DEV2 - Le Langage Java

P. Bettens (pbt) M. Codutti (mcd)

Haute École Bruxelles-Brabant
Département des Sciences Informatiques
Campus de l'École Supérieure d'Informatique

Année académique 2024 / 2025

Séance 1

Introduction
L'orienté objet

Crédit photo

Introduction

Objectifs
Moyens
Évaluation

Crédit photo

Objectifs

L'introduction à la programmation est faite . . .

- Programmation orientée objet
- Apprentissage du langage Java
- Familiarisation avec les spécifications du langage
- Approche de l'API Java
- Lecture et compréhension de la documentation

Supports et ressources

Rien

- pas de syllabus
- pas de livre

Quoique

- sur poÉSI
  - les slides ;
  - « The Java Language Specification » ;
  - des liens ;
  - des documents. . .

Évaluation

cfr. modalités de l'UE

Crédit photo

L'orienté objet

Motivation
Vocabulaire
Classe
Constructeur
Instanciation
Membre
Accesseur-Mutateur
Visibilité
this
Encapsulation

Crédit photo

Orienté objet - Motivation

Un langage **orienté objet** permet de créer ses propres types, liés au problème à résoudre.

Exemples : Étudiant, Produit, Partie, Vidéo...

Un type (pensez à **int**) c'est :

- des valeurs possibles ;
- ce que l'on peut en faire.

Crédit photo

Orienté objet - Vocabulaire

Un **objet** représente un élément du problème à résoudre.

Exemples : un étudiant en particulier, un produit bien précis, une vidéo donnée...

Un programme peut **créer** et **manipuler** des objets.

Orienté objet - Vocabulaire

Un type de données OO est défini par une **classe**.

Un objet est une **instance** d'une classe :

- construit à partir de la définition donnée par la classe ;
- appartenant au type défini par la classe.

Exemple : dev2 est un objet, instance de la classe UniteEnseignement.

Orienté objet - Vocabulaire

Un objet se caractérise par :

- son **état** (ses données) ;
  - stocké dans des **attributs** (des variables liées à l'objet)
- son **comportement** (ce que l'on peut en faire).
  - défini par des **méthodes** (du code)

membres = attributs + méthodes

Orienté objet - Exemple

Illustrons ces concepts avec la notion de vidéo sur un site diffusant des vidéos (vision fortement simplifiée).

- Une vidéo :
  - possède un auteur
  - possède un titre
  - est publiée ou non
  - a un certain nombre de "likes"
- On peut :
  - la liker
  - la publier

Orienté objet - Exemple - La classe

Exemple : Représentation graphique d'une classe

Orienté objet - Exemple - Les objets

Exemple : Représentation graphique de 2 vidéos (objets/instances) possibles :

kaamelott : Video
- author = "Alexandre Astier" - name = "Kaamelott" - publish = true - nbLikes = 2870
+ like() + publier()

dontlookup : Video
- author = "Adam McKay" - name = "Don't look up" - publish = true - nbLikes = 0
+ like() + publier()

Orienté objet - Définir la classe

Une classe se définit dans un fichier portant le nom de la classe.

Exemple : La classe Video se trouve dans le fichier `Video.java`.

```
public class Video {

}
```

- les {} déterminent le début et la fin de la classe

Orienté objet - Définir la classe

Les **attributs** s'écrivent directement dans la classe.

Exemple : Les attributs de la classe Video.

```
public class Video {

    private String author;
    private String title ;
    private boolean publish;
    private int nbLikes;
}
```

- Ici, les attributs sont **privés**
- ils ont un **type**

Orienté objet - Définir la classe

Les **méthodes** aussi s'écrivent directement dans la classe.

Exemple : Les méthodes de la classe Video.

```
public class Video {
    // [cut]

    public void like () {
        nbLikes++;
    }

    public void publish () {
        publish = true;
    }
}
```

- une méthode a un **nom**, un type de retour et des paramètres éventuels

Orienté objet - Constructeur

Instancier un objet c'est le construire en mémoire :

- lui réserver de l'espace en mémoire (sur le tas)
- initialiser son état (ses attributs)

L'initialisation est prise en charge par un **constructeur**.

Orienté objet - Constructeur

Un constructeur est une méthode particulière qui crée l'objet.

- Pas de type de retour déclaré
- En pratique, retourne la référence de l'objet
- A le même nom que celui de la classe

Orienté objet - Constructeur

Exemple

```
public Video (String aAuthor, String aTitle) {
    author = aAuthor;
    title = aTitle;
    publish = false;
    nbLikes = 0;
}
```

- Ce constructeur se trouve dans la classe,
- entre **public class** Video { et }

Orienté objet - Instanciation

Pour instancier un objet :

- utiliser l'opérateur **new**
- fournir les paramètres au constructeur

Exemple : instanciation d'une vidéo

```
Video kaamelott = new Video("Alexandre Astier", "Kaamelott");
```

- Crée un nouvel objet kaamelott de type Video
- Appelle le constructeur pour l'initialiser

Orienté objet - Instanciation

Une classe est un type **référence**

- une variable locale est crée sur la pile et
- l'instance de l'objet qu'elle référence, sur le tas.

Orienté objet - Instanciation

Exemple :

```
Video kaamelott;
// variable locale créée sur la pile
```

kaamelott [?]

```
kaamelott = new Video("Alexandre Astier", "Kaamelott");
// objet créé sur le tas
```

Orienté objet - Appel d'une méthode

Utilisation de la notation *pointée* (opérateur **.**)

Exemple

```
Video kaamelott = new Video("Alexandre Astier", "Kaamelott");
kaamelott.publish();
kaamelott.like();
```

- Appeler une méthode sur un objet peut changer son état.

Orienté objet - Membre privé

Impossible d'accéder aux attributs (privés)

- Pourquoi
  Pour éviter de donner des valeurs invalides

Exemple

```
Video kaamelott = new Video("Alexandre Astier", "Kaamelott");
System.out. println (kaamelott. title ); // ne compile : attribut privé
kaamelott.nbLikes = 1_000_000;           // idem
```

- System.out. println () permet d'afficher sur *stdout*

Orienté objet - Accesseur

Définition : **Accesseur** (**getter**), méthode donnant la valeur d'un attribut

Exemple

```
public String getAuthor() { return author; }
public String getTitle() { return title ; }
public int getNbLikes() { return nbLikes; }
public boolean isPublish() { return publish ; }
```

Par convention, l'accesseur de attribut est getAttribut (isAttribut pour un booléen)

Orienté objet - Accesseur

Exemple : Utilisation

```
Video kaamelott = new Video("Alexandre Astier", "Kaamelott");
System.out. println (kaamelott.getNbLikes());  // 0
kaamelott. like ();
System.out. println (kaamelott.getNbLikes());  // 1
```

Orienté objet - Mutateur

Définition : **Mutateur** (**setter**) méthode permettant de modifier l'état d'un objet (modifier un attribut)

Exemple : Un mutateur possible pour *Video*

```
public void setTitle(String aTitle) { title = aTitle;}
```

Par convention, le mutateur de attribut est setAttribut

Orienté objet - Mutateur

Exemple : Appel d'un mutateur

```
Video kaamelott = new Video("Alexandre Astier", "Kaamelott");
System.out.println( kaamelott.getTitle() );
kaamelott.setTitle("Kaamelott - Livre 1 - Tome II");
System.out.println( kaamelott.getTitle() );
```

Orienté objet - Mutateur

Est-ce qu'il faut fournir « ce » mutateur ?

- Est-ce que le statut (publié) peut changer ? Oui !
- Est-ce que le titre peut changer ? Pourquoi pas !
- Est-ce que l'auteur peut changer ? Euh !
- Est-ce que le nb de likes peut changer ? Oui !
  - Mais est-ce qu'il faut permettre de le changer directement ?
  - ou uniquement via des méthodes comme like () ?

Orienté objet - Visibilité

Chaque membre peut avoir comme **visibilité** :

- privé : n'est accessible que de **la** classe (**private**)
- paqueté : visible dans toutes les classes du *package* (pas de mot clé)
- protégé : visibilité liée à l'héritage (**protected**)
- public : visible dans **toutes** les classes (**public**)

Orienté objet - État par défaut

Et si un constructeur oublie d'initialiser un attribut ?

Les attributs ont une **valeur par défaut** :

- 0 pour les nombres ;
- **null** pour les références ;
- **false** pour les booléens.

Orienté objet - Constructeur par défaut

Et si on oublie de fournir un constructeur ?

Il existe un **constructeur par défaut** :

- sans paramètre ;
- qui ne fait rien.

Remarque Il n'est plus disponible si un autre constructeur est fourni.

Orienté objet - Surchage de constructeur

On peut fournir plusieurs constructeurs :

- On appelle cela la **surcharge** (**overloading**)
- Doivent se différencier par le nombre ou le type des paramètres

Exemple Un deuxième constructeur avec 4 paramètres

```
public Video (String aAuthor, String aTitle,
        boolean isPublish, int likes) {
    author = aAuthor;
    title = aTitle;
    publish = isPublish;
    nbLikes = likes;
}
```

Orienté objet - Surcharge de méthode

La surcharge (*overloading*) est permise aussi pour les méthodes

Exemple

```
public void like () {
    like (1);
}

public void like (int times) {
    nbLikes += times;
}
```

this
EX NCTEUR
Crédit photo

Orienté objet - this

this est une référence à soi-même.

Elle apparait dans différents contextes :

- constructeur **this**() ;
- attribut **this**. title ;
- méthode **this**. like ().

Orienté objet - this

Exemple Dans le contexte du constructeur

```
public Video (String author, String title, boolean publish, int nbLikes) {
    // [cut]
}

public Video (String author, String title) {
    this(author, title, false, 0); // must be first statement

}
```

- Appelle « l'autre » constructeur

Orienté objet - this

Exemple Dans le contexte d'un attribut

```
public Video (String author, String title, boolean publish, int nbLikes) {
    this.author = author;
    // [cut]
}
```

- Différencie l'attribut author du paramètre author lorsqu'ils portent le même nom

Orienté objet - this

Exemple Dans un contexte d'appel de méthode

```
public void like () {
    this . like (1);
}

public void like (int times) {
    nbLikes += times;
}
```

- Permet de préciser que c'est bien la méthode de **cette** classe
- Peu usité

Encapsulation

Crédit photo

Orienté objet - Encapsulation

Premier principe de l'orienté objet : l'**encapsulation**

La cohérence de l'objet est assurée par la classe

- les attributs sont **privés**
- les méthodes permettant de modifier l'état de l'objet sont **publiques** ; elles s'assurent que l'objet reste cohérent

Orienté objet - En résumé

En résume,
notre première classe a l'allure suivante...

Orienté objet - En résumé

```
public class Video {
    private String author;
    private String title ;
    private boolean publish;
    private int nbLikes;

    public Video (String author, String title ,
        boolean publish, int nbLikes) {
    this.author = author;
    this. title = title ;
    this.publish = publish;
    this.nbLikes = nbLikes;
    }
```

...

Orienté objet - En résumé

```
public Video (String author, String title ) {
    this(author, title , false, 0);
}

public String getAuthor() { return author; }
public String getTitle () { return title ; }
public int getNbLikes() { return nbLikes; }
public boolean isPublish() { return publish ; }

public void setTitle (String title ) {
    this. title = title ;
}
```

...

Orienté objet - En résumé

```
    public void like () {
        like (1);
    }

    public void like (int times) {
        nbLikes += times;
    }

    public void publish () {
        publish = true;
    }
}
```

Séance 2

Hello World
Orienté objet (suite)

Crédit photo

Hello world

Crédit photo

Hello world

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

- La classe peut porter « n'importe quel » nom (**CamelCase**).
- Le fichier doit porter le nom de la classe (ici `Main.java`).

Hello world

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

- La méthode main est le **point d'entrée du programme**
- args est un paramètre qui correspond aux arguments du programme (cf. infra)

Le fichier **source** (`java`) doit être compilé.
Le compilateur génère un **bytecode** compréhensible par la machine virtuelle `java`.

