Java Initiation

Arnaud Delafont

08/07/19

Plan

- Présentation
- Syntaxe du langage
- Programmation Orientée Objet
- Agrégation et encapsulation
- Héritage, polymorphisme et interface
- Classes essentielles
- Exceptions
- Les collections
- Entrées/Sorties

Présentation

Historique

- 90th : Sun Microsystems & James Gosling
- Première version : langage OAK
- 1995 : Lancement public de Java
- 2000 : version 1.2 avec JavaSE et JavaEE
- 2006 : Java devient open source
- 2009 : Java est racheté par Oracle

Caractéristiques

- Simple, orienté objet et familier
- Interprété
- Portable et indépendant des plates-formes
- Robuste et sûr
- Distribué
- Dynamique et multithread

Plateforme java

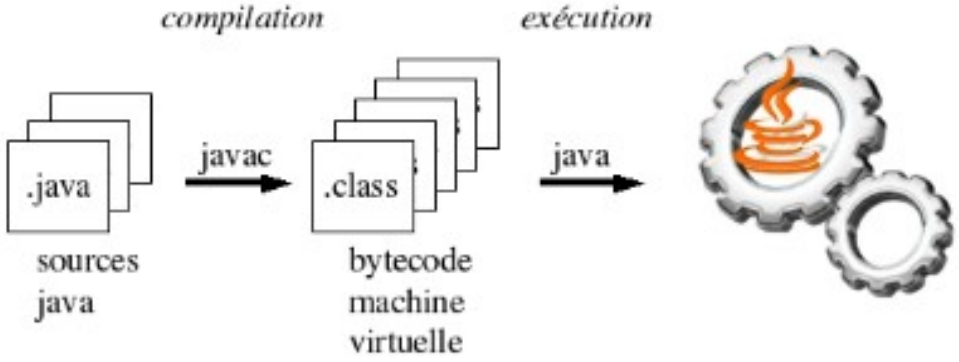
- **Java SE** : Java Standard Édition
- **Java EE** : Java Entreprise Édition
  ajoute des API pour écrire des applications installées sur les serveurs dans des applications distribuées : Servlet, JSP…
- **Java ME** : Java Micro Édition
  destiné aux applications pour systèmes embarqués et mobiles (microcontrôleur, capteur, passerelle, smartphone, assistant personnel numérique, décodeur TV, imprimante)
- L'évolution de java est gérée par le **JCP** (Java Community Process) qui émet des **JSR** (Java Specification Requests). elles décrivent les spécifications et les technologies proposées pour un ajout à la plateforme Java

Développement Java

Écrire un programme Java ne nécessite pas d'outils sophistiqués :

- un éditeur de texte tel notepad suffit
- un JDK (Java Development Kit)

  (http://java.sun.com/javase/downloads/index.jsp)



Kit de développement Java

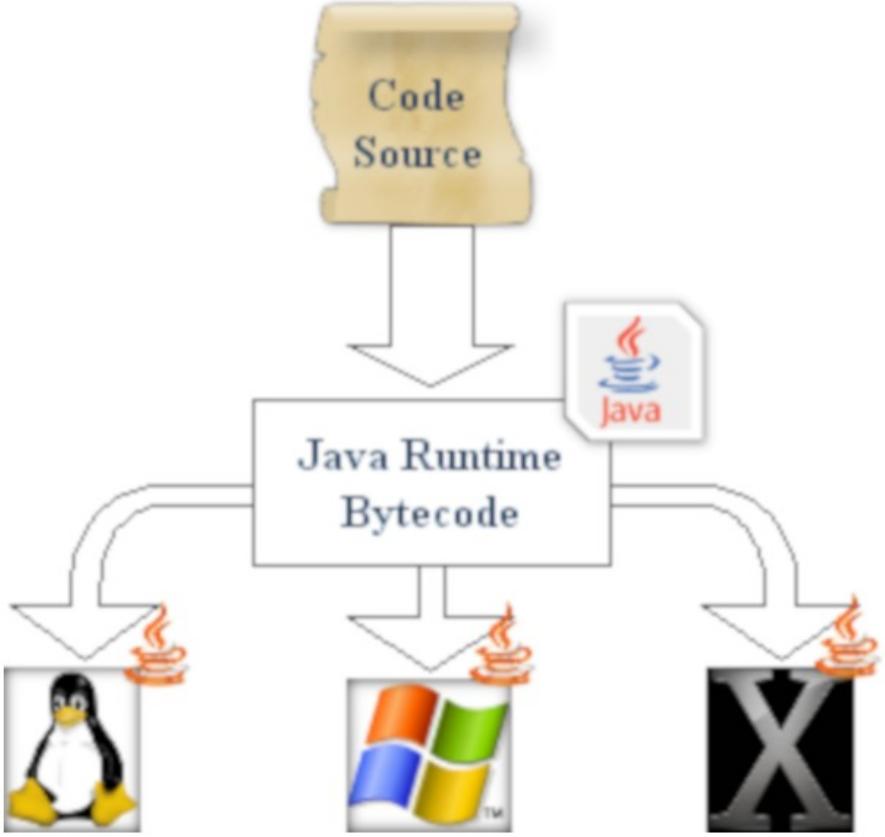
Ce kit de développement comprend de nombreux outils :

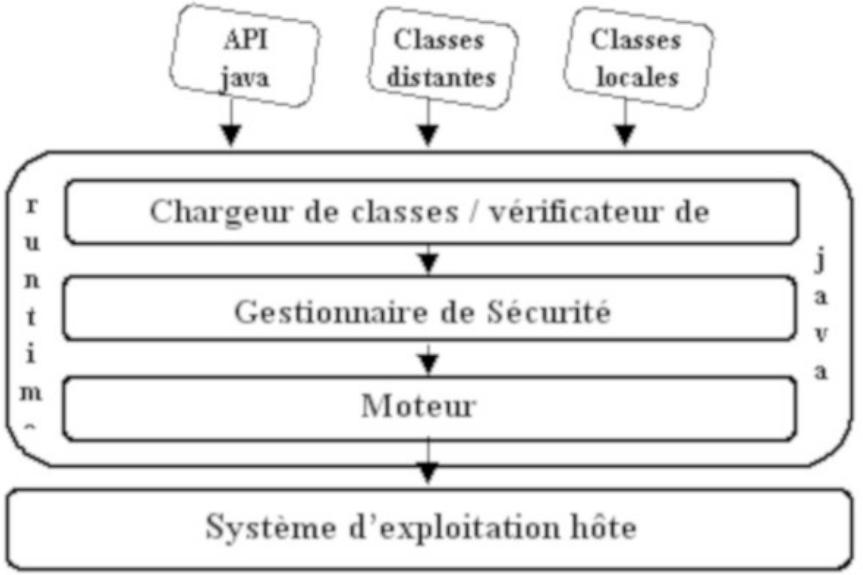
- le compilateur : **javac**
- l'interpréteur d'application : **java**
- le débogueur : **jdb**
- le générateur de documentation : **javadoc**
- le désassembleur : **javap**

Packages Java (API)

java.lang	Classes fondamentales
java.io	Entrées/sorties, gestion des flux, sérialization...
java.util	Collections framework
java.awt	Abstract Windowing Toolkit (IHM).
java.security	Framework de Sécurité
java.net	Applications Réseau
javax.swing	Composants Graphiques
java.text	Textes, dates, nombres et messages
java.sql	Accès aux données stockées dans des Bdd

Java Virtual Machine





Environnements de développement

- Hormis l'outil de base de développement (JDK), il existe des environnements de développement intégrés (IDE) :
  - Eclipse (eclipse.org)
  - NetBeans (netBeans.org)
  - IntelliJ Idea (jetbrains.com/idea/)

Première Application Java

- Installation d'un JDK
- Paramétrages des variables d'environnement
  JAVA_HOME= C:\Program Files\Java\jdk1.8.0_201
  PATH=%JAVA_HOME %\bin
- Écriture et exécution du programme en Java : HelloWorld.java
  - Minuscules / majuscules différentiées
  - Espaces / CR / LF / Tabulations sans conséquences

```
public class HelloWorld{
   public static void main(String [] args){
      System.out.println("Hello world") ;
   }
}
```

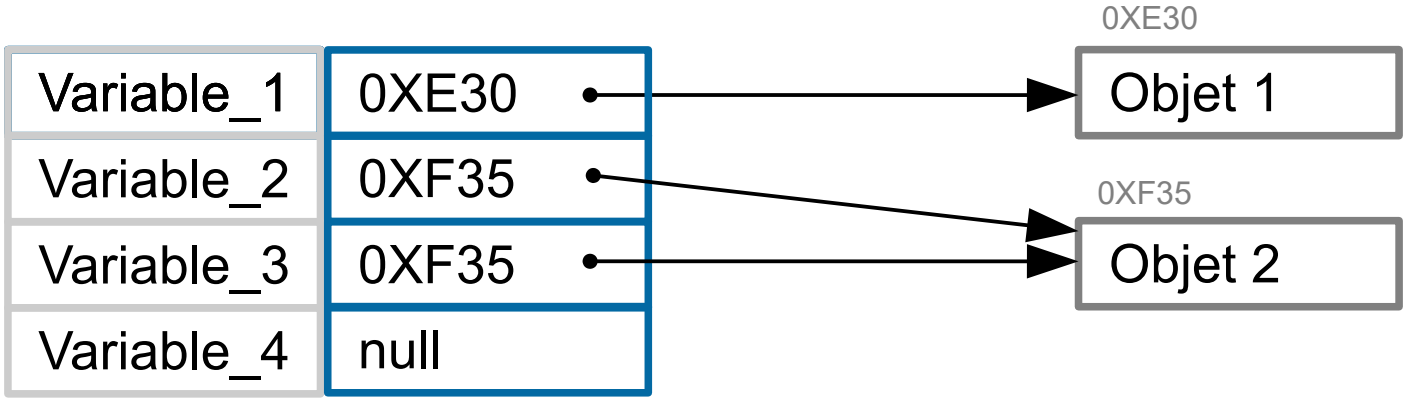
Syntaxe du langage

Types de données

- Types primitifs

Variable_1	45
Variable_2	true
Variable_3	c
Variable_4	56.5

- Types références



Types primitifs

Type	Valeurs	Portée	Exemple
boolean	true , false		true
byte	entier signé sur **8** bits	-127 à 128	123
short	entier signé sur **16** bits	-2^{15} à $2^{15}-1$	123
int	entier signé sur **32** bits	-2^{31} à $2^{31}-1$	123
long	entier signé sur **64** bits	-2^{63} à $2^{63}-1$	123**L**
float	réel signé sur **32** bits	$\{-3,4028234^{38}..3,4028234^{38}\}$ $\{-1,40239846^{-45}..1,40239846^{-45}\}$	4.23**f**
double	réel signé sur **64** bits	$\{1,797693134^{308}..1,797693134^{308}\}$ $\{-4,94065645^{-324}..4,94065645^{-324}\}$	4.23 5.34e3
char	caractère unicode sur **16** bits	\u0000 à \FFFF	‘a’

