h e g

Haute école de gestion
Genève

Programmation Orientée Objets

63-31 - Programmation Collaborative

Claude Chaudet

Hes·so GENÈVE
Haute Ecole Spécialisée
de Suisse occidentale

Définition - Programmation Orientée Objets

Programmation "classique" (procédurale, impérative, ...)

- Division d'un programme en plusieurs fonctions
- Beaucoup d'inter-dépendances
- Listes d'arguments des fonctions longue et redondante
- Recopie des variables (changements de contexte)
- ⇒ Code très rigide

Philosophie POO : mettre l'accent sur les données plutôt que sur la logique

- Une des critiques principales de la POO est qu'on oublie l'algorithmique

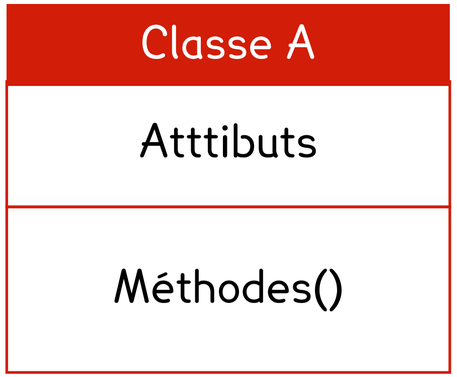
En fait plutôt une division

- Conception : que veut-on manipuler plutôt que comment le manipuler
- Implémentation : comment réaliser des tâches de manière efficace

Définitions

Une Classe est un modèle (représentation abstraite) d'une entité manipulée par un programme informatique

Définition de la structure des objets manipulés par le programme



Un objet est une instance d'une classe (une entité particulière, une variable)

Une classe définit:

- Des attributs : données caractérisant chaque objet (variables) ; son identité
- Des méthodes : les actions que l'objet peut réaliser (fonctions membres) ; son comportement

Avantages & Inconvénients

😀 Proche du modèle de données

😀 Modularité

- ✦ Meilleure organisation du code, modules indépendants
- ✦ Facilité de travailler en équipe

😀 Réutilisation plus facile du code

- ✦ Héritage
- ✦ Frameworks cohérents

🤨 Critiques

- ✦ Mettre trop l'accent sur les données et non sur les algorithmes
- ✦ Code plus lourd et parfois plus long à compiler

Structure Typique Programme Java

Nom du Programme ; doit correspondre au nom du fichier (MonProgramme.java)

Méthode main ; point d'entrée dans le programme (uniqu

```
public class MonProgramme {
    public static void main(String[] args) {
        int a = 20;
        a = a * 2;
        System.out.println("La valeur de a est : " + a);
    }
}
```

Affichage ; affiche du texte sur la console

Variable ; déclaration & initialisation

Déclaration et utilisation des classes


```
class MaClasse {
    // Attributs
    String          attr1;
    int             attr2 = 10;
    String          attr3;

    // Constructeur
    void MaClasse() { … }

    // Méthodes
    void meth2() { … }

    void meth3() { … }

    // Finaliseur (destructeur)
    void finalize() { … }
}
```

```
public class MonProgramme {
  public static void main(String[] args)
  {

    // Création d'un objet
    MaClasse x = new MaClasse()

    // Accès à un attribut
    x.attr1 = "Hello";

    // Appel d'une méthode
    x.meth2();
  }
}
```

TBD : Quiz sur le vocabulaire & les concepts clés

Concepts clés

Encapsulation

Définition : un objet contient toute l'information dont il a besoin et n'expose que ce qui est nécessaire

En pratique : définition des classes comme collection de méthodes & attributs

Avantage : cohérence

Objectif ultime : méthodes ne prenant aucun paramètre

Encapsulation - Intérêt

```
float prixHT = 100;
float tauxTVA = 6;
float prixExpedition = 5;
float prixAssurance = 10;
String nom = "Claude Chaudet"
String adresse = "Rue de la Tambourine, 17"
String ZIP = "1227"
String ville = "Carouge"

Facture(prixHT, prixExpedition, prixAssurance, tauxTVA, nom, adresse, ZIP, ville)
```

```
c = new Client("Claude Chaudet", "Rue de la Tambourine, 17", "1227", "Carouge")
m = new Commande()
m.prixHT = 100;
m.prixExpedition = 5;
m.prixAssurance = 10

m.Facture();
```

Visibilité de l'information

On range souvent sous la notion d'encapsulation la question de la visibilité de l'information

- privé, public, protégé, ...

