

Objectifs

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

JUnit 5

Pré-requis

- Syntaxe C
- Concepts objets

Cadre

- Le Java vu par SUN et uniquement

Sun Microsystems

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 30
3. Concepts objets en Java 49
4. Les exceptions 102
5. Généricité 118
6. Collections 132
7. Plus de langage 139
8. Introspection 158
9. Programmation fonctionnelle –Streams 162
10. Fichiers & flux, sérialisation 169



Introduction



- Île ?
- Javascript (ECMA) ?

- Langage
- Machine virtuelle
- Plateforme



Motivations

- Langage créé en 1995
  - Patrick Naughton
  - James Gosling

Sun ORACLE

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

Concepts difficiles du C++ évacués

1ER PROGRAMME





CLERMONT AUVERGNE INP ISIMA

Premier programme (Java 25+)

```
/* mon premier code */

void main() {
    // afficher un message
    IO.println("Bonjour");
}
```

Fichier source (texte) *cequelonveutici.java*

Par exemple: Bonjour.java

Pour voir le résultat…

1. Compiler le programme

```
javac Bonjour.java
```

2. Lancer le programme

```
java Bonjour
```

.exe ? .class ?

Et si j'ai pas Java 25 ?

```
/* ma première classe */
public class Exemple {

  public static void main(String[] argv) {

    // afficher un message
    IO.println("Bonjour");
  }
}
```

Fichier source (texte) *Exemple.java*

Fichier source

- Extension : .java
- Nom du fichier = nom de la classe publique
- Respecter la casse Exemple
- 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

Java 7 hotspot
250 000 lignes
C / C++

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

```
java Exemple
```

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

https://github.com/openjdk/jdk







S'il y a plusieurs main() ?

Classe implicite ou publique

Héritées ou déclarées

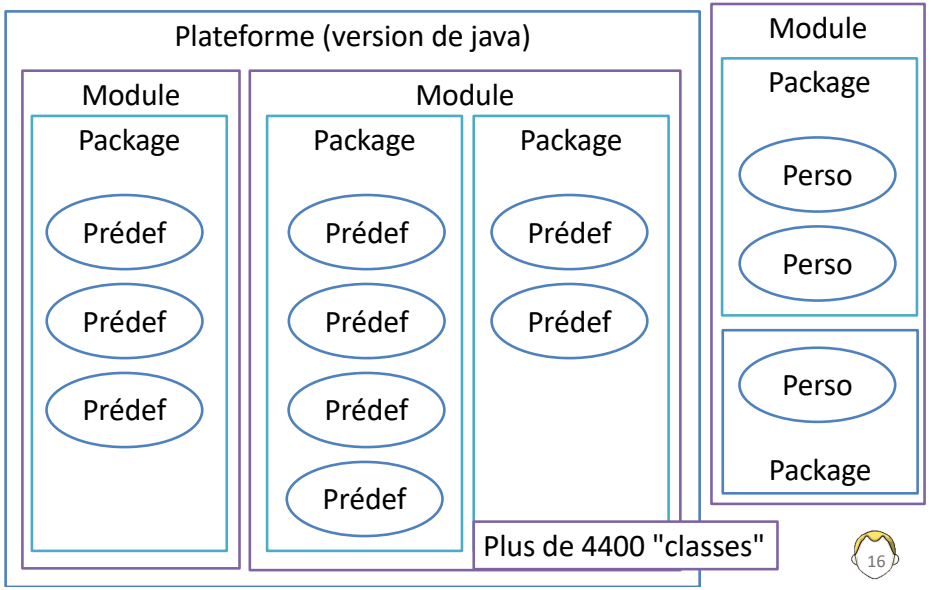
```
void main(String args[])
static void main(String args[])
void main()
static void main()
```

launchable main

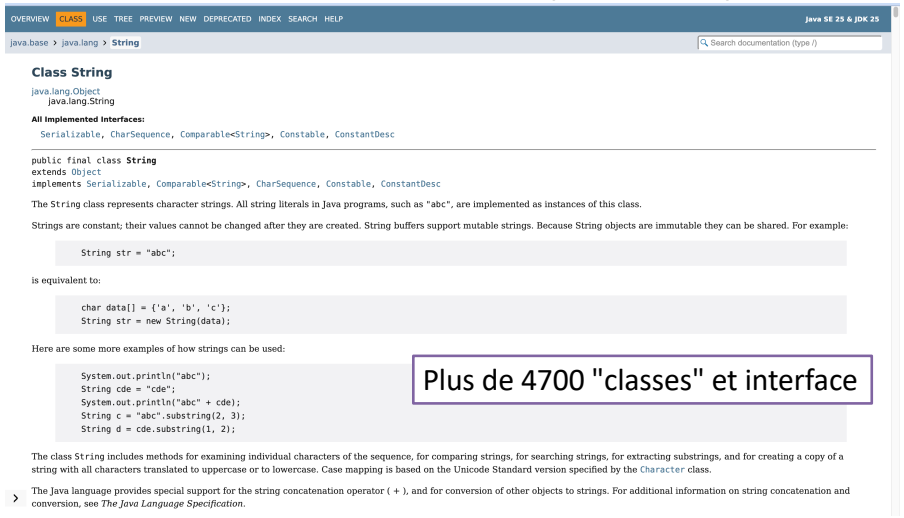
https://openjdk.org/jeps/512

Un langage à objets ?

