



Objectifs

- Écrire un programme JAVA
- Utiliser les concepts objets avec JAVA
- Utiliser des packages classiques
- Exécuter des tests unitaires



Pré-requis

- Syntaxe C
- Concepts objets



Cadre

- Le Java vu par ~~SUN~~ et uniquement **ORACLE**
- Préparer les premières certifications



Évaluation

- Présence
- Examen final



EDI ?

- Editeur de texte simple
- Netbeans
 - Oracle -> Apache
 - Prochaine version ?
- Eclipse
 - Plugin ?
 - Récupération de Java EE / Jakarta EE
- IntelliJ
 - Licence gratuite pour étudiant
 - Modèle pour Android Studio



Les EDI cachent des choses ...



Plan

1. Premier programme	8
2. Notions de base & syntaxe	28
3. Concepts objets en Java	47
4. Les exceptions	100
5. Généricité	116
6. Collections	130
7. Plus de langage	137
8. Introspection	156
9. Programmation fonctionnelle –Streams	159
10. Fichiers & flux, sérialisation	166



Introduction



- Île ?
- Javascript (ECMA) ?

- Langage
- Machine virtuelle
- Plateforme



Motivations

- Langage créé en 1995



ORACLE

- Patrick Naughton
- James Gosling

Concepts difficiles du
C++ évacués

- Simple
- Sécurisé (réseaux, Internet)
- Portable
- Performant



1. Premier programme

ISIMA

Premier programme

```
/* ma première classe */  
public class Exemple  
{  
    public static void main(String[] argv)  
    {  
        // afficher un message  
        java.lang.System.out.println("Bonjour");  
        System.out.println("; -");  
    }  
}
```

Fichier source (texte) *Exemple.java*



Pour voir le résultat...

1. Compiler le programme

```
javac Exemple.java
```

2. Lancer le programme

```
java Exemple
```

.exe ?

.class ?



Fichier source

- Extension : .java
- Nom du fichier = nom de la classe publique
- Respecter la casse **E**xemple
- 1 classe publique par fichier
- Mélange déclaration + implémentation



+ commentaires



Compilation

- Fichier compilé : .class
- Pseudo-code (byte-code)
- ≠ Code machine



```
javac Exemple.java
```

Certains compilateurs transforment le code java en code natif :

- Portabilité nulle
- Gestion de la mémoire ?



Exécution / JVM

- Pseudo-code **interprété** par la *Java Virtual Machine (JVM)*

java Exemple

- voire compilé en natif à la volée (JVM hotspot)
- Programme seul (*standalone*)
 - Spring
- Serveur applicatif, bases de données, IA
- Processeur JAVA (Smart Cards, Blu-ray)
- Systèmes *Android*
- Portabilité totale si bonne JVM



Java 7 hotspot
250 000 lignes
C / C++

<https://github.com/openjdk/jdk>

Documentation (11->22)

Module java.base
Package java.lang
Class String

java.lang.Object
java.lang.String

All implemented interfaces:
Serializable, CharSequence, Comparable<String>, Constable, ConstantDesc

```
public final class String
    extends Object
    implements Serializable, Comparable<String>, CharSequence, Constable, ConstantDesc
```

The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

Here are some more examples of how strings can be used:

```
System.out.println("abc");
String cde = "cde";
System.out.println("abc" + cde);
String c = "abc".substring(2, 3);
```

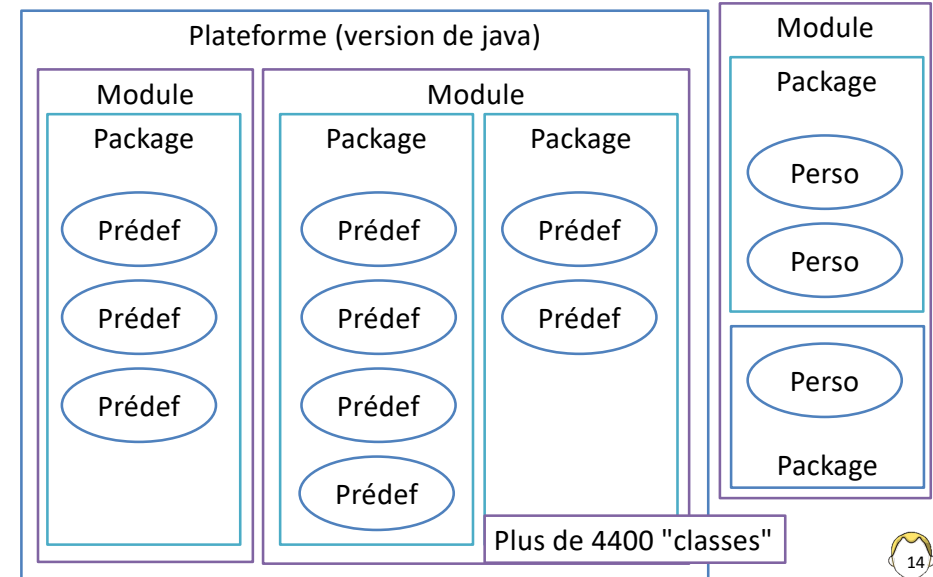
Plus de 4400 "classes"

<https://docs.oracle.com/en/java/javase/22/docs/api/>



Un langage à objets ?

Classe



Paquetage / package (1)

- Un ensemble de classes/fichiers rassemblés pour une finalité besoin fonctionnel
- Répertoire (ou fichier jar)

Version

- par défaut (java.lang)
- Standard (gestion E/S, graphisme)
- Personnel / Tiers

Plateforme





Paquetage / *package* (2)

- Nom spécifique suivant le type :
 - `java.lang` (sys), `java.awt` (std)
 - `javax.swing` (std), `javax.xml` (std)
 - `org.w3c.dom` (tiers/std)
 - `loic.classeperso` (perso)
- Sous-package
 - Mécanisme arborescent comme les répertoires
 - Séparateur : le point
- Retrouver les packages : `classpath`
 - Variable système
 - Paramètres en ligne de commande (`-cp` ou `-classpath`)



Module (1)



classe \subset package \subset module

- Ensemble de packages
- Projet Jigsaw – Java 9
- Charger une JVM adaptée
 - Plus rapide, plus économe
- Regroupés par famille

Pas de rétrocompatibilité

Java 11-13
21
37
1



Clause import

- Spécification complète d'une classe d'un package qui n'est pas chargé par défaut

`ArrayList` \Rightarrow `java.util.ArrayList;`

- Facilité : clause import

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

- Enumération
 - À l'unité
 - Par package (*) non récursif ;-(

Attention à l'import automatique des EDIs

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



Module (2)



- Répertoire de packages avec fichier descripteur
- Peut contenir des ressources
- Encapsulation forte
- Nom unique
- Dépendances
- Packages explicitement fournis
- Les services offerts et consommés




Plateforme

- ⊗ Plus grosse difficulté du java
 ➡ connaître ces classes standards
 classes *deprecated*
- 😊 Documentation bien faite :
 javadoc & tutoriels



```
java -showversion
javac -version
// version > 1.3,
// options : -source et -target
```




- 
 - 1.8 – Streams, Lambdas [2014]
 - Code dans les interfaces
 - 4240 classes
 - 1.9 – Modularité - Performances [2017]
 - Jshell
 - 6005 classes
- Nouveau cycle de développement


 - 1.10 – Performances [2018]
 - Variables locales implicites
 - 6002 classes
 - 1.11 – Version LTS

 [2018]
 - JDK light sans JavaFX, Java EE, CORBA
 - 4410 classes
 - 1.12 / 1.13 – Améliorations GC



LEGACY

- 1.0 (1.1) – *applet* , jni, awt [1995]
 - 236 classes pour 1.0.2
- 
 - 1.2 – *swing* (**version 2**) [1998]
 - 1524 classes
 - 1.3 – débogage [2000]
 - 1.4 – performances – nio [2002]
 - 1.5 - patrons / templates

 [2004]
 - 3279 classes
 - 1.6 – sécurité, scripts, performance [2006]
 - 3795 classes
 - Acquisition de Sun par **ORACLE** [2010]
 - 1.7 – open JDK - 4024 classes [2011]
- 
 - 1.14 [2020]
 - record
 - 1.15 [2020]
 - Classes et interfaces scellées (pour 1.17)
 - 1.16 [2021]
 - jpackage / améliorations GC
 - Support du C++ 14 pour l'OJDK
 - 1.17 – version LTS

 [2021]
 - Opérations virgule flottante strictes
 - API Générateurs nombres aléatoires
 - Classes et interfaces scellées
 - 1.18 – 1.19 [2022]
 - 1.20 [2023]



- 1.21

- Virtual threads, Sequenced collections

[2023]

- 1.22

- record

[2023]

- 1.23

[2024]

- 1.24

[2024]

- 1.25 – version LTS

[2025]



Tutoriel Java

- Bases du Java
- Version 8 de la plateforme

- <https://docs.oracle.com/javase/tutorial/>
- <https://docs.oracle.com/javase/8/docs/api/>



Distribution ?

- Usage

- Exécution seule (JRE)
- Développement (JDK < 2, SDK v ≥ 2)
- Modules externes complémentaires ?

Télécharger le JDK pour java SE
/ openjdk

- Cibles

- **Standard** **Java SE** / J2SE
- **Entreprise** **Java EE** / J2EE / Jakarta EE
- **Micro** **Java ME** / J2ME (v ≥ 5)

Spring[Boot]
WebSphere
GlassFish,
Jboss, ...