C'est en réalité une notion transversale (utilisée pour l'encapsulation, mais aussi l'abstraction)

Avantage : améliorer la sécurité du programme, éviter les corruptions accidentelles de données

Concepts : visibilité (public, private, protected, package-protected), méthodes get/set

Java : Visibilité de l'Information


```java
public class MaClasse {
    // Attributs
    private String          attr1;
    private int             attr2;
    protected String        attr3;

    // Constructeur - public
    public void MaClasse() {
    }

    // Méthodes
    void meth2() {
    }

    private void meth3() {
    }
}
```

	Classe elle-même	Package	Classes héritières	Autres (monde)
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
(rien)	✓	✓	✗	✗
private	✓	✗	✗	✗

Abstraction

Définition : On définit des objets par leur interface externe, pas par leurs détails ou leur mécanique

En pratique : On se pose la question de ce dont on a besoin d'exposer

Exemples : dans une voiture, le conducteur manipule le volant et les pédales, peu importe le type d'énergie (essence, électrique, ...) ou les détails du mélange essence-air

Avantages :

#1 : Interface plus intuitive

#2 : On réduit l'impact des changements

exemple: changement de moteur de la voiture

#3 : Liberté d'implémentation & de division interne des tâches

Objectif ultime : Pas besoin d'explications ou de documentation pour utiliser une classe

Différence Encapsulation / Abstraction / Visibilité

Encapsulation = regroupes l'information dans des entités cohérentes

- On divise le programme (ou ses données) en blocs qui ont un sens

  Voiture = moteur + boîte de vitesse + carrosserie + ...

Abstraction = n'exposer que les fonctions pertinentes, cacher les détails

- On montre le "quoi" mais pas le "comment"

  Voiture = levier de vitesses, accélérateur, frein, volant, clignotants, ...

Visibilité = restriction d'accès ; mécanisme pour rendre tout ça plus "propre"

- On s'autorise à avoir plus que ce qu'on expose

  Voiture = ce qui est dans le capot ou dans l'habitacle

Héritage

Définition : Les classes peuvent réutiliser du code écrit pour d'autres classes

En pratique : définition d'une hiérarchie, certains objets n'étant que des extensions ou spécialisations d'autres plus généraux.

Avantages : On peut réutiliser du code sans faire des objets "couteau suisse"

Objectif ultime : On n'écrit chaque algorithme qu'une seule fois

Java : Héritage

```java
public class MaClasse {
    // Attributs
    private String          attr1;
    private int             attr2;
    protected String        attr3;

    // Constructeur - public
    public void MaClasse() {
    }

    // Méthodes
    void meth2() {
    }

    private void meth3() {
    }
}
```

```java
public class MaClasseFille extends Maclasse {
    // Dispose déjà de attr3 et de meth2()

    // Constructeur - public
    public void MaClasseFille() {
            // Appel du constructeur parent
            super();
    }

    // Nouvelles méthodes
    private void meth4() {
    }

    // Redéfinition de méthodes
    void meth2() {
            // Appel de la méthode parente
            super.meth2();
            …
    }

}
```

Polymorphisme

Définition : si plusieurs objets fournissent la même fonction, ils peuvent être interchangés à l'exécution

En pratique : Spécifier des types génériques plutôt que des types concrets afin de n'être dépendant que du strict minimum.