Variables

- Déclaration de variable

```
double valeur;
boolean tst;
int i, j;
char c ;
```

- Initialisation de variable

```
int hauteur;
hauteur = 10;
double t1 = 1.25, t2 = 1.26;
boolean test = true;
char c = 'a';
```

Règles de nommages

- Le nom doit commencer par : **une lettre, _ ou $**
- Les **nombres** sont autorisés **sauf en tête**
- Ne doit **pas** être un **mot réservé**

Correct :	identifier	conv2Int	_test	__test2
Faux :	3dPoint	*$coffe	public	while

- Pour java: minuscule ≠ MAJUSCULE

  Valeur ≠ valeur

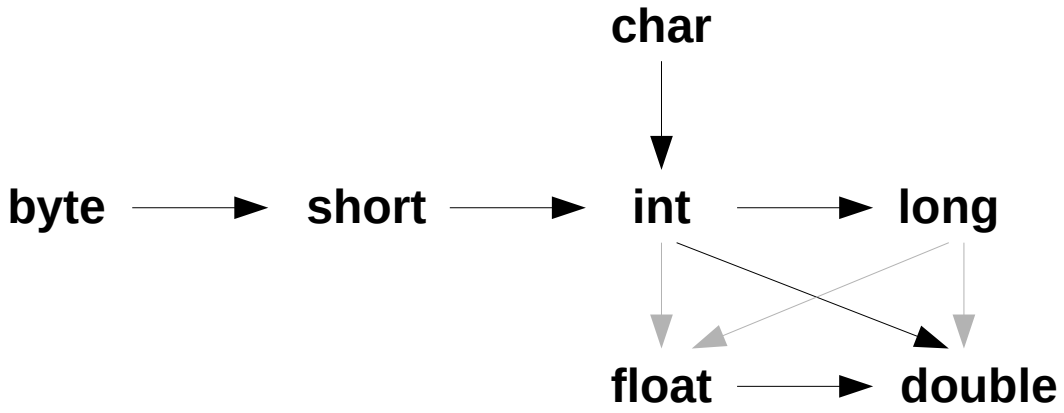
- Par convention les variables utilisent le **c**amel**C**ase

```
int nombreDeVisiteur, anneeNaissance
```

Changement de type (typecasting)

Transtypage implicite (automatique)

- type inférieur vers un type supérieur
- entier vers un réel



Avec perte de précision

```
int i = 42;
double d = i;
```

Changement de type (typecasting)

Transtypage explicite (cast)

- type supérieur vers un type inférieur
- réel vers un entier

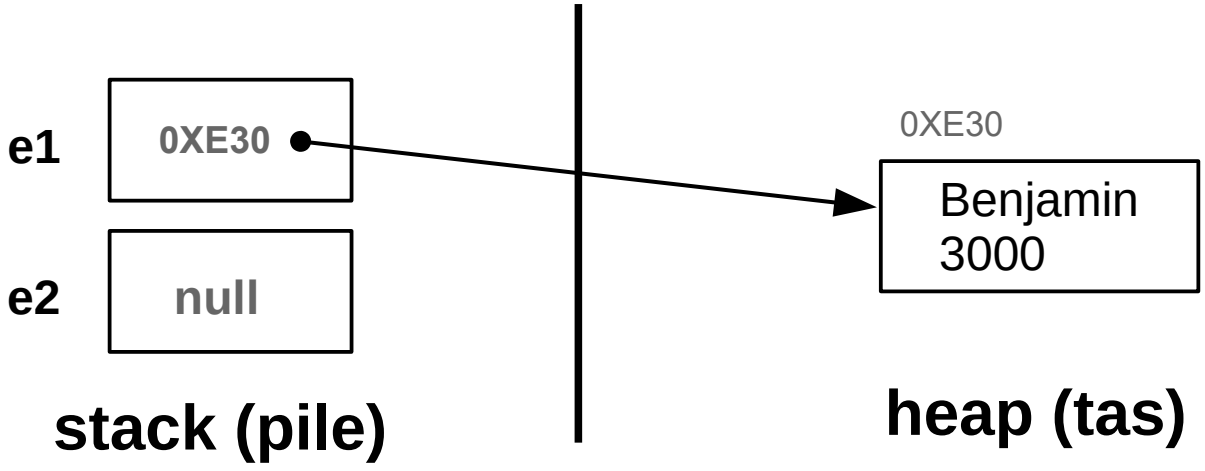
```
int i = 123;
short s = (short)i;

double d = 44.95;
int i = (int)d;

// Attention !!
int i = 130;
byte b = (byte)i;  // b vaut -126
```

Objets

Il existe des types complexes :

- String (chaîne de caractère)
- Integer (encapsulation d'un entier)
- Tableau
- ...

```
Employe e1= new Employe("Benjamin",3000);
Employe e2= null ;
```



Wrappers (types enveloppes)

Les **wrappers** sont des objets identifiants des variables primitives

Type primitif	Wrapper
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

```
char c = 'a';
Character ch =new Character(c);
```

```
int i = 3;
Integer iW = new Integer(i);
int k = Integer.parseInt("10");
Double dW= Double.valueOf("10.5");

double d=iW.doubleValue();
i=iW.intValue();
String s=Integer.toHexString(iW);
```

Opérateurs

- Opérateurs mathématiques : **- + * / %**
- Opérateurs d'incrémentations : **-- ++**
- Opérateurs d'affectations : **= += -= *= /= %= &=**
- Opérateurs de comparaisons : **== != < > <=**

  instanceof
- Opérateurs logiques : **! && ||**
- Opérateurs binaires : **~ & ^ |**
- Opérateurs de décalage : **>> << >>>**

Promotion numérique

La promotion numérique permet de rendre compatible le type des opérandes avant qu'une opération arithmétique ne soit effectuée.

- Le type le + petit est promu vers le + grand type des deux
- Si une variable est entière et une autre en virgule flottante, la valeur entière est promu en virgule flottante
- **byte**, **short**,**char** sont promus en **int** à chaque fois qu'ils sont utilisé avec un opérateur
- Après une promotion le résultat aura le même type.

Méthodes

- Déclaration d'une méthode

type_de_retour nomDeLaMethode(arguments) {

instructions

}

```
int somme(int a, int b){
   return a+b ;
}
```

- Appel d'une méthode

```
int a=somme(3,45);
somme(2,3);
```

Méthodes

- Si aucune valeur n'est renvoyée, on utilise le pseudo-type **void**

```
void setSalaire(double salaire){
    ...
}
```

- Il peut y avoir plusieurs **return** dans une méthode
- Les arguments d'une méthode sont passés **par valeur**
- Les règles de nommage sont les même que pour les variables

Instructions

- se terminent par un ;
- peut contenir : une affectation, une déclaration ou un appel de méthode…

```
int x = 10;                 // Déclaration et affectation
x = (x+30)/2;               // Opération
setName("John");            // Appel de méthode
```

- bloc d'instruction

```
{
    instructions
}
```

- Les blocs d'instruction définissent la portée des identificateurs. Elle commence où l'identificateur est déclaré et va jusqu'a la fin du bloc dans lequel il est défini.

Conditions

- **if**

```
if (condition) {
      // bloc d'instructions1 (condition vrai)
} else {
      // bloc d'instructions2 (condition fausse)
}
```

- **opérateur ternaire**

```
condition ? condition_vraie : condition_fausse ;
```

```
int j = (i > 2) ? i : 0;
//j=i si i est supérieur à 2 sinon j=0
```

Conditions

- **switch**

```
int jours = 7;
switch (jours) {
    case 1:
            System.out.println("Lundi");
            break;
    case 6:
    case 7:
            System.out.println("week end !");
            break;
    default:
            System.out.println("autre jour");
}
```

- La condition peut être de type:
  byte, Byte, short,Short,char, Character, int, Integer, String et enum
- la valeur de **case** doit avoir une **valeur constante**
- **default** n'est pas obligatoire

Boucles

- **while**
  Tant que la condition est vérifiée, le bloc d'instructions est exécuté

```
while(condition){
   instructions
}
```

```
int i = 10, somme = 0, j = 0;
while (j <= i) {
   somme = somme + j;
   j = j + 1;
}
System.out.println("Somme= " + somme);
```

Boucles

- **do while**

  identique à while, sauf que le test est réalisé après l'exécution des instructions

```
do {
   instructions
}
while(condition)
```

```
int i = 10; int j = 0; int somme = 0;
do {
   somme = somme + j;
   j = j + 1;
} while (j <= i);
System.out.println("Somme = " + somme) ;
```

Boucles

- **for**

```
for (initialisation ; condition ; opération){
   instructions
}
```