Classe



Documentation (11->26)



Plus de 4700 "classes" et interface

https://docs.oracle.com/en/java/javase/25/docs/api/

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)

Plateforme

- Personnel / Tiers

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)

Clause import

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

ArrayList ⇒ **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 (1)

classe ⊂ package ⊂ module

- Ensemble de packages
- Projet Jigsaw – Java 9

Pas de rétrocompatibilité

- Charger une JVM adaptée
  - Plus rapide, plus économe
- Regroupés par famille
  - `java : java.base`
  - `jdk`
  - `Autres`

Java 11-13
21
37
1

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

```
import module java.base; // JDK 25+
```


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
```



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]





- 1.8 – 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



- 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 [2023]
  - Virtual threads, Sequenced collections
  - Simplification main()



- 1.22 [2023]
  - record
- 1.23 [2024]
- 1.24 [2024]
- 1.25 – version LTS [2025]
  - Plus de 32 bits
  - Main / modules
  - Code avant this() ou super()
- 1.26
- 1.27 [2026]

Distribution ?

- Usage
  - Exécution seule (JRE)
  - Développement (JDK < 2, SDK v ≥ 2)
  - Modules externes complémentaires ?
- Cibles
  - **Standard** **Java SE** / J2SE
  - Entreprise **Java EE** / J2EE / Jakarta EE
  - Micro **Java ME** / J2ME (v ≥ 5)

Télécharger le JDK pour java SE / openjdk

Spring[Boot] Websphere GlassFish, Jboss, …

Tutoriels officiels Java



- Bases du Java

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

- https://dev.learn.java





{ Accolades } et commentaires

```
public class Exemple {

   public static void main(String[] argv) {
     //  afficher un commentaire monoligne
     /*  commentaire
         sur plusieurs lignes              */
     /** commentaire javadoc (≈ doxygen) */
     /// commentaire Markdown (Java 25)
   }
}
```

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

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"

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)



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

```
void main() {
  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"

Manipulation de variables primitives

```
void main() {
  int i = 0;
  i = i + 1;
  i += 1 ;
  i *= 2 ;

  IO.println(i) ;
  IO.println(++i) ;
  IO.println(i) ;
  IO.println(i++) ;
  IO.println(i) ;

  i = (int) 10.6;
}
```

Exemple de classes : les chaînes de caractères

- `String ≠ char[]`

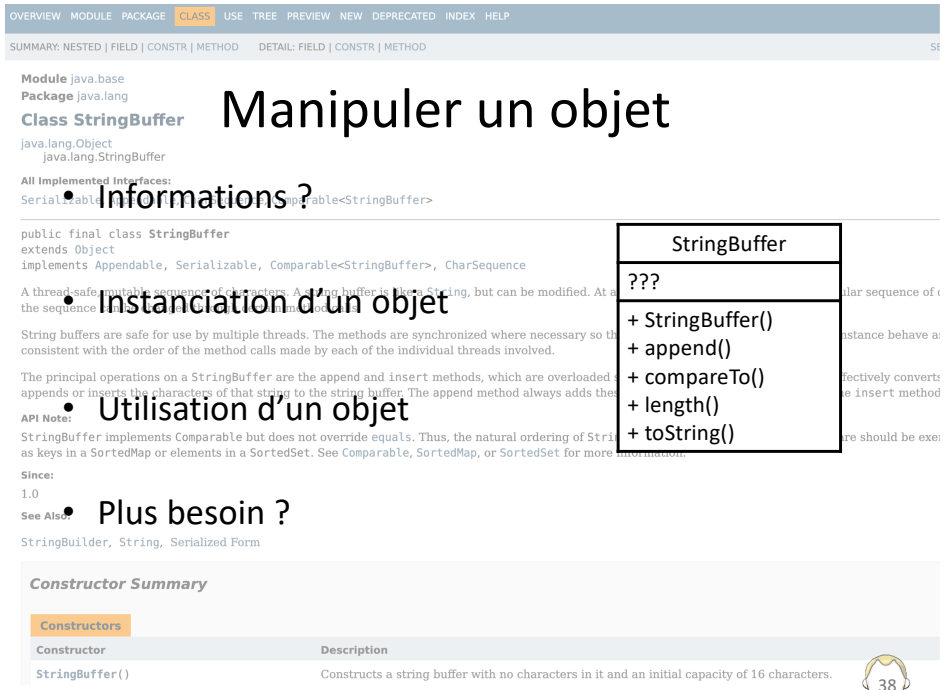
- **constante** `String`
- **modifiables** `StringBuffer` `StringBuilder`

- UTF-16
- Bibliothèque fournie
  - Comparaison de chaînes : equals( ), compareTo( )
  - Recherche : indexOf( )
  - Extraction : substring( ), StringTokenizer, split, regexp
  - Transformation aisée de type scalaire vers *StringBBBB*

Manipuler un objet

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

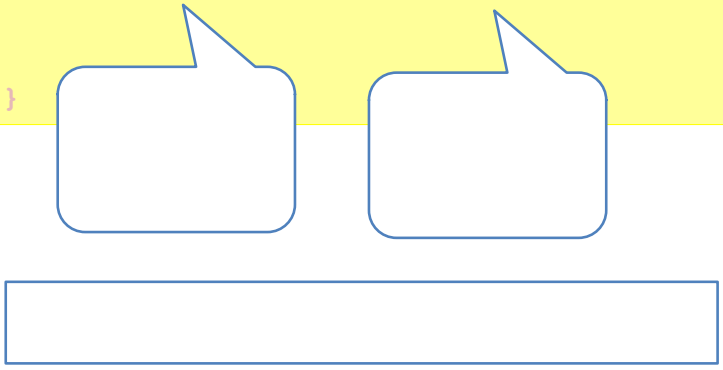
StringBuffer
???
+ StringBuffer() + append() + compareTo() + length() + toString()