Avantages : Flexibilité du code ; on peut décider de quel objet va fournir une fonction au moment de l'exécution

> Pas besoin de bloquer les choses au moment de la compilation ou de traiter des cas différents avec des conditions

Objectif ultime : réduire sensiblement le nombre de "if" et de "switch"

Java : Polymorphisme

On déclare une variable comme étant une instance d'une classe abstraites

Au moment de l'instanciation, on utilise n'importe laquelle des classes dérivées

```
abstract class Animal {… void manger(); …}
class Chien extends Animal {…}
class Chat extends Animal {…}
…
Animal a = new Chien();
a.manger();
a = new Chat();
a.manger();
```

Intérêt principal : bonne conception. Ce qui nous intéresse ici c'est la capacité à manger, par le fait que l'animal soit un chien ou un chat => pas besoin d'introduire de contrainte.

Java : Interfaces

Une interface est une collection de méthodes abstraites

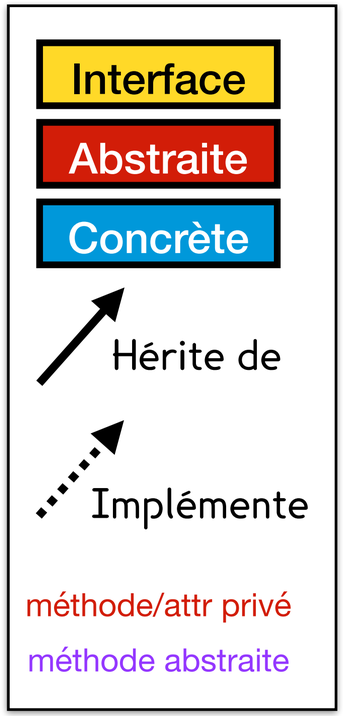
```
public interface MonInterface { … }
…
public class MaClasse implements MonInterface { … }
```

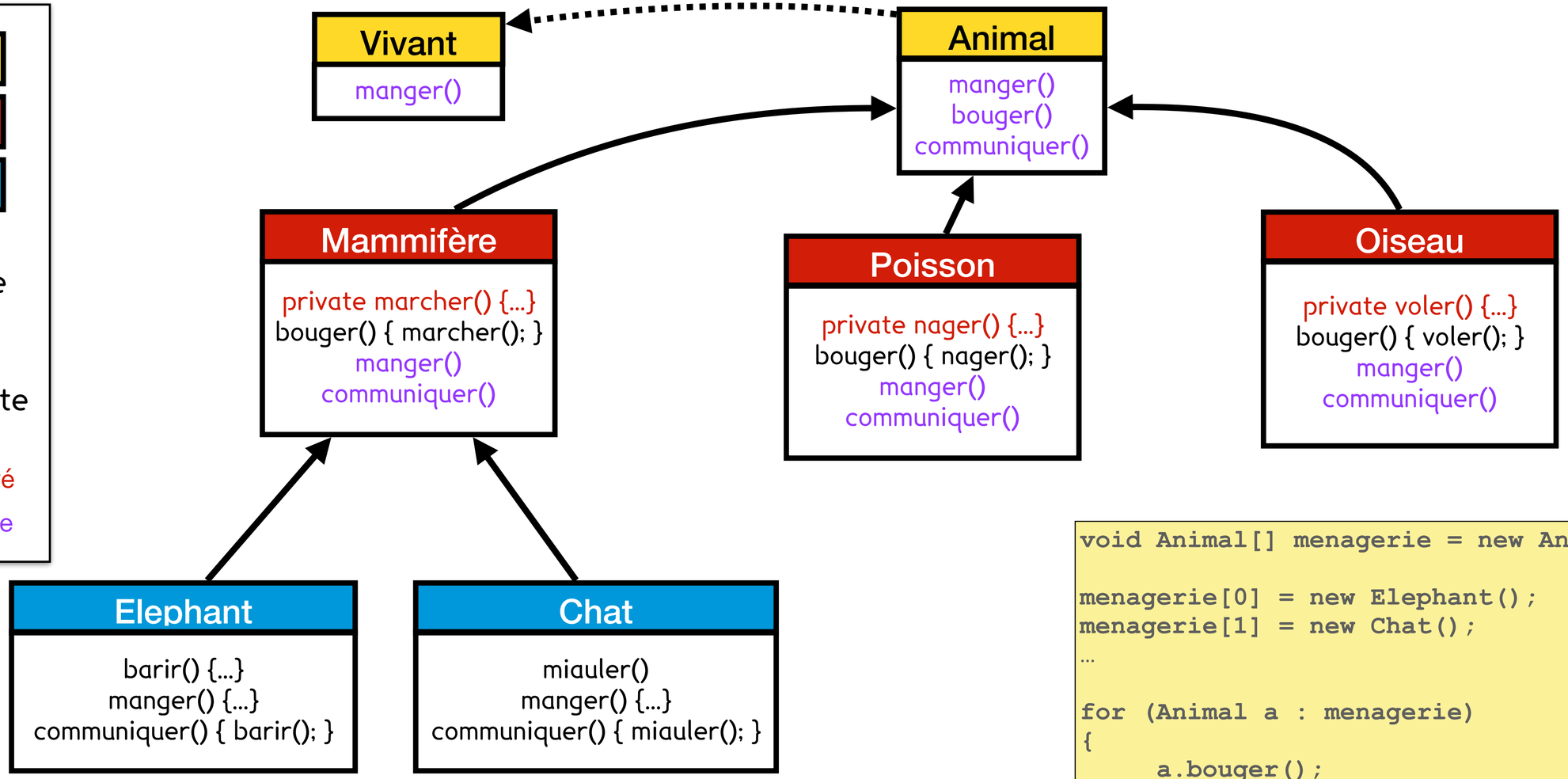
Philosophie : c'est un contrat que doivent respecter les classes qui implémentent l'interface (ou les interfaces)