1) **initialisation** est exécuté
2) si la **condition** est fausse ➡ on sort de la boucle
3) le bloc d'**instruction** est exécuté
4) **opération** est exécutée
5) retour à l'étape 2

```
for(int i = 0; i < 10; i = i + 1){
   maMethode(i);
   System.out.println(i) ;
}
```

Instructions liées aux boucles

- **break** permet de sortir de la boucle et continu l'exécution après le bloc d'instruction
- **continue** force l'exécution à l'itération suivante
- **break** et **continue** peuvent être suivis d'un nom d'étiquette : **LABEL :**

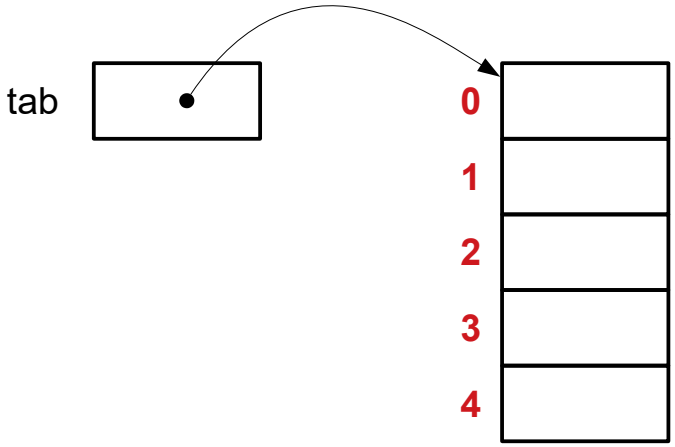
```
PARENT_LOOP: for(...){
    for(...){
        if(...){
            ...
            break PARENT_LOOP;
        }
    }
}
```

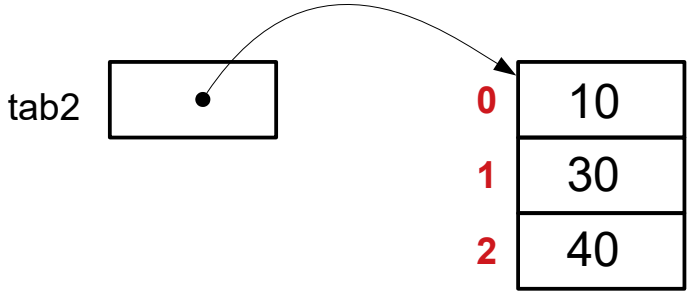
Tableaux

- **Déclaration d'un tableau**

```
type[] nom = new type [taille] ;
type[] nom = {valeur1, valeur2,..} ;
```

```
int [] tab= new int[5];
int [] tab2= {10,30,40};
```





type[] tab équivaut à type tab[]

Tableaux

- **Valeur d'un élément du tableau**

```
nom [indice]
```

l'indice d'un tableau commence à **0**

```
int [] tab= {10,30,40};
System.out.println(tab[0]);   // affiche 10
```

- **Taille d'un tableau : length**

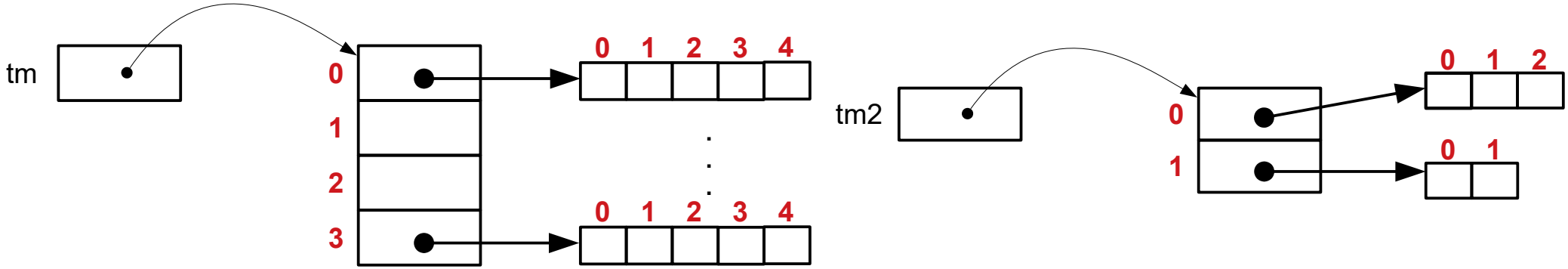
```
int [] tab= new int[20];
int n = tab.length;      // n a pour valeur 20
```

Tableaux

- **Tableau à deux dimensions**

```
type[][] nom = new type [taille][];
nom[0] = new type [taille2];
```

```
int [][] tm= new int[4][5];
int [][] tm2 = new int [2][];
tm2[0]=new int [3];
tm2[1]=new int [2];
int [][] tm3= {{1,4},{3},{9,8,6}};
System.out.println(tm3[0][1]);   // affiche 4
```



Tableaux

- **Itération complète**

```
for(type instance : tableau) {
    ...
}
```

```
int [] tab= {2,4,6};
for(int val : tab){
    System.out.println(val) ;
}
```

```
Affiche :
2
4
6
```

Méthode main

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

- Point d'entrée du programme
- Doit être statique
- Peut recevoir des paramètres depuis une ligne de commande

```
java MonJavaProgram I am 35
```

```
args[0]➡ MonJavaProgram
args[1]➡ I
args[2]➡ am
args[3]➡ 35
```

Commentaires

- sur une ligne

```
// commentaire sur une ligne
```

- sur plusieurs ligne

```
/* commentaire sur plusieurs lignes
   suite du commentaire
*/
```

- documentation automatique avec javadoc

```
/**
  * Cette méthode calcule
  * @author James Gosling
 */
```

@author	Nom du développeur
@param	Définit un paramètre de méthode
@return	Documente la valeur de retour
@version	Donne la version d'une classe ou d'une méthode.

Exercices

- Déclarer un tableau de 10 nombres.
- Écrire des méthodes qui calculent le minimum, le maximum, la moyenne des valeurs de ce tableau.
- Bonus : Écrire une méthode pour trier le tableau dans l'ordre croissant.
- Tester ces méthodes.

Programmation Orientée Objet

Définition

L'orienté-objet = approche de résolution algorithmique de problèmes permettant de produire des programmes modulaires de qualité.

Objectifs :

- Développer une partie d'un programme sans qu'il soit nécessaire de connaître les détails internes aux autres parties
- Apporter des modifications locales à un module, sans que cela affecte le reste du programme
- Réutiliser des fragments de code développés dans un cadre différent

Qu'est ce qu'un objet ?

Objet = élément identifiable du monde réel

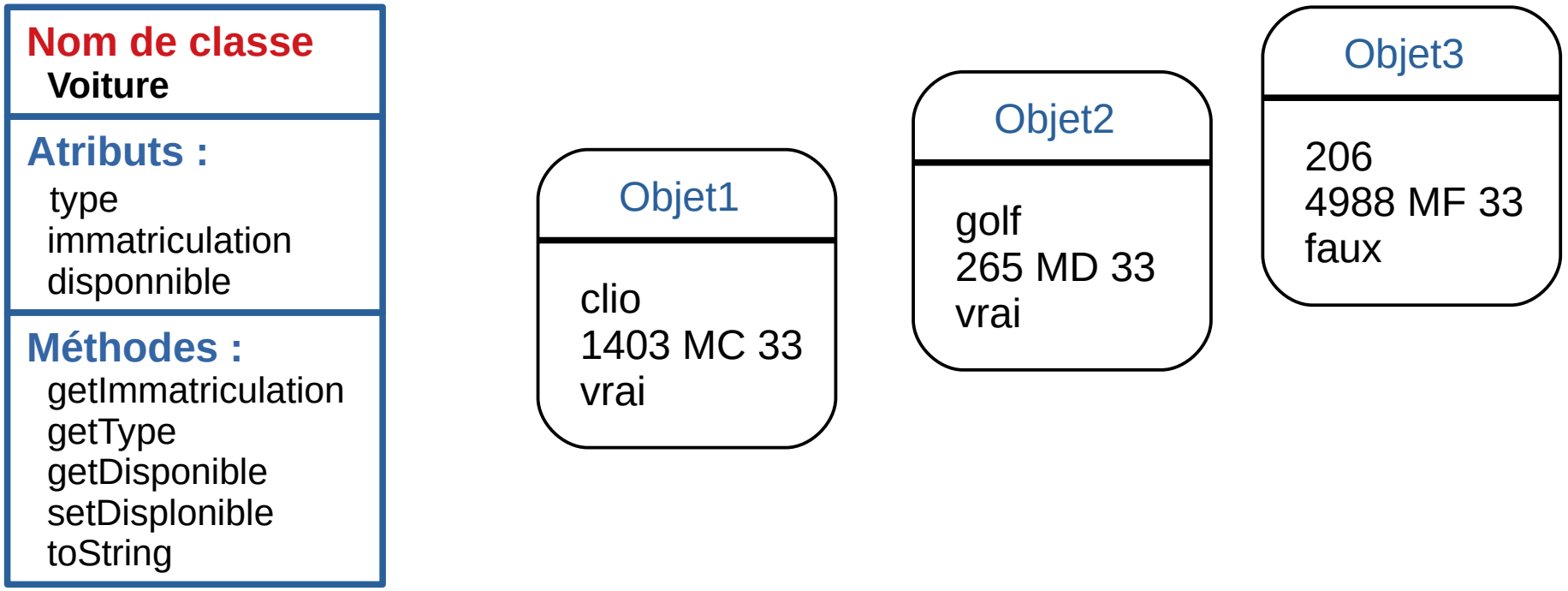
- concret (voiture, stylo,...),
- abstrait (entreprise, temps,…)

Un objet est caractérisé par :

- **son identité**
- **son état** ➞ les données de l'objet
- **son comportement** ➞ ce qu'il sait faire

Qu'est-ce qu'une Classe ?