Instanciation d'objet

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


}
```



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
  - Plus utilisée (optionnel)

```
StringBuffer s;
s = null;
```

- Affection à une valeur admissible
  - Allocation mémoire

Toujours possible ?

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

Manipulation d'objet (1)

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

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

Essayer si s1 est nulle

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

```
IO.println(s1.toString());
IO.println(s1.length());
IO.println(s1.capacity());
```

```
StringBuffer s2 = s1;
IO.println(s2.toString());
```


Manipulation d'objet(2)

```
IO.println(s1 == s2);
IO.println(s1.compareTo(s2));
IO.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)
   IO.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)   { … }
}
```



3. CONCEPTS OBJETS





CLERMONT AUVERGNE INP UCA Isima

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 …


```
public class Etudiant {

  private String nom;
  int id;


  public String getNom() {
     return nom;
  }

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

VISIBILITÉ : public / ⃠ (package) / private

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




```
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);

  }
}
```



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


```
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);
  }
```



```
public class Cours {
  // …
  public static void main(String[] argv) {
     Cours c1 = new Cours();

     Cours c2 = new Cours(12, true);


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

     IO.println(c2);
     IO.println(c2.toString());
  }
}
```

Membres de classe

```
public class Etudiant {

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

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




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

  Etudiant e = new Etudiant();

  // IO.println(compteur);
  // IO.println(getCompteur());

  // IO.println(e.compteur);
  // IO.println(e.getCompteur());

  IO.println(Etudiant.compteur);
  IO.println(Etudiant.getCompteur());
}
```




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)



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;
```

```
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);
   }
}
```



Digression[1]

Afficher **i et t[4]**


```
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;
   }

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

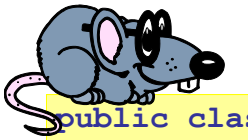


Digression[2]



Passage des paramètres

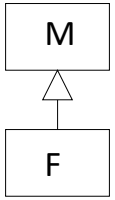
- Passage des paramètres par valeur/copie
  - Evident pour types primitifs
  - Toujours ?
- Copie de référence/pointeur
  => pas de copie d'objet
  - Tableaux
  - Objets



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 ?
  }
}
```

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
    // initial...tions spécifiques
  }
}
```


Héritage sélectif

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

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

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

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


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()

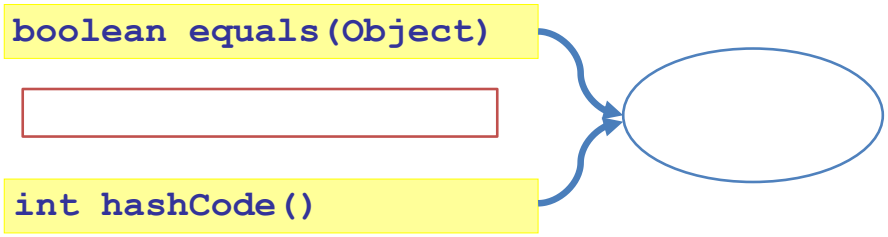
`Classe@hashcode`

- getClass().getName()
  - Connaître à l'exécution le nom de la classe



`objet instanceof Classe`

java.lang.Object (2)

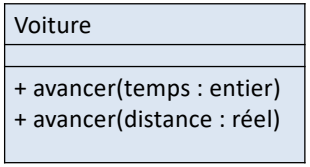


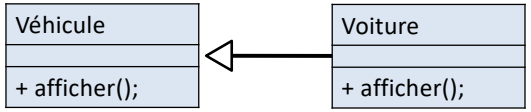
- Deux objets égaux ont le même hashcode
- Ne doit pas changer pour une exécution
- Deux objets distincts peuvent avoir le même
- Souvent l'adresse mémoire de l'élément



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





Parentalité (1)



```
public class M {
  private String a;

  public M(String s) {
    a = s;
  }
  public void m() {
   S.o.p(a);
  }
}
```

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

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


Parentalité (2)



```
public class M {
  private String a;

  public M(String s) {
    a = s;
  }
  public void m() {
   S.o.p(a);
  }
}
```

```
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



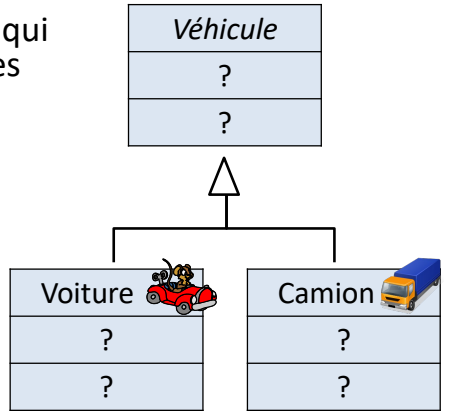
Noms qualifiés

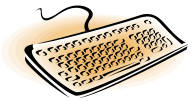
```
class M {
  protected int a;
}
public class F extends M {
  protected double a;
  public void toto() {
    a
    this.a
    super.a
    ((M)this).a
    ((F)this).a
  }
}
```



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");
  }
}
```



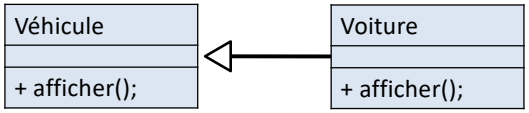
Le polymorphisme marche directement !

Appeler les méthodes afficher() des objets

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

Méthode virtuelle ?

- Compilation
  - Validité du message
  - Version inconnue ?



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


```
public class Vehicule {
  public void afficher() {
    IO.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() {
    IO.println("Sportive");
  }
  public void special() {
    IO.println("special");
  }
}
```



Que se passe-t'il ?

