

DEV2 - Le Langage Java

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



Haute École Bruxelles-Brabant
École Supérieure d'Informatique

Année académique 2020 / 2021



Séance 1

Introduction L'orienté objet

Introduction

Objectifs
Moyens
Évaluation



Crédit photo

Objectifs

L'introduction à la programmation est faite . . .

- ▶ Continuité de l'apprentissage du langage
- ▶ Familiarisation avec les spécifications du langage
- ▶ Approche de l'API Java
- ▶ Programmation orientée objet

Supports et ressources

Rien

- ▶ pas de syllabus
- ▶ pas de livre

Quoique

- ▶ sur poÉSI
 - les slides ;
 - des liens ;
 - des documents...



Évaluation

cfr. modalités de l'UE

L'orienté objet

Motivation

Vocabulaire

Classe

Constructeur

Instanciation

Membre

Accesseur-Mutateur

Visibilité

this

static

Encapsulation



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 qu'on peut en faire.

Connaissez-vous des types OO prédéfinis ?

Orienté objet - Motivation

Apparu suite aux limites de la programmation procédurale.

Permet d'écrire des programmes plus :

- ▶ **lisibles** ;
- ▶ **compacts** ;
- ▶ **robustes**.



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** les 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 qu'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 :

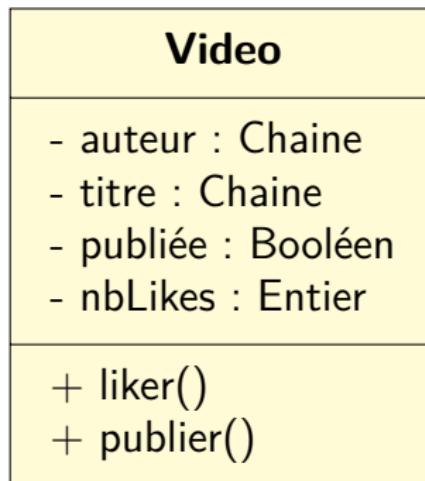
- possède un auteur
- possède un titre
- est publiée ou non
- a un certain nombre de "likes"

► 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
- auteur = "Alexandre Astier" - nom = "Kaamelott" - publiée = vrai - nbLikes = 2870
+ liker() + publier()

micmathFoot : Video
- auteur = "Mickaël Launay" - nom = "Dimensions idéales terrain foot" - publiée = faux - nbLikes = 0
+ liker() + publier()

Orienté objet - Définir la classe

```
public class Video {  
  
    private String auteur;  
    private String titre ;  
    private boolean publiée;  
    private int nbLikes;  
  
    public void liker () {  
        nbLikes++;  
    }  
  
    public void publier () {  
        publiée = true;  
    }  
}
```

- ▶ Ne pas confondre attributs et variables locales
- ▶ Pas de **static** pour les méthodes

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)

Initialisation prise en charge par un **constructeur**.

Exemple

```
public Video (String unAuteur, String unTitre) {  
    auteur = unAuteur;  
    titre = unTitre;  
    publiée = false;  
    nbLikes = 0;  
}
```

Orienté objet - Instantiation

Pour instancier :

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

Exemple : instantiation 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 - Instantiation

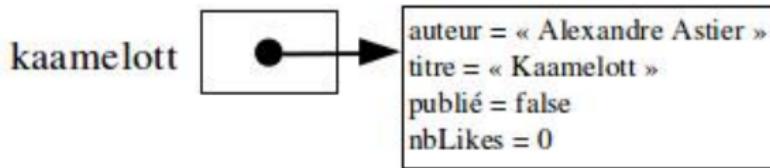
Une classe est un type **référence** (comme les tableaux)

Exemple :

```
Video kaamelott; // référence créée sur la pile
```

kaamelott ?

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



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 unAuteur, String unTitre) {  
    if(unAuteur==null || unAuteur.length()==0) {  
        throw new IllegalArgumentException("Auteur invalide: " + unAuteur);  
    }  
    auteur = unAuteur;  
    titre = unTitre;  
    publiée = false;  
    nbLikes = 0;  
}
```

⇒ Un objet est toujours créé dans un **état valide**

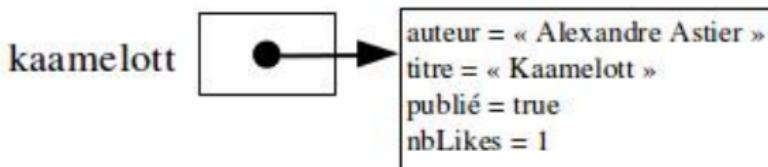
« Le développeur ou la développeuse est responsable de la cohérence de l'objet au sein de la classe. »

Orienté objet - Appel d'une méthode

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

Exemple