- Une classe est un type de structure ayant :
  - des attributs
  - des méthodes
- On peut construire plusieurs **instances** d'une classe



Déclaration d'une Classe

```
public class Voiture {
    // Attributs
    …
    // Méthodes
    …
}
```

Fichier : Voiture.java

- Les règles de nommages sont les mêmes que pour les variables
- Le nom d'une classe commence toujours par une majuscule
- Une classe public par fichier, le nom du fichier est celui de cette classe.

Variables d'instance

- Définissent l'état de l'objet :
- On les appelle également **attributs**.
- La valeur d'un attribut est propre à chaque instance.

```
public class Voiture {
    String marque;
    String plaque;
    String couleur;
    …
}
```

- Accès au attribut :
  instance.attribut

```
clio.couleur;
```

Initialisation des variables d'instance

- Si elle n'est pas initialisée, elle prend une valeur par défaut

Type	Valeur par défaut
boolean	false
byte, short, int, long	0
float, double	0.0
char	\0000
référence	null

```
public class Voiture {
    String marque;              //marque a pour valeur null
    int nbKilometre=1000;
    …
}
```

Variables locales

- Variables temporaires qui existent seulement pendant l'exécution de la méthode.

```
public class MaClasse {
    // …
    public void maMethode() {
        int monNombre = 10;
    }

    public void maMethode2(){
        System.out.println(monNombre);
        // Erreur de compilation
    }
}
```

- Les variables locales n'ont pas de valeur par défaut et doivent être initialisées

Méthodes d'instances

- Bloc d'instructions définissant un comportement d'une instance.
  - déclarées à l'intérieur d'une classe.
  - peuvent être surchargées (même nom, différents paramètres, …)

```
public class MaClasse {
    public void maMethode() {
    // les actions
    }
}
```

- L'appel d'une méthode :

  instance.methode();

```
clio.déplacer();
```

Constructeur

- Le constructeur est une méthode spéciale dans la classe appelée à la création d'instances.

- Un constructeur:
  - porte le nom de la classe
  - n'a pas de type de retour

```
public class Voiture {
    public Voiture() {
        // Corps du constructeur
    }
…
}
```

Constructeur multiple

Il est possible de déclarer plusieurs constructeurs différents pour une même classe afin de permettre plusieurs manières d'initialiser un objet.
Les constructeurs diffèrent alors par leur signature.

```
public class Voiture {
    // Contructeur par défaut sans paramètres
    public Voiture() {
        // Corps du contructeur
    }
    // Contructeur avec un paramètre
    public Voiture(String couleur) {
        // Corps du contructeur
    }
    …
}
```

Constructeur par défaut

```
public Classe() { }
```

- Lorsqu'une classe ne comporte pas de constructeur, java ajoute automatiquement un constructeur par défaut

- Si un constructeur est ajouté à la classe, java n'ajoute plus automatiquement de constructeur par défaut. Il devra être ajouté explicitement

Cycle de vie d'un objet

- Instanciation d'un objet : new

```
String str=new String("hello wolrd") ;
```

- Garbage collector
  - Travail en arrière plan
  - Libère la mémoire des instances non référencées
  - Intervient lorsque le système a besoin de mémoire ou de temps en temps (priorité faible)
- Méthode **finalize**() peut-être appelée (0 ou une fois), si le garbage collector essaye de collecter l'objet. Déprécié dans les futures versions de java

Le mot clé this

- Fait référence à l'objet en cours
- On peut l'utiliser pour :
  - Manipuler l'objet en cours

```
maMethode(this);
```

  - Faire référence à une variable d'instance

```
this.maVariable ;
```

  - Faire appel au constructeur propre de la classe

```
this("w44","BMW");
```

Exercices

- Créez une classe Voiture.
- Ajoutez un constructeur avec comme paramètres sa marque et sa plaque d'immatriculation.
- Ajoutez un constructeur avec comme paramètres sa marque, sa plaque d'immatriculation et sa couleur.
- Écrivez une méthode principale permettant de créer deux objets avec les différents constructeurs.

Variables de classes

Variables partagées par toutes les instances de classe.

- Variables déclarées avec le mot clé **static**
- Pas besoin d'instancier la classe pour utiliser ses variables statiques (**static**).
- Chaque objet détient la même valeur de cette variable.

```
public class Voiture {
   String type;
   static int nbVoitures;
   …
}
```

- L'appel de ses variables : Classe.variableStatic;

```
Voiture.nbVoiture ;
```

Méthodes de classes

Bloc d'instructions définissant un comportement global ou un service particulier.

- Déclarées avec le mot clé **static**
- Peuvent être surchargées (même nom, ≠ paramètres).

```
public class MaClasse {
   // …
   public static void maMethode() {
      // les actions
   }
}
```

- N'utilisent pas de variables d'instances parce qu'elles doivent être appelées depuis la classe.
- L'appel de ses méthodes : **MaClasse.maMethode;**

Méthode principale (main)

- La méthode main représente le point d'entrée d'une application en exécution
- Elle peut être intégrée dans une classe existante ou écrite dans une classe séparée.

```
public class Launch {
   // Méthode principale
   public static void main (String [] args) {
   // Création d'une instance de la classe Voiture
      Voiture Clio = new Voiture();
   }
}
```

Exercices

- Ajout d'une variable statique « Nombre de voitures » à la classe Voiture.
- Cette variable devra être incrémentée à chaque instanciation de la classe
- Utilisation du mot clé this à l'intérieur du constructeur pour changer la valeur de ses attributs
- Écrivez la méthode permettant de connaître le nombre de voitures

Packages
ou Espaces de noms

Package = groupement de classes qui traitent un même problème pour former des « bibliothèques de classes »

- Une classe appartient à un package s'il existe une ligne au début renseignant cette option :

```
package nompackage;
```

- Pour utiliser une classe (au choix) :
  - Être dans le même package
  - Préfixer par le package (à chaque utilisation)

```
package.Class variable ;
```

  - Au début du fichier, importer la classe, ou le package entier *

```
import nompackage.nomClasse;
import nompackage.*;
```

Les conventions

- Indentation significative (tabulation),
  80 caractères par ligne, opérateur en début

- Accolades : ouvrantes en fin de ligne, fermantes isolées

```
public Voiture() {
        …
}
```

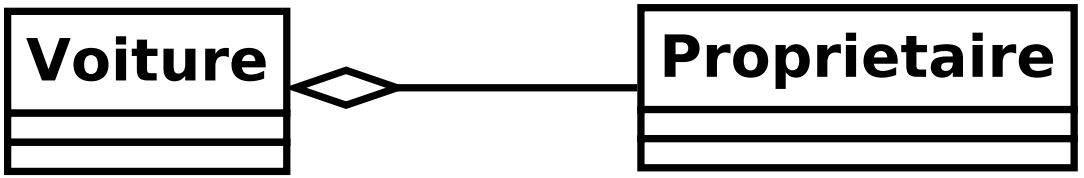
- Ordre de déclaration : attributs : statiques → d'instance → constructeurs → méthodes : public → protégé → privé

- Intitulés : un.package, UneClasse, UneInterface, uneMethode, uneVariable, unAttribut, UNE_CONSTANTE

Agrégation et encapsulation

Agrégation

- Agrégation = associer un objet avec un autre
  ex : Objet Propriétaire à l'intérieur de la classe Voiture




```
public class Proprietaire{
    …
}

Public class Voiture() {
    Proprietaire owner ;
    …
}
```

Accessibilité

Accessibilité : utilisation de facteurs de visibilité

- **public:** accessible par toutes les classes
- **protected**: accessible par toutes les sous-classes et les classes du même package
- **rien ou default**: accessible seulement par les classes du même package.
- **private**: accessible seulement dans la classe elle-même

```
public class Point{
    private int x,y ;

    public Point(int x, int y){
        …
    }
}
```

Encapsulation

Encapsulation =

- Regroupement de code et de données.
- Masquage d'information par l'utilisation d'accesseurs (getters et les setters) afin d'ajouter du contrôle

L'encapsulation permet de restreindre les accès aux membres d'un objet, obligeant ainsi l'utilisation des membres exposés.

Qu'est-ce qu'un JavaBean ?

- Un type de classe ayant :
  - un constructeur public sans arguments (et éventuellement d'autres constructeurs)
  - Des getters et setters pour chaque attribut
  - Un moyen de sérialisation (généralement, implémente java.io.Serializable)

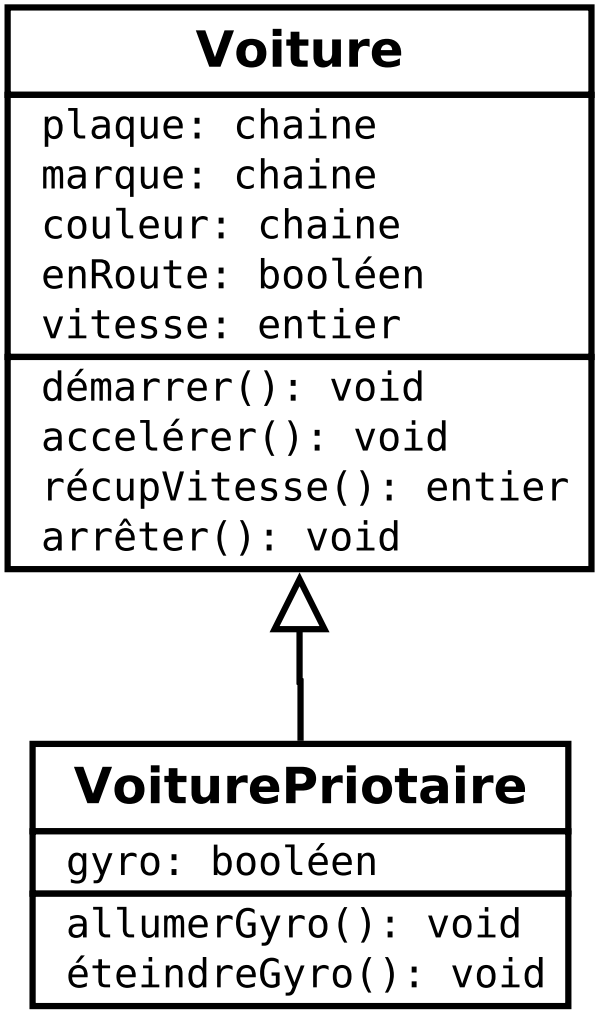
Exercices

- Implémentation d'une agrégation :
- Voiture – Propriétaire - Adresse
- Ajout de getters et setters.
- Création d'instances de la classe Voiture