2. Notions de base
Syntaxe

{ Accolades } et commentaires

```
public class Exemple
{
    public static void main(String[] argv)
    {
        //  afficher un commentaire monoligne
        /*  commentaire
            sur plusieurs lignes          */
        /** commentaire javadoc (≈ doxygen) */
    }
}
```

- Classe
- Méthode
- Bloc : ensemble séquentiel d'instructions



Types de données primitifs

- char
 - type caractère
 - [] ≠ String
 - UTF-16 '\u0000'
- boolean
 - true ou false.
 - non homomorphe aux entiers
- types entiers
 - byte (8 bits)
 - short (16 bits)
 - int (32 bits)
 - long (64 bits)
- types réels
 - float (32 bits)
 - double (64 bits)



Attribut / Variable / Paramètre ?

- Type fixé à la compilation
- Primitive / scalaire
 - utilisation directe
 - entier / réel / booléen / caractère 'A'
 - pour l'efficacité
 - doublé par un type objet
- Objet
 - Manipulation par « références » (pointeurs ?)
 - Prédéfini ou utilisateur
 - Chaîne de caractères : String "Essai"



Encapsulation des types de données primitifs

- Modèle objet
 - Character
 - Number : Double Float
Byte Short Integer Long
- Conversion implicite : [un]boxing
- Interface riche
 - Conversion vers d'autres types
 - Opérations



Déclaration de variables locales

```
public static void main(String[] argv) {  
    int    i = 0;  
    char   c = 'A';  
    double d = 1.0;  
    float  f = 1.3f;  
}
```

- N'importe où dans le bloc
- Initialisation d'une variable **pas automatique**
 - Erreur : "might not be initialized"



Exemple de classes : les chaînes de caractères

- String ≠ char[]
- **constante** String
- **modifiables** StringBuffer
StringBuilder
- UTF-16
- Bibliothèque fournie
 - Comparaison de chaînes : equals(), compareTo()
 - Recherche : indexOf()
 - Extraction : substring(), StringTokenizer, split, regexp
 - Transformation aisée de type scalaire vers StringBBBB



Manipulation de variables primitives

```
public static void main(String[] a) {  
    int i = 0;  
    i = i + 1;  
    i += 1 ;  
    i *= 2 ;  
  
    System.out.println(i) ;  
    System.out.println(++i) ;  
    System.out.println(i) ;  
    System.out.println(i++) ;  
    System.out.println(i) ;  
  
    i = (int) 10.6 ;  
}
```



OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

SE

Module java.base
Package java.lang
Class StringBuffer

java.lang.Object
java.lang.StringBuffer

All Implemented Interfaces:
Serializable, Appendable, CharSequence, Comparable<StringBuffer>

public final class StringBuffer
extends Object
implements Serializable, Comparable<StringBuffer>, CharSequence

A thread-safe, mutable sequence of characters. A string buffer is like a String, but can be modified. At a given time, it contains one particular sequence of characters, but the sequence can be changed through certain method calls.

String buffers are safe for use by multiple threads. The methods are synchronized where necessary so that no two threads access the same data simultaneously. This is done by locking the object, and thus, no threads involved.

The principal operations on a StringBuffer are the append and insert methods, which are overloaded to append or insert the characters of that string to the string buffer. The append method always adds the characters of that string to the end of the string buffer.

API Note:
StringBuffer implements Comparable but does not override compareTo. Thus, the natural ordering of StringBuffers as keys in a SortedMap, SortedSet, or SortedList is not guaranteed.

Since:
1.0

See Also:
StringBuilder, String, Serialized Form

Constructor Summary

Constructor	Description
StringBuffer()	Constructs a string buffer with no characters in it and an initial capacity of 16 characters.

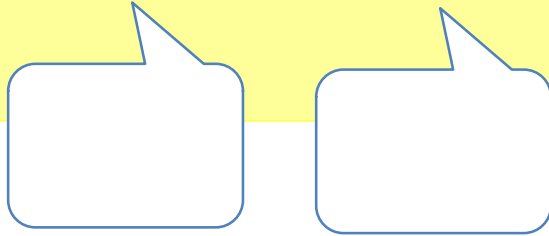
- Informations ?
- Instanciation d'un objet
- Utilisation d'un objet
- Plus besoin ?

StringBuffer
???
+ StringBuffer()
+ append()
+ compareTo()
+ length()
+ toString()



Instanciation d'objet

```
public static void main(String[] a) {  
    StringBuffer s  
        = new StringBuffer("essai");  
}
```



Manipulation d'objet (1)

```
StringBuffer s1 =  
    new StringBuffer("Bonjour");
```

- Appel de méthode sur référence **non nulle**
→ opérateur "point"

```
s1.append(" les ZZ");  
s1.append(2);
```

Essayer si s1 est
nulle

```
System.out.println(s1.toString());  
System.out.println(s1.length());  
System.out.println(s1.capacity());
```

```
StringBuffer s2 = s1;  
System.out.println(s2.toString());
```



Référence ...

- PAS de manipulation directe des objets ou tableaux

→ manipulation par « référence »

- ≈ pointeur en C/C++
- Valeur **null** si initialisation par défaut

▪ Pas encore de contenu valide

```
StringBuffer s;
```

▪ Plus utilisée (optionnel)

```
s = null;
```

- Affection à une valeur admissible

▪ Allocation mémoire

Toujours possible ?

```
s = new StringBuffer("texte");
```



Manipulation d'objet(2)

```
System.out.println(s1 == s2);  
System.out.println(s1.compareTo(s2));  
System.out.println(s2.compareTo(s1));
```

```
s1[0] = 'c';
```

```
s1.setCharAt(0, 'b');
```



```

{
    int i = 0 ;
    {
        int j= 3 ;
        // i est utilisable dans ce bloc
    }
    // j n'est plus disponible ici
}

```

```

{
    int i = 0 ;
    boolean b = true;
    {
        double i= 3 ;
        boolean b = false;
    }
}

```



Condition (1)

```

if (test) {
    ...
}

```

Test avec i entier

```

(i==0)
(i!=0)

```

```

(i)
(!i)

```

```

if (test) {
    ...
} else {
    ...
}

```

```

boolean b1 =(i==5) ;
boolean b2 = !b1;

```

Opérateur ternaire

```

(test) ? VRAI : FAUX

```

Un test est un booléen :
true ou false



Condition (2)

```

if (test) instruction1;
else instruction2;

if (b1) ...
if (!b1)...      Opérateur NON

if (b1 || b2) ... Opérateur OU
if (b1 && b2) ... Opérateur ET ALORS

```

Une séquence de test n'est pas complètement évaluée si ce n'est pas nécessaire.



Condition (3)

```

switch (variable) {
    case valeur1 :
        instructions;
        break;
    case valeur2 :
    case valeur3 :
        instructions;
        break;
    default:
        instructions;
        [break;]
}

```

- Variable de type simple (**String** possible dans 1.7)
- Oubli du break ? (≠ C#)
- **default** facultatif



Boucles conditionnelles

```
for (initialisation;test;incrémentation) {  
    ...  
}
```

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

```
while (test) {  
    ...  
}
```

Accolades facultatives
s'il n'y a qu'une instruction

```
do {  
    ...  
} while (test);
```



Variable de boucle et visibilité...

```
{  
    int i;  
    for(i=0; i<10; ++i) { ... }  
}
```

```
{  
    for (int i=0; i<10; ++i) { ... }  
    for (int i=0; i<10; ++i) { ... }  
}
```

```
{  
    int i;  
    for (int i=0; i<10; ++i) { ... }  
}
```



Déclaration d'une classe

- Nom unique (package)
- Moule / Modèle / Fabrique à objets
- Caractéristiques / Attributs
- Messages / Méthodes
- De classe / d'instance
- Visibilité et encapsulation

- Relations

Java 9 change
pas mal de choses ...



3. Concepts objets



```
public class Etudiant {
```

VISIBILITÉ : public / (package) / private

```
private String nom;
int id;
```

```
public String getNom() {
    return nom;
}
```

```
public String getId() {
    return id;
}
```

Etudiant
- nom : chaîne ~ id : entier
+ getNom() : chaîne + getId() : entier

49

```
public class Etudiant {
    // ...
    public static void main(String[] argv) {

        Etudiant e = new Etudiant();

        S.o.p ("Nom"+ e.getNom());

        S.o.p ("Nom"+ e.nom);

    }
}
```

50



Constructeur

- Initialiser les attributs d'un nouvel l'objet
- Syntaxe différente d'une méthode
 - Porte le même nom que la classe
 - Pas de type de retour
- Constructeur sans argument
 - fourni automatiquement si pas d'autres constructeurs
- Surcharge de constructeur
 - Appel de constructeurs ≠ avec paramètres

❗ PAS d'héritage de constructeur

```
public class Cours {
    private int nbEtudiants;

    public void setNbEtudiants(int n) {
        nbEtudiants = n;
    }

    public int getNbEtudiants() {
        return nbEtudiants;
    }

    public Cours() {
        nbEtudiants = 0;
    }

    public Cours(int n) {
        nbEtudiants = n;
    }
}
```

Cours
- nbEtudiants : entier
+ constructeurs + getNbEtudiants () : entier

51

52

```

public class Cours {
    private int    nbEtudiants;
    private boolean passionnant;
    public void setPassionnant(int p) {
        passionnant = p;
    }
    public boolean isPassionnant() {
        return passionnant;
    }
    public Cours() {
        this(0, true);
    }
    public Cours(int n, boolean b) {
        setNbEtudiants(n);
        // setPassionnant(b);
    }
}