Crédit photo

Hello world

Exemple,
le code slide 50 doit se trouver dans un fichier Main.java que l'on compile,

```
javac Main.java
```

- crée un fichier Main.class
- ce fichier contient le **bytecode**

Hello world

Exemple (suite) ,

La classe est exécutée

```
java Main
```

- la machine virtuelle java (JVM) exécute le bytecode de la classe Main
- en commençant par le code de la méthode main

Hello world

Remarque 1

En **python**, le code est **interprèté**

- un fichier contient le code source hello.py
- le fichier est interprèté par l'interprèteur python

```
python3 hello.py
```

Hello world

Remarque 2

En Java, si j'écris java Main.java « ça marche »

- uniquement depuis JDK11 ;
- le *bytecode* est généré et se trouve en mémoire ;
- pour un programme d'une seule classe

L'orienté objet (part 2)

static
Robustesse

Crédit photo

Orienté objet - static

Quelle est l'utilité du mot clé ?

static

- Sans **static** : fait référence aux objets,
- avec **static** : Pas de référence à un objet. Ce sont des « fonctions »

Orienté objet - static

Un membre **static** :

- fait référence à la classe et non à une instance ;
- il ne dépend pas d'un objet.

Un **membre** peut être un **attribut**, une **méthode** ou un **constructeur** (qui ne sera jamais **static**).

Orienté objet - static

Attribut statique

- existe en un seul exemplaire
- est initialisé lors du chargement de la classe (une seule fois)
- utilisation courante : constantes

```
public class Math {
    public static final double PI = 3.14159265358979323846;
    public static final double E = 2.7182818284590452354;
}
```

Orienté objet - static

Attribut (non statique)

- un exemplaire par instance
- est initialisé par le constructeur
- utilisation courante : les objets

```
public class Video {
    private String author;
    private String title ;
    private boolean publish;
    private int nbLikes;

    // ...
}
```

Orienté objet - static

Méthode statique

- ne peut pas accéder aux membres des instances
- seuls les paramètres et les variables locales sont visibles
- utilisation courante : méthodes non objets (fonctions)

```
public class Utils {
    public static int abs(int nb) {
        return nb < 0 ? -nb : nb;
    }
}
```

Remarque : opérateur conditionnel ? :

Remarque ? : (grammaire simplifiée)

ConditionalOperator ? : :
BoleanExpression ? *Expression* : *Expression*

L'opérateur conditionnel ? : utilise la valeur booléenne de la première expression pour décider laquelle des deux autres expressions sera évaluée.

Remarque : opérateur conditionnel ? :

Exemple

```
int value = n < 0 ? -n : n;
```

est équivalent à

```
int value;
if (n < 0){
    value = -n;
} else {
    value = n;
}
```

Orienté objet - static

Méthode (non statique)

- accède aux membres de l'instance courante
- utilisation courante : méthodes

```
public class Video {
    // ...

    public void like (int times) {
        this.nbLikes += times;
    }
}
```

« Le développeur ou la développeuse est responsable de la cohérence de l'objet au sein de la classe. »

Orienté objet - Constructeur robuste

Le constructeur peut vérifier la validité des paramètres

Exemple : Permettrait de refuser

```
Video kaamelott = new Video("", "Kaamelott"); // refusé
Video kaamelott = new Video(null, "Kaamelott"); // aussi
```

null :

- littéral de type référence
- indique qu'il n'y a pas d'objet (référence vers **rien**)

Ne pas confondre **null** et **""**

(faire un schéma mémoire)

Orienté objet - Constructeur robuste

Il suffit d'ajouter des tests en début de constructeur

Exemple :

```
public Video (String aAuthor, String aTitle) {
    if (aAuthor == null
        || aAuthor.length() == 0) {
            throw new IllegalArgumentException(
                "Author invalid : " + aAuthor);
        }
    // some for aTitle
    author = aAuthor;
    title = aTitle;
    publish = false;
    nbLikes = 0;
}
```

Lorsqu'un objet risque d'être dans un état incohérent,
le code lance une exception
Crédit photo

Orienté objet - Exception

Lancer une exception, c'est

- créer un objet,
- *lancer l'exception* grâce à **throw**

Le code est interrompu ce qui permettra de confiner les problèmes. L'exception sera attrapée ailleurs (cf. infra).

Orienté objet - Mutateur robuste

Le mutateur aussi peut vérifier le paramètre

Exemple : Mutateur avec test de validité

```
public void setTitre(String aTitle) {
    if (aTitle == null || aTitle.length() == 0) {
        throw new IllegalArgumentException("Titre invalide : " + aTitle);
    }
    title = aTitle;
}
```

Permettrait de refuser

```
kaamelott.setTitle("");  // refusé
kaamelott.setTitle(null);  // aussi
```

Orienté objet - Appel d'une méthode

Appel d'une méthode :

- **statique**
  par le biais de la **classe**

  Math.sqrt(4)

- **non statique**
  par le biais d'une **instance de la classe**

  kaamelott. like ()

Question : Pourquoi ne peut-on pas instancier un objet de la classe Math ?

Étude de cas

Orienté objet - Étude de cas - Point

Représentons un point dans le plan

Orienté objet - Étude de cas - Point

Premier projet

Orienté objet - Étude de cas - Point

Ajout d'un **constructeur**

Point
- x : Real - y : Real
+ Point(x : Real , y : Real) + display() + getX() : Real + getY() : Real

Orienté objet - Étude de cas - Point

- Instancier le point $P(-2, 5)$
- Afficher le point P
- ~~Changer une coordonnée du point P~~
- Obtenir la valeur de l'abscisse du point P

Orienté objet - Étude de cas - Point

Donnons la possibilité à un point d'être **déplacé**.
Donnons également la possibilité de calculer sa **distance** à un autre point.

Point
- x : Real - y : Real
+ Point(x : Rael , y : Real) + display() + getX() : Real + getY() : Real + move(deltaX : Real, deltaY : Real) + distance(other : Point) : Real

Orienté objet - Étude de cas - Point

- Instancier les points $P_1(-2, 5)$ et $P_2(1, 1)$
- Afficher les points P_1, P_2
- Afficher la distance entre P_1 et P_2
- Afficher la distance entre l'origine et P_2

- Ajouter une méthode (à moindre frais) `distance()` donnant la distance à l'origine

Orienté objet - Étude de cas - toString

String toString()

- fournit une représentation textuelle basique de l'état
- nom standardisé
- appelée automatiquement par println
  ou lors d'une concaténation
- version par défaut existe mais pas intéressante

Exemple

```
public String toString() {
    return "(" + x + "," + y + ")";
}
```

```
System.out.println(myPoint);
System.out.println("point = " + myPoint);
```

Orienté objet - Étude de cas - Cercle

Pour aller plus loin...

- Écrire une classe `Circle` où un cercle est représenté par son centre (un point) et son rayon (un nombre).
- Proposer une méthode `move` permettant de déplacer le cercle... en déplaçant son centre.

Séance 3

Grammaire

Crédit photo

Grammaire

Principe
Fonctionnement
Lexicale
Syntaxique

« When I use a word,
it means just what I choose it to mean
- neither more or less »
Java Langage Specification Introduction

Crédit photo

Grammaire - Motivation

La nécessité de pouvoir écrire un **compilateur**, la nécessité de pouvoir **décrire** le langage imposent que les règles soient clairement décrites.

C'est le rôle de la **grammaire**.

Grammaire - Description

La grammaire est décrite dans
The Java Language Specification

The Java® Language Specification
Java SE 21 Edition

James Gosling
Bill Joy
Guy Steele
Gilad Bracha
Alex Buckley
Daniel Smith
Gavin Bierman

Grammaire - Fonctionnement

Une grammaire est une description finie de l'infinité des programmes

- Un mot (*token*) doit être légal,
  grammaire lexicale
- Une séquence de mots doit être légale,
  grammaire syntaxique
- Le tout doit avoir un sens,
  sémantique

Grammaire - Compilation

Chaîne de compilation

El tahc tse rion

Le est noir chat

Le chat est noir

Le parapluie mange l'ascenseur

Grammaire - Fonctionnement

Fonctionnement d'une grammaire :

1. symbole de départ
2. règles de productions (***productions***)
3. symboles terminaux (***token***)

Un code est correct s'il peut être **produit** par la grammaire

2.1 Context-Free Grammars

A *context-free grammar* consists of a number of *productions*. Each production has an abstract symbol called a *nonterminal* as its *left-hand side*, and a sequence of one or more nonterminal and *terminal* symbols as its *right-hand side*. For each grammar, the terminal symbols are drawn from a specified *alphabet*.

Starting from a sentence consisting of a single distinguished nonterminal, called the *goal symbol*, a given context-free grammar specifies a language, namely, the set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

Extrait de [jls8] p 9(29)

Crédit photo

Grammaire - Fonctionnement

Exemple : Un nombre décimal naturel

Nombre :
    Chiffre
    Chiffre Nombre

Chiffre : one of `0 1 2 3 4 5 6 7 8 9`

Grammaire - Fonctionnement

Exemple : Un palindrome binaire

Palindrome :

0
1
00
11
0 *Palindrome* 0
1 *Palindrome* 1

Grammaire lexicale

(des caractères aux mots)

Grammaire lexicale

La **grammaire lexicale** (*lexical grammar*)

- Les symboles terminaux sont les **caractères** (*characters of Unicode characters set*)
- Les règles de production forment les mots (**tokens**), éléments d'entrée (*input elements*) de la grammaire syntaxique.

Grammaire - Les caractères en entrée

Les caractères

UnicodeInputCharacter :
    RawInputCharacter
    UnicodeEscape

RawInputCharacter :
    any Unicode character

UnicodeEscape :
    UnicodeMarker HexDigit HexDigit HexDigit HexDigit

UnicodeMarker :
    `u`
    UnicodeMarker `u`

HexDigit : one of `0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F`

Grammaire - Les caractères en entrée

Exemples

a A û \u0070 Ê

Unicode versus UTF16

Grammaire - Input elements

Ce que produit la grammaire lexicale

InputElement:
    Comment
    WhiteSpace
    Token

Remarque : Les commentaires et espaces ne passent pas la phase suivante. Seuls les *tokens* seront traités par l'analyse syntaxique.

Grammaire - Commentaires

Deux sortes de commentaires différents

```
// Commentaire sur une ligne
```

```
/* Commentaire sur
   plusieurs lignes */
```

(voir [jls8] p21(41))

*/** commentaire */* est un commentaire de la deuxième sorte, pris en compte par `javadoc`.

Grammaire - WhiteSpace

WhiteSpace :
the ASCII SP character, also known as "space"
the ASCII HT character, also known as "horizontal tab"
the ASCII FF character, also known as "form feed"
LineTerminator

LineTerminator :
the ASCII LF character, also known as "newline"
the ASCII CR character, also known as "return"
the ASCII CR character followed by the ASCII LF character

Grammaire - Token

Token :
- *Keyword*
- *Separator*
- *Operator*
- *Literal*
- *Identifier*

Grammaire - Token

Keyword :
ReservedKeyword
ContextualKeyword

ReservedKeyword : one of
`abstract assert boolean break byte case catch char class const continue default do double else enum extends final finally float for if goto implements import instanceof int interface long native new package private protected public return short static strictfp super switch synchronized this throw throws transient try void volatile while _(underscore)`

ContextualKeyword : one of
`exports module non-sealed open opens permits provides record requires sealed to transitive uses var when with yield`

Grammaire - Token

Separator : one of

```
( ) { } [ ] ; , . ... @ ::
```

Operator : one of

```
=  >  <  !  ~  ?  :  ==  <=  >=  !=  &&  ||
++  --  +  -  *  /  &  |  ^  %  <<  >>  >>>
+=  -=  *=  /=  &=  |=  ^=  %=  <<=  >>=  >>>= ->
```

Grammaire - Literal

Un littéral représente une valeur d'un certain type.

Literal:
- *IntegerLiteral*
- *FloatingPointLiteral*
- *BooleanLiteral*
- *CharacterLiteral*
- *StringLiteral*
- *NullLiteral*

Grammaire - Literal

BooleanLitteral: one of
    `true` `false`

NullLiteral:
    `null`

IntegerLiteral:
    DecimalIntegerLiteral
    ...

DecimalIntegerLiteral:
    DecimalNumeral [IntegerTypeSuffix]

[jls21] p 33-38(49-55)

Grammaire - Literal

Exemples

- littéraux entiers en base 10
  1, 2_000, 3L, 4_5_6
- la quantité 100
  100, 1_0_0, 0144, 01_44, 0x64, 0b0110_0100

Consulter la spécification ...