Héritage, polymorphisme et interface

Héritage

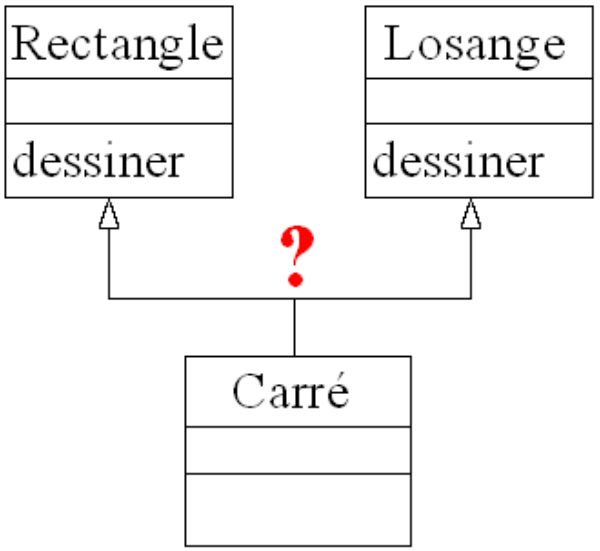
- L'héritage permet de créer la structure d'une classe à partir des membres d'une autre classe
- La sous-classe hérite de tous les attributs et méthodes de sa classe mère.



Héritage multiple

L'héritage multiple permet a une classe d'heriter simultanément de plusieurs autres classes.

Problème :



Non disponible dans Java

L'héritage multiple peut être partiellement comblé par une approche d'héritage en cascade des classes.

Héritage en Java

- Utilisation du mot clé extends

```
public class VoiturePrioritaire extends Voiture{
private boolean gyrode;
    // les actions
}
```

- Utilisation du mot clé super (référence à la classe mère)

```
private void demarrer() {
    gyrode = true;
    super.demarrer();
}
```

Surcharge et redéfinition

- La **redéfinition (overriding)** consiste à réimplémenter une version spécialisée d'une méthode héritée d'une classe mère (les signatures des méthodes dans la classe mère et la classe fille doivent être identiques)

- La **surcharge (overloading)** consiste à proposer, au sein d'une même classe, **plusieurs « versions » d'une même méthode** qui diffèrent simplement par le nombre et le type des arguments (le nom et le type de retour doivent être identiques).

final

- Le mot clé final permet :
  - d'interdire l'héritage à partir de la classe.
  - d'interdire la redéfinition d'une méthode
  - de déclarer une constante

```
public final class C1 { …
}

public final void M1() { …
}

public static final int X = 3;
```

Exercices

- Modifiez la classe VoiturePrioritaire afin que la variable « Nombre de voitures » renvoie le nombre de voitures prioritaires
- Surchargez la méthode démarrer() de la classe Voiture
- Écrivez la méthode permettant de savoir si le véhicule est prioritaire (en marche et gyrode allumé)

Polymorphisme

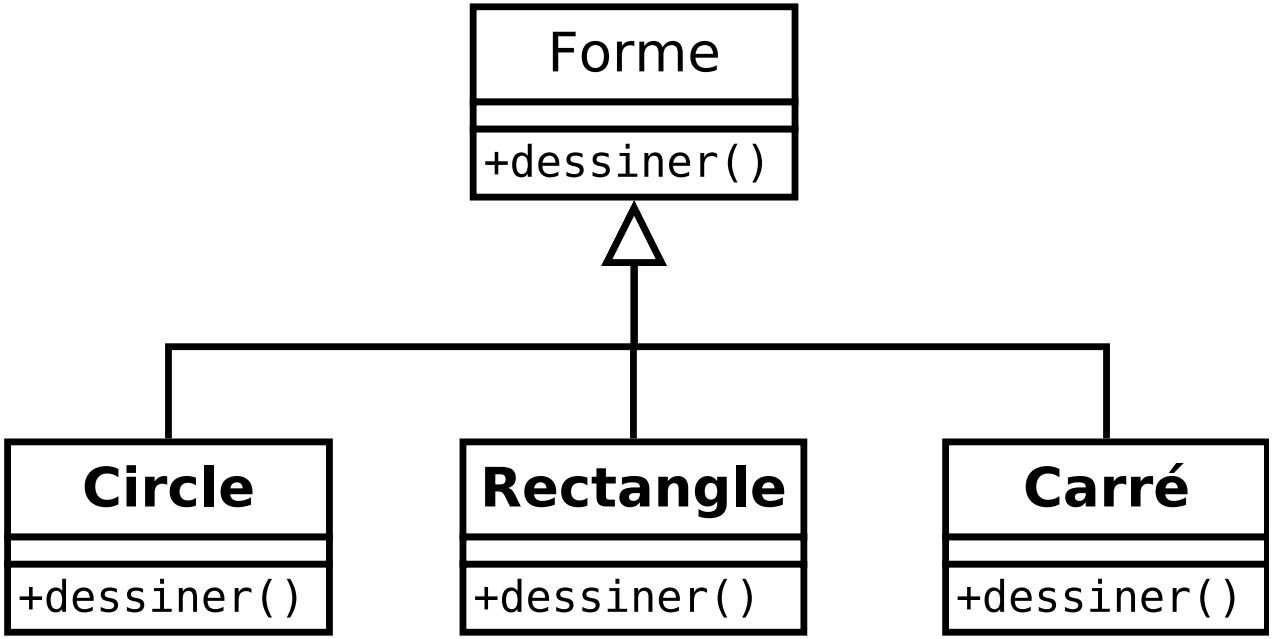
Le polymorphisme est la propriété d'une entité de pouvoir se présenter sous diverses formes. Ce mécanisme permet de faire collaborer des objets entre eux sans que ces derniers aient déclarés leur type exact.

Exemples :

- On peut avoir une voiture prioritaire avec le type Voiture
- On peut créer un tableau de Voitures et placer à l'intérieur des objets de type Voiture et d'autres de type VoiturePrioritaire

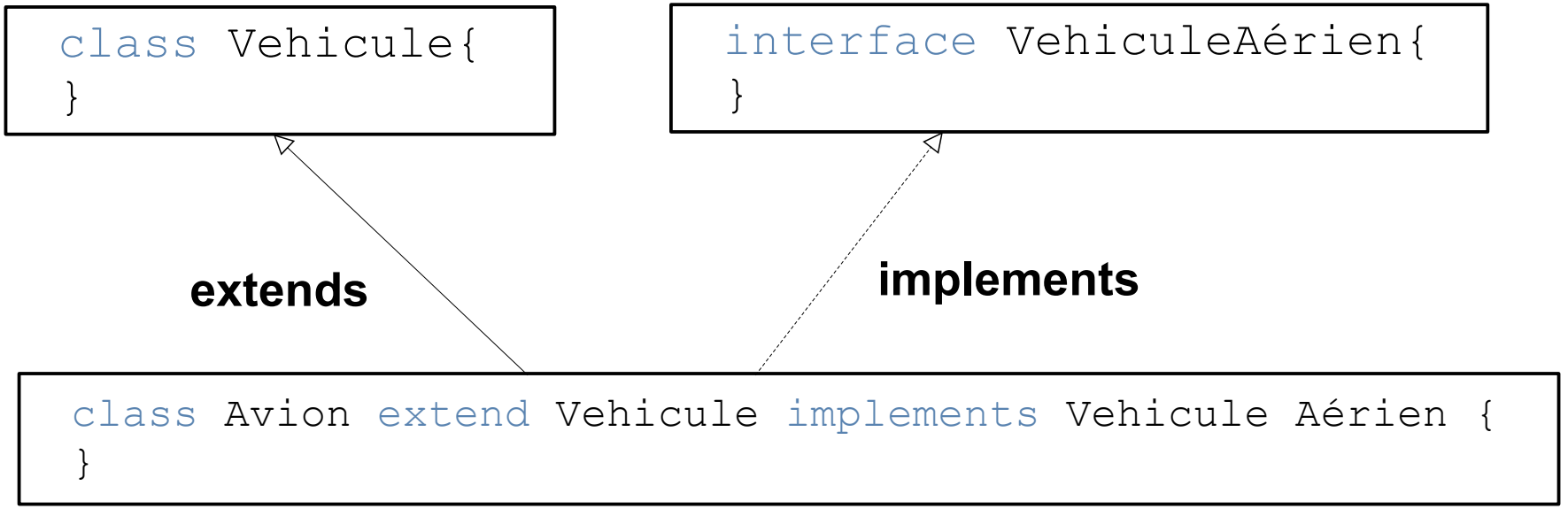
Classe Abstraite

- Une classe qui **ne peut être instanciée**.
  - Définit un type de squelette pour les sous-classes
    - Si elle contient des méthodes abstraites, les sous-classes doivent implémenter le corps des méthodes abstraites.
  - Déclarée avec le mot clé **abstract**.