```



53

```

public class Cours {
    // ...
    public static void main(String[] argv) {
        Cours c1 = new Cours();
        Cours c2 = new Cours(12, true);

        S.o.println("Classe #"
                    +c2.getNbEtudiants());
        if (!c1.isPassionnant())
            S.o.println("bof");

        S.o.println(c2);
        S.o.println(c2.toString());
    }
}

```

54

Membres de classe

```

public class Etudiant {

    private String nom;
    ⓪ int id;
    static private int compteur = 0;

    public String getNom() {
        return nom;
    }
    static public int getCompteur() {
        return compteur;
    }
}

```



55

```

public class Etudiant {
    public static void main(String[] argv) {

        Etudiant e = new Etudiant();

        // S.o.p(compteur);
        // S.o.p(getCompteur());

        // S.o.p(e.compteur);
        // S.o.p(e.getCompteur());

        S.o.p(Etudiant.compteur);
        S.o.p(Etudiant.getCompteur());
    }
}

```

56



Détruire une instance

- Pas de destruction manuelle
- Destruction automatique par la JVM
 - Ramasse-miettes (*Garbage Collector*)
 - Le développeur peut demander un nettoyage, enfin ...
- Plus de fuites de mémoire ?
 - Tables de hachage complexe
 - Boucle infinie
 - Aider la JVM en mettant à `null`
- Méthode `finalize()`
 - Destructeur ?
 - Peut ne pas être appelée (si `gc` non exécuté)



57

```
public class Passage {
    static void methode1(int i) {
        i = 3;
    }

    static void methode2(int [] t) {
        t[4] = 7;
    }

    static void methode3(int [] t) {
        t = new int[10];
        t[4] = 9;
    }

    public static void main(String[] param) {
        int i = 5;
        int[] t = new int[10];
        methode1(i);
        methode2(t);
        methode3(t);
    }
}
```



Digression¹

Afficher i et t[4]

59

Encapsulation

// classe A avec encapsulation brisée

```
class A {
    public int valeur2;
    public A(int i) { valeur = i; }
}
```

A a = new A(2);
a.valeur = 5;

// classe A avec encapsulation

```
class AE {
    private int valeur2;
    public AE(int i) { setValeur(i); }
    final public int getValeur() { return valeur; }
    final public void setValeur(int v)
    { valeur = v; }
}
```

AE ae = new AE(0);
ae.setValeur(3);
ae.valeur = 5;

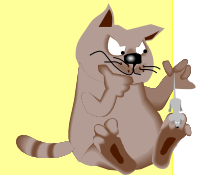


58

```
public class Passage {
    static void methode4(AE a) {
        a = new AE(2);
    }

    static void methode5(AE c) {
        c.setValeur(3);
    }

    static AE methode6(AE b) {
        b = new AE(4);
        return b;
    }
}
```



Digression²

```
public static void main(String[] param) {
    AE a = new AE(1);
    methode4(a);
    methode5(a);
    a = methode6(a);
}
```



60

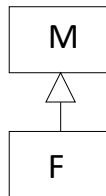
Passage des paramètres

- Passage des paramètres par valeur/copie
 - Evident pour types primitifs
 - Toujours ?
- Copie de référence/pointeur
 - => pas de copie d'objet
 - Tableaux
 - Objets

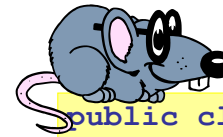
Héritage

- F hérite de M / dérive de M / spécialise M
- Héritage **SIMPLE** seulement

```
public class F extends M {
    public F() {
        super(); // appel du constructeur de M
        // initialisations spécifiques
    }
}
```



61



Référence **this**

```
public class C {
    String chaine1, chaine2;
    public C() {
        chaine1 = "CHAINE1";
        chaine2 = "CHAINE2";
    }
    void methode1(String chaine1, String c) {
        this.chaine1 = chaine1;
        chaine2 = c;
    }
    void methode2() {
        methode1("e", "f");
        this.methode1("", ""); // utile ?
    }
}
```

62

```
public class M {
    public String att1;
    protected String att2;
    private String att3;

    public void methode1() {...}
    protected void methode2() {...}
    private void methode3() {...}
}
```

Héritage sélectif

```
public class F extends M {
}
```

```
public class A {
    static void m() {
        F f = new F();
        f.methode1();
    }
}
```

64

63

Conditions pour l'héritage

- M doit être visible de F (public ou même package)
- M doit être dérivable (non finale)

```
[public] class M1 {
    public M1(String s) {}
    public M1 () {}
}
```

```
final class M2 {
    public M2() {}
}
```

```
class F1 extends M1 {
}
```

```
class F2 extends M2 {
}
```

```
public class A {
    static void m() {
        F1 f1a = new F1();
        F1 f1b = new F1("essai");
    }
}
```



java.lang.Object (1)

• clone()

• finalize()

• toString()

Classe@hashcode

• getClass().getName()

- Connaître à l'exécution le nom de la classe

objet instanceof Classe



java.lang.Object (2)

boolean equals(Object)

int hashCode()

- Deux objets égaux ont le même hashcode
- Ne doit pas changer pour une exécution
- Deux objets distincts peuvent avoir le même
- Souvent l'adresse mémoire de l'élément



Polymorphisme

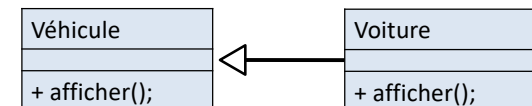
• Forme faible

- Surcharge de méthode – *overloading*
- Statique (compilation)
- Méthodes de signatures différentes

• Forme forte

- Redéfinition – *overriding*
- "Surcharge" *dynamique* (abus de langage)
- Actions différentes pour des classes d'une même hiérarchie

Voiture
+ avancer(temps : entier)
+ avancer(distance : réel)



```
public class M {
    private String a;

    public M(String s) {
        a = s;
    }

    public void m() {
        S.o.p(a);
    }
}
```

Parentalité (1)



```
public class F extends M {
    public F(String s) {
        super(s);
    }
}
```

```
F f = new F();
f.m();
```



```
public class M {
    private String a;

    public M(String s) {
        a = s;
    }

    public void m() {
        S.o.p(a);
    }
}
```

Parentalité (2)

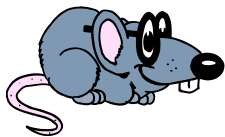


```
public class F extends M {
    public F(String s) {
        super(s);
    }

    @Override
    public void m() {
        S.o.p("Fille");
        super.p();
    }
}
```

Annotation

- Optionnelle
- Compilo & dev
- Bonne pratique



Noms qualifiés

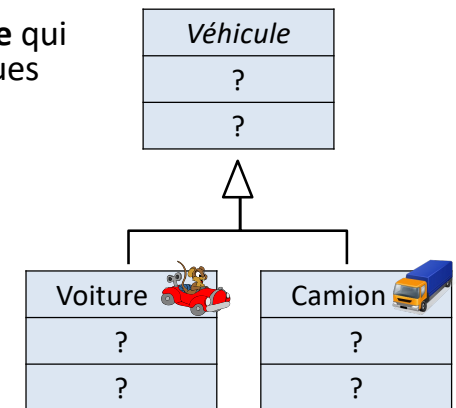
```
class M {
    protected int a;
}

public class F extends M {
    protected double a;
    public void toto() {
        a
        this.a
        super.a
        ((M) this).a
        ((F) this).a
    }
}
```



Hériter ...

- Écrire une classe **Véhicule** qui reprend les caractéristiques communes des classes **Voiture** et **Camion**
- Modifier les classes pour tester le polymorphisme



```
class Vehicule {
    String immat;
    public Vehicule(String im) {
        immat = im;
    }
    public void afficher() {
        S.o.p("Je suis un vehicule "+immat);
    }
}
```

```
Voiture v = new Voiture("300 ISI 63");
v.afficher();
```

```
class Voiture extends Vehicule {
    String immat;
    public Voiture(String im) {
        super(im);
    }
    // afficher ?
}
```



```
public class Polymo {
    public static void main(String[] param) {
        Vehicule u = new Vehicule();
        Voiture v = new Voiture();
        Camion w = new Camion();
        Vehicule x = new Voiture();
        Voiture y = new Vehicule();
    }
}
```

```
class Vehicule {
    public void afficher() {
        S.o.p ("Vehicule");
    }
}
```

```
class Voiture extends Vehicule {
    public void afficher() {
        S.o.p("Voiture");
    }
}
```

Appeler les méthodes afficher() des objets

```
class Camion extends Vehicule {
}
```

Le polymorphisme marche directement !

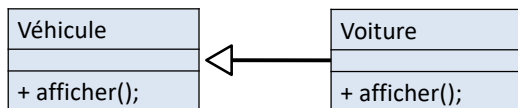


74

Méthode virtuelle ?



- Compilation
 - Validité du message
 - Version inconnue ?