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



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

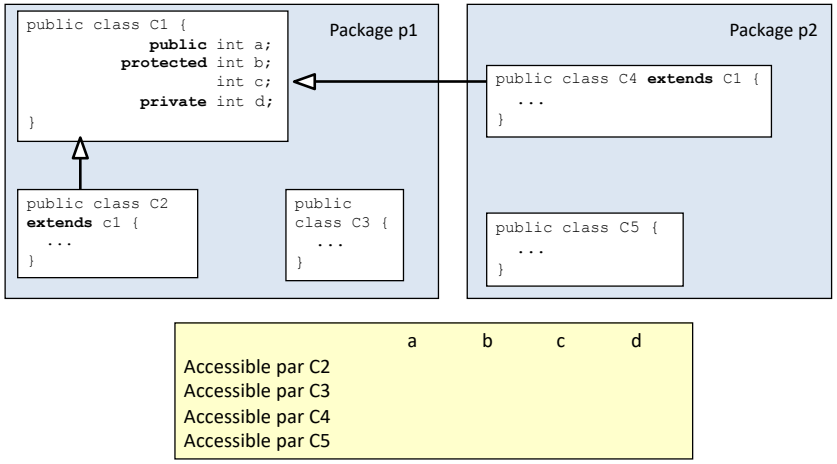
Dernier mot : la JVM hotspot
avec ses nombreuses optimisations

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



Encapsulation & visibilité



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



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


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

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

```
class Voiture1a extends Vehicule1 {
}
```

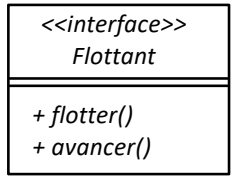
```
ERREUR : must implement the inherited method
```

```
abstract class Voiture1b extends Vehicule1 {
}
```

```
class Voiture1c extends Vehicule1 {
   public void afficher() {}
}
```




Interface (1)

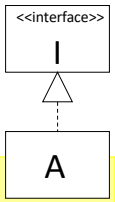


- Description / contrat
  - Liste de méthode(s) sans code
  - "Constantes" autorisées (*public static final* par défaut)
  - Pas de variable/attribut [UML]
  - « Classe virtuelle pure » [C++]
- Respecter le contrat = IMPLEMENTER l'interface

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

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

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();
}

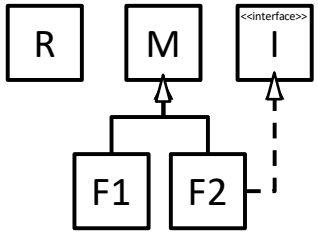
class Waymo extends Vehicule implements Autonome {
  public void rouler() {
    IO.println("Pilote automatique");
}}
```

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();
    IO.println(Autonome.COMPLET);
    IO.println(Waymo.COMPLET);
}
```



Waymo
(Google car)



instanceof

```
a instanceof X ?
```

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

Implémentation multiple

```
class M {
  void mM() {}
  void m1() {}
}
```

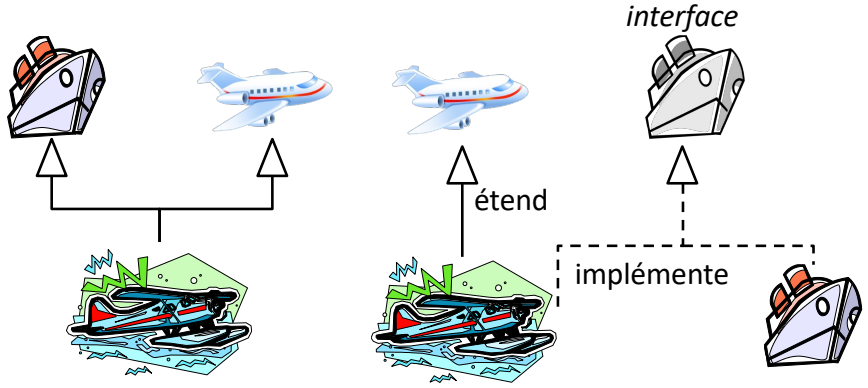
```
interface IC {
  void mIC();
  void m1();
  void m2();
}
```

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

```
public class F extends M implements IC, ID {
  public void mIC() {}
  public void mID() {}
  public void m2 () {}
  @Override
  public void m1 () {}
}
```

Héritage multiple ?

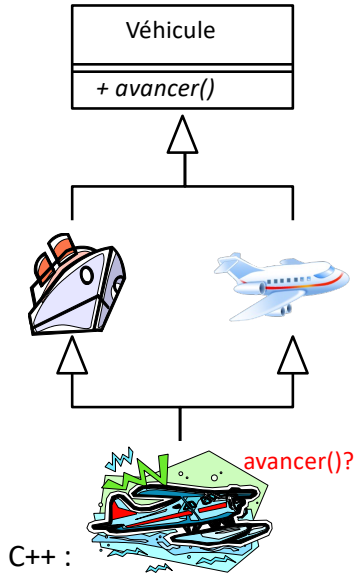
Relation non symétrique
= raison fonctionnelle



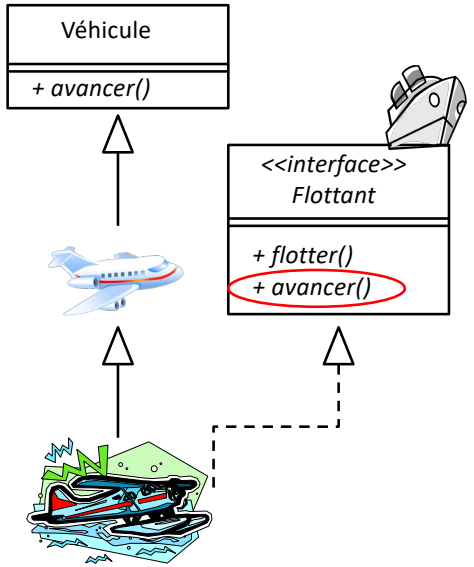

```
class H extends A, B {
      …
}
```

```
class H extends A implements F {
      …
}
```

```
class H extends B implements V {
}
```