Interfaces

- Une pseudo classe abstraite marquée par le mot clé **interface** contenant **juste** des **signatures de méthodes** (et des constantes) afin de montrer les capacités



Exercices

- Définissez une classe abstraite, dont va hériter la classe Voiture
- Ajout d'une méthode abstraite à la classe nouvellement créée
- Implémentez la méthode dans la classe Voiture
- Définissez une interface pour la classe Voiture

Classes essentielles

Object

- Toutes les classes héritent implicitement de la classe Object
- **toString()→** pour représenter un objet sous forme d'une chaine de carctère, par défaut retourne : **nomDeLaClasse@hashcode**
- **clone()→** pour dupliquer un objet, interdit par défaut
  pour l'utiliser,il faut :
  - Surcharger la méthode clone() par une méthode public
  - La classe implémente l'interface clonable
- **equals(Object obj)→** permet de comparer deux objets
  par défaut, retourne vrai, si this a la même référence que obj,
  elle doit être redéfinit.
- **hashcode()→** calcul un code numérique pour l'objet, par défaut l'adresse mémoire de l'objet.
  si equals est redéfinit, hashcode doit aussi être redéfinit

Package java.lang

- Le package java.lang est importé automatiquement

- Contient les services que l'on retrouve éparpillés dans les langage proceduraux :
  - Wrappers (Integer, Double...)
  - Exception/RuntimeException/Error
  - Chaînes de caractères
  - Divers (Class, Math, Thread, System...)

String

- Chaîne non mutable de caractères unicodes ➞Une fois créé, elles ne sont plus modifiables
  String nom="test" stocké dans le StringPool
  String nom= new String("test") stocké dans le tas
- **length** ➞retourne la longueur de la chaîne

```
int i = "abcd".length();          //4
```

- **char charAt**(int index)➞retourne le caractère à l'index
  L'index d'une chaîne commence à 0

```
char chr = "abcd".charAt(2);  // c
```

- **Int indexOf**(String str)➞ retourne l'indice de la première correspondante

```
int i = "abcd".indexOf("bc"); //1
```

String

- String **substring**(int beginIndex, int endIndex)➞ endIndex n'est pas inclus

```
String s = "abcd".substring(2);     //cd
String s = "abcd".substring(2,3);   //c
```

- boolean **contains**(String str)➞ teste si la chaîne contient une sous-chaîne.
- boolean **replace**(String)➞ recherche et remplace

```
String s = "aba".replace("a","c"); //cbc
```

- trim() ➞ retire tous les caractères blanc de début et de fin

```
String s = " ab cd ".trim(); //ab cd
```

On peut chaîner les méthodes

StringBuilder

- La chaîne peut être modifiée (pas de création de chaîne intermédiaire). Elle retourne une référence à elle même
- **charAt**(), **indexOf**(), **length**() et **substring** ➞idem String
- **append**(String str) ➞ ajouter une chaîne
- **insert**(int offset, String str)➞ insérer (offset commence à 0)
- **delete**(int start, int end)➞ supprimer (end non inclut)
- **toString**() ➞ convertie un StringBuilder en String

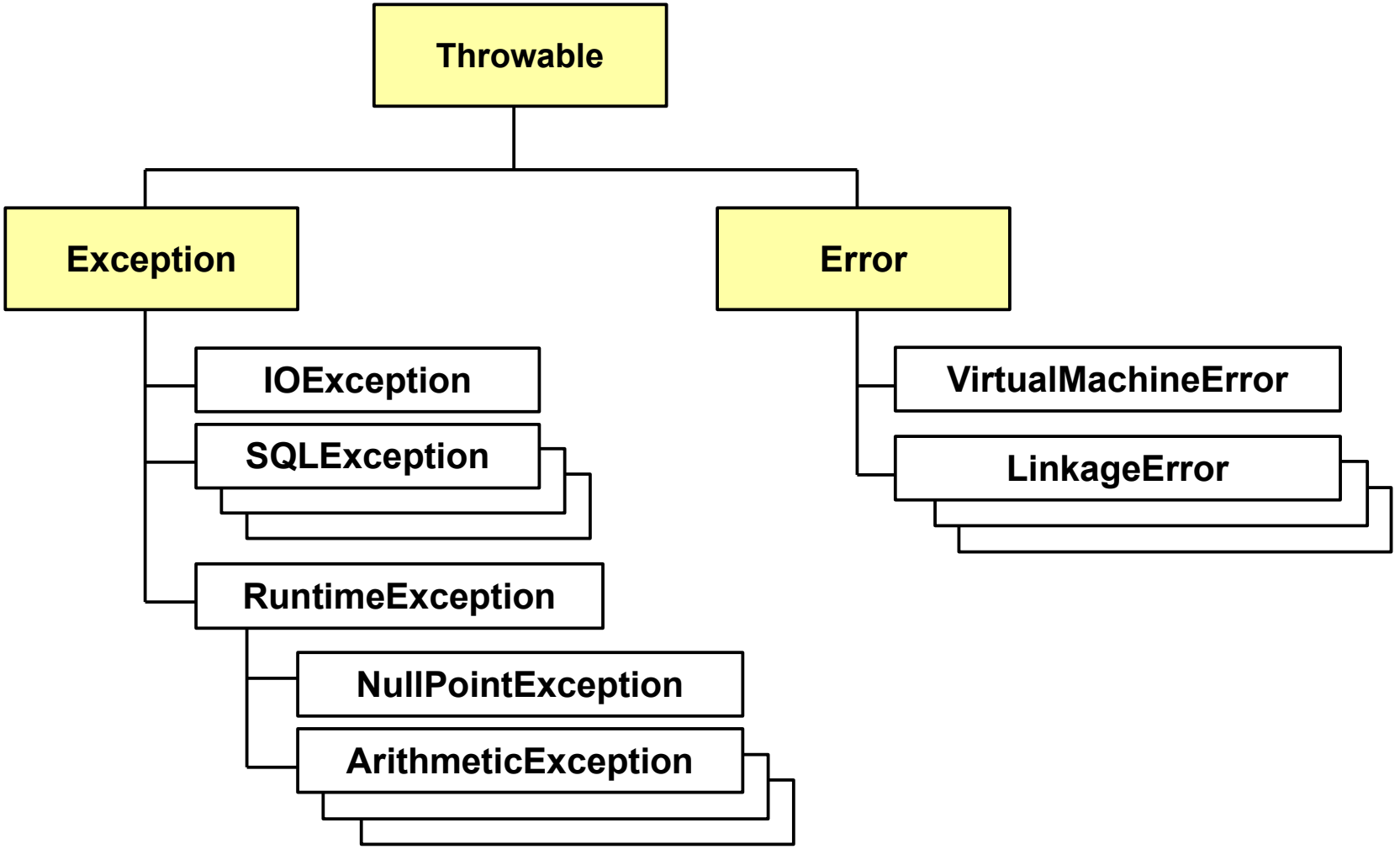
Maths

- Classe contenant des implémentations standard (abs(), cos(), floor(), min(), round()...)
- Les méthodes sont toutes statiques
- Constantes E et PI

```
int i = (int)Math.floor(3.99/2);
long l = Math.round(.51);
double d = Math.sin(Math.PI * .75);
double d2 = Math.pow(-1,.5);
```

Exceptions

La classe Throwable



Types d'Exceptions

- **Checked exceptions**
  - Le développeur doit les anticiper et coder des lignes pour les traiter.

  Exemple : Ouvrir un fichier qui n'existe pas.
- **Errors**
  - On ne doit pas les identifier et le programme s'arrête en les rencontrant.

  Exemple : La JVM charge une classe inexistante.
- **Runtime exceptions**
  - Ne peuvent être prévues (dans certains cas)

  Exemple : Essayer de lire une valeur en dehors d'un tableau.

Le bloc « try » et « catch »

- Utilisé pour encadrer un bloc susceptible de déclencher une exception

```
try {
    // des lignes de code susceptibles
    // de lever une exception
}
catch (IOException e) {
    // traitement de l'exception de type IO
}
finally {
    // toujours exécuté, même sans
    // exception ou une exception imprévue
}
```

Les mots clés « throw » et « throws »

- Le mot clé **throw** est utilisé pour déclencher une exception à n'importe quel moment
  Ex : test d'une valeur positive

- Le mot clé **throws** est utilisé pour dire à la méthode de ne pas récupérer l'exception localement mais plutôt l'envoyer dans la méthode appelante.

```
public void methode3() throws IOException {
    throw new IOException("Fichier manquant");
}
```

Créer ses propres exceptions

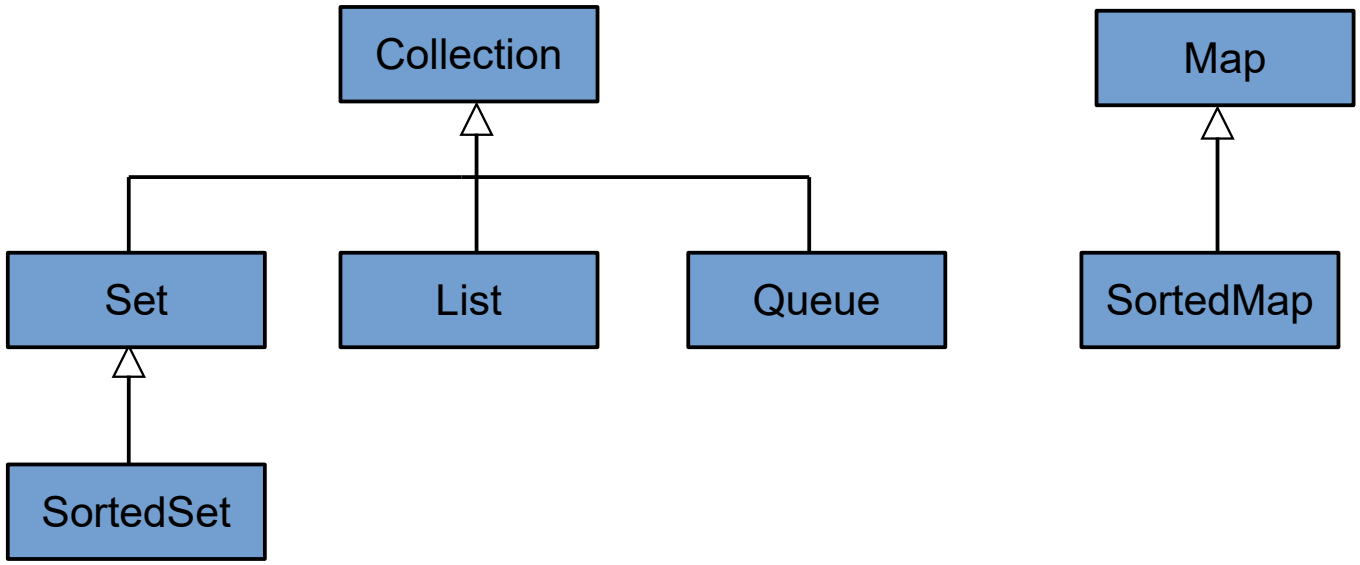
- Il faut seulement hériter de la classe **Throwable** ou une sous-classe (généralement la classe **Exception**)

```
public class MonException extends Exception {
    // Une exception valide !
}
```

```
public class NegativeNumberException extends NumberException {

    public NegativeNumberException(int num) {
        super("Le nombre "+num+"est négatif");
    }
    // C'est une exception valide également !
}
```