[jls21] p 32-42(48-59)

Grammaire - Identifier

Identifier :
    IdentifierChars
    but not a *Keyword* or *BooleanLitteral* or *NullLitteral*

IdentifierChars :
    JavaLetter
    IdentifierChars JavaLetterOrDigit

JavaLetter :
    any Unicode character that is a Java letter (_ et $ sont compris)

JavaLetterOrDigit :
    any Unicode character that is a Java letter-or-digit

Grammaire syntaxique

(des mots au programme)

Grammaire syntaxique

La **grammaire syntaxique** (*syntactic grammar*)

- Les symboles terminaux sont les **tokens**
- Les règles de production permettent de définir ce qu'est un programme syntaxiquement correct

Grammaire syntaxique

Parmi les éléments importants d'un programme, on trouve :

- Les **expressions**
  - Représentent les « *calculs* »
  - Ont une valeur et un type
  - On y reviendra continuellement sans jamais tout faire en une fois
  - Voir [jls21] p 563(579)
- Les **instructions**
- Les **expressions-instructions**

Séance 4

Les instructions
Instruction-expression

Crédit photo

Instructions
Revue des troupes. . .
Crédit photo

Les instructions

L'exécution d'un programme est contrôlée par ses instructions (*statement*).

- une instruction est exécutée pour son **effet** et n'a pas de valeur ($\neq$ expression)
- une instruction se termine par un ; (excepté le block)

Les instructions

Statement :
- *EmptyStatement*
- *Block*
- *LabeledStatement*
- *BreakStatement*
- *ContinueStatement*
- *ExpressionStatement*
- *IfThenStatement*
- *IfThenElseStatement*
- *IfThenElseStatementNoShortIf*
- *SwitchStatement*
- *WhileStatement*
- *DoStatement*
- *ForStatement*
- *ReturnStatement*
- *YieldStatememt*
- ...

Version simplifiée.

[jls21] p491-493 (507-509)

L'instruction vide

EmptyStatement :
;

Cette instruction ne fait rien... et le fait toujours bien.

Exemple :

```
return 1;;;  // 3 instructions
```

Le bloc

Block :
{ *BlockStatements*(opt) }

BlockStatements :
BlockStatement
BlockStatement BlockStatements

BlockStatement :
LocalVariableDeclarationStatement
Statement

Représente un ensemble d'**instructions** ou de **déclarations** de variables locales (au bloc)

Le bloc

Exemple : Que penser de ceci ?

```
public static void main(String[] args) {
    int i=1;
    {
        int j=2;
        System.out.println(i+j);
    }
    System.out.println(i+j);
}
```

Return

ReturnStatement :
`return` *[Expression]* `;`

L'instruction **return** rend le contrôle au code appelant avec ou sans valeur de retour.

Les choix

if
switch

Crédit photo

If

IfThenStatement:
`if` (*Expression*) *Statement*

IfThenElseStatement:
`if` (*Expression*) *StatementNoShortIf* `else` *Statement*

IfThenElseStatementNoShortIf:
`if` (*Expression*) *StatementNoShortIf* `else` *StatementNoShortIf*

Pourquoi pas de { } dans la grammaire ?

If

Exemple :

```
if (door.isOpen())
    house.enter();
else
    door.knock();
```

Bonne pratique : mettre les accolades

```
if (door.isOpen()) {
    house.enter();
} else {
    door.knock();
}
```

If

Exemples : Que penser de ceci ?

```
if (door.isOpen())
    house.enter();
    resident.greet("Hello");
else
    door.knock();
```

If

Exemples : Que penser de ceci ?
Problème du *dangling else* ?

```
if (door.isOpen())
    if (resident.isVisible())
        resident.greet("Hello");
else
    door.knock();
```

[jls21] p491-493 (507-509)

Switch

SwitchStatement:
`switch` (*Expression*) *SwitchBlock*

SwitchBlock:
`{` {*SwitchBlockStatementGroup*} {*SwitchLabel*} `}`
`{` *SwitchRule* {*SwitchRule*} `}`

- L'expression est de type ; **byte**, **char**, **short**, **int** ou un **type référence** (JDK21)
- Distinguer `SwitchStatement` et `SwitchExpression` (JDK14)

Switch Statement

SwitchStatement:
`switch` (*Expression*) *SwitchBlock*

SwitchBlock:
`{` {*SwitchBlockStatementGroup*} {*SwitchLabel*} `}`

SwitchBlockStatementGroup:
SwitchLabel `:` { *SwitchLabel* `:` } *BlockStatements*

SwitchLabel:
`case` *CaseConstant* { `,` *CaseConstant* }
`default`

- `CaseConstant` est une expression conditionnelle
- **break** dans `BlockStatement` permet de « sortir » du **switch**

Switch Statement

Exemple 1

```
switch (n) {
    case 3:
        System.out. println ("Hi !");
    case 2:
        System.out. println ("Hi !");
    case 1:
        System.out. println ("Hi ! ? again");
    default:
}
```

Switch Statement

Exemple 2

```
switch (n) {
    case 1:
        System.out. println ("Only one");
        break;
    case 2:
        System.out. println ("I'm case 2. I'm done");
        break;
}
```

Switch Expression

SwitchExpression:
`switch` (*Expression*) *SwitchBlock*

SwitchBlock:
`{` *SwitchRule* {*SwitchRule*} `}`

SwitchRule:
SwitchLabel -> *Expression* ;
SwitchLabel -> *Block*
SwitchLabel -> *ThrowStatement*

- En tant qu'expression, le `SwitchExpression` a une valeur (et un type)

Switch Expression

Exemple

```
Season s = Saison.WINTER;
String message = switch(s){
    case SPRING, SUMMER -> "It's hot";
    case WINTER, AUTUMN -> "It's cold";
};
```

```
enum Season {SPRING, SUMMER, AUTUMN, WINTER;}
```

Yield Statement

YieldStatememt:
`yield` *Expression* `;`

- l'instruction yield transfère le contrôle pour permettre à un *switch* de produire une valeur

Yield Statement

Exemple

```
enum Season { SPRING, SUMMER, AUTUMN, WINTER ;}
Season s = Season.WINTER;
String v = switch (s) {
    case SPRING -> 42;
    case SUMMER -> 43;
    case AUTUMN -> 44;
    case WINTER -> {
        var len = s.toString().length();
        yield len * len;
    }
};
```

- Sans yield , c'est une expression et pas une instruction

Les boucles

while
do-while
for

Crédit photo

While

`while` (*Expression*) *Statement*

- Expression est booléen
- C'est *Statement* qui se charge de « l'incrément »
- **Exemple**

```
int indice = 0;
while (indice < tableau.length && tableau[indice] == 0) {
    indice++;
}
```

Do - While

`do` *Statement* `while (` *Expression* `);`

- C'est un **répéter tant que**
- **Exemple**

```
int nb;
do {
    nb = clavier.nextInt();
} while (nb<=0);
```

Le for

BasicForStatement
EnhancedForStatement

For - BasicForStatement

BasicForStatement :
`for` (*ForInit*(opt) ; *Expression*(opt) ; *ForUpdate*(opt)) *Statement*

ForInit :
StatementExpressionList
LocalVariableDeclaration

ForUpdate :
StatementExpressionList

For - BasicForStatement

1 *ForInit*

Étape d'initialisation de la boucle

- liste de **StatementExpression**

StatementExpressionList :
    StatementExpression
    StatementExpressionList , *StatementExpression*

- déclaration de variable locale
  (version simplifiée)

LocalVariableDeclaration :
    { `final` } *Type VariableDeclaratorList*

For - BasicForStatement

Une déclaration de variables

VariableDeclaratorList :
VariableDeclarator { **,** *VariableDeclarator*}

VariableDeclarator:
VariableDeclaratorId [= *VariableInitializer*]

Exemple

```
int i, j = 5, k, l = m = 5
```

For - BasicForStatement

2 Évaluation de l'*Expression*

Évaluation de l'expression représentant le **test**

- expression booléenne
- si **true**, *Statement*, *ForUpdate* et recommencer
- si **false**, instruction suivant la boucle *for*

3 Évaluation du *ForUpdate*

- liste de **StatementExpression**

For - BasicForStatement

For - EnhancedForStatement

EnhancedForStatement :
`for` (*Type Identifier* `:` *Expression*) *Statement*

Permet de parcourir un tableau ou un **Iterable**

Iterable → séance ultérieure

Exemple

```
String [] toys = {"Hamm", "Slink", "Potato", "Woody", "Sarge", "Etch",
    "Lenny", "Squeeze", "Wheezy", "Jessie", "Stretch", "Buster", "Bullseye"}
for (String toy : toys) {
    // do something with toy
}
```

For - EnhancedForStatement

```
for (String toy : toys) {
    // do something with toy
}
```

est un raccourci pour

```
for (int i=0; i<toys.length; i++) {
    String toy = toys[i];
    // do something with toy
}
```

Plus concis et plus rapide mais :

- **Pas** accès à l'**indice** ;
- **Impossible de modifier** un élément.

Ruptures
étiquette
break
continue
Crédit photo

Label

Toute instruction peut recevoir une **étiquette** (*label*)

LabeledStatement :
    Identifier **:** *Statement*

- Permet de nommer (étiqueter) une instruction
- N'est connue que dans l'instruction qui la suit
- Permettra de quitter brutalement (**break**) ou de réitérer (**continue**) certaines instructions

Break

BreakStatement :
`break` *Identifier*(opt) `;`

- Permet d'**arrêter** brutalement une instruction
- Si étiquette → arrête l'instruction étiquetée
- Si pas d'étiquette → arrête la boucle englobante

Break

Exemple

```
int i= 1;
lab1 : {if(i==1) break lab1 ; System.out.println (2);}
System.out.println (3);        // affiche : 3
```

- Si l'on retire les accolades ?
- Si l'on retire le label ?

Continue

ContinueStatement :
`continue` *Identifier*(opt) `;`

- Permet de passer **directement** à l'itération suivante
- Si pas d'étiquette → recommence la première instruction **répétitive englobante**
- Si étiquette → recommence la boucle étiquetée

Continue

Exemples

```
for (int i=0; i<10; i++) {
    if (i%2==0) continue;
    System.out.println(i);
}
```

```
bcli: for (int i=0; i<10; i++) {
    bclj: for (int j=0; j<10; j++) {
        if( (i*j)%2==0 ) continue bcli;
        System.out.println(j);
    }
    System.out.println(i);
}
```

Crédit photo

```java
public class Meule{
    public static int inbuff(char[] meule, char[] aiguille) {
        int i,j, t = -1, sizem = meule.length, sizea = aiguille.length;

        for (i=0; i <= sizem-sizea; i++) {
            for (j=0; j < sizea; j++) {
                if (meule[i+j] != aiguille[j])
                    break;
                t = j;
            }
            if (t == sizea-1) {
                t = i;
                break;
            }
            else
                t = -1;
        }
        return t;
    }
}
```

~
~
~
~
~
~
~
~

Crédit photo

Expression-instruction

Assignation
Incrémentation
Appel de méthode
Instanciation

Certaines expressions peuvent devenir une instruction dès l'ajout du ;
statement expression

Crédit photo

Crédit photo

Expression-instruction

ExpressionStatement:
    StatementExpression **;**

StatementExpression:
    Assignment
    PreIncrementExpression
    PreDecrementExpression
    PostIncrementExpression
    PostDecrementExpression
    MethodInvocation
    ClassInstanceCreationExpression

Assignation

Crédit photo

Expression-instruction - Assignation

Assignment :
    LeftHandSide AssignmentOperator Expression

LeftHandSide :
    Identifier
    ArrayAccess

L'assignation est avant tout une **expression**

- elle a un **type** (celui de la variable)
- elle a une **valeur** (celle du *left hand side*)

On en fait une instruction avec le ;

Expression-instruction - Assignation

Exemples

```
élément = 1
éléments[i] = j

éléments[i] = j;
i = j = k = l = 0;
i = (j = i+j) + 1;
f(i=1,j=0);
```

Expression-instruction - Assignation

Il existe d'autres **opérateurs d'assignation**

Assignment :
LeftHandSide *AssignmentOperator* *Expression*

AssignmentOperator : one of
`=` `*=` `/=` `%=` `+=` `-=`

- var += expr équivaut à var = (Type) (var + expr)
- **Ex** : i+=1 équivaut à i = i + 1

```
i = 2;
i = i = (i*=2) + 1;
(i+1) -= 2;
```

Crédit photo

Pré/post in/décrémentation

++ permet d'**incrémenter** une variable
−− permet de **décrémenter** une variable

- Peut se placer avant ou après la variable
- i++; ≅ ++i; ≅i+=1; ≅ i=i+1;

++i et i++ sont des expressions

- Elles ont une valeur
- Mais elles ont aussi un effet (l'incrémentation)
- Dans quel ordre ?