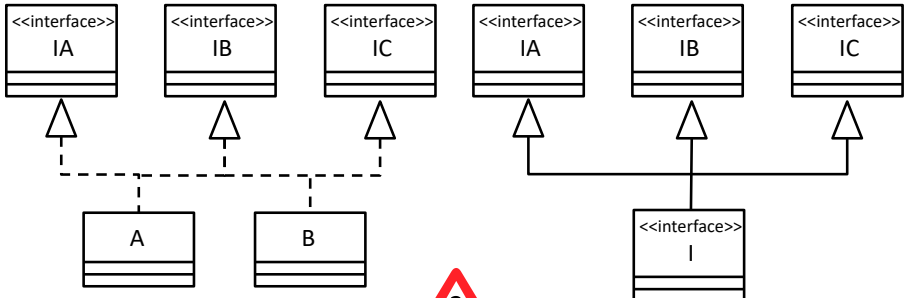
C++ :
- héritage virtuel
- conflit de méthode ?
- conflit d'attribut ?



pas de conflit



Héritage multiple d'interface




```
interface I
  extends IA, IB, IC {
  // …
}
```


Héritage multiple d'implémentation

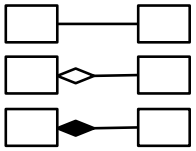
8

```
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() {}
}
```

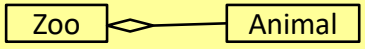
```
public class Voiture2 {
   private Moteur m;

   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;
}}
```






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

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

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


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



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 **final**e 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

```
class CoursGenial
```

- Attribut écrit en minuscule
  - Pas de tiret
- Méthode
  - Verbe pour action
  - Premier mot en minuscule
  - Majuscules à la première lettre des mots suivants

```
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
```

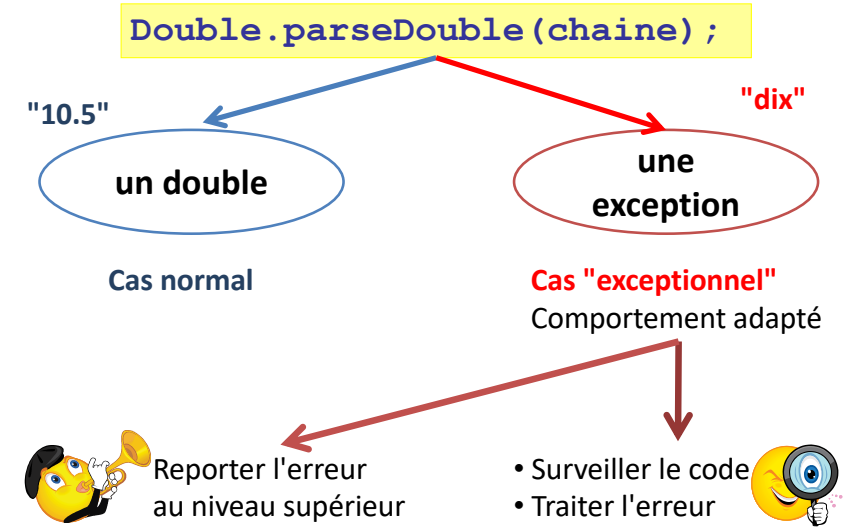



Exceptions

- Manière élégante et efficace de gérer les erreurs potentielles d'exécution

- Une erreur potentielle ≡ une exception
- Hiérarchie des exceptions
- Une erreur = une instance d'exception

- Partie intégrante de la signature d'une méthode
- Obligation de gérer les exceptions



OBLIGATION DE TRAITER UNE EXCEPTION

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();
  }
}
```



Attraper une exception

```
public void somme(String chaine) {
  double res = .0;
  try {
    // bloc à surveiller
    res = Double.parseDoub
    total += res;

  } catch (NumberFormatException e) {
    IO.println(e.getMessage());
    // ou e.printStackT
  } finally {
    // Clause TOUJOURS exécutée
  }
}
```



Reporter l'erreur …

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

}
```

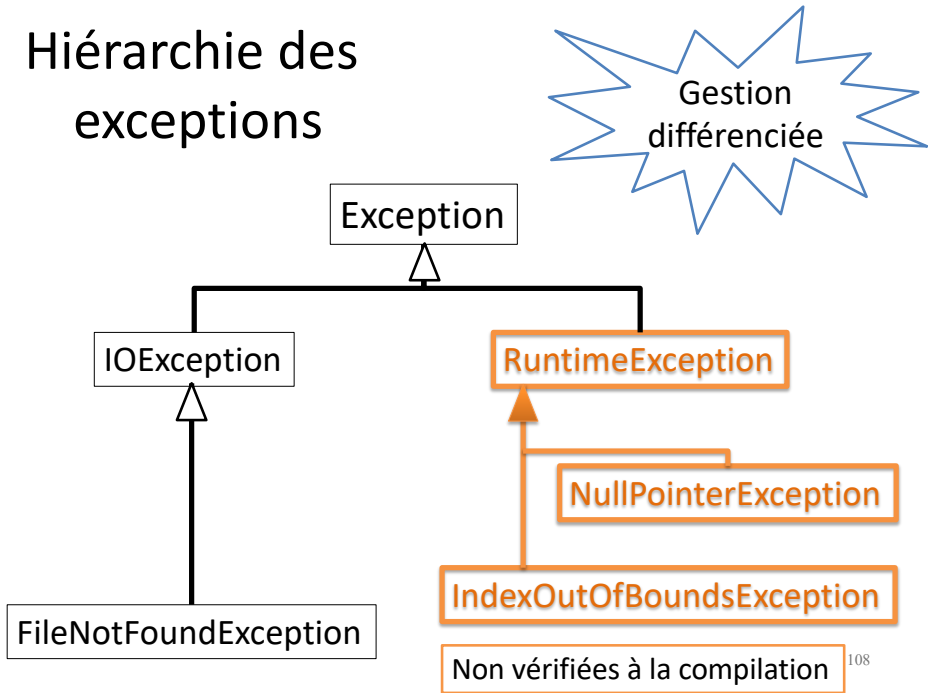
```
public void saisie() {

    somme(chaine);

}
```



