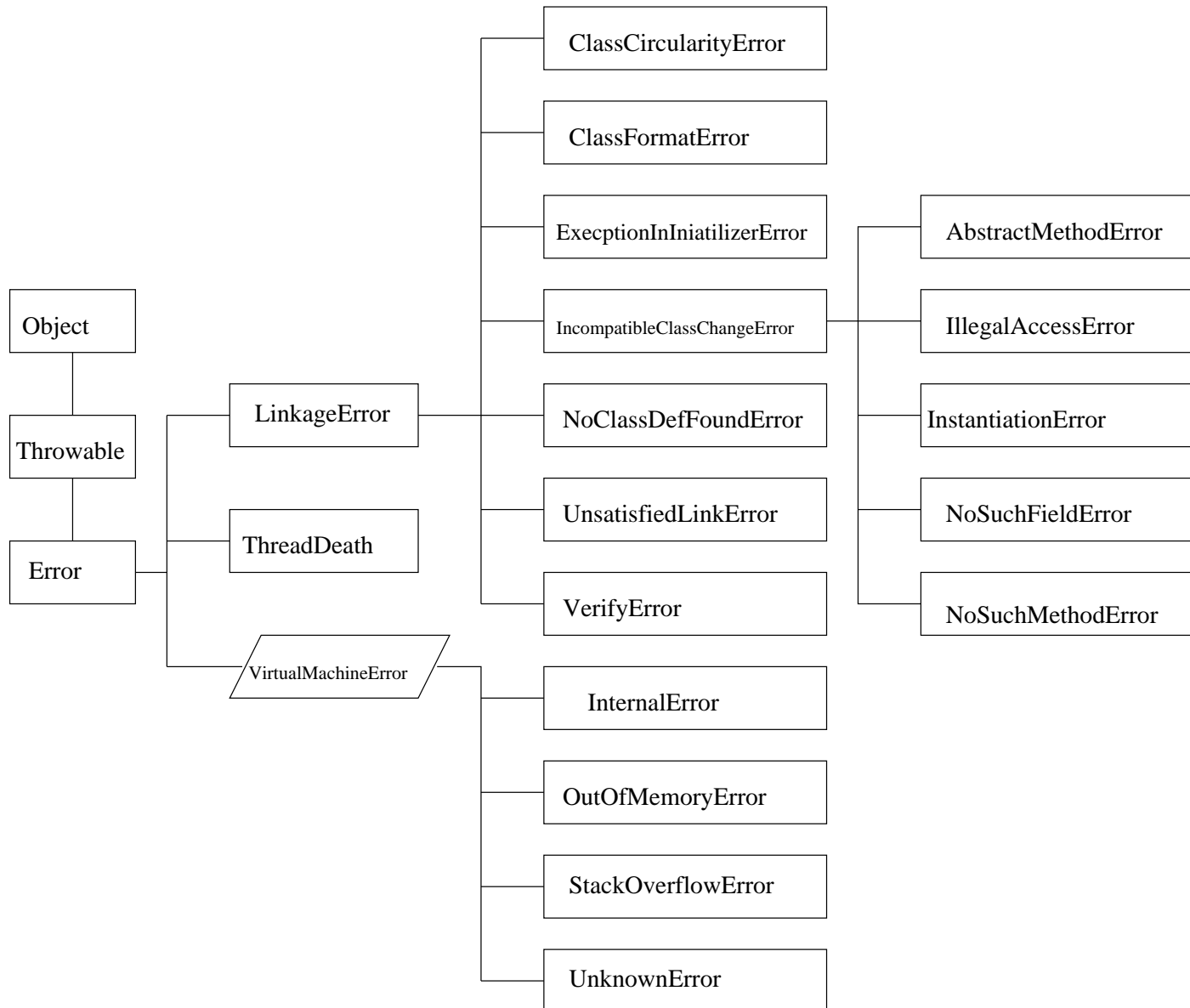


- Les erreurs graves
- Les exceptions non contrôlées par le compilateur
- Les exceptions contrôlées par le compilateur

Classe Error

- Une exception Error est levée si
 - ◆ une erreur interne se produit
 - ◆ un manque de ressources (mémoire ...) apparaît
- ne doit pas être attrapée
- Assez rare