- Exécution
 - Construction d'une table des méthodes virtuelles
 - Recherche puis appel de la bonne méthode



Méthode finale

```
class M {
    public final void m() {
    }
}
```

- Méthode non redéfinissable dans les classes filles
- Pas de création d'entrée dans la table des méthodes virtuelles OU limitation du parcours
- "Appel plus rapide" que les méthodes virtuelles



Dernier mot : la JVM hotspot avec ses nombreuses optimisations



```

public class Vehicule {
    public void afficher() {
        System.out.println("Vehicule");
    }
    public static void main(String[] param) {
        Vehicule v = new Sportive();
        v.afficher();
        v.special();
        ((Sportive)v).special();
        ((Camion)v).afficher();
    }
}

class Sportive extends Vehicule {
    public void afficher() {
        System.out.println("Sportive");
    }
    public void special() {
        System.out.println("special");
    }
}

```



Que se passe-t'il ?

```

if (v instanceof Sportive)
    ((Sportive)v).special();

```



77

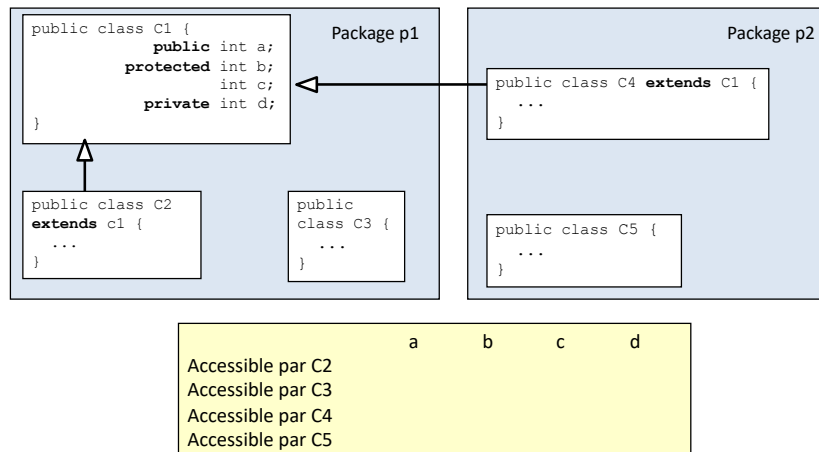
Niveaux d'accès

- **Privé** | **private** : même classe
- **Protégé** | **protected** : même package ou sous-classe d'un package différent
 - Moins restrictif que le C++ !
 - Différent en UML également
- **Package** | - (par défaut) : package
 - Sorte de friend du C++
 - DANGEREUX
- **Public** : tout le monde



78

Encapsulation & visibilité



79



Méthodes & classes abstraites

- Mot-clé **abstract** (modificateur) OBLIGATOIRE
- Méthode abstraite
 - Sans implémentation
- Classe abstraite
 - Toute classe avec au moins une méthode abstraite OU ALORS toute classe déclarée comme telle
- Non instanciable
- Permet d'implémenter la notion de concept

80



Interface (1)

<<interface>>
Flottant

+ flotter()
+ avancer()

- Description / contrat
 - Liste de méthode(s) sans code
 - "Constantes" autorisées (*public static final* par défaut)
 - Pas de variable/attribut [UML]
 - « Classe virtuelle pure » [C++]
- Respecter le contrat = IMPLEMENTER l'interface

```
[public] interface Flottant {
    public static final int CONSTANCE = 30;
    double PI = 3.14;

    public abstract void flotter();
    public          void avancer();
}
```



```
public abstract class Vehicule1 {
    abstract public void afficher() ;
}
```

```
public abstract class Vehicule2 {
    public void afficher() {
        System.out.println("Vehicule");
    }
}
```

```
class Voiturela extends Vehicule1 {
}
```

ERREUR : must implement the inherited method

```
abstract class Voiturelb extends Vehicule1 {
}
```

```
class Voiturelc extends Vehicule1 {
    public void afficher() {}
}
```

81

Interface (2)

- [Vocabulaire] IMPLEMENTER une interface

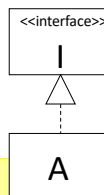
```
class Radeau implements Flottant {
    public void flotter() {}
    public void avancer() {}
}
```

- Instancier une classe ?

```
abstract class Bateau implements Flottant {
    public void flotter() {}
}
```

Voilier

BateauAMoteur



Interface & polymorphisme

```
interface Autonome {
    public static final boolean COMPLET = true;
    public void rouler();
}

class Waymo extends Vehicule implements Autonome {
    public void rouler() {
        System.out.println("Pilote automatique");
    }
}
```

la classe est du **type** de l'interface qu'elle implémente

```
public static void main(String[] param) {
    Vehicule v = new Waymo();
    ((Waymo)v).rouler();
    ((Autonome)v).rouler();
    System.out.println(Autonome.COMPLET);
    System.out.println(Waymo.COMPLET);
}
```





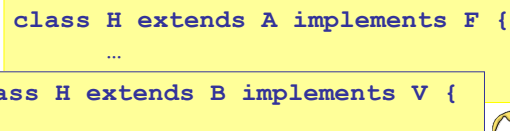
	R	M	I	Object
<code>X a = new R ()</code>				
<code>X b = new M ()</code>				
<code>X c = new F1 ()</code>				
<code>X d = new F2 ()</code>				
<code>X e = new I ()</code>				

Implémentation multiple

```
interface ID {
    void mID();
    void m1();
    void m2();
}
```



Relation non symétrique
= raison fonctionnelle



The diagram is divided into two parts, labeled 'a)' and 'b)', illustrating different ways to structure a class hierarchy for the 'avancer()' method.

Part a) Concrete Class Hierarchy:

- At the top is a class box for 'Véhicule' with the method '+ avancer()'. It has a solid line inheritance arrow pointing to it from a base class box below.
- The base class box has two children: a ship (carrack) and an airplane.
- Below these are two more airplane icons, each with a dashed line inheritance arrow pointing to the base class box.
- At the bottom left, a seaplane icon is shown with a red text label 'avancer()?' next to it, indicating a question about its inheritance.

Part b) Interface Hierarchy:

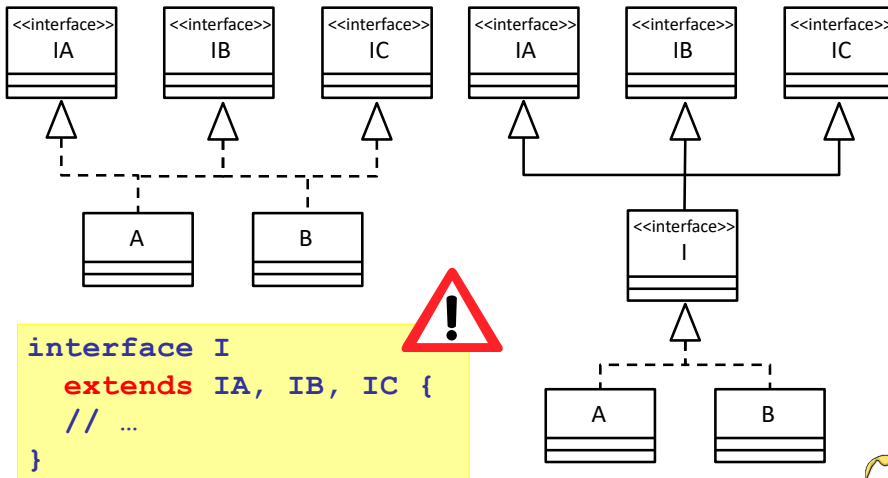
- At the top is a class box for 'Véhicule' with the method '+ avancer()'. It has a solid line inheritance arrow pointing to it from a base class box below.
- The base class box has two children: an airplane and an interface box.
- The interface box is labeled '<<interface>>' and 'Flottant'. It contains two methods: '+ flotter()' and '+ avancer()'. The '+ avancer()' method is circled in red.
- Below the interface box is a seaplane icon with a dashed line inheritance arrow pointing to the interface box.

- héritage virtuel
- conflit de méthode ?
- conflit d'attribut ?





Héritage multiple d'interface



Héritage multiple d'implémentation

8

```

interface I {
    default void m1() {}
    static void m2() {}
}

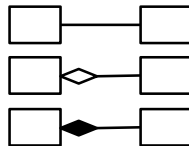
```

- Implémentation par défaut
 - Ajout de fonctionnalités sans casser du code existant
 - Cachée si la méthode est implémentée
 - Ambiguïté à lever dans certains cas
- Code statique



Relations entre objets

- Relation
- Agrégation
- Composition



- Modélisation par un attribut
 - Référence
 - Tableau de références
 - Conteneur spécifique / Collection
 - Ex: java.util.ArrayList



Agrégation / composition simple

```

public class Voiture1 {
    private Moteur m;

    public Voiture1() {
        m = new Moteur();
    }

    public void demarrer() {}
}

```

```

public class Moteur {
    public void demarrer() {}
}

```

```

public class Voiture2 {
    private Moteur m;

    public Voiture2(Moteur m) {
        this.m = m;
    }

    public void demarrer() {}
}

```

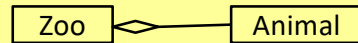
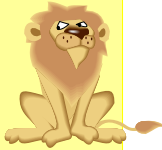


```

public class Zoo {
    static final int NB_ANI = 50;
    Animal[] animaux;
    public Zoo() {
        // pas de création d'objet, sinon le constructeur
        // par défaut serait obligatoire
        animaux = new Animal[NB_ANI];
    }
    public void setAnimal(int i, Animal a) {
        // if ((i>=0) && (i<NB_ANI))
        animaux[i] = a;
    }
    public static void main(String[] chaines) {
        Zoo zoo = new Zoo();
        zoo.setAnimal(0, new Animal("lion"));
    }
}

class Animal {
    String nom;
    // public Animal() { nom="INCONNU"; }
    public Animal(String nom) {
        this.nom = nom;
    }
}