- On l'utilise pour indiquer qu'une classe "est un"... ou possède certaines capacités

Possibilité d'avoir une interface vide : **tagging interface**

Exemple






```
void Animal[] menagerie = new Animal[10];

menagerie[0] = new Elephant();
menagerie[1] = new Chat();
…

for (Animal a : menagerie)
{
     a.bouger();
}
```

Java : attributs statiques

Un attribut peut être déclaré comme statique

On définit alors une variable de classe, par exemple nombre d'objets instanciés

```
class MaClasse {
	static int nombreObjets = 0;
}
```

On peut aussi créer un bloc statique

Bout de code qui va être exécuté au moment de la création du premier objet

```
class MaClasse {
	static { nombreObjets = 0 }
}
```

On peut aussi le déclarer comme statique et final => Constante

Doit être initialisé à la déclaration ou dans un bloc statique

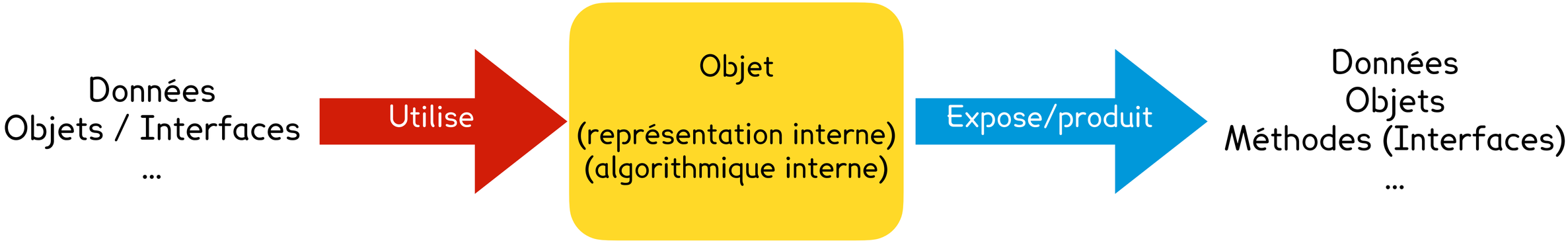
```
class MaClasse {
	final static int nombreObjets = 0;
}
```

Au final, la POO c'est quoi ?