Pré/post in/décrémentation

i++ : lorsqu'elle est évaluée

1. i donne sa valeur à l'expression i++
2. i est incrémentée

Exemple

```
int i = 1;
System.out.println(i++);
System.out.println(i);
```

Pré/post in/décrémentation

++i : lorsqu'elle est évaluée

1. i est incrémentée
2. i donne sa valeur à l'expression ++i

Exemple

```
int i = 1;
System.out.println(++i);
System.out.println(i);
```

Pré/post in/décrémentation

Exemples

```
int i = 5;
j = i++;
j = ++i;

int i = 5;
i = i++;
i = ++i;
i = i++ + ++i;

int i = 2;
f(i++,--i); // équivaut à f(2,2)
f(--i,i++); // équivaut à f(1,1)

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

Appel de méthode

MethodInvocation

Crédit photo

Expression-instruction - Appel de méthode

L'appel de méthode est une **expression**

- elle a un **type** (le *return type*)
- elle a une **valeur** (celle retournée)

```
Math.sqrt(4)
o1.foo(o2)
```

Instanciation de classe

ClassInstanceCreationExpression

Crédit photo

Expression-instruction - Instanciation

Créer une instance d'une classe est une expression...
ne pas utiliser sa valeur n'a pas (beaucoup) de sens

- elle a un **type** (celui de l'objet créé)
- elle a une **valeur** (la *référence* vers l'objet)

```
new Video("author"," title ")
new Point(2,3)
```

Séance 5

Les types
Les tableaux
Les tableaux de tableaux

Crédit photo

Les types

Crédit photo

Les types

Java est un langage a **typage statique** :
chaque expression et chaque variable a un type connu à la compilation.

Java est un langage **fortement** typé :
le type d'une variable est fixé ainsi que la valeur qu'elle peut recevoir. *Idem* pour une expression.

Type :
PrimitiveType
ReferenceType

- Types auxquels s'ajoute **null** qui a pour type le *type null*

Les types primitifs

```
int value = 7;
```

Les types primitifs

[JLS21] p56-64 (72-80)

Les types primitifs entiers

Les types primitifs entiers : **byte**, **short**, **int** et **long**

- nombres signés en compléments à 2
- codés sur 8, 16, 32 et 64 bits
- intervales :
  - -128 - 127
  - -32 768 - 32 767
  - -2 147 483 648 - 2 147 483 647
  - -9 223 372 036 854 775 808 - 9 223 372 036 854 775 807

Les types primitifs entiers

char est aussi un type numérique entier... représentant un **caractère**

- entier non signé sur 16 bits
- sous ensemble BMP (*Basic Multilingual Plane*) des caractères Unicode codé en UTF-16.
  Soient \u0000 à \ uffff

Littéral :

- caractère entre guillemets simples (*single quote*),
- séquence d'échappement \n, \t, \' ou \"

Les types primitifs à virgule flottante

Les types primitifs à virgule flottante : **float** et **double**

- sous-ensemble des réels
- respectent la norme IEEE754
- codés sur 32 bits et 64 bits respectivement

Les types références

ReferenceType :
ClassOrInterfaceType
TypeVariable
ArrayType

- Les « autres »types définis dans le langage, dans l'API ou par læ développeurice et,
- les tableaux (*array*)

La classe Object

La classe Object est la classe parent de toutes les classes (cf. héritage infra).

```
Object o = new Object();
```

- déclarer un variable de ce type a peu d'intérêt
- toutes les classes héritent de Object

La classe String

La classe String représente les **chaines de caractères**

```
String value = "Hello world";
```

La classe String

Littéral de type String (**string literal**)
Caractère(s) entre guillemets doubles (*double quote*)

```
String value = "Hello ... \nworld";
```

Bloc de texte (**text block**)
Caractères entre 3 guillemets doubles

```
String value = """
        Hello ...
        world
        """;
```

Les tableaux

« Should array indices start at 0 or 1 ?
My compromise of 0.5 was rejected without,
I thought, proper consideration. »
Stan Kelly-Bootle

Crédit photo

Les tableaux

```
int [] values = {3, 1, −4, 1, 5, −9};
```

- Type[] est le type « tableau de Type »
- Accès à un élément par son **indices** entre **crochets** [] (à partir de 0)

Les tableaux

Un tableau est de type référence (et hérite de Object)

Les tableaux

déclaration - création - initialisation

- déclaration : Type[] identifier
- création : identifier = **new** Type[taille]
- initialisation (lors de la création) :
  identifier = **new** Type[] {a1, a2};

Les tableaux

Exemples

```
double[] ds;
ds = new double[] {4.2, -1};

int [] is = new int[] {0x2A, 021, -5};
String [] girls = new String[]
    {"Victoria", "Melanie", "Melanie", "Emma", "Geri"};
```

- la taille du tableau ne fait pas partie de son type, elle peut être déterminée à l'exécution
- la déclaration et la création peuvent être combinées
- les éléments sont initialisés à 0

Les tableaux

Exemples

```
double[] ds;
ds = {4.2, -1}; // ERROR

int [] is = {0x2A, 021, -5};
String [] girls =
    {"Victoria", "Melanie", "Melanie", "Emma", "Geri"};
```

- lorsque la déclaration et l'initialisation sont faites en même temps, la création explicite est optionnelle

Les tableaux

Erreurs fréquentes

- tentative d'accès à un tableau non créé

```
String [] ss;
ss [0] = "Hello world"; // EXCEPTION
```

NullPointerException

- tentative d'accès à un élément hors tableau

```
double[] ds = {3.14, 2.91};
System.out. println (ds [2]); // EXCEPTION
```

ArrayIndexOutOfBoundException

Les tableaux à plusieurs dimensions

Crédit photo

Tableaux de tableaux

- Les éléments d'un tableau peuvent être des tableaux. . .

  . . .de tableaux de tableaux de tableaux de tableaux de tableaux

- **Exemple** : **int** [][] t

Tableau - Création

ArrayCreationExpression :
`new` *TypeName Dims ArrayInitializer*
`new` *TypeName DimExprs Dims*(opt)

On peut créer un tableau en donnant :

- les valeurs ;
- les tailles.

Création en utilisant un *ArrayInitializer*

Tableau - Création

ArrayCreationExpression :
`new` *TypeName Dims ArrayInitializer*

Dims :
`[ ]`
Dims `[ ]`

ArrayInitializer :
`{` *VariableInitializers*(opt) `,`(opt) `}`

VariableInitializers :
VariableInitializer
VariableInitializers `,` *VariableInitializer*

VariableInitializer :
Expression
ArrayInitializer

Tableau - Création

Exemples

```
int [] is;
is = new int[] {1, -2, 3}; // or
int [] is = new int[] {1, -2, 3}; // or
int [] is = {1, -2, 3};
```

```
Video[] videos;
videos = new Video[] {
    new Video("Alexandre Astier", "Kaamelott"),
    new Video("Mike Judge", "Idiocracy"),
    null
};
```

Tableau - Création

Exemples

```
Piece [][] pieces = {                                   // if class Piece exists
    {new Piece(), new Piece(), new Piece()},
    {new Piece(), new Piece(), new Piece()},
    {new Piece(), new Piece(), new Piece()},
};
```

```
int [][] pascal = {
    {1},
    {1, 1},
    {1, 2, 1},
    {1, 3, 3, 1},
    {1, 4, 6, 4, 1}
};
```

Tableau - Création

Rappel : Les tableaux sont des types **références**

Tableau - Création

Remarques :

- chaque case a un type ;
- il s'agit bien d'un tableau de tableaux ;
- chaque élément peut être de taille différente ;
- **chaque** tableau (intermédiaire) **connait sa taille** pascal.length et aussi pascal[i].length

Création en donnant des tailles

Tableau - Création

ArrayCreationExpression :
`new` *TypeName DimExprs Dims*(opt)

DimExprs :
DimExpr
DimExprs DimExpr

DimExpr :
`[` *Expression* `]`

Dims :
`[ ]`
Dims `[ ]`

- Expression est un **int**, positif, peut être nul

Tableau - Création

Exemples

```
int []  is  = new int[3];
int [][]  iss  = new int [3][2];
```

Rappel : Valeur par défaut (0, **false**, **null**)

Tableau - Création

Exemples

```
int [][] iss ;
iss = new int [3][];
iss [1] = new int[] {3, 14};
```

Tableau - Création

Quizz : Les créations suivantes sont-elles correctes ?

- ☐ **int** [][] entierss = **new int**[2];
- ☐ **int** [][] entierss = **new int**[] {1,2};
- ☐ **int** [][] entierss = **new int**[] {{1},{2}};
- ☐ **int** [][] entierss = **new int**[3][2] {{1,2},{3,4}};
- ☐ **int** [][] entierss = **new int**[][] {1,2};
- ☐ **int** [][] entierss = **new int**[][] {**null**, **null**};
- ☐ **int** [][] entierss = **new int**[][] {**null**,{}};
- ☐ **int** [][] entierss = {**null**,{}};

Parcours d'un tableau

Crédit photo

Tableaux - Parcours

Exemple de parcours

```
MyObject[][] moss = new MyObject[4][2];

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

- Possibilité de modification des éléments du tableau
  moss[i][j] = **new** MyObject(...)
- Comment parcourir colonne par colonne ?

Tableaux - Parcours

Exemple de parcours avec un foreach

```
MyObject[][] moss = new MyObject[4][2];

for (MyObject[] mos : moss) {
    for (MyObject mo : mos) {
        System.out.print(" " + mo);
    }
    System.out.println("");
}
```

- Possibilité d'envoyer un message à un objet : mo.foo()
- Pas de parcours colonne par colonne

Tableaux - Arrays

La classe java.util.Arrays est une classe utilitaire

binarySearch()	recherche dans le tableau
copyOf	copie tout ou partie du tableau
fill ()	remplit le tableau
sort()	trie le tableau
toString	représentation du tableau
deepToString	représentation « en profondeur »
equals	égalité des valeurs de deux tableaux
deepEquals	égalité « profonde »
...	

Copie de tableau

Crédit photo

Tableaux - Copie

Définissons un objet pour les besoins de la présentation

```
public class MyObject {
    private String description ;
    private int number;

    public MyObject(String description , int number) {
        this . description = description ;
        this .number = number;
    }

    public static MyObject newInstance(MyObject mo) {
        if (mo == null) {return null;}
        return new MyObject(mo.description, mo.number);
    }

    public void brol () {number++;}
}
```

Tableaux - Copie

Situation initiale des exemples qui suivent

```
MyObject[][] moss;
MyObject[][] copy;

moss = new MyObject[][]{
    {new MyObject("zéro", 0), new MyObject("zéro", 1), new MyObject("zéro", 2)},
    {new MyObject("un", 0), new MyObject("un", 1)},
    {new MyObject("deux", 0)},
    {null},
    null
};
```

- faire la représentation mémoire de moss et de toutes les copies qui vont suivre

Tableaux - Copie

1

Copie très superficielle

```
copy = moss;
```

Tableaux - Copie

2

Copie en utilisant java.util.Arrays.copyOf()

```
copy = Arrays.copyOf(moss, moss.length);
```

- copyOf permet de copier tout le tableau, une partie ou plus (en complétant avec **null**)

Tableaux - Copie

3

Copie en profondeur en utilisant Arrays.copyOf

```
copy = new MyObject[moss.length][];
for (int i = 0; i < moss.length; i++) {
    copy[i] = mos[i]==null
        ? null
        : Arrays.copyOf(moss[i], moss[i].length );
}
```

Tableaux - Copie

4

Copie en profondeur en utilisant ces « bonnes vielles boucles »

```
copy = new MyObject[moss.length][];
for (int i = 0; i < moss.length; i++) {
    copy[i] = new MyObject[moss[i].length];
    for (int j = 0; j < moss[i].length; j++) {
        copy[i][j] = moss[i][j];
    }
}
```

Tableaux - Copie

5

Copie profonde défensive

```
copy = new MyObject[moss.length][];
for (int i = 0; i < moss.length; i++) {
    copy[i] = new MyObject[moss[i].length];
    for (int j = 0; j < moss[i].length; j++) {
        copy[i][j] = MyObject.newInstance(moss[i][j]);
    }
}
```

Remarque : on suppose qu'aucune ligne n'est vide (**null**)

Tableaux - Copie

```
copy [0][0] = new MyObject("FOO", -1);

copy[1] = new MyObject[]{ new MyObject("BAR", -2),
                          new MyObject("BAR", -3) };

copy [2][0]. brol ();
```

- quel est l'effet pour chacune des situations ?

Séance 6

Les collections

Crédit photo

Les collections

Collection
List - Interface
ArrayList - LinkedList
Polymorphisme
Wrapper
Collections

Crédit photo

Collection

Définition : Une **collection** représente un groupe d'objets : ses éléments.