```



93

```

// ajouter dans la classe Zoo
public Animal getAnimal(int i) {
    return animaux[i];
}

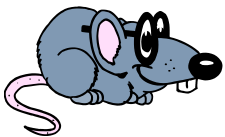
// ajouter dans la classe Animal
public void afficher() {
    System.out.println(nom);
}

// ajouter dans la methode main()
zoo.getAnimal(0).afficher(); // c'est bon

zoo.getAnimal(1).afficher();
zoo.getAnimal(60).afficher();

```

94



Tableau

- De types primitifs
 - int, double, char, ...
 - Une case = une valeur utilisable directement
- D'objets
 - Un tableau de références sur des objets de la classe
 - Références initialisées à **null**
 - Pas de création d'objets par défaut [C++]
 - Initialiser chaque élément du tableau pour l'utiliser

Résumé

```

[Modificateur]* class identifiant
    [extends     classe_de_base ]
    [implements  interface {, interface}* ] {
}

```

- Héritage simple **seulement**
- Implémentation multiple d'interfaces
- *Toutes* les méthodes sont virtuelles
- Une classe **finale** n'est pas dérivable
- Tous les classes dérivent de `java.lang.Object`

Conventions (1)

- Documentation officielle
- Tutoriaux SUN/Oracle
- Respect à l'écriture, facilité de lecture
- Production rapide
- Intégrée dans les EDI classiques
 - Génération automatique de code



Conventions (2)

- Nom de classe ou interface
 - Première lettre majuscule
 - Reste en minuscules
 - Majuscules aux mots composés
- Attribut écrit en minuscule
 - Pas de tiret
- Méthode
 - Verbe pour action
 - Premier mot en minuscule
 - Majuscules à la première lettre des mots suivants

```
class  
CoursGenial
```

```
int attribut;
```

```
void ronfler();
```



Conventions (3)

- Accesseur / Accessor
 - get + nom de l'attribut
 - is pour un booléen
- Mutateur/Mutator
 - set + nom de l'attribut
- "Constante"
 - Tout en majuscules
- Package
 - Tout en minuscules

```
getAttribut()  
isAttribut()
```

```
setAttribut()
```

```
CONSTANTE
```

```
fr.isima.paquetage
```



❌ Exceptions

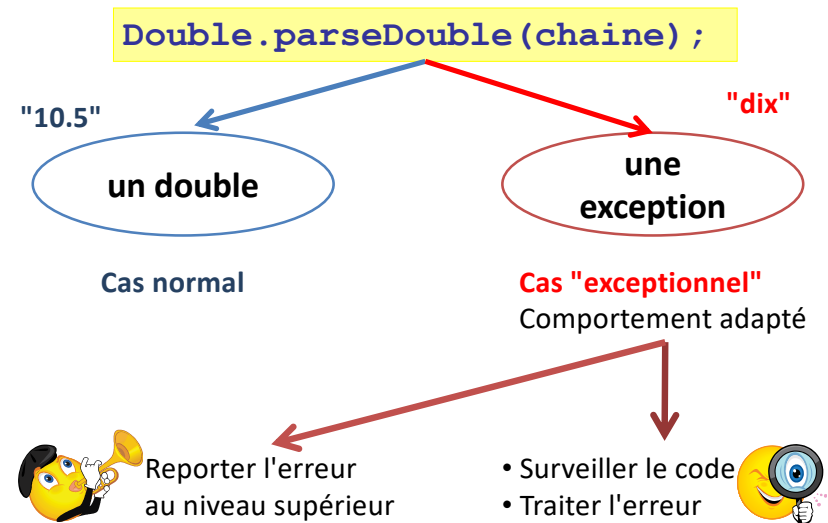
- Manière élégante et efficace de gérer les erreurs potentielles d'exécution
- Une erreur potentielle \equiv une exception
- Hiérarchie des exceptions
- Une erreur = une instance d'exception
- Partie intégrante de la signature d'une méthode
- Obligation de gérer les exceptions



Exemple

```
public void somme(String chaine) {  
    res = Double.parseDouble(chaine);  
    total += res;  
}
```

```
public void saisie {  
    String chaine = System.console().readLine();  
    while (!chaine.isEmpty()) {  
        somme(chaine);  
        chaine = System.console().readLine();  
    }  
}
```



OBLIGATION DE TRAITER UNE EXCEPTION



Attraper une exception

```
public void somme(String chaine) {  
    double res = .0;  
    try {  
        // bloc à surveiller  
        res = Double.parseDouble(chaine);  
        total += res;  
    } catch (NumberFormatException e) {  
        System.out.println(e.getMessage());  
        // ou e.printStackTrace();  
    } finally {  
        // Clause TOUJOURS exécutée  
    }  
}
```



Reporter l'erreur ...

```
public void somme(String chaine)
    throws NumberFormatException {
    double res = .0;
    res = Double.parseDouble(chaine);
    total += res;
}
```

```
public void saisie() {
    somme(chaine);
}
```



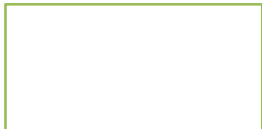
105



Ordre des blocs *catch*

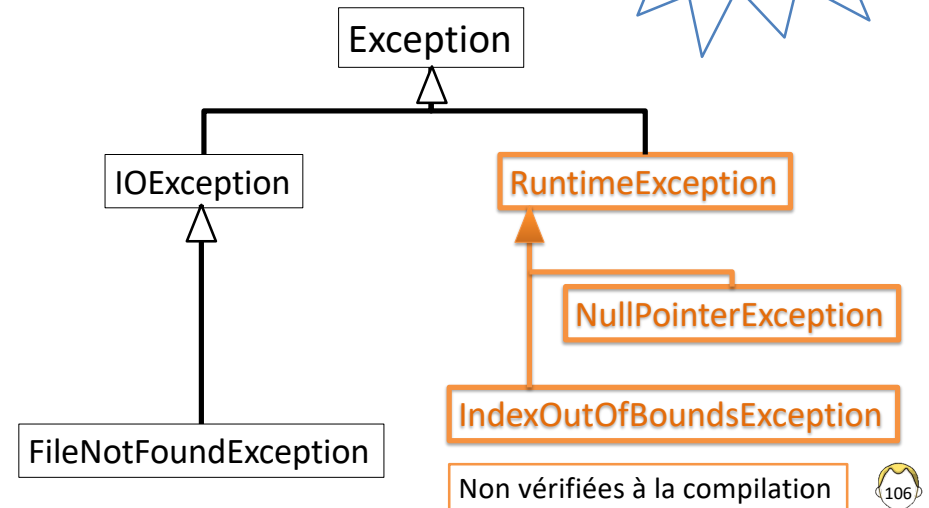
```
try {
    // code à tester
} catch (Exception e) {
    e.printStackTrace();
} catch (IOException e) {
    // traitement adapté
}
```

```
try {
    // code à tester
} catch (IOException e) {
    // traitement adapté
} catch (Exception e) {
    e.printStackTrace();
}
```



7

Hiérarchie des exceptions



106

Même traitement ?

```
try {
    // bloc à surveiller
} catch (NumberFormatException e) {
    e.printStackTrace();
    throw e;
} catch (IOException e) {
    e.printStackTrace();
    throw e;
}
```

Même traitement

```
try {
    // bloc à surveiller
} catch (NumberFormatException |
        IOException e) {
    e.printStackTrace();
    throw e;
}
```

7

108

Bloc *finally*

- Optionnel
- TOUJOURS exécuté
 - Même si aucune exception n'a été levée
 - Même si une instruction *continue*, *break* ou *return* se trouve dans le bloc *try*
 - Sauf fin de thread ou de JVM
- Utilité avec un langage doté d'un ramasse-miettes ???
 - Libérer les ressources
 - Fermer des fichiers, par exemple
 - *Try-with-resources*

7



```
class AutorisationException extends Exception {  
    public String getMessage() {  
        return "Op impossible : découvert trop grand";  
    }  
}
```

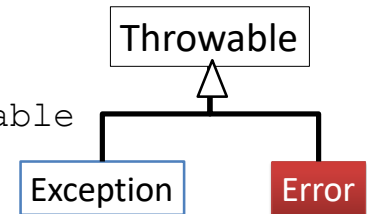
```
public class CompteBancaire {  
    double solde = 0.0;  
    double decouvert = -700.; // ... autorise  
  
    public void retrait(double montant)  
        throws IllegalArgumentException, AutorisationException {  
        if (montant < 0.0)  
            throw new IllegalArgumentException();  
        double nouveau = solde - montant;  
        if (nouveau < decouvert)  
            throw new AutorisationException();  
        solde = nouveau;  
    }  
}
```



111

Exception personnalisée

- Exception dérive de Throwable



- Dériver de `java.lang.Exception`
 - Surcharger Constructeur (`String message`)
 - OU redéfinir `getMessage()`
- Lancer une exception

```
throw new MonException();
```



Clonage

- Copier un objet pour ne pas le modifier
 - Pas de constructeur de copie
- Implémenter `Cloneable`
 - Sert seulement à prévenir le compilateur
- Appeler la méthode `clone()` de la classe mère en public
- S'assurer que la méthode `clone()` d'`Object` est également appelée en haut de l'échelle
- Traiter les exceptions dans `clone()`



```

class Trooper implements Cloneable {
    public Object clone() {
        Trooper object = null;
        try {
            object = (Trooper) super.clone();
        } catch (CloneNotSupportedException cnse) {
            cnse.printStackTrace(System.err);
        }
        // s'occuper des attributs "compliqués"
        // pour éviter la copie de surface
        // (shallow copy) si object != null

        return object;
    }
}

```



Copie des types primitifs
Copie des références
Objets non mutables (String)