Concevoir la solution à un problème comme :

- Un ensemble d'objets qui peuvent exister indépendamment des autres
  Approche diviser pour mieux régner
- Un travail sur les interactions entre ces objets
  Gestalt - le tout est différent de la somme des parties
  Atome –> molécules –> cellules –> organes –> être vivant

On obitent au final un ensemble d'objets qui communiquent



Connaissances & compétences - Checklist

- ☐ Créer et exécuter un programme simple en Java
- ☐ Expliquer la différence entre une classe et un type de base (`int`, `float`, …)
- ☐ Déclarer une classe
- ☐ Instancier des objets et les utiliser
- ☐ Définir ce qui doit être exposé (public, private, protected)
- ☐ Faire la différence entre une classe abstraite et une interface
- ☐ Créer une classe héritant d'une autre classe
- ☐ Déclarer qu'une classe implémente une ou plusieurs interfaces
- ☐ Utiliser le polymorphisme
- ☐ Déclarer et utiliser des attributs et des méthodes statiques

Philosophie SOLID

Principes SOLID (Robert C. Martin - Design Principles and Design Patterns, 2000)

- S => Single-responsibility

  Chaque classe, module, ou fonction a un et un seul rôle.

  On doit fournir une chose (pas de couteau suisse) et la fournir bien.

- O = Open-Closed

  On doit créer nos classes afin de favoriser l'héritage et l'abstraction (Open to extension) mais décourager la modification (Closed to modification)

  Principe pour assurer un certaine stabilité au code

- L = Liskov Substitution

  Quand on utilise un objet d'une certaine classe, on doit pouvoir utiliser un objet dérivant de la classe sans le remarquer

- I = Interface Segregation

  On doit éviter de rendre le code dépendant de méthodes qu'on n'utilise pas.

  Par exemple : si je veux un objet collection, il vaut mieux écrire "implements Collection<E>" que "ArrayList"

- D = Dependency Inversion

  Minimiser les dépendances explicites entre modules - passer plutôt par des interfaces

  Exemple : plutôt que "A dépend de B", on préférera "A fournit une fonctionnalité F et B dépend de la fonctionnalité F"