- tous les éléments sont de **même type** (collection de String $\neq$ collection de Video)
- certaines collections sont **triées**, d'autres **ordonnées**
- certaines collections permettent les **doublons**, d'autres garantissent l'**unicité** des éléments

List

Définition : Une **liste** est une collection d'éléments **ordonnés** accessibles par leur indice.

- la taille s'adapte à son contenu ($\neq$ tableau)
- les éléments ne sont pas nécessairement différents
- ordonnés, pas nécessairement triés
- possibilité d'**ajouter**, de **supprimer**, d'**accéder**, de **remplacer** un élément

List

Une liste satisfait au **contrat** suivant :

boolean add(E e)	ajout à la fin
void add(**int** index, E e)	insère à la position
E get(**int** index)	retourne l'élément à la pos.
E remove(**int** index)	supprime l'élément
int size ()	donne la taille de la liste
...	(cf. API `java.util.List`)

List est **générique** : le type des éléments, E est spécifié lors de la déclaration/création.

Interface

Définition : Le code qui définit un contrat s'appelle une **interface**

Exemple

```
public interface CheapList {
    void add(String s);
    int size();
}
```

Interface

Une classe peut **implémenter** une interface.

- l'indiquer via le mot clé **implements**
- définir le code de **toutes** les méthodes

Exemple

```
public class CheapListImpl implements CheapList {

    @Override
    public void add(String s) {
        // do something
    }

    @Override
    public int size() {
        return 0; // something usefull
    }
    // others methods if you want
```

Interface

Une interface

- définit un type de données
- on peut donc déclarer un objet de ce type
- mais il faut instancier une classe concrète

Exemple

```
CheapListImpl o = new CheapListImpl(); // OK
CheapListImpl o = new CheapList();     // NOK
CheapList o = new CheapListImpl();     // OK
CheapList o = new CheapList();         // NOK
```

ArrayList

La classe java.util.ArrayList est une classe qui implémente List.

Exemple

```
List<String> nombrils = new ArrayList<>();
nombrils.add("Vicky");
nombrils.add("Jenny");
nombrils.add(1, "Karine");
System.out.println(nombrils);   // ["Vicky", "Karine", "Jenny"]
```

ArrayList - get

get(**int**)	retourne un élément (utile par ex. pour le parcours)

Exemple :

```
public static void display(ArrayList<String> liste) {
    for (int i=0; i < liste.size(); i++) {
        System.out.println( i + ": " + liste.get(i) );
    }
}
```

ArrayList - foreach

Une liste est **Iterable**.
Elle peut être parcourue par un **foreach**

Exemple :

```
for (String mot : dictionnaire){
    System.out.println(mot);
}
```

- la variable `mot` prend chaque valeur de la liste
- la position de `mot` est inconnue : remplacements et suppressions impossibles

Une classe qui implémente **Iterable** peut être parcourue et donc se trouver dans un *foreach*.

LinkedList

La classe java.util.LinkedList est une classe qui implémente (aussi) List.

Exemple

```
List<String> nombrils = new LinkedList<>();
nombrils.add("Vicky");
nombrils.add("Jenny");
nombrils.add(1, "Karine");
System.out.println(nombrils); // ["Vicky", "Karine", "Jenny"]
```

- implémentation différente
- quel intérêt ?

Polymorphisme

Crédit photo

Polymorphisme

Second principe de l'orienté objet : le **polymorphisme**

Une instance et une variable peuvent être de types différents. C'est la « bonne méthode » qui sera exécutée

- la variable est de type List
- l'instance est de type ArrayList

→ c'est la méthode de la classe ArrayList qui est exécutée

Polymorphisme

Permet une définition polymorphique de méthode.

```
public static void display(List<String> liste) {
    for (int i=0; i < liste.size(); i++) {
        System.out.println(i + ": " + liste.get(i));
    }
}
```

```
List<String> l1 = new ArrayList<>();
List<String> l2 = new LinkedList<>();
...
display(l1);
display(l2);
```

Polymorphisme

La méthode `display` pourrait aussi s'écrire :

```
public static void display(List<String> liste) {
    for (String s : liste) {
        System.out.println(s);
    }
}
```

- On perd alors la valeur de l'indice

Types primitifs et listes

Crédit photo

Les wrappers

Seuls les objets sont permis dans les listes.
Peut-on avoir une liste de **int** ?

- existence de **wrapper** (**enveloppe**)
- englobe une valeur primitive dans un objet

boolean :	Boolean	**int** :	Integer
byte :	Byte	**long** :	Long
char :	Character	**float** :	Float
short :	Short	**double** :	Double

Les wrappers

Exemple

```
List<Integer> list = new ArrayList<>();
list.add(1);
int nombre = list.get(0);
Integer nombre2 = list.get(0);
```

Les conversions du type primitif vers son *wrapper* et vice versa sont automatiques (**boxing** / **unboxing**)

Les wrappers

Les *wrappers* sont aussi des classes utilitaires

- `Integer.parseInt(String s)`
- `Integer.valueOf(String s)`
- `Integer.toBinaryString(int i)`
- ...

Collections

java.util.Collections propose des services pour les listes

max (List l)	donne le maximum d'une liste
sort (List l)	trie une liste
reverse (List l)	inverse une liste
shuffle (List l)	mélange une liste
...	...

Exemple :

```
List<Card> cards = new ArrayList<>();
cards.add(new Card(CardColor.DIAMOND, 1));
// continue to fill the deck
Collections.shuffle(cards);
```

Autres collections

Il existe d'autres collections classiques :

- les **ensembles**, java.util.Set, sont des collections d'éléments *non ordonnés* et *uniques* ;
- les **tableaux associatifs** ou dictionnaire (*map*), java.util.Map, sont des collections d'éléments qui associent à un ensemble de *clés*, une valeur. Chaque *clé* est unique. Les valeurs correspondantes non.

Séance 7

Héritage

Crédit photo

Héritage

Principe super Object(s)

Crédit photo

Héritage - Principe

Troisième principe de l'orienté objet : l'**héritage**

L'héritage est la possibilité qu'a une classe « enfant » de réutiliser les membres de sa classe « parent ».

- grâce au mot clé **extends**

Héritage - Principe

Lorsqu'une classe **hérite** d'une autre[1], elle

- possède les mêmes attributs
  - peut en ajouter
- possède les mêmes méthodes
  - peut en ajouter
  - peut les **réécrire**[2]
- les visibilités restent de mise (**protected**)

1. Lorsqu'elle peut hériter. Cfr. `final`, **sealed classes**
2. Sous certaines conditions. Cfr. `final`

Héritage - Principe

Exemple

Reprenons la classe `Point` ...

Héritage - Principe

...et définissons la classe `ColoredPoint`

super
Crédit photo

Héritage - super

super, à l'instar de **this** permet d'accéder aux membres du parent

- dans un contexte d'attribut **super**.foo,
- dans celui de constructeur **super**(x,y) et
- dans le contexte de méthode **super**.bar()

```java
package pbt.cours;

import java.awt.Color;

public class ColoredPoint extends Point {
    private Color color;

    public ColoredPoint(double x, double y) {
        super(x,y);
        color = Color.BLACK;
    }

    @Override
    public void display(){
        // Attributs privés
        // System.out.println("(" + x + "," + y + ") - " + color);
        System.out.println("(" + getX() + "," + getY() + ") - " + color);
    }




    public static void main(String[] args) {
        Point p1 = new Point(1,4);
        Point p2 = new ColoredPoint(2,3);

    }

}
```

Crédit photo

Héritage et polymorphisme

Rappel (polymorphisme) : là où on attend une classe on peut trouver une classe enfant

- À la compilation : type déclaré
- À l'exécution : type réel

Exemple

```
Point p = new ColoredPoint(...); // OK
ColoredPoint cp = new Point(...); // NON
System.out.println( p.getColor() ); // Refusé par le compilateur
p.display(); // méthode de ColoredPoint
```

Un point coloré est un point...
avec des fonctionnalités supplémentaires

Object

La classe parent de toutes les classes est Object

Toutes les classes héritent de cette classe et donc de :

String toString()	représentation textuelle
boolean equals(Object o)	égalité sémantique
int hashCode()	*hash* associé à chaque objet
Object clone()	retourne une copie de l'objet

Object

String toString() | représentation textuelle

- cette méthode « doit » être **réécrite** (*override*)
- par défaut retourne le type et le hash de l'objet

Exemple

```
@Override
public String toString() {
    return "(" + x + "," + y + ")";
}
```

Object

boolean equals(Object o) | égalité sémantique

- permet de **définir** quand deux objets sont égaux
- par défaut, retourne ==
- réflexive, symétrique et transitive

Not all implementations of equals() are equal.
Angelika

Object

Exemple

```
public boolean equals(Object o) {
    if (this == o) return true;      // Réponse rapide si même objet
    if (o == null || getClass() != o.getClass()) return false;
    // ou if (!(o instanceof Point)) return false;
    Point p = (Point) o;
    return this.x == p.x && this.y == p.y;
}
```

- Que faire avec un point coloré ?

Object

L'implémentation de equals est polémique

- utilisation de getClass
- utilisation de **instanceof**
- rendre la classe ou la méthode equals **final**

Stackoverflow, prefer getClass over instanceof
Angelica Lanker, secrets of equals

Object

int hashCode() | *hash* associé à chaque objet

- deux objets « equals » $\Rightarrow$ même hashcode
- hashcodes différents $\Rightarrow$ objets differents (*contraposée*)
- objets différents $\not\Rightarrow$ hashcodes différents (*mais c'est mieux*)

```
Objects.hash( <attributs> )
```

Object

Object clone() | retourne une copie de l'objet

- méthode mal conçue dont l'usage est polémique
- il est préférable d'utiliser un **constructeur par copie**

Copy Constructor versus Cloning. Josh Bloch
Essayer d'implémenter correctement la méthode clone. *pbt*
Java practice, copy constructors

Object

Constructeur par copie

Exemples

```
public Point(Point point) {
    this(point.x, point.y);
}

// alternative, use of static method
public static Point newInstance(Point point) {
    return new Point(point.x, point.y);
}
```

Objects

Objects

La classe Objects est une classe utilitaire

- **static boolean** equals(Object o1, Object o2)
- **static boolean** deepEquals(Object o1, Object o2)
- **static** T requireNonNull(T t)
- **static int** hash(Object ... values)

Objects

- **static boolean** equals(Object o1, Object o2)

Exemple

```
Objects.equals(o1, o2)
// remplace
o1 == o2 || o1 != null && o1.equals(o2)
```

- **static boolean** deepEquals(Object o1, Object o2)

idem que equals sauf dans le cas de tableaux

Objects

- **static** T requireNonNull(T t)

Exemple

```
this.bar = Objects.requireNonNull(bar);

// remplace
if (bar == null) {
    throw new NullPointerException("bar is nul");
}
this.bar = bar;
```

Objects

- **static int** hash(Object ... values)

Exemple

```
@Override
public int hashCode() {
    return Objects.hash(x,y);
}
```

varargs

varargs

Principe java permettant d'écrire des méthodes acceptant un **nombre variable d'arguments**.

Une même méthode foo(String ... ss) peut être appelée

```
foo("one");
foo("one","two");
String [] ss = {"one", "two", "three"};
foo(ss);
```

... et le paramètre reçu est un **simple tableau**

Java permet d'hériter d'**une** seule classe et d'implémenter **plusieurs** interface

JDK8> Une interface peut implémenter des **default methods**.

If it quacks, it's a duck

Si ça fait coin-coin, c'est un canard

Crédit photo

Héritage

Héritage ou implémentation ?