Les collections

Les collections



- Une collection est un objet qui contient d'autres objets
- Les interfaces et les classes se trouvent dans le paquetage Java.util

Interface Collection

- **add**(T t) et **remove**(T t) ➞ permet d'ajouter et de retirer un objet à la collection
- **contains**(T t) ➞ true si l'objet passé en paramètre est contenu dans la collection
- **size()** ➞ retourne le nombre d'éléments de la collection
- **isEmpty**() ➞ retourne true si la collection est vide.
- **clear**() ➞ efface la collection
- **toArray**(T[] a) ➞ convertit la collection en tableau.
- **addAll**(collection) et **removeAll**(collection) ➞ permet d'ajouter et de retirer l'ensemble des objets passés dans la collection en paramètre.
- **retainAll**(collection) ➞ permet de retirer tous les éléments de la collection qui ne se trouvent pas dans la collection passée en paramètre.
- **containsAll**(collection) ➞ retourne true si tous les éléments de la collection passée en paramètre se trouvent dans la collection courante.

Parcours d'une collection

- **iterator**() ➞retourne une instance d' Iterator.
- Il n'a que 3 méthodes :
  - **hasNext**() : retourne true si la collection possède encore des éléments à itérer
  - **next**() : retourne l'élément suivant
  - **remove**() : permet de retirer de la collection l'élément courant. (pas supportée par toutes les implémentations UnsupportedOperationException)
- avec

```
for(type element : collection) {
   ...
}
```

Interface List

- L'interface **List** correspond à un groupe d'objet indexé (commençant à 0)

- Il a trois implémentations:
  - **ArrayList** : un tableau à taille variable
    ➞accès rapide à un élément donné
  - **LinkedList** : liste chaînée
    ➞insertion en début et en fin de liste rapide
  - **Vector** : synchronisé

Méthode de l'interface List

- **add**(int index, T t) et **addAll**(int index, collection) ➞insérer un ou plusieurs éléments à la position de l'index
- **remove**(int index) ➞ retire l'élément à la position de l'index. L'élément est retourné par la méthode
- **set**(int index, T t) ➞ remplace l'élément placé à la position index par celui passé en paramètre. L'élément retiré est retourné par la méthode
- **get**(int index) ➞ retourne l'élément placé à l'index.
- **indexOf**(Object o) et **lastIndexOf**(Object o) ➞ retournent respectivement le premier et le dernier index de l'objet passé en paramètre
- **subList**(int debut, int fin) ➞ retourne la liste composé des éléments compris entre debut, et fin - 1.
  ne retourne pas une copie de la liste, mais une vue sur cette liste

Interface Set

- L'interface Set correspond à un ensemble d'objets qui n'accepte pas de doublons (2 objets égaux avec equals).
- L'ajout d'un élément dans un Set peut échouer, si cet élément s'y trouve déjà. (dans ce cas, add(T t) retourne false)
- Les principales implémentation :
  - **HashSet** : fonctionne avec une table de hachage, dont les clés sont les codes de hachage des objets ajoutés, et les valeurs les objets eux-mêmes.
    Offre des performances constantes pour les opérations add, remove, contains et size.
    Par contre, il faut éviter l'itération sur de grand ensemble
  - **LinkedHashSet** :La classe est une extension de HashSet. qui améliore les performances de l'itération
  - **TreeSet** : garantit que les éléments sont rangés dans leur ordre naturel

Interface SortedSet

- Avec l'interface **SortedSet** tous les objets sont automatiquement triés dans un ordre que l'on peut définir
- Il y a deux façons de faire pour comparer deux objets :
  - Implémenter l'interface **Comparable**.
    il n'a qu'une méthode : **compareTo**(T t), qui retourne un entier 0 → égal, -1→plus petit que l'argument 1→plus grand que l'argument .
  - Fournir au constructeur de SortedSet, une instance de **Comparator**. Il n'a qu'une méthode **compare**(T t1, T t2), idem compareTo(T t)

Méthode de SortedSet

- **comparator**() : retourne l'objet instance de Comparator.
- **first**() et **last**() : retournent le plus petit objet de l'ensemble, et le plus grand.
- **headSet**(T t) : retourne une instance de SortedSet contenant tous les éléments strictement plus petit que l'élément passé en paramètre.
- **tailSet**(T t) : retourne une instance de SortedSet contenant tous les éléments plus grands ou égaux que l'élément passé en paramètre
- **subSet**(T inf, T sup) : retourne une instance de SortedSet contenant tous les éléments plus grands ou égaux que inf, et strictement plus petits que sup.

Interface Queue

L'interface Queue modélise une file d'attente simple

- **add**(T t) et **offer**(T t) ➞ajouter un élément à la liste.
- Si la capacité maximale de la liste est atteinte :
  - **add**() lance une exception IllegalStateException
  - **offer**() retourne false.
- **remove**() et **poll**() ➞ retirer un élément de la file d'attente. Si aucun élément n'est disponible
  - **remove**() lance une exception NoSuchElementException
  - **poll**() retourne null.
- **element**() et **peek**() examinent toutes les deux l'élément disponible, sans le retirer de la file d'attente. Si aucun élément n'est disponible, alors element() lance une exception **NoSuchElementException**, tandis que **peek**() retourne null

Interface Map

- L'interface **Map** correspond à un groupe de clé/valeur
- Une clé repère une et une seule valeur
- Implémentation : HashMap, TreeMap
- Méthode :
  - **put**(K key, V value) et **get**(K key) ➞ associer une clé à une valeur et récupérer cette valeur à partir de cette clé
  - **remove**(K key) ➞supprimer la clé passée en paramètre de la table, et la valeur associée
  - **keySet**() ➞ retourne un Set contenant toutes les clés de la table de hachage
  - **values**() ➞ retourne l'ensemble de toutes les valeurs stockées dans la table
  - **clear**() ➞ efface tout le contenu de la table.
  - **size**() ➞ retourne le cardinal de la table, et

Interface Map

- **isEmpty**() ➞indique si la table est vide
- **putAll**(Map map) ➞ajouter toutes les clés de la table passée en paramètre
- **containsKey**(K key) et **containsValue**(V value) ➞tester si la clé ou la valeur passée en paramètre sont présentes dans cette table.
- **entrySet**() ➞retourne un Set, dont les éléments sont des Map.Entry

  Map.Entry permet de modéliser les couples (clé, valeur) d'une table de hachage. **getKey**() et **getValue**() retournent la clé et la valeur de ce couple. **seValues**() modifie la valeur associé à une clé durant l'itération

Collections

- Classe utilitaire pour travailler avec des collections
  - Tris (liste)
  - Recherche (liste)
  - Copies
  - Minimum et maximum

Arrays

- Classe utilitaire qui traite des tableaux
- Int **binarySearch**(int[] tab, int key)
  ➞recherche d'un élément dans un tableau, retourne l'index de l'élément recherché
- boolean **equals**(int[] tab1, int[] tab2)
  ➞comparaison de deux tableaux
- **fill**(int[] tab, int val)
  ➞initialisation d'un tableau
- **sort**(int[] tab)
  ➞Tri d'un tableau
- **toString**(int[] tab)➞méthode toString() pour les tableaux