h e g
Haute école de gestion
Genève

Autres Concepts Java

Reader

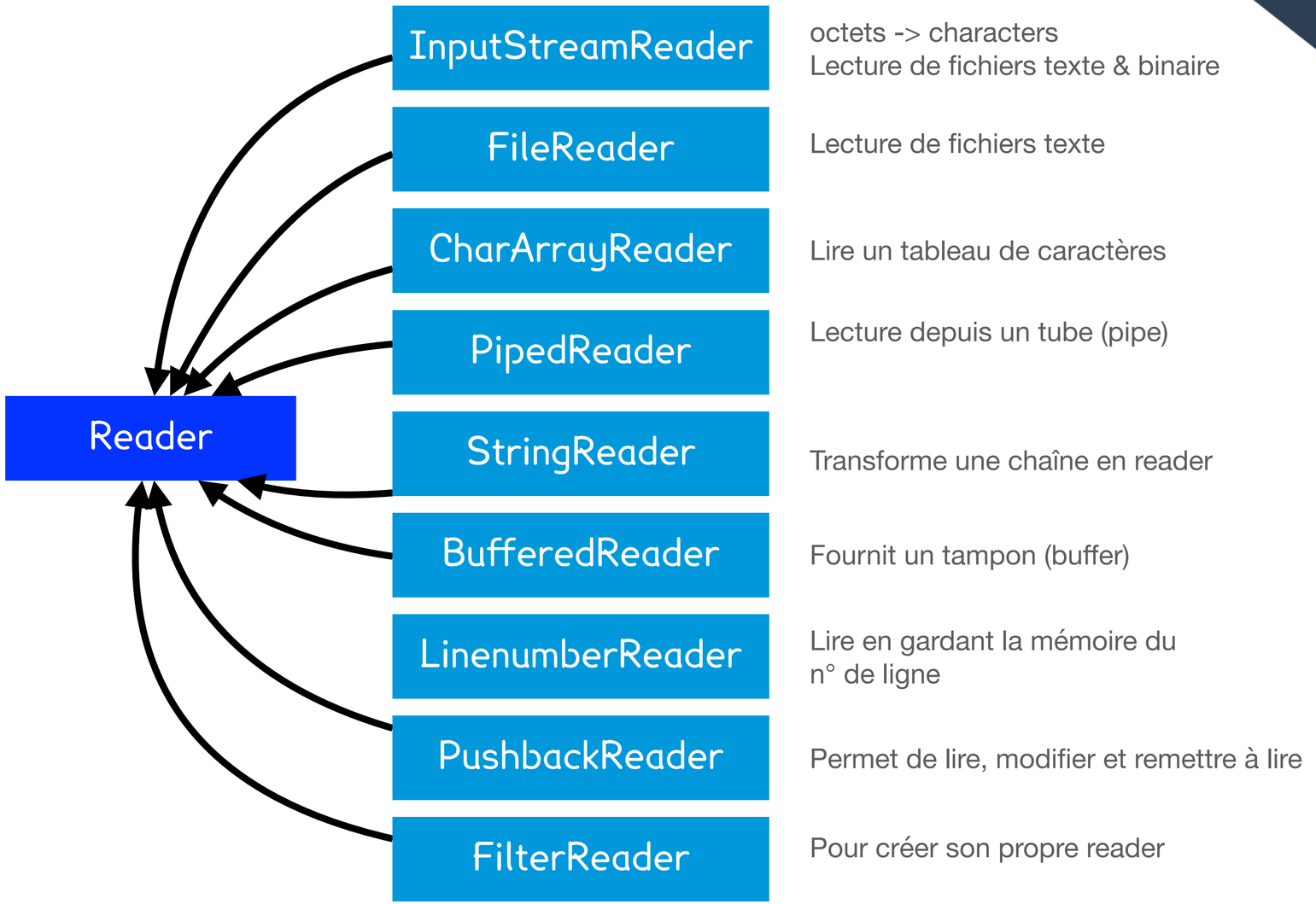
Super-classe Reader

méthode read();

Permet de lire un flux de données (liée aux streams)

Octets bruts (binaire) ou interprétation (caractères)

Source : réseau, fichier, variable, autres processus



Reader : exemple (lecture fichier texte)


```
Reader fileReader = new FileReader("monFichier.txt");

int monTexte = fileReader.read();

while(monTexte != -1) {

 // Utiliser ce qu'on a lu dans monTexte

  monTexte = fileReader.read();
}

fileReader.close();
```

```
InputStream inputStream       = new FileInputStream("monFichier.txt");
Reader      inputStreamReader = new InputStreamReader(inputStream);

int monTexte = inputStreamReader.read();

while(monTexte != -1){

    char monChar = (char) monTexte;
    // Utiliser ce qu'on a lu dans monChar

    monTexte = inputStreamReader.read();
}

inputStreamReader.close();
```

Exceptions

Une exception est une erreur d'exécution qui possède un type qui permet de comprendre ce qui se passe

On peut l'intercepter et faire autre chose que de laisser le programme s'arrêter

```
try {
   // Instructions qui peuvent provoquer une exception
} catch TypeException e {
   // Que faire si l'exception survient
}
```

A noter : les exceptions se propagent

si la méthode où elle survient ne l'intercepte pas, son exécution s'arrête et la méthode appelante recevra l'exception, etc. jusqu'au main()