Conclusion exceptionnelle



Pour toute exception déclenchée, le compilateur **impose** un traitement


1. Bloc `try/catch` qui gère cette exception
2. Passage de l'exception au niveau supérieur (appelant).
 - L'exception apparaît alors dans la **signature** de la méthode



```



public void lire(BufferedReader br)
    throws IOException
{
    String lecture;
    try {
        while((lecture= br.readLine()) != null) {
            S.o.p(lecture);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Java

5. Généricité

ISIMA

Programmation générique



- Écrire des types de données ou des algorithmes paramétrés par des TYPES

Musser, Stepanov, 1988

- Choix d'un type => instantiation
- Éviter la duplication de code
- Popularisée par les langages à objets

https://link.springer.com/chapter/10.1007/3-540-51084-2_2

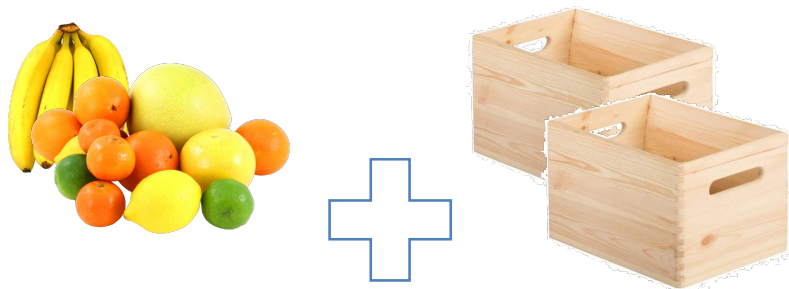


Généricité en Java

5

- Classes / interface paramétrée
 - Méta-modèle (pas métaclasse)
 - 1 définition, N invocations
- Méthodes paramétrées
- Gabarit / patron / *template* / **generics**
- Ajout fondamental au java 1.5
 - Incompatibilité avec code plus ancien ?
 - Type Erasure
 - Types primitifs exclus

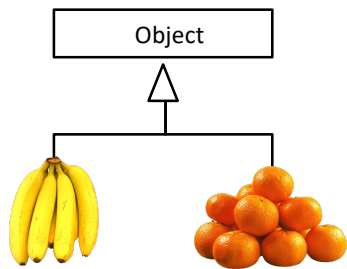
<https://www.oracle.com/technetwork/java/javase/generics-tutorial-159168.pdf>



```
class CaisseBanane {  
    Banane[] bananes;  
    CaisseBanane() {}  
    void ajouter(Banane b) {}  
    Banane enlever() {}  
    void porter()  
}
```

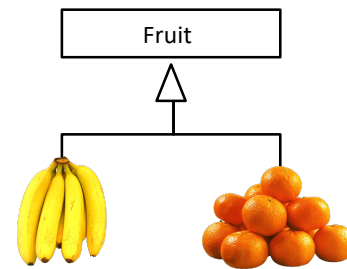
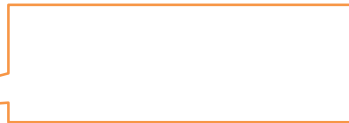
```
class CaisseOrange {  
    Oranges[] oranges;  
    CaisseOrange() {}  
    void ajouter(Orange b) {}  
    Orange enlever() {}  
    void porter()  
}
```





```

class Caisse {
    Object[] nimp;
    Caisse() {}
    void ajouter(Object b) {}
    Object enlever() {}
    void porter()
}
  
```



```

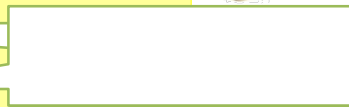
class CaisseFruit {
    Fruit[] fruits;
    CaisseFruit() {}
    void ajouter(Fruit b) {}
    Fruit enlever() {}
    void porter()
}
  
```



Condition sur un type ?

```

class Caisse<F> {
    F[] f;
    Caisse() {}
    void ajouter(F b) {}
    F enlever() {}
    void porter()
}
  
```



```

public class Classe<E> {
}
  
```



```

public class Caisse<F extends Fruit> {
}
  
```



```

public class Port<N extends Flottant> {
}
  
```

Méthode paramétrée

```
class Exemple {  
    public <T> void method1(T t) {  
    }  
  
    public <F extends M> void method2(F f) {  
    }  
  
    public static <T> T method3() {}  
}
```



Une interface utile !

```
interface Comparable<T> {  
    int compareTo(T t);  
}
```

```
new Integer(3).compareTo(new Integer(5));
```

```
new Integer(3).compareTo(new String("pas"));
```

```
class MaClasse  
    implements Comparable<MaClasse> {  
}
```



"Effacement" de type

- *Type erasure*
- Remplacement de type par un type borné ou *Object* à la compilation

```
UneClasse -> Object  
Comparable<MaClasse> -> Comparable
```

- Conversion si nécessaire
- Méthodes supplémentaires (*bridge*)
- Pas de surcoût à l'exécution



Problèmes ?



```
class ArrayList<E> implements List<E> {}  
List<String> l1 = new ArrayList<String>();  
List<Object> l2 = l1;  
List<?> l3 = l1;
```

```
class Forme {  
    public void afficher();  
}  
class Cercle extends Forme {}  
class Rectangle extends Forme {}
```

```
public afficherTout(List<Forme> l) {}
```

```
public afficherTout(List<? extends Forme> l) {}
```



Aller plus loin...



- Réécrire son code 1.4 en générique ou l'inverse
- Comparer les invocations ?

```
ArrayList<String> l1;  
ArrayList<Integer> l2;  
l1.getClass() == l2.getClass();
```

- Créer un objet ou un tableau d'objets ?

```
T t = new T();  
T[] a = new T[10];
```

- Dépendance de plusieurs types

```
public <F extends Object & M1 & M2> void m(T t) {}
```

1^{er} type utilisé pour l'erasure



Collections

- Interfaces / implémentations
 - Développer plus vite avec des outils fiables
 - Paquetage `java.util`
- Collection = groupe d'**éléments**
 - Peut être parcourue (itérable)
 - Générique
 - Sérialisable ?
- Algorithmes
 - Tri / Recherche / Manipulation



Conteneur



6. Collections

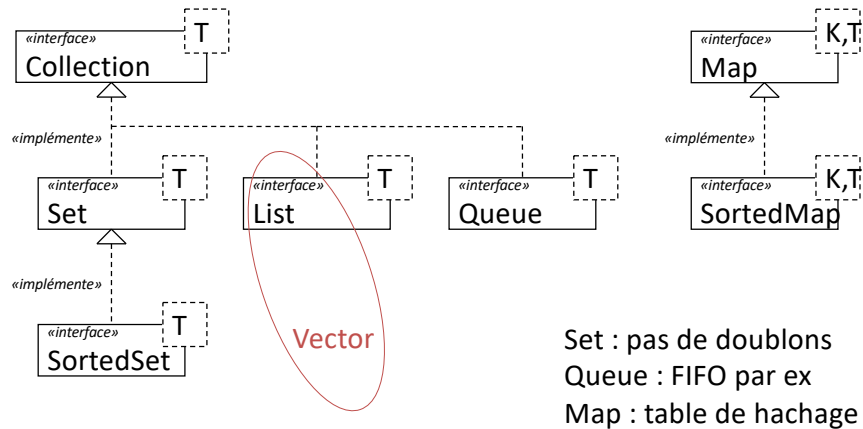


ISIMA

Types de collections

- *Backing collections*
 - "Classique"
 - Stockage d'éléments
- *View collections*
 - Pas de stockage
- *Unmodifiable Collections*
 - Les éléments peuvent être mutables
- Supportent la concurrence
- Ou pas ☹

Types de données ?



133

Quelques implémentations

General purpose implementations					
Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue, Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

ArrayList or Vector ?

134

Exemple

```

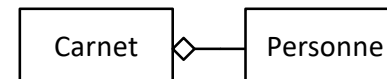
Vector<Integer> v = new Vector<Integer>();

for (int i=0; i<10; ++i)
    v.addElement(new Integer(i));
// plus de transtypage

int somme = 0;
for (int i=0; i<10; ++i)
    somme += v.elementAt(i).intValue();

// avec une énumération, elle aussi paramétrée
Enumeration<Integer> e = v.elements();
while (e.hasMoreElements())
    somme += e.nextElement().intValue();
  
```

135



```

class Personne {
    String nom;
    String telephone;

    equals()
    hashCode()
}
  
```

```

class Carnet {
    // agrégation

    public Carnet() {
    }
}
  
```

Personne[] p1;

List<Personne> p2;

Set<Personne> p3;

Pas de duplication d'élément

p1 = new Personne[50];

p2 = new
ArrayList<Personne>();

p3 = new
HashSet<Personne>();

136



7. Plus de langage...



Terminologie : classe imbriquée

- Classe imbriquée / *nested class*
 - Classe définie à l'intérieur d'une autre classe
 - 4 types de *nested class*
 - Classe membre statique
 - Classe membre non statique
 - Classe locale (définie dans une méthode)
 - Classe anonyme (locale sans nom)
- } *Inner class*
- N'existe qu'avec une instance de la classe
 - Pas de membres statiques
 - Peut accéder aux variables / attributs de l'englobant

138

Classes imbriquées

```
class E {  
    static class S {  
    }  
  
    class I {  
    }  
}
```

```
E.S s = new E.S();
```



```
E e = new E();  
E.I i = new e.I();
```

139

Classe anonyme (1)



```
interface Tester {  
    public boolean test(Pokemon p)  
}
```

```
banc.enlever(new IsKO());
```

Enlever du banc tous les pokemons "testés"...