Les entrées/sorties

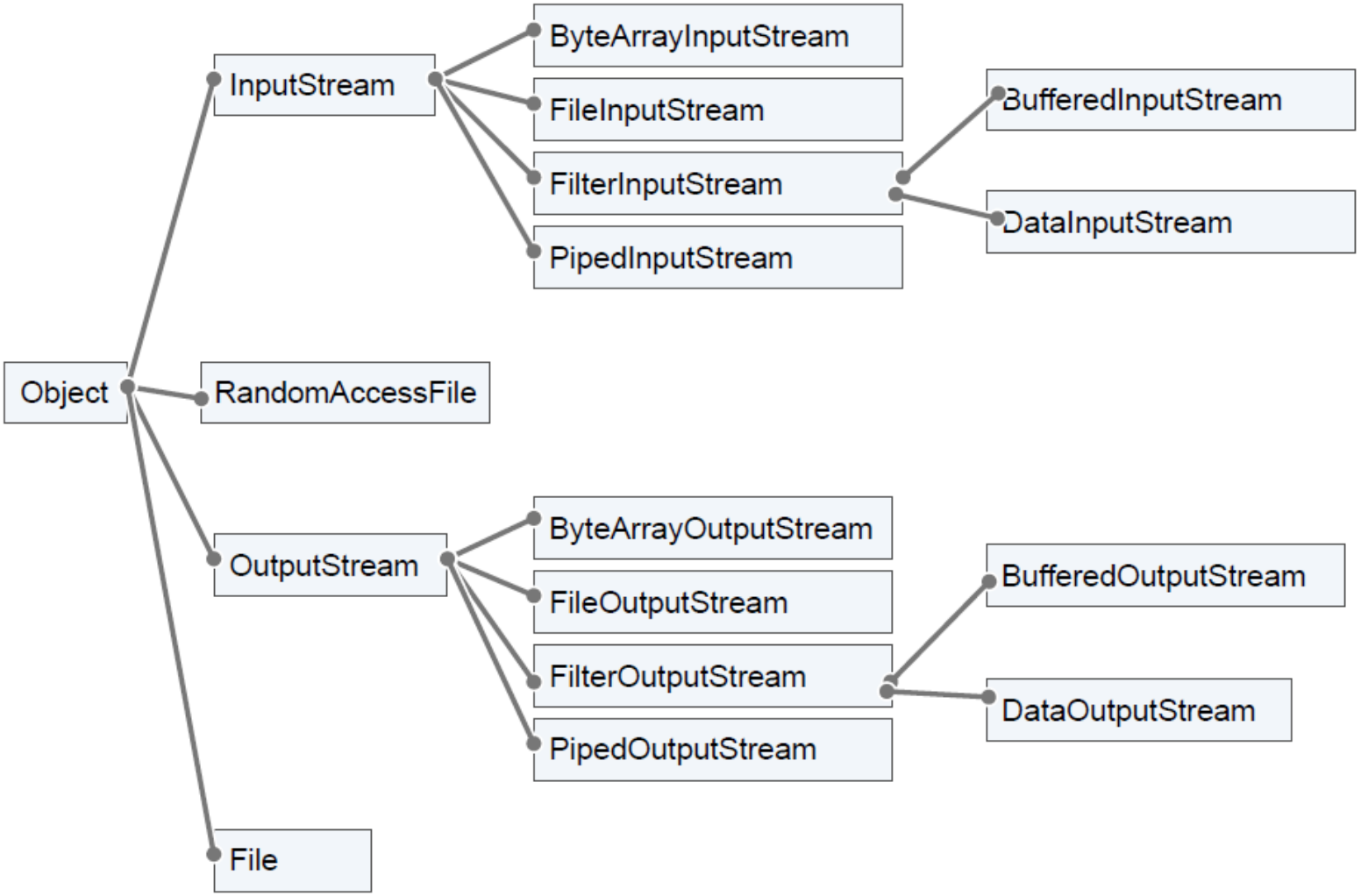
Définition

- **Stream** = flux de données (sériel, temporaire, en entré ou sortie
- Le package java.io englobe les classes permettant la gestion des flux

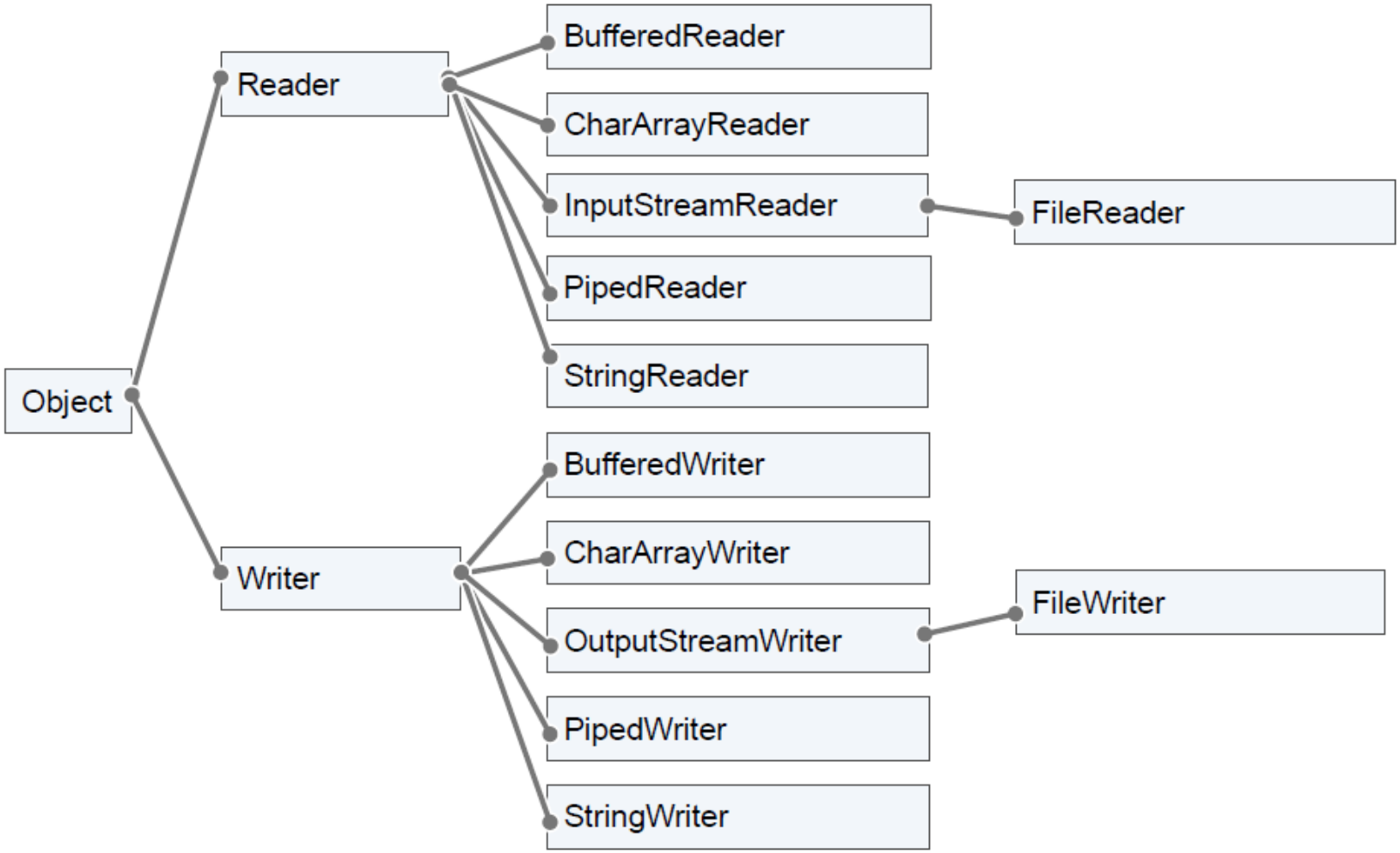
- Principe d'utilisation d'un flux:
  - Ouverture du flux
  - Lecture/écriture de l'information
  - Fermeture du flux

- InputStream et OutputStream pour la gestion d'un flux binaire, Reader et Writer pour un flux de caractères

Aperçu des classes



Aperçu des classes

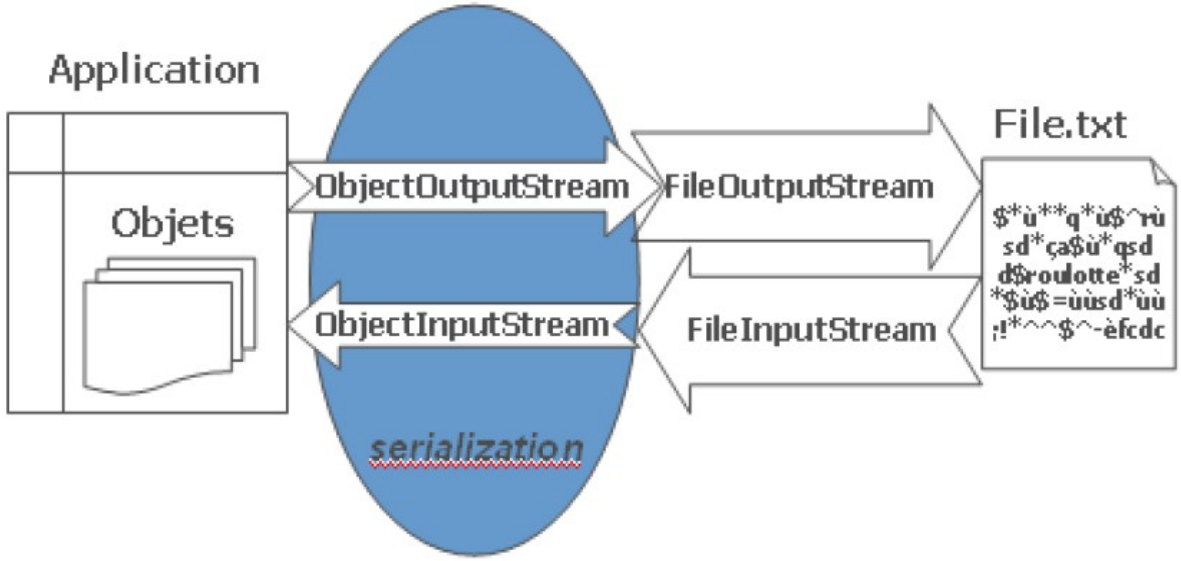


Exercices

- Manipulation des différentes méthodes de la classe File : création d'un fichier, vérification si répertoire et lister les différents fichiers d'un répertoire.
- Manipulation des différents types de flux : Object, Data, byte, char etc.
- Écriture de programmes permettant la copie de fichiers : utilisant Input/Output puis Reader/Writer

Sérialisation d'objets

- La sérialisation permet de sauvegarder l'état d'un objet dans un support persistant.




```
class Test implements java.io.Serializable{…}
```

Sérialisation d'objets

- transient pour définir un attribut non sérialisable :
  public transient String password = "";

- ObjectOutputStream pour la sérialisation :
  void writeObject(Object o);

- ObjectInputStream pour la désérialisation :
  Object readObject()

Exercices

- Sérialisation d'une instance de Voiture dans un fichier.

- Désérialisation à partir d'un fichier