Exceptions principales

Exception	Situation
ArithmeticException	Erreur de calcul (division par 0, racine d'un nombre négatif, …)
ArrayIndexOutOfBoundsException	Dépassement des limites d'un tableau (accès élément 100 sur un tableau de taille 10)
FileNotFoundException	Fichier inexistant sur le disque (erreur de chemin, …)
IOException	Echec d'une opération d'entrée/sortie
NullPointerException	Tentative d'accès à un objet dont la valeur est `null` (e.g.: `String s = null;`)
NumberFormatException	Exemple : conversion en entier d'une chaîne non entière (`Integer.parseInt ("hello");`)
RuntimeException	Exception générale
StringIndexOutOfBoundsException	Tentative d'accès à un caractère au delà de la longueur d'une chaîne

Définir sa propre exception

C'est une méthode qui va déclencher une exception, on la déclare comme suit:

```
int maMethode(…) throws ExceptionPersonnalisee {
    …
    if (…) throw new ExceptionPersonnalisee(12);
}
```

On doit, bien sûr, avoir défini notre type d'exception :

```
public class ExceptionPersonnalisee extends Exception  {
    // On peut avoir des attributs
    int exVal;

    public ExceptionPersonnalisee(int x) {
        exVal = x;
   }

   public String toString() {
        return "ExceptionPersonnalisee : valeur = " + exVal;
   }
}
```

toString() sera appelée par printStackTrace() qui provient de la classe Throwable dont hérite la classe Exception

Hiérarchie Classes Java

Toutes les classes héritent de la classe Object

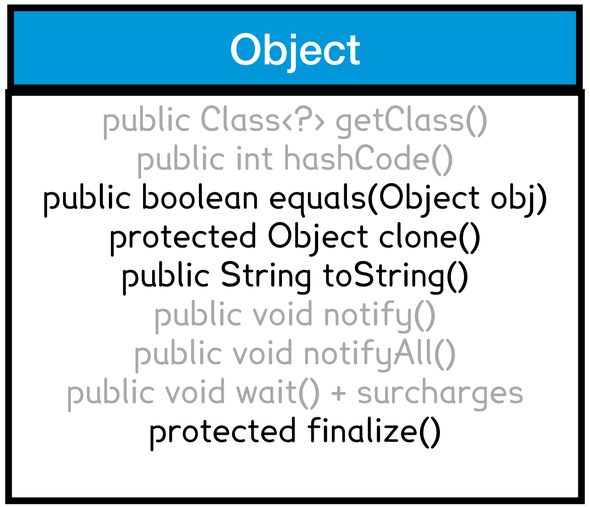
Méthodes importantes

equals() => permet de décider si deux objets (et non leurs références) sont égaux