```
public static void main(String[] args) {  
    Video kaamelott = new Video("Alexandre Astier", "Kaamelott");  
    kaamelott.publier();  
    kaamelott.liker();  
}
```



Orienté objet - Membre privé

Impossible d'accéder aux attributs (privés)

Exemple

```
public static void main(String [] args) {  
    Video kaamelott = new Video("Alexandre Astier", "Kaamelott");  
    System.out.println (kaamelott.titre); // ne compile : attribut privé  
    kaamelott.nbLikes = 1_000_000; // idem  
}
```

- ▶ Pourquoi
Pour éviter de donner des valeurs invalides
- ▶ Que faire ?
Fournir des méthodes publiques

Orienté objet - Accesseur

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

Exemple

```
public String getAuteur() {return auteur;}  
public String getTitre() {return titre ;}  
public int getNbLikes() {return nbLikes;}  
public boolean isPubliée() {return publiée ;}
```

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

Orienté objet - Accesseur

Exemple : Utilisation

```
public static void main(String [] args) {  
    Video kaamelott = new Video("Alexandre Astier", "Kaamelott");  
    System.out.println (kaamelott.getNbLikes()); // 0  
    kaamelott.liker ();  
    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 setTitre(String unTitre) { titre = unTitre;}
```

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.getTitre() );
kaamelott.setTitre ("Kaamelott – Livre 1 – Tome II");
System.out.println( kaamelott.getTitre() );
```

Orienté objet - Mutateur robuste

Le mutateur peut vérifier le paramètre

Exemple : Mutateur avec test de validité

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

Permettrait de refuser

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

⇒ Un objet reste toujours dans un **état valide**

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 `liker ()` ?

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 - Surcharge de constructeur

On peut fournir plusieurs constructeurs :

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

Exemple

```
public Video (String unAuteur, String unTitre,  
              boolean publiée, int likes) {  
    auteur = unAuteur;  
    titre = unTitre;  
    publiée = publiée;  
    nbLikes = likes;  
}
```

Orienté objet - Surcharge de méthode

La surcharge est permise aussi pour les méthodes

Exemple

```
public void liker () {  
    liker (1);  
}  
  
public void liker (int nbFois) {  
    nbLikes = nbLikes + nbFois;  
}
```

Des exemples dans l'API Java ?

this



Crédit photo

Orienté objet - this

this est une référence à soi-même.

Elle apparait dans différents contextes :

- ▶ constructeur **this()** ;
- ▶ attribut **this. titre** ;
- ▶ méthode **this. liker ()**.

Orienté objet - this

Exemple

```
public Video (String auteur, String titre , boolean publiée, int likes ) {  
    this.auteur = auteur;  
    this.titre = titre ;  
    this.publiée = publiée;  
    nbLikes = likes ;  
}  
  
public Video (String auteur, String titre ) {  
    this(auteur, titre , false ,0);      // doit être la première instruction !  
}  
  
public void liker () {  
    this.liker (1);  
}
```

static



Credit photo

Orienté objet - static

Quelle différence avec ce que l'on faisait avant ?

static

Trois types de classes :

- ▶ classe utilitaire (`Math`) ;
- ▶ classe « objets » (`String`, `Scanner`) ;
- ▶ classe mixte.

Orienté objet - static

Un membre **static** :

- ▶ fait référence à la classe et non à une instance ;
- ▶ est **partagé** par toutes les instances (éventuelles).

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

Méthode statique

- ▶ ne peut pas accéder aux membres des instances
- ▶ utilisation courante : méthodes non objets

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

Remarque : opérateur conditionnel ? :

? : (grammaire simplifiée)

ConditionalOperator ? :

BooleanExpression ? *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 - 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. liker ()`

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

Orienté objet - import static

import static crée un raccourci pour l'accès aux membres statiques

Exemple

```
import static java.lang.Math.log;
import static java.lang.Math.E;
public class Test {
    public static void main( String [] args ) {
        System.out.println ( log(E) );
    }
}
```

Exemple

```
import static org.junit.Assert.*;
```

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 - Et la suite...

Y-a-t'il d'autres principes ?

Oui ! Notre étude de l'OO n'est pas finie ;)

Nous aborderons encore :

- ▶ l'héritage ;
- ▶ le polymorphisme ;
- ▶ les interfaces.

Orienté objet - Exercice



Donnez des exemples de code OO écrit en DEV1.