Éviter l'héritage par implémentation (*implementation inheritance*, **extends**) et préférer l'héritage par interface (*interface inheritance*, **implements**)

Why extends is evil

Héritage

La méthode display du slide 228 peut devenir en utilisant l'héritage par interface et le fait que les listes sont **itérables** :

```
private static void display( Iterable <String> iter){
    for (String s : iter ) {
        System.out. print (s + " ");
    }
    System.out. println ();
}
```

- l'appel avec une liste : display (l)
- l'appel avec un ensemble (`Set`) : display (s)

Séance 8

Énumération
Exceptions
`var`

Crédit photo

Énumération

Crédit photo

Énumération

Définition : Une **énumération** est un ensemble fixe et petit de valeurs sémantiquement liées

- Printemps - Été - Automne - Hiver
- Pique - Cœur - Carreau - Trèfle
- Janvier - Février - Mars - Avril - Mai - ...
- Haut - Bas - Gauche - Droite
- Rouge - Vert - Bleu

Énumération

En Java :

- Un type à part entière
- Les instances sont décrites dans la classe
- Elles portent un nom et sont constantes (nom en majuscule)
- Impossible d'en créer d'autres par la suite (constructeur privé)

Énumération

ClassDeclaration:
    NormalClassDeclaration
    EnumDeclaration

EnumDeclaration:
    {*ClassModifier*} **enum** *Identifier* [*Superinterfaces*] *EnumBody*

Exemple

```
$cat Saison.java
```

```
public enum Season {
    SPRING, SUMMER, AUTUMN, WINTER;
}
```

Énumération

L'énumération est comme une classe...
avec des fonctionnalités en plus :

- comme une classe :
  - c'est un type à part entière ;
  - elle peut avoir des **attributs** et des **méthodes** ;
- avec des fonctionnalités en plus :
  - conversion automatique vers une chaine ;
  - peut apparaitre dans un **switch** ;
  - fournit un tableau des valeurs de l'énumération ;

Énumération

Exemples

- Saison, version élémentaire [Saison-1.pdf]
- Saison, version « classe » [Saison-2.pdf]

Énumération

Anciennement Java utilisait des **constantes numériques** pour simuler la notion d'énumération.

```
final int SEASON_SUMMER = 1;
final int SEASON_AUTUMN = 2,
final int SEASON_WINTER = 3,
final int SEASON_SPRING = 4;
```

Les exceptions

Crédit photo

Les exceptions

Quand le code risque de devenir incohérent

- La méthode attend un nombre entre 1 et 5 et reçoit -1.
- La méthode doit déplacer en position (4,5) mais les règles du jeu ne le permettent pas.
- Il faut ouvrir **ce** fichier... qui n'existe pas.
- ...

Les exceptions

Lancer une exception

```
throw new IllegalArgumentException("Why ?");
```

Attraper une exception

```
try {
    // code that may throw an exception
} catch (IllegalArgumentException e){
    // error handling
}
```

Les exceptions

Hiérarchie des exceptions

Source

Les exceptions

- Throwable, tous les objets « jetables »
- Error, les exceptions qui **ne doivent pas** être gérées
- Exception, les exceptions qui **doivent** être gérées (exceptions **contrôlées**)
- RuntimeException, les exceptions qui **peuvent** être gérées

Les exceptions

Exceptions **contrôlées** par le compilateur

- il faut préciser qu'une telle exception est lancée
- utilisation de **throws**

```
public void myMethod() throws MyException {
    // ...
    throw new MyException("My raison");
    // ...
}
```

Les exceptions

Une **exception contrôlée** doit :

- être gérée (**try catch**)

```
try {
    myMethod();
} catch (MyException ex) {
    // handle exception
}
```

- ou être relancée (**throws**)

```
public void otherMethod() throws MyException {
    myMethod();
}
```

Comment créer ses propres exceptions ?

Les exceptions

Créer une exception

```
public class MyException extends Exception{
    public MyException(String s){
        super(s);
    }
}
```

- MyException est une sous-classe de Exception, elle est **contrôlée**, si c'est une sous-classe de RuntimeException, elle **n'est pas** contrôlée
- utilisation du mot clé **extends**
- le constructeur fait appel au constructeur parent via le mot clé **super**

(re) catch

Attraper une exception

Une exception est un **objet**

- méthode getMessage()
- méthode printStackTrace()

```
e.getMessage();
```

Attraper une exception

Plusieurs catch

Un **try** peut avoir plusieurs **catch**

```
try {
    // code
} catch (MyException e1) {
    // code
} catch (Exception e2) {
    // code
}
```

Attraper une exception

Catch multiple

Un **try** peut avoir un *catch multiple*

- utilisation de l'opérateur |

```
try {
    // code
} catch (MyException | IOException e) {
    // code
}
```

Attraper une exception

Try with resources

Certains objets Java sont **closeable**

```
String path = "elsewhere/myfile";
try (BufferedReader br = new BufferedReader(new FileReader(path))) {
    return br.readLine();
} catch (NoSuchFileException e){
    // traitement de l'absence du fichier
}
```

Inférence de type

Inférence de type

L'**inférence de type** pour les déclarations de variables locales avec initialiseurs est la possibilité de **ne pas répéter** le type d'une variable locale lors de sa déclaration et d'utiliser le mot réservé var.

Inférence de type

Exemples

```
int i = 5;
String s = "Hello world";
ArrayList<Integer> l = new ArrayList<>();
Reader reader = new BufferedReader(
    new InputStreamReader(connection.getInputStream()));
```

```
var i = 5;
var s = "Hello world";
var l = new ArrayList<Integer>();
var reader = new BufferedReader(
    new InputStreamReader(connection.getInputStream()));
```

Le respect des conventions de nommage est *encore plus* important.

Inférence de type

Lignes de conduite

1 Choisir un nom de variable qui donne de l'information utile.

```
// Before (and bad)
List<Customer> l = connexion.executeQuery(query);
```

```
// Good
var customers = connexion.executeQuery(query);
```

Inférence de type

2 Minimiser la portée des variables.

```
var values = new int[4];
values[2] = 42;
for(var value : values){
        System.out.print(value + " - ");
}
```

```
var values = new ArrayList<Integer>();
```

Changer le type de `values` introduit un bug immédiatement visible.

Inférence de type

...ce qui ne serait pas le cas si la déclaration de variable est loin de son utilisation

```
var values = new ArrayList<Integer>();

// a lot of code

values[2] = 42;
for(var value : values){
    System.out.print(value + " - ");
}
```

Inférence de type

3 Utiliser **var** quand l'initialiseur donne suffisament d'informations.

```
// Original
ByteArrayOutputStream outputStream = new ByteArrayOutputStream(...);
BufferedReader reader = Files.newBufferedReader(...);
List<String> stringList = List.of("a", "b", "c");

// Good
var outputStream = new ByteArrayOutputStream(...);
var reader = Files.newBufferedReader(...);
var stringList = List.of("a", "b", "c");
```

Inférence de type

4 Ne pas trop s'inquiéter de l'utilisation des interfaces.

```
// Good
List<String> words = new ArrayList<>(...);
// Bad
ArrayList<String> words = new ArrayList<>(...);
```

Dans le cadre d'une **variable locale** et non d'un attribut, de paramètres de méthodes ou d'un type de retour, ce n'est pas important.

```
var words = new ArrayList<String>(...);
```

Inférence de type

5 Être attentif avec l'opérateur *diamond* et les génériques.

```
List<String> words = new ArrayList<String>(...);
List<String> words = new ArrayList<>(...);
var words = new ArrayList<String>(...);
var words = new ArrayList<>(...);
```

1. redondant. Répétition inutile (depuis JDK8)
2. *good*
3. *good*
4. la liste, est une liste d'`Object` :-(

Et aussi...

Lire le code est plus important que de l'écrire.

Le code devrait être clair lors d'une lecture d'un extrait.

La lisibilité du code ne doit pas dépendre d'un IDE.

Séance 9

Un peu de fonctionnel...

Crédit photo

Un peu de fonctionnel

Expression lambda
Étude de cas 1 : Trier
Étude de cas 2 : For each
Étude de cas 3 : Filtrer
Récapitulons

Crédit photo

Le lambda calcul

Expression λ (lambda) et **programmation fonctionnelle** (JDK8>)

```
x -> x*x
(a,b) -> a>b
(a,b) -> a>b ? a : b
```

Une expression λ est « une écriture compacte pour une classe anonyme implémentant une interface ne contenant qu'une seule méthode »

Introduisons le concept au travers de quelques études de cas

Étude de cas 1
Trier

Crédit photo

Un tri simple

L'API fournit des méthodes pour trier tableaux et listes

```
int [] tab = {23, 42, 7, 14, 16, 3};
Arrays.sort(tab);
```

```
List<String> l = Arrays.asList("Pomme","Poire","Abricot");
Collections.sort(l);

// better
var fruits = Arrays.asList("Pomme","Poire","Abricot");
Collections.sort( fruits );
```

Trier des objets personnels

Comment trier des **objets non standards** (ex : Video) ?
Il faut que la classe soit Comparable (cf. API)

- Interface : **int** compareTo(T o)
- Définit l'**ordre naturel**
- Utilisée par l'algorithme pour comparer deux éléments

Trier des objets personnels

Exemple

```
public class Video implements Comparable<Video> {
    @Override
    public int compareTo(Video o) {
        return this.auteur.compareToIgnoreCase(o.auteur);
    }
```

```
var videos = Videos.getRandomVideos();
Collections.sort(videos);
```

Un ordre personnalisé

Comment trier suivant un **ordre personnalisé** ?
(ex : les vidéos selon le nb de likes)

Il faut fournir un Comparator (cf. API)

- Interface : `int compare(T o1, T o2)`
- Définir une classe implémentant cette interface
- Passer une instance à une autre version de sort
  `void sort(T[] t, Comparator<T> c)`
- Utilisée par l'algorithme pour comparer deux éléments

Un ordre personnalisé

Exemple

```
public class VideoLikesComparator implements Comparator<Video>{
    @Override
    public int compare(Video o1, Video o2) {
        return Integer.compare(o1.getNbLikes(), o2.getNbLikes());
    }
}
```

```
public static void main(String[] args) {
    Video[] tab = {
        new Video("Alexandre Astier", "Kaamelott", true, 1_236_722),
        new Video("Dominique A", "Au revoir mon amour", true, 455_262),
        new Video("Michael Launey", "Dimensions Stade foot", true, 64_598)
    };
    Arrays.sort(tab, new VideoLikesComparator());
    System.out.println(Arrays.toString(tab));
}
```

Class anonyme

Lourd si la classe n'est utilisée qu'**une seule fois**.

La solution ? Une **classe anonyme** (à usage unique)

Exemple

```
public static void main(String[] args) {
    // ...
    Arrays.sort(tab, new Comparator<Video>() {
        public int compare(Video o1, Video o2) {
            return Integer.compare(o1.getNbLikes(), o2.getNbLikes());
        }
    });
}
```

Mais pas très lisible...

Une expression λ

Java 8 a introduit les **expressions λ** (lambda)

- Peut être vu comme une écriture compacte pour une classe anonyme implémentant une interface ne proposant qu'une seule méthode
- Donc un bout de code que l'on peut passer à une méthode pour qu'elle l'utilise.

Exemple

```
Arrays.sort(tab, (v1, v2) -> Integer.compare(v1.getNbLikes(),v2.getNbLikes()) );
```

Étude de cas 2

Itérer une collection

Crédit photo

Présentation du problème

Supposons que l'on veuille liker toutes les vidéos d'une liste

```
List<Video> videos = Arrays.asList(
    new Video("Alexandre Astier", "Kaamelott", true, 1_236_722),
    new Video("Dominique A", "Au revoir mon amour", true, 455_262),
    new Video("Michael Launey", "Dimensions Stade foot", true, 64_598)
);
```

Via un for each

On peut utiliser un **for each**

```
public void likerAll (List<Video> videos) {
    for (Video v : videos) {
        v.liker();
    }
}
```

Via la méthode forEach

Java 8 : **méthode forEach** et λ

```
public void likerAll (List<Video> videos) {
    videos.forEach( v -> v.liker() );
}
```

Que l'on peut aussi raccourcir en (**method reference**)

```
public void likerAll (List<Video> videos) {
    videos.forEach( Video::liker );
}
```

En parallèle

Code précédent plus compact mais pas plus rapide.

On peut **paralléliser** l'exécution

```
public void likerAll (List<Video> videos) {
    videos.parallelStream().forEach( Video::liker );
}
```

List.of *versus* Arrays.asList

Remarque Deux méthodes pour créer facilement une liste :

- `Arrays.asList` crée une liste de taille fixe. Les éléments sont modifiables

```
var values = Arrays.asList("a", "b");
values.set(1,"c");
values.add("d"); // ERROR
```

- `List.of` crée une liste immuables.

```
var values = List.of("a", "b");
values.set(1,"c"); // ERROR
values.add("d"); // ERROR
```

Étude de cas 3

Filtrer une collection

Crédit photo

Présentation du problème

Supposons que l'on veuille ne garder que la liste de vidéos les plus likées

En Java classique, on écrirait

```
public List<Video> mostLiked(List<Video> videos, int limit) {
    var mostLiked = new ArrayList<Video>();
    for(Video v : videos) {
        if( v.getNbLikes() >= limit ) {
            mostLiked.add( v );
        }
    }
    return mostLiked;
}
```

Via un stream

Java 8 : utilisation de Stream.

```
public List<Video> mostLiked(List<Video> videos, int limit) {
    return videos.stream()
        . filter ( v -> v.getNbLikes() >= limit )
        . collect ( Collectors . toList ());
}
```

Un stream parallèle

À nouveau, l'exécution peut être **parallélisée**

```
public List<Video> mostLiked(List<Video> videos, int limit) {
    return videos.parallelStream()
        .filter( v -> v.getNbLikes() >= limit )
        .collect( Collectors.toList());
}
```

Récapitulons...