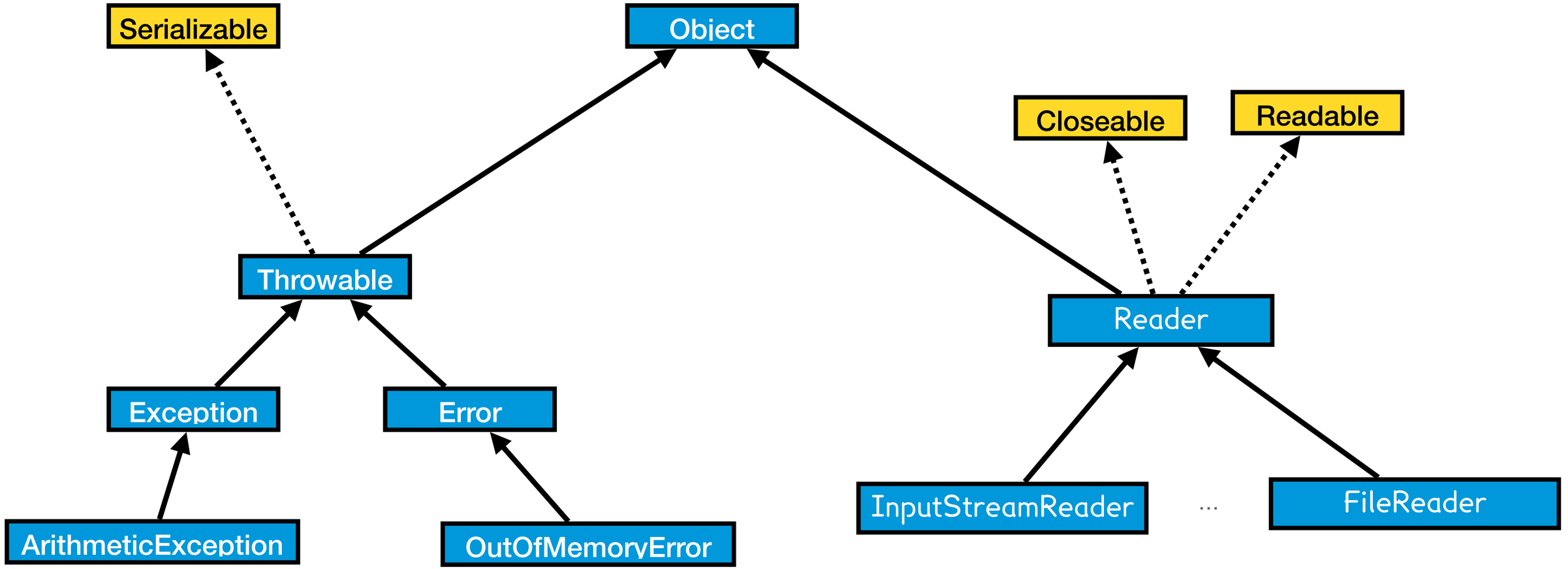
clone() => crée une copie d'un objet

toString() => renvoie une représentation texte de l'objet

finalize() => appelée quand le garbage collector supprime l'objet



Hiérarchie classes Java (petit extrait)



Quelques Interfaces à connaître

Serializable : l'objet peut être converti en un tableau d'octets afin d'être stocké, transmis, etc.
méthodes à surcharger : readObject() et writeObject()

Cloneable : l'objet peut être dupliqué (cloné)
méthodes à surcharger : clone()

Closeable : l'objet peut être fermé (libération des ressources)
méthodes à surcharger : close()

Readable : l'objet peut être lu (source de caractères)
méthodes à surcharger : int read(CharBuffer cb)

Iterable<T> : l'objet peut être parcouru avec une boucle "for each"

Collection : groupe d'objets (éléments)

Comparable<T> : il existe un ordre total sur les éléments de la classe
méthodes à surcharger : int compareTo(T o)

Comparable

L'interface Comparable s'assure qu'il existe un ordre total entre les éléments de la classe

Pour tous les éléments a et b appartenant à la classe, on peut toujours dire si a < b ; a == b ou a > b

La définition de ce que ça veut dire "être inférieur à" incombe au programmeur

Surcharge de la méthode int compareTo(T o)

Renvoie un nombre négatif si this < o

Renvoie 0 si (et seulement si) this == o

Renvoie un nombre positif si this > o

Transitif (x < y et y < z => x < z)

Permet de déclencher un tri (d'un tableau par exemple)

Collections

Collection‹E› est une interface qui est à la racine d'une bibliothèque de classes

- Groupes d'objets
- Peut être ordonné ou non
- Peut permettre d'avoir un objet en double ou non

Quelques méthodes

bool add(E e)	: ajoute (si besoin) un élément à la collection (optionnel)
bool clear()	: vide la collection (optionnel)
bool contains(Object o)	: indique si un objet fait partie de la collection
bool isEmpty()	: indique si la collection est vide ou non
Iterator‹E› iterator()	: renvoie un itérateur sur les éléments de la collection
bool remove(Object o)	: supprime un objet de la collection (optionnel)
int size()	: renvoie le nombre d'élément présents dans la collection
Object[] toArray()	: renvoie un tableau contenant tous les éléments de la collection

Collections - Hiérarchie (très) partielle