Hiérarchie des exceptions

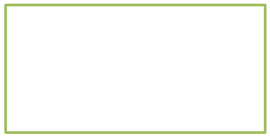




Ordre des blocs *catch*

```
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();
}
```

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

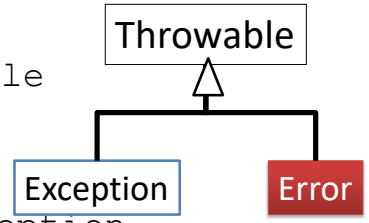
Bloc *finally*

- 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

Exception personnalisée

- Exception dérive de `Throwable`



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

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

```
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 IllegalArgException, AutorisationException {
    if (montant<0.0)
      throw new IllegalArgumentException();
    double nouveau = solde - montant;
    if (nouveau<decouvert)
      throw new AutorisationException();
    solde = nouveau;
  }
}
```





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

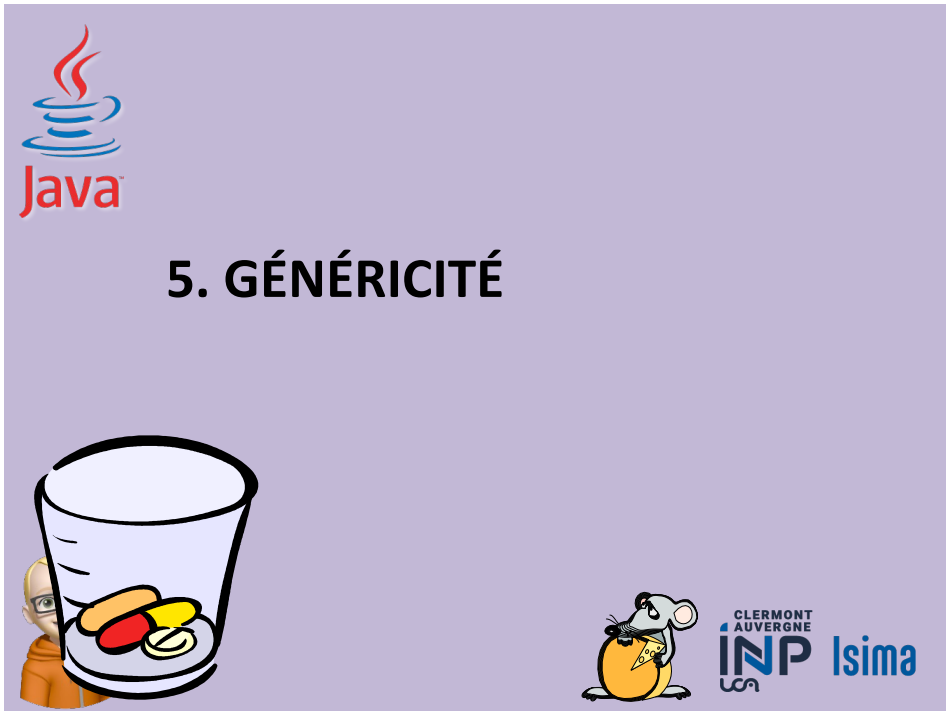


1. Bloc `try/catch` qui gère cette exception
2. Passage de l'exception au niveau supérieur (appelant).
   - L'exception apparait alors dans la **signature** de la méthode

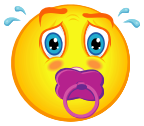


```
public void lire(BufferedReader br)
    throws IOException
{
   String lecture;
   try {
      while((lecture= br.readLine())!= null) {
         S.o.p(lecture);
   } catch(IOException e) {
      e.printStackTrace();
   }
}
```





Programmation générique



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

  Musser, Stepanov, 1988

- Choix d'un type => instanciation
- 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

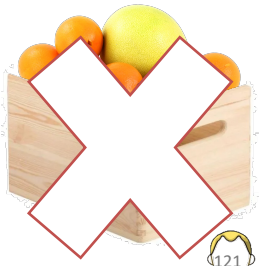










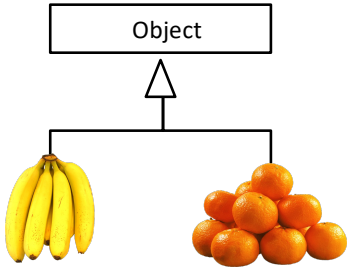




```
class CaisseBanane {
  Banane[] bananes;
  CaisseBanane(){}
  void ajouter(Banane b){}
  Banane enlever() {}
  void porter()
}
```

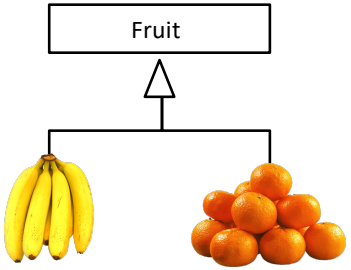
```
class CaisseOrange {
  Orange[] oranges;
  CaisseOrange(){}
  void ajouter(Orange o){}
  Orange enlever() {}
  void porter()
}
```







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



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

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



Condition sur un type ?

```
public class Classe<E> {
}
```

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

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


Méthode paramétrée

```
class Exemple {
  public <T> void methode1(T t) {
  }

  public <F extends M> void methode2(F f) {
  }

  public static <T> T methode3() {}
}
```


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'éxécution

https://docs.oracle.com/javase/tutorial/java/generics/bridgeMethods.html

Problèmes ?