Crédit photo

Stream

Le **stream** Java est très puissant

- Étape 1 : **Création**
- Étape 2 : Opérations intermédiaires : **Manipulations**
  - transforment le stream
  - peuvent être chainées
- Étape 3 : Opération finale : **Réduction**
  - produit autre chose qu'un stream
  - une seule permise

Étape 1 - Créer un Stream

De nombreuses possibilités d'en **créer** un

- Via une liste (déjà vu)
- Via un tableau : Arrays.stream(monTab)
- Via des méthodes de génération

```
Stream<Integer> s1 = Stream.generate(clavier::nextInt );
Stream<Integer> s2 = Stream.iterate(1, n->n+1);
IntStream si2 = new Random().ints(1, 100);
```

Remarques

- streams **infinis**
- IntStream : version spécialisée de Stream avec des méthodes numériques en plus

Étape 2 - Opérations intermédiaires

Opération **intermédiaire** :

- filter : déjà vu (ne garde que certains éléments)
- limit : ne garde que les premiers éléments

```
Stream<Integer> si = Stream.iterate(1, n->n+1).limit(100);
```

- sorted : trie le stream

```
Stream<Video> s = videos.stream().sorted();
```

- map : applique une méthode sur chaque élement

```
Stream<String> s = videos.stream().map(Video::getAuteur);
Stream<Integer> si = Stream.iterate(1, n -> n + 1).limit(100)
                                .map(x -> x*x);
```

- ...

Étape 3 - Opérations terminales

Opération **terminale** :

- forEach : déjà vu (équivalent du for each)
- collect : déjà vu (convertit le stream en liste)
- (any| all |none)Match : teste le stream

```
var hasUnder100Likes = videos.stream()
    .anyMatch( x->x.getNbLikes()<100 );
```

- find (First |Any) : cherche un élement

```
// what ?
var anything = videos.stream().findAny();
```

Étape 3 - Opérations terminales

Opération **terminale** :

- count : compte le nombre d'éléments

```
int nbBests = videos.stream()
    . filter ( v -> v.getNbLikes() >= limite )
    .count();
```

- sum/average : somme/moyenne (sur des numériques)

```
int sumLikes = videos.stream().map(Video::getNbLikes)
    .mapToInt(Integer::intValue).sum();
```

- max/min : cherche le maximum/minimum

```
int maxLikes = videos.stream().map(Video::getNbLikes)
    .mapToInt(Integer::intValue)
    .max().orElse(-1);
```

Stream

Références

- « Understanding Java 8 Streams API » par Amit Phaltankar
- « Java 8, Sreams et Collectors » par José Paumard
- « JDK8, les nouveautés », blog de Pierre Bettens
- « Java Streams Tutorial », sur `java2s.com`

Stream - Exercices

Pour aller plus loin...

1. Lire la documentation de Comparator et voir comment indiquer que l'on veut inverser l'ordre d'un tri.
2. Récrire l'instruction qui trie un tableau de vidéos suivant le nb de likes en utilisant une référence de méthode (::).

Stream - Exercices

Pour aller plus loin

3. Écrire une méthode qui reçoit une liste de vidéos et compte combien elle contient d'auteurs différents.
4. Écrire une méthode qui reçoit une liste de vidéos et affiche les 3 vidéos les plus likées.

Stream - Exercices

Pour aller plus loin... et au delà

5. Écrire une méthode qui retourne un tableau des n premiers nombres premiers.
6. Écrire une méthode qui retourne un tableau des n premiers nombres satisfaisant une propriété passée en paramètre via une expression lambda.

Séance +1

Le codage des fichiers
Trouver son chemin
Entrées-sorties

Crédit photo

Le codage des fichiers

Binaire
Textuel

Crédit photo

Le codage des fichiers

Coder l'information de manière **binaire** ou **textuelle**

- **binaire** - représentation mémoire
- **texte** - utilisation d'une suite de caractères

Imaginons un instant que l'on sache (déjà) écrire dans un fichier et écrivons la valeur

16

binaire - texte

Le codage des fichiers

binaire

```
0000000 0010
0000001
```

- le fichier fait 1 byte
- la valeur stockée est 16 = 0x10

Le codage des fichiers

texte

```
0000000 3136
0000002
```

- le fichier fait 2 bytes
- les valeurs stockées sont les codes unicodes de 1 et 6
- `hexdump` respecte *little big endian* (`hexdump -C`)

Et si j'écrivais œ plutôt que 16 ?

Le codage des fichiers

texte

```
0000000 93c5
0000002
```

- le fichier fait 2 bytes alors qu'il s'agit d'un seul caractère
- quel est le rapport entre **93c5** et **u0153** ?

Le codage des fichiers

Le **fichier** est encodé en UTF8

Définition du nombre d'octets utilisés dans le codage (uniquement les séquences valides)

Caractères codés	Représentation binaire UTF-8	Premier octet valide (hexadécimal)	Signification
U+0000 à U+007F	0xxxxxxx	00 à 7F	1 octet codant 1 à 7 bits
U+0080 à U+07FF	110xxxxx 10xxxxxx	C2 à DF	2 octets codant 8 à 11 bits
U+0800 à U+0FFF	11100000 101xxxxx 10xxxxxx	E0 (le 2e octet est restreint de A0 à BF)	3 octets codant 12 à 16 bits
U+1000 à U+1FFF	11100001 10xxxxxx 10xxxxxx	E1	
U+2000 à U+3FFF	1110001x 10xxxxxx 10xxxxxx	E2 à E3	
U+4000 à U+7FFF	111001xx 10xxxxxx 10xxxxxx	E4 à E7	
U+8000 à U+BFFF	111010xx 10xxxxxx 10xxxxxx	E8 à EB	
U+C000 à U+CFFF	11101100 10xxxxxx 10xxxxxx	EC	
U+D000 à U+D7FF	11101101 100xxxxx 10xxxxxx	ED (le 2e octet est restreint de 80 à 9F)	
U+E000 à U+FFFF	1110111x 10xxxxxx 10xxxxxx	EE à EF	
U+10000 à U+1FFFF	11110000 1001xxxx 10xxxxxx 10xxxxxx	F0 (le 2e octet est restreint de 90 à BF)	4 octets codant 17 à 21 bits
U+20000 à U+3FFFF	11110000 101xxxxx 10xxxxxx 10xxxxxx		
U+40000 à U+7FFFF	11110001 10xxxxxx 10xxxxxx 10xxxxxx	F1	
U+80000 à U+FFFFF	1111001x 10xxxxxx 10xxxxxx 10xxxxxx	F2 à F3	
U+100000 à U+10FFFF	11110100 1000xxxx 10xxxxxx 10xxxxxx	F4 (le 2e octet est restreint de 80 à 8F)	

Le codage des fichiers

Démonstration

œ	
Code unicode	0153
En binaire	00000001 01|010011
Représentation binaire UTF-8	**110**00101 **10**010011
Représentation hexa UTF-8	c5 93

Trouver son chemin

Principe
Path(s)
Files

Avant d'utiliser un fichier, il faut le trouver...

Crédit photo

Trouver son chemin - Principe

Un **fichier** est identifié par son chemin à travers le *filesystem*, on parle aussi de

- son nom complètement qualifié (FQN - *Fully Qualified Name*),
- son Path (qui signifie *chemin*)

Exemple

- `/home/alice/java/Hello.java` ou
- `C:\Users\alice\java\Hello.java`

Trouver son chemin - Principe

Rappels

- Le séparateur (*delimiter*) est différent en fonction du *filesystem*
- Un chemin (*path*) peut être **relatif** ou **absolu**
- Certains systèmes de fichiers autorisent la notion de **lien symbolique** (*symbolic link*)

Trouver son chemin - Path

L'interface Path en java représente un chemin (*path*) et permet de le manipuler

- créer un *path*
- utiliser l'information contenue dans un *path*
- convertir un *path*
- comparer deux *path*
- ...

Trouver son chemin - Paths

La classe **Paths** est une classe utilitaire permettant de créer un *path* (une *Factory*)

Exemple

```
Path path1 = Paths.get("/tmp/foo");
Path path2 = Paths.get(System.getProperty("user.home"), "logs", "foo.log");
```

Le fichier que le chemin représente peut ne pas exister

Arguments variables

La signature de Paths.get est

```
public static Path get(String first, String ... more)
```

... (**varargs**)

- indiquent un nombre variable d'arguments
- reçoit les arguments sous forme d'un tableau

```
public void foo(String ... args) {
    System.out.println("Nb de paramètres reçus: " + args.length);
    for(String arg : args) {
        System.out.println(arg);
    }
}
```

Arguments variables

L'avantage est la facilité d'écriture dans l'appel.

Exemples

```
foo("one");
foo("one","two");
String [] ss = {"one", "two", "three"};
foo(ss);
```

Trouver son chemin - Path

Path fournit des méthodes pour interroger un chemin

Exemples

Soit la déclaration (dans un contexte linux)

```
Path path = Paths.get("/home/alice/foo");
```

- `toString` ⟶ /home/alice/foo
- `getFileName` ⟶ foo
- `getParent` ⟶ /home/alice
- `getRoot` ⟶ /

Trouver son chemin - Path

Convertir un *path*

- **`toAbsolutePath()`** : vers un chemin absolu

```
// Si pwd = /home/alice
Path path = Paths.get("foo");
System.out.println(path.toAbsolutePath());          // /home/alice/foo
```

- **`resolve(Path)`**
  crée un chemin sur base de deux chemins incomplets

```
// Si pwd = /somewhere
Path path = Paths.get(".").toAbsolutePath();        // /somewhere
System.out.println("Path: " + path.resolve("file"));
                                                    // /somewhere/file
```

Trouver son chemin - Files

La classe Files est une classe utilitaire du package NIO.2.

Elle propose les méthodes newInputSream, newOutputStream, newBufferedReader et newBufferedWriter nous y reviendrons.

Trouver son chemin - Files

Outre ces méthodes, on y trouve aussi :

exists (Path, LinkOption ...) - notExists (Path, LinkOption ...) - isReadable(Path) - delete (Path) - copy(Path, Path, CopyOption...) - move(Path, Path, CopyOption...) - getLastModifiedTime(Path, LinkOption ...) - size (Path) (et autres métadonnées) - createTempFile(String, String)

...et plein d'autres

Entrées-sorties bas niveau

Principe
Fichiers binaires
Fichiers textes

101010

Crédit photo

I/O bas niveau - Types de fichiers

En Java, les fichiers sont des flux (stream) :

- **uniformes**
- **non structurés**
- **binaire** ou **texte**

I/O bas niveau - Vue d'ensemble

Java fournit des classes de bas niveau
pour lire / écrire des **byte** / **char**

I/O bas niveau - Lecture binaire

Lecture binaire dans un fichier

Exemple

```
int b;
try (InputStream in = Files.newInputStream( Paths.get("FILE"),
                                            StandardOpenOption.READ)) {
    b = in.read();
    while (b != -1) {
        System.out.print(" " + b);
        b = in.read();
    }
} catch (IOException e) {
    System.out.println("Error: " + e.getMessage());
}
```

I/O bas niveau - Lecture binaire

Lecture binaire dans un fichier (suite)

- InputStream permet la lecture binaire de bas niveau
- Files est une classe utilitaire
- newInputStream retourne un *input stream* sur le fichier concerné
- StandardOpenOption est une énumération
- l'InputStream est **Closeable**, il sera fermé automatiquement par le *try with resources*
- IOException sont les exceptions liées aux I/O

I/O bas niveau - Lecture binaire

Exercice

- lire avec ce code un fichier contenant 16
- interpréter le résultat obtenu

```
49 54 10
```

I/O bas niveau - Écriture binaire

Écriture binaire dans un fichier

Exemple

```
try (OutputStream out = Files.newOutputStream( Paths.get("FILE2"),
                                                StandardOpenOption.CREATE)) {
    out.write(64);
} catch (IOException e) {
    System.out.println("error : " + e.getMessage());
}
```

`int InputStream.read()` et `OutputStream.write(int)` traitent des **bytes** alors que leur paramètre est un **int**.

Pourquoi ?