Types d'erreurs



Les exceptions non contrôlés

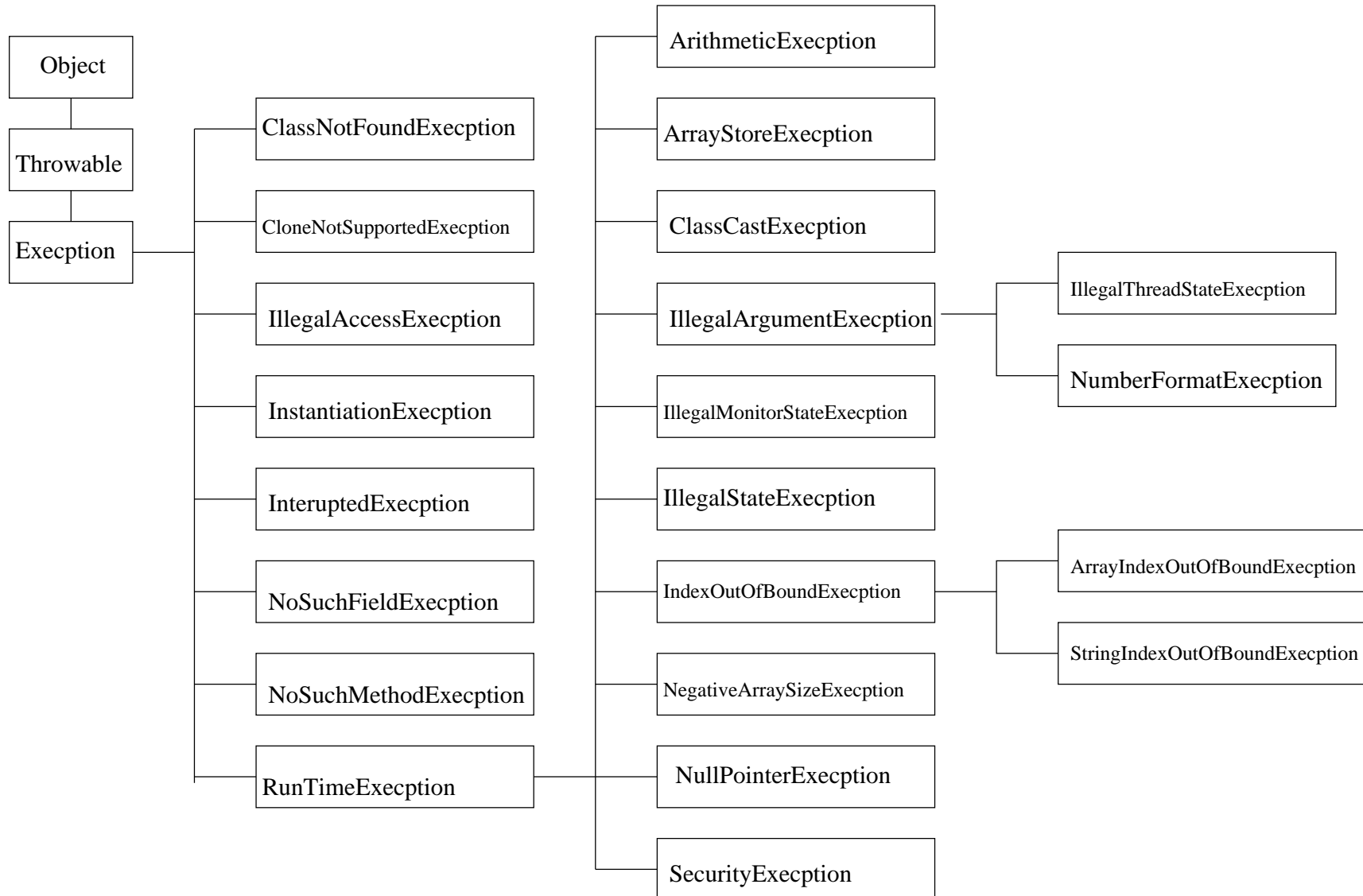
- issues de la classe `RuntimeException` : définies dans SDK
 - ◆ Peuvent être lancées par la JVM
 - ◆ Peuvent être attrapées, ne doivent pas être signalées
 - ◆ exemple : `ArrayIndexOutOfBoundsException`, `NullPointerException`...
 - ◆ Heureusement, car sinon le code java serait rempli de try et catch et serait extrêmement lent...

Les exceptions contrôlés

- Héritent de la classe `Exception`, mais pas de `RuntimeException`
- Le compilateur vérifie que les méthodes les utilisent correctement (les attrapent ou les relancent)
- Toute méthode qui peut lancer une telle exception doit le déclarer

```
void lance(..) throws TotoException
```
- Exemple : `ClassNotFoundException`, `SQLException`

Types d'exceptions



Créer des nouvelles exceptions

- Intéressant lorsque l'on rencontre un problème qui ne correspond pas de façon adéquate à l'une des exceptions standards
- Il suffit de créer une classe héritant de exception
- On crée deux constructeurs
 - ◆ Un sans arguments
 - ◆ Un avec une chaîne de caractère comme argument qui contient un message détaillé de l'erreur
- C'est tout

Exemple

```
class DivisionByZeroException extends ArithmeticException {  
    DivisionByZeroException() {  
        super("Division by 0");  
    }  
    DivisionByZeroException(String s) {  
        super(s);  
    }  
}
```

- C'est tout : `DivisionByZeroException` est créée, je peux la lancer avec `throw`

Exemple

```
int lireDenominateur() throws DivisionByZeroException {  
    int x = Mediator.readInt();  
    if(x==0) throw new DivisionByZeroException();  
    return x;  
}
```

Exceptions et performances

- Si aucune exception n'est levée, l'impact sur les performances est négligeable
- La levée d'une exception peut au contraire avoir un impact non négligeable sur les performances

Conseils

- Les exceptions doivent être utilisées dans des cas exceptionnels
- Il existe de nombreuses exceptions, essayez de les utiliser avant d'en créer des nouvelles
- Elles ne doivent pas remplacer des tests simples tels que l'appartenance d'un index dans les bornes d'un tableau !

Exemple

```
for(int i=0 ; i < a.length ; i++) {  
    System.out.println(a[i]);  
}
```

```
try {  
    int i = 0 ;  
    while(true) {  
        System.out.println(a[i]);  
        i++;  
    }  
    catch(ArrayIndexOutOfBoundsException e) { }
```

1. C'est Horrible
2. C'est 20 fois plus lent à l'exécution

Exceptions et Constructeur

- Aucune instance n'est créée si l'exception est levée dans un constructeur
- Exemple..