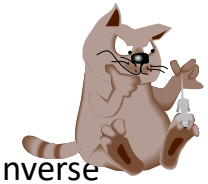
```
class ArrayList<E> implements List<E> {}
List<String> l1 = new ArrayList<String>();
List<Object> l2 = l1;
List<?>      l3 = l1;
```

```
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) {}
```

1er type utilisé pour l'*erasure*



6. COLLECTIONS





CLERMONT AUVERGNE INP ISIMA

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

https://docs.oracle.com/javase/tutorial/collections/index.html

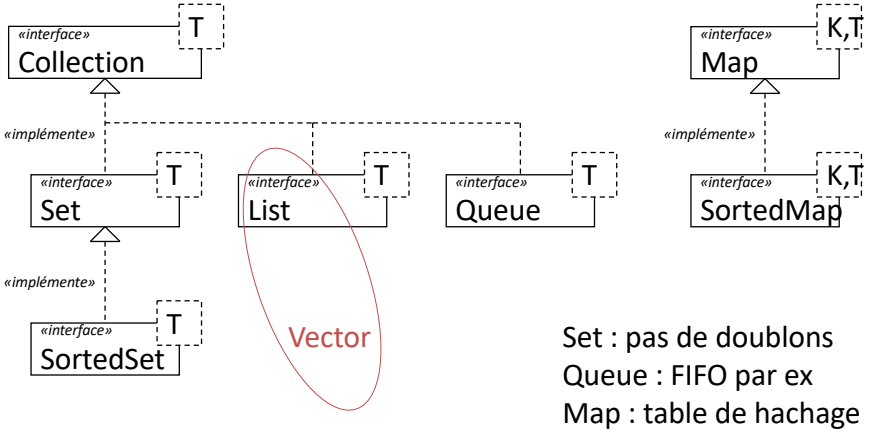
Types de collections

- *Backing collections*
  - "Classique"
  - Stockage d'éléments
- *View collections*
  - Pas de stockage
- *Unmodifiable Collections*
  - Les éléments peuvent être mutables

- Supportent la concurrence
- Ou pas ☹

5

Types de données ?



Set : pas de doublons
Queue : FIFO par ex
Map : table de hachage

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 ?

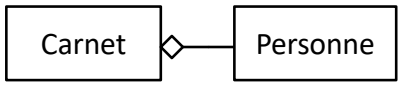
Exemple

```
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();
```


```
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>();
```




7. PLUS DE LANGAGE





CLERMONT AUVERGNE INP Isima

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

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();
```


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)
}
```


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 ?


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

```
for (Semaine jour : Semaine.values())
IO.println(jour);
```

```
for (Semaine j : EnumSet.range(
Semaine.LUNDI,  Semaine.VENDREDI))
   IO.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);
}
```


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

Exemples d'annotations

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




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

```
class Fille extends Mere {
  @Override
  public void Methode() {
    IO.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



Méthode à nombre d'arguments variable

5

```
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 (1)

- Limiter les classes dérivées

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

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

- Troi

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
- Pet implémenter des interfaces
- On peut ajouter des méthodes et des attributs de classes

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

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;
   }
}
```




8. INTROSPECTION



CLERMONT AUVERGNE INP Isima

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

```java
public class Vehicule {
  public void afficher() {
    IO.println("Vehicule");
  }
  public static void main(String[] param) {
    Vehicule v = new Sportive();

    if (v instance of Sportive)
      ((Sportive)v).special();

  }
}

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

Java 25
Tests sur types primitifs



9. PROGRAMMATION FONCTIONNELLE

- Streams



CLERMONT AUVERGNE INP ISIMA

Programmation fonctionnelle ?

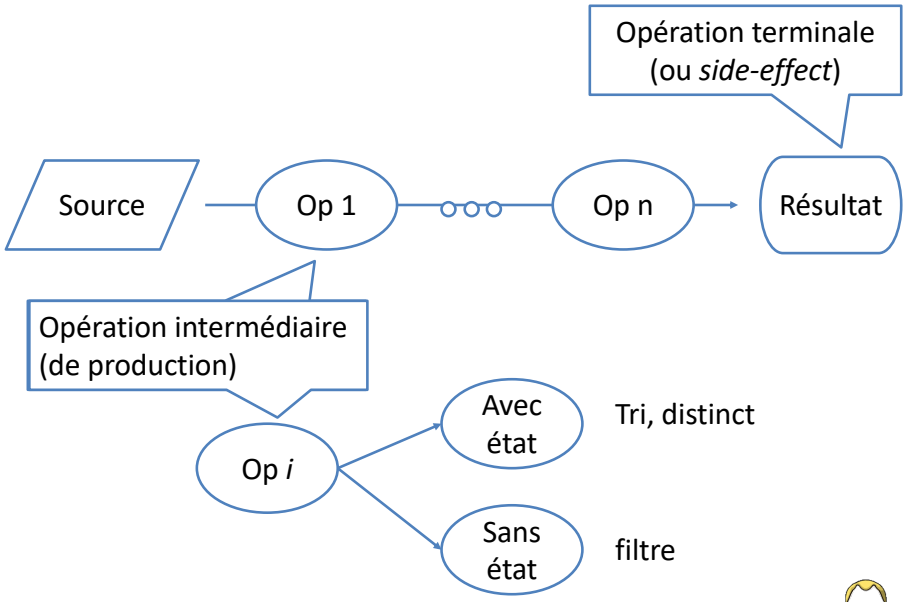
- Lambdas
- Interface fonctionnelle
- Package java.util.function
- Référence de fonction
- Streams