I/O bas niveau - Lecture/Écriture texte

Mêmes principes que pour les fichiers binaires mais avec des classes adaptées

- FileReader pour lire un fichier texte
  - **int** read() lit un caractère (-1 si fin de fichier)
- FileWriter pour écrire un fichier texte
  - **void** write(**int** c) écrit le caractère stocké dans c
- BufferedReader et BufferedWriter
  - versions bufférisées
  - cf. Files .newBufferedReader/Writer

I/O bas niveau - Lecture/Écriture texte

Exercices

- Écrire un bout de code permettant de lire un fichier texte
- Écrire un bout de code permettant d'écrire dans un fichier texte la phrase « Hello world »

Les fichiers (haut niveau)

- Flux englobants
- Données primitives
- Expressions régulières
- Scanner
- Flux standards
- Console
- Sérialisation

Crédit photo

I/O haut niveau - Flux englobant

L'API java propose une série de **flux englobants**

- pour *bufferiser* ;
- traiter les objets ;
- ...

Un flux englobant se construit à partir d'un autre flux

I/O haut niveau - Données primitives

Pour **écrire** des données **primitives** dans un fichier **binaire** on se base sur la classe DataOutputStream

Exemple

```
try (DataOutputStream out = new DataOutputStream(
            Files.newOutputStream(Paths.get("file.data"),
            StandardOpenOption.CREATE))) {
    out.writeBoolean(true);
    out.writeUTF("Hello");
    out.writeDouble(2.5);
} catch (IOException e) {
    System.err.println("Error: " + e.getMessage());
}
```

I/O haut niveau - Données primitives

Pour la **lecture** à partir d'un fichier **binaire**, c'est la classe DataInputStream

Exemple

```
try (DataInputStream in = new DataInputStream(
            Files.newInputStream(Paths.get("file.data"),
            StandardOpenOption.READ))) {
    boolean b = in.readBoolean();
    String s = in.readUTF();
    double d = in.readDouble();
    System.out.println("values:" + b + " " + s + " " + d);
} catch (IOException e) {
    System.err.println("Error: " + e.getMessage());
}
```

I/O haut niveau - Données primitives

Remarques

- Pas de valeur sentinelle ; génère une EOFException si tentative de lecture au-delà de la fin du fichier.
  - ajout d'un **catch**

```
catch (EOFException e) {
    // all data are read
}
```

- Le *try with resource* s'occupe du close

Les expressions régulières

Crédit photo

I/O haut niveau - Expressions régulières

Les **expressions régulières** (*regex*) permettent de vérifier qu'une chaine correspond à un certain schéma (*pattern*)

- voir la classe Pattern
- la classe String propose une méthode matches

`g\d{5}, \d+, [a-z]{5-}`

I/O haut niveau - Scanner

Lecture de données **primitives** depuis un fichier **texte**, *via* la classe Scanner

- le constructeur accepte : String, Path, InputStream, Reader, ...

```
try (Scanner scanner = new Scanner(Paths.get("file"))) {
    int i = scanner.nextInt();
    System.out.println("i: " + i);
} catch (IOException e) {
    System.err.println("Error: " + e.getMessage());
}
```

I/O haut niveau - PrintWriter

Écriture de données **primitives** dans un fichier **texte**, *via* la classe PrintWriter

- PrintWriter est un flux englobant
- il propose les méthodes println , print et printf

```
try (PrintWriter out = new PrintWriter(Files.newBufferedWriter(
            Paths.get(" file "), StandardOpenOption.CREATE))) {
    out. println (10);
    out. printf ("%04d\n%4.2f", 12, 12.2);
} catch (IOException e) {
    System.err. println ("Error: " + e.getMessage());
}
```

Nous avons déjà utilisé ces classes, méthodes avec System.out et System.in

Qui sont-ils ?

I/O haut niveau - Flux standards

Les 3 flux standards

- l'entrée standard, System.in est un InputStream
- la sortie standard System.out et la sortie d'erreur standard System.err sont des PrintStream

Exercice : Détailler System.out. println ()

I/O haut niveau - Console

Pour des entrées sorties *via* la console. La classe Console peut se substituer à Scanner **et** à des System.out

Exemple

```
Console console = Sytem.console();
// if console not null
String name = console.readLine("Enter name: ");
console.format("Your name is %s", name);
char[] password = console.readPassword("Password: ");
// some work
Arrays. fill (password, ' ');
```

Sérialisation
Crédit photo

I/O haut niveau - Sérialisation

Dès lors qu'un objet est **sérialisable** (Serializable), il pourra être transformé en une suite d'octets

- chaque attribut doit être sérialisable
- Serializable est une interface de *tag*
- la classe doit être la même à l'écriture et à la lecture (serialVersionUID)

I/O haut niveau - Sérialisation

Exemple

```
try (ObjectOutput out = new ObjectOutputStream(
                Files.newOutputStream( Paths.get("file.ser"),
                StandardOpenOption.CREATE))){
    out.writeObject(new MyObject("Anonymous object", 7));
} catch (IOException e) {
    System.err.println ("Error: " + e.getMessage());
}

// extrait d'une lecture
MyObject mo = (MyObject) in.readObject();
```

Séance +2

Le temps

Crédit photo

Le temps

LocalDate
LocalTime
LocalDateTime
Instant
ZoneId
ZonedDateTime
Mise en page
Calendriers

Crédit photo

Le temps - Un sujet très complexe

- Que signifie "10/05/2012" ? (différentes **notations**)
- Quand exactement était-on le 12 janvier 2016 à 10h ? (**fuseau horaire**)
- Quand exactement était-on le 25 octobre 2015 à 2h30 à Bruxelles ? (**heure d'hiver**)
- Est-ce que ajouter 1 jour, c'est ajouter 24 heures ? (vision calendrier vs vision **durée**)
- Quelle année sommes-nous ? (tout le monde n'utilise pas le même **calendrier**)

Le temps - Plusieurs tentatives

Java 1.0 (1996) : java.util.Date.
Juste un point dans le temps ; pas de fuseau

Java 1.1 (1997) : java.util.Calendar.
Fuseaux horaires mais de nombreux défaut

Joda Time (2001) : Un projet indépendant, de qualité

Java 8 (2014) : Date and Time API. (java.time.*)
Standard largement inspiré de Joda Time

Le temps - Écriture normalisée

Un **temps** s'écrit sous la norme **ISO8601**

yyyy-mm-ddThh :mm :ss[TZ]

Exemple

1946-02-01T00:00:00Z
1995-05-23T12:00:00Z
1977-04-22T01:00:00-05:00

Lien Wikipedia

Le temps - LocalDate

LocalDate : représente une date

- Pas d'heure ni de fuseau horaire
- Pas de constructeur ;
  utilisation de **fabrique statique** (*static factory*)

```
LocalDate d1 = LocalDate.now();
LocalDate d2 = LocalDate.parse("1946-02-01");
LocalDate d3 = LocalDate.of(2016, Month.JUNE, 10);
```

- Des getters

```
int  year  = d3.getYear();                  // 2016
Month month = d3.getMonth();                // JUNE
int  dom = d3.getDayOfMonth();              // 10
DayOfWeek dow = d3.getDayOfWeek();          // TUESDAY
```

Le temps - LocalDate

- Des méthodes utiles

```
int  len  = d3.lengthOfMonth();          // 30 (jours en juin)
boolean leap = d3.isLeapYear();          // false (pas bissextile)
boolean before = d1.isBefore(d3);
```

- Pas de mutateur (**immuable**) :
  toute modification crée un nouvel objet

```
LocalDate date = LocalDate.of(2016, Month.JUNE, 10);
date = date.withYear(2015);        // 2015-06-10
date = date.plusMonths(2);         // 2015-08-10
date = date.minusDays(1);          // 2015-08-09
```

- On peut donc **chainer** les appels

```
date = date.withYear(2015).plusMonths(2).minusDays(1);
```

Le temps - LocalTime

LocalTime : représente un moment dans la journée

- Pas de jour ni de fuseau horaire
- Fonctionnement similaire à LocalDate

```
LocalTime t1 = LocalTime.now();
LocalTime t2 = LocalTime.parse("10:30:00");
LocalTime t3 = LocalTime.of(20, 30);
int hour = t3.getHour();            // 20
int minute = t3.getMinute();        // 30
t1 = t3.withSecond(6);              // 20:30:06
t1 = t3.plusMinutes(3);             // 20:33:06
boolean after = t1.isAfter(t3);
```

Le temps - LocalDateTime

LocalDateTime : combine les deux

```
LocalDateTime dt1 = LocalDateTime.of(2014, Month.JUNE, 10, 20, 30);
LocalDateTime dt2 = LocalDateTime.of(d1, t1);
LocalDateTime dt3 = d1.atTime(20, 30);
LocalDateTime dt4 = d1.atTime(t1);
```

Le temps - Instant

Instant : représente un **temps « machine »**

- Un « point » sur la droite du temps (**timestamp**)
- Représente le nombre de secondes écoulées depuis le 1 janvier 1970 à 00h00 au méridien de Greenwich
- Stocké dans un **long**
- Non lié à un fuseau horaire
  $\longrightarrow$ impossible de poser des questions liées au calendrier (jour, heure...)

```
Instant now = Instant.now();
Instant parse = Instant.parse("1946-02-01T01:30:00Z");
```

Le temps - ZoneId

ZoneId : représente un fuseau horaire (**timezone**)

- Relatif à GMT ou lié à un endroit

```
ZoneId plus1 = ZoneId.of("GMT+1");
ZoneId bxl = ZoneId.of("Europe/Brussels");
// Quel est le décalage pour l'instant ?
System.out.println(bxl.getRules().getOffset(Instant.now()));
// Quand aura lieu le prochain changement d'heure ?
Instant chgt = bxl.getRules().nextTransition(Instant.now()).getInstant();
```

Le temps - ZonedDateTime

ZonedDateTime :
représente une date dans un fuseau donné

```
ZonedDateTime now = Instant.now().atZone(ZoneId.of("Europe/Brussels"));
System.out. println (now);
System.out. println (now.getDayOfMonth());
System.out. println (now.getHour());
```

Le temps - Mise en page

De nombreuses possibilités de **mettre en page** un temps

```
ZonedDateTime now = Instant.now().atZone(ZoneId.of("Europe/Brussels"));
DateTimeFormatter fs = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
DateTimeFormatter fl = DateTimeFormatter.ofLocalizedDate(FormatStyle.LONG);
DateTimeFormatter fc = DateTimeFormatter.ofPattern("dd MMMM");
System.out. println (now.format(fs ));
System.out. println (now.format( fl ));
System.out. println (now.format(fc ));
System.out. println (now.format( fl .withLocale(Locale.ENGLISH)));
```

Le temps - Calendriers

Local* (ISO) se basent sur le calendrier Grégorien mais il y en a d'autres

Exemples

```
ZonedDateTime now = Instant.now().atZone(ZoneId.of("Europe/Brussels"));
System.out. println (Chronology. getAvailableChronologies ());
System.out. println (HijrahDate.now());
System.out. println (JapaneseDate.now());
```

Le temps - Conclusion

La spécification java 310 (**JSR 310**) définit l'*usage du temps* :

5 packages - 39 classes - 13 enums - 4 exceptions - 13 interfaces

. . .nous en avons parcouru une partie

Le temps - Conclusion

Références

- « Intuitive, Robust Date and Time Handling, Finally Comes to Java » par Stephen Colebourne
- « Les dates en Java 8 » par Lemoine
- « Java Date Time Introduction », sur `java2s.com`

Séance +3

Mise à jour

Crédit photo

Mises à jour

JDK 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19

Crédit photo

Mises à jour

Quelques notions non inclues dans ces notes ?

- JDK 9 [lien]
  - `JShell`
- JDK 10, 11, 12 [lien]
  - *Launch single-file source-code programs* (JDK 11)
- JDK 13, 14, 15, 16, 17 [lien]
  - *Helpful NullPointerExceptions* (JDK 14)
  - *Pattern matching for instanceof* (JDK 16)
  - *Records* (JDK 16)
  - Les classes scellées (*sealed class*) (JDK 17)
- JDK 18, 19 *todo*

Crédits

Ces slides sont le support pour la présentation orale de l'activité d'apprentissage **DEV2-JAV** à HE2B-ESI

Crédits

Les distributions **Ubuntu** et/ou **debian**
du système d'exploitation GNU Linux.
LaTeX/Beamer comme système d'édition.
Git et **GitHub** pour la gestion des versions et le suivi.
GNU make, rubber, pdfnup, ... pour les petites tâches.

Images et icônes

deviantart, **flickr**, **The Noun Project**

BY-NC-SA