```
class IsKO implements Tester {  
    public boolean test(Pokemon p) {  
        return p.getPV() <= 0  
    }  
}
```

140

Classe anonyme (2)



- Déclarée à la volée / utilisée qu'une seule fois
- **Spécialise** la classe donnée ou **implémente** l'interface donnée



```
banc.enlever(new Tester() {  
    public boolean test(Pokemon p) {  
        return p.getName().equals("pikachu");  
    }  
});
```

- Peut capturer l'environnement d'appel
- Compilée avec un nom arbitraire
nomclasseenglobante\$nombre.class



Lambda

- Classe anonyme avec une seule méthode ?
- Closure (environnement ?)

```
banc.enlever(  
    (Pokemon p) -> p.getName().equals("pikachu")  
);
```

- Parenthèses non obligatoire si 1 argument inférable
- Return implicite si une seule instruction

<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>



Énumération ? (<1.5)

- Ensemble de valeurs données

```
public static final int LUNDI    = 0;  
public static final int MARDI    = 1;  
public static final int MERCREDI = 2;  
public static final int JEUDI    = 3;
```



Des énumérations ?

5

```
enum Semaine { LUNDI, MARDI, MERCREDI,  
    JEUDI, VENDREDI, SAMEDI, DIMANCHE}
```

```
for (Semaine jour : Semaine.values())  
    System.out.println(jour);
```

```
for (Semaine j : EnumSet.range(  
    Semaine.LUNDI,    Semaine.VENDREDI))  
    System.out.println(j);
```

Dangereux mais utile : le static import

<https://docs.oracle.com/javase/1.5.0/docs/guide/language/enums.html>



Des enum OK, mais des objets !

```
enum Nom { VAL1(1), VAL2 (2);
    private int valeur;
    Nom(int i) { this.valeur = i };
}
```

```
enum Nom {
    VAL1 { retour methode(params) {...} },
    VAL2 { retour methode(params) {...} };

    abstract retour methode(params);
}
```



Exemples d'annotations

- @Override
- @SuppressWarnings
 - @SuppressWarnings(value="deprecation")
 - @SuppressWarnings("deprecation")
 - @SuppressWarnings({"unchecked", "deprecation"})
- @Deprecated
- @Documented
 - Documentation
- @Retention(RetentionPolicy.RUNTIME)
 - Exécution



Annotation

- Méta données
 - Développeur
 - Compilateur : génération de code, documentation
 - Déploiement
 - Machine virtuelle
 - Configuration
- Plus simple et plus léger que le XML
- Utilisation intensive
 - JUnit
 - Java EE dont Spring, JPA



```
class Mere {
    public void methode() {
        System.out.println("Methode de Mere");
    }
}
```

```
class Fille extends Mere {
    @Override
    public void Methode() {
        System.out.println("Methode de Fille");
    }
}
```

```
Fille f = new Fille();
f.methode();
```

Annotations personnalisées

```
@interface ClassPreamble {
    String author();
    String date();
    int currentRevision() default 1;
    String lastModified() default "N/A";
    String lastModifiedBy() default "N/A";
    String[] reviewers();
    // utilisation possible des tableaux
}
```

<https://docs.oracle.com/javase/tutorial/java/annotations/declaring.html>



Classes scellées (1)

- Limiter les classes dérivées

```
public sealed class Forme
    permits Rectangle, Cercle {}
```

```
public sealed interface Forme
    permits Rectangle, Cercle {}
```

- Trois stratégies pour les classes filles



Méthode à nombre d'arguments variable

```
public void somme(double ... nombres) {
    double s = 0.0;

    for(int i=0; i < nombres.length; ++i)
        s += nombres[i];

    for (double nb: nombres) s+= nb;
}
```



Classes scellées (2)

```
public final class Cercle
    extends Forme {
}
```

Classe finale

```
public non-sealed class Cercle
    extends Forme {
}
```

Classe ouverte

```
public sealed class Cercle
    extends Forme
    permits Ovale {
}
```

Classe scellée



Records

```
public record Personne  
    (String nom, String prenom) {}
```

- Aide à la création d'objets non mutables
 - Code par défaut
- Encapsulation ?
 - « État et seulement état de l'objet »
- Classe non dérivable
- Ne peut étendre d'autres classes
- Peut implémenter des interfaces
- On peut ajouter des méthodes et des attributs de classes



Au chargement de la classe...

- Initialisation des attributs de classes

```
class Exemple1 {  
    static int[] tab1 = new int[20];  
}
```

- Instructions spécifiques exécutées au **chargement** de la classe dans la JVM

```
class Exemple2 {  
    static int[] tab2;  
    static {  
        // exécuté au chargement de la classe  
        tab2 = new int[20];  
        for(int i=0; i<20; ++i) tab2[i] = 2*i;  
    }  
}
```



Equivalent du "record"

```
final public class Personne {  
    final String nom;  
    final String prenom;  
  
    public Personne(String nom, String prenom) {}  
  
    public prenom() {}  
    public nom() {}  
  
    public boolean equals(Object o) {}  
    public int hashCode() {}  
  
    public String toString() {}  
}
```

Attributs non
modifiables

Getters avec nom
non standards



Reflection



- Introspection

- Informations sur la classe à l'exécution

```
o instanceof Point  
o.getClass().getName()
```

- Gestion dynamique des classes

```
Class.forName("MaClasseInconnueALaCompil")
```



- Intercession

- Modification dynamique des classes
- Gestionnaire de sécurité
- Limité



Classes disponibles

- AccessibleObject

- Class

- Connaître le type et la classe mère
- Lister les méthodes / constructeurs / interfaces
- Lister les attributs
- Lister les annotations (pas toutes)
- Chargement dynamique

- Constructor

- Method

- Field



Programmation fonctionnelle ?

- Lambdas
- Interface fonctionnelle
- Package java.util.function
- Référence de fonction
- Streams



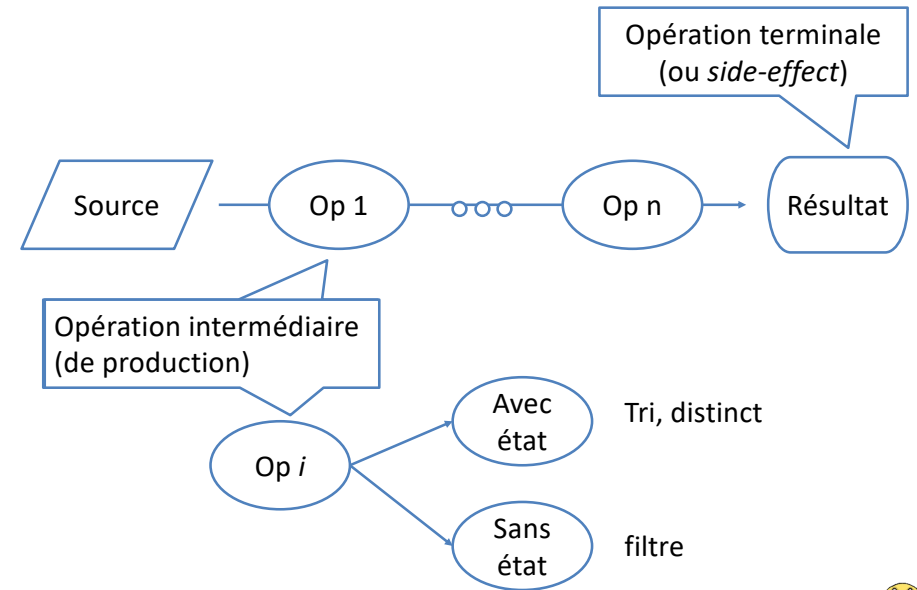
9. Programmation fonctionnelle
Streams



Streams

- Suites d'opérations à effectuer
 - Pas de stockage
 - Les données viennent d'une collection / source
- Parallélisation possible
- Evaluation paresseuse si possible (efficacité)
- Traitement sur partie des données si possible
- Streams
- Int/Long/DoubleStream

<https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/stream/package-summary.html>



Exemple

```
List<Integer> liste =  
    Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

```
System.out.println(liste);
```

```
liste.stream()  
    .forEach(System.out::println);
```



```
var l = liste.stream()  
    .collect(Collectors.toList());
```



Interface fonctionnelle

- Une unique méthode abstraite
- Avec annotation

```
@FunctionalInterface  
interface MonInterface {  
    public abstract void apply(Object);  
}
```
- Ou sans annotation si le compilateur le décide
- Très utilisée avec lambda et références de méthode



Filtrage par stream

- Predicate<T> -> boolean test(T t)

```
stream.filter(n-> n% 2 ==0);
```

```
stream.filter(new Predicate<Integer> () {  
    public boolean test(Integer i) {  
        return Integer.valueOf(i) % 2 == 0;  
    }  
});
```



Gestion des entrées/sorties (1)

- Flux E/S de données binaires
- Flux E/S de caractères
- Flux E/S d'objets
- Communication avec des fichiers
- Communication avec des ressources Internet
- Sérialisation (=> réseau)