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);
```

```
IO.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;
});
```



ENTREES-SORTIES

- Fichiers & Flux
- Sérialisation





CLERMONT AUVERGNE INP UCA Isima

Gestion des entrées/sorties (1)

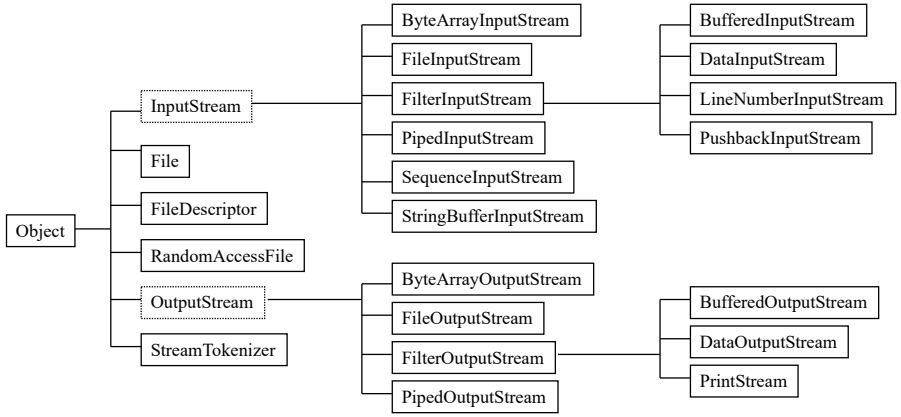
- Flux E/S de données binaires
- Flux E/S de caractères
- Flux E/S d'objets
- Communication avec des fichiers
- Communication avec des ressources Internet
- Sérialisation (=> réseau)

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

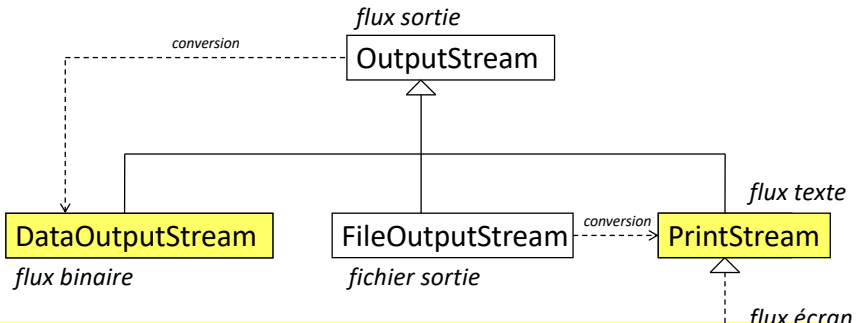


java.io.*



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

Flux de sortie

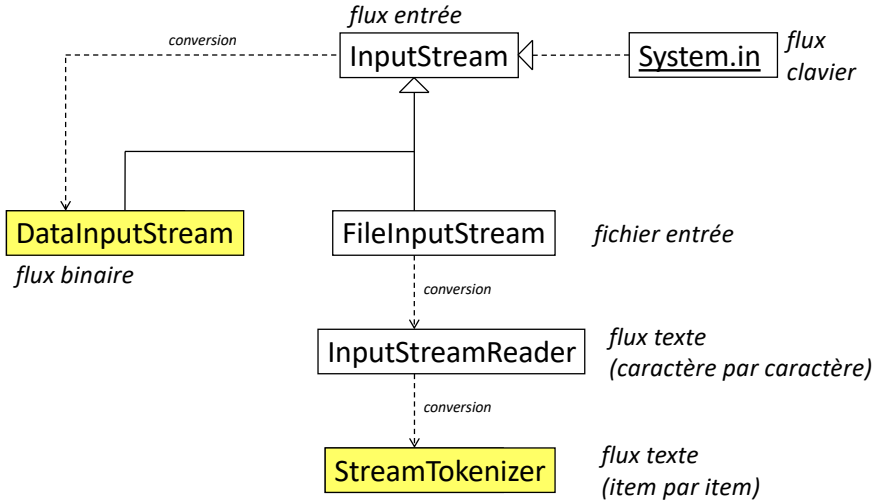



```
File fic = new File("log.txt");
FOS  fos = new FileOutputStream(fic);
PS   ps  = new PrintStream(fos);

ps.println("une chaine de caractères");
ps.close();
fos.close();
```

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();
```

6

System.console() == null sous Eclipse !


```
try {
    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();
    }
} catch(Exception e) {
```

Exceptions possibles ?

Pas la bonne place

Pas la bonne place non plus
= duplication de code

Bonne place
Mais exception possible

Try-with-resources

7

- 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


```
// 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.

Sérialisation (1)

- Transformer un **objet** présent en mémoire en bits
  - Sur un disque (Stockage - Persistance)
  - 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*

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
```

```
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 {
   if
   if
}
```

Écrire

```
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();
}
```

Classe de l'objet

Lire

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>
```




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, …