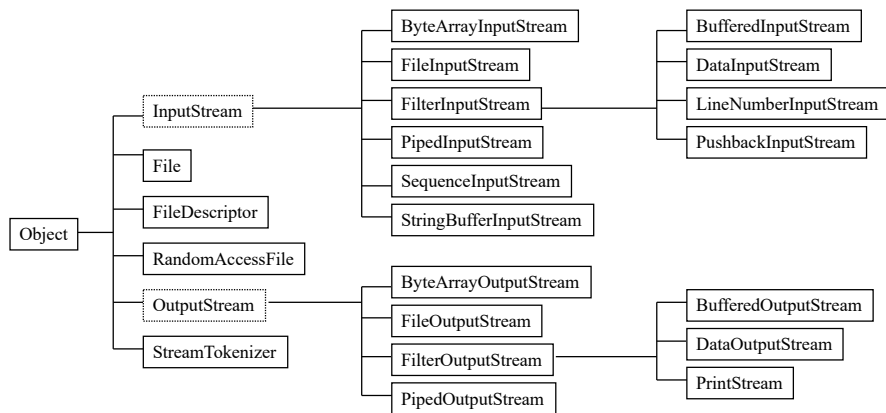


Gestion des entrées/sorties (2)

- java.io
 - Flux de données (fichier, pipe/threads, ...)
 - Sérialisation
 - Système de fichiers
- java.nio (java 4)
 - Mémoire tampon Buffer
 - Canaux + Sélecteurs : hautes performances
 - Traduction des jeux de caractères
- java.nio2
 - Simplification
 - Path



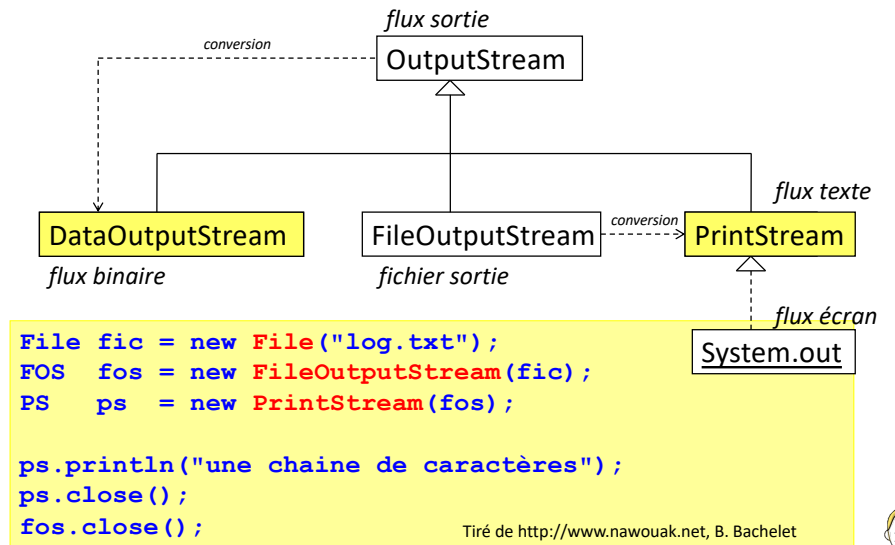
java.io.*



Tiré de « Eléments de programmation JAVA », Olivier Dedieu, INRIA



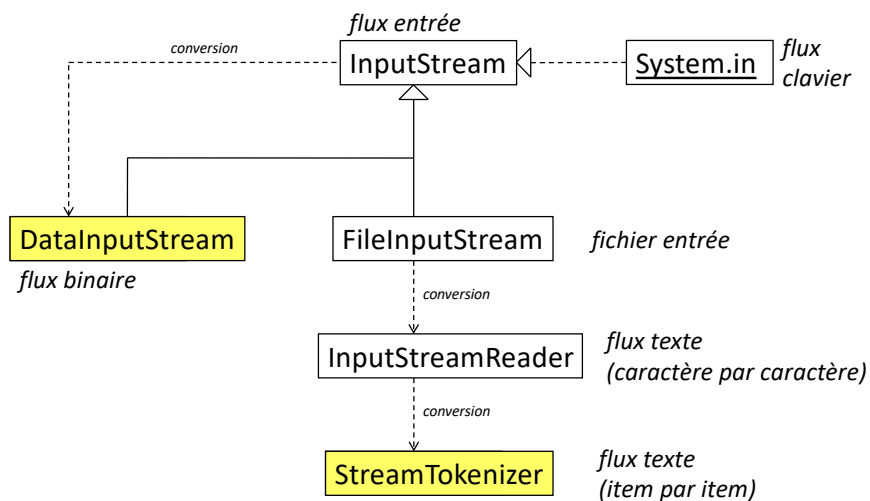
Flux de sortie



Tiré de <http://www.nawouak.net>, B. Bachelet



Flux d'entrée



Tiré de <http://www.nawouak.net>, B. Bachelet



Saisie clavier

```

// Version 1 standard
// Attention aux exceptions soulevées
InputStreamReader isr =
    new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
br.readLine();
  
```

```

// Version 1.6
// les exceptions sont gérées par la Console
System.console().readLine();
  
```

System.console() == null sous Eclipse !

6



try {

7

Try-with-resources

```
BufferedReader br = null;
FileReader fr = null;
try {
    fr = new FileReader(nomFichier);
    br = new BufferedReader(fr);
    String lecture;
    while ((lecture = br.readLine()) != null) {
        //
    }
    br.close();
    fr.close();
} catch (Exception e) {
    e.printStackTrace();
    if (br != null) br.close();
} finally {
    if (br != null) br.close();
    if (fr != null) fr.close();
}
```

Exceptions possibles ?

Pas la bonne place

Pas la bonne place non plus = duplication de code

Bonne place Mais exception possible

} catch(Exception e) {

173

- Ressource
 - Tout objet à fermer / libérer après utilisation
 - Fichier, flux, chaussette, requête
- Méthode classique (< 1.7)
 - `close()` dans bloc `finally` => exception levable
 - Bloc `try-catch` englobant supplémentaire ou relance
- Fermeture et libération automatique (1.7+)
 - Syntaxe simplifiée
 - Interface `AutoCloseable`

7

174

Sérialisation (1)

```
// bloc try-with-resources
try (
    FileReader fr = new FileReader(nomFichier);
    BufferedReader br = new BufferedReader(fr);
){
    String lecture;
    while ((lecture = br.readLine()) != null) {
        //
    }
} // Les ressources sont fermées automatiquement
catch (Exception e) {
    e.printStackTrace();
}
```

Certaines exceptions peuvent être lancées mais elles sont dissimulées. Elles sont toutefois récupérables en cas de besoin.

175

- Transformer un **objet** présent en mémoire en bits
 - Sur un disque (Stockage - Persistence)
 - Sur le réseau (Communication – RMI)
 - Les infos de classe ne sont pas transmises
- Implémenter l'interface *Serializable*
 - Ne fait rien, prévient le compilateur
- Proposer une version de sérialisation
 - `static final long serialVersionUID = 110L;`
- Attention à la protection des données
 - Données *transient* : données non copiées
 - Ou implémenter *Externalizable*

176

Sérialisation (2)

Méthodes à redéfinir pour un comportement particulier

```
void writeObject(ObjectOutputStream out)
    throws IOException;
void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException;
void readObjectNoData()
    throws ObjectStreamException;
```

Flux (streams) à utiliser ...

```
FileOutputStream/FileInputStream
ObjectOutputStream/ObjectInputStream
```



Sérialisation en XML

- XMLEncoder pour les objets respectant les conventions NetBeans
- X-Stream pour les autres ;-)
 - Une bibliothèque tiers sur github

```
<personne>
  <nom>yon</nom>
  <prenom>loic</prenom>
</personne>
```



```
FileOutputStream fos = null;
ObjectOutputStream oos = null;
try {
    fos = new FileOutputStream("fichier.dat");
    oos = new ObjectOutputStream(fos);
    oos.writeObject(objects);
    oos.flush();
} catch (IOException e) {
    e.printStackTrace();
} finally {
```

Écrire

```
    if (FileInputStream fis = null;
        ObjectInputStream ois = null;
    try {
        fis = new FileInputStream("fichier.dat");
        ois = new ObjectInputStream(fis);
        objects = (Composite) ois.readObject();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (ois != null) ois.close();
        if (fis != null) fis.close();
    }
}
```

Classe de l'objet

Lire



```
FileOutputStream fos = null;
XStream xstream = null;
try {
    fos = new FileOutputStream(name);
    xstream = new XStream(new StaxDriver());
    xstream.toXML(objects, fos);
} catch (Exception e) {
    e.printStackTrace();
} finally {
    if (fos != null) fos.close();
}
```

Écrire

```
FileInputStream fis = null;
XStream xstream = null;
try {
    fis = new FileInputStream(name);
    xstream = new XStream(new StaxDriver());
    objects = (Composite) xstream.fromXML(fis);
} catch (Exception e) {
    e.printStackTrace();
} finally {
    if (fis != null) fis.close();
}
```

Lire

Classe de l'objet





Bibliographie (1)

- Documentation JAVA de Sun/Oracle
- Tutoriaux Web de Sun/Oracle
<http://download.oracle.com/javase/tutorial/>
- Cours JAVA d'Olivier Dedieu, INRA
<http://www-sor.inria.fr/~dedieu/java/cours/>
- Intro JAVA, Bruno Bachelet
<http://www.nawouak.net/?doc=java+lang=fr>



Bibliographie (2)

- Thinking in Java, 2nd ed, Bruce Eckel
- Head First Java, 2nd ed, Kathy Sierra, Bert Bates , O'Reilly, 2005



Aller plus loin...



- Java standard
 - Réseau, RMI
 - Threads avancés
 - JNLP
- Outils
 - Ant, Maven
- Java Entreprise
 - Glassfish, tomcat
 - Servlets, jsp, beans
 - Persistance (JPA, Hibernate)
 - Facelets (JSF)
 - Frameworks : Struts, Spring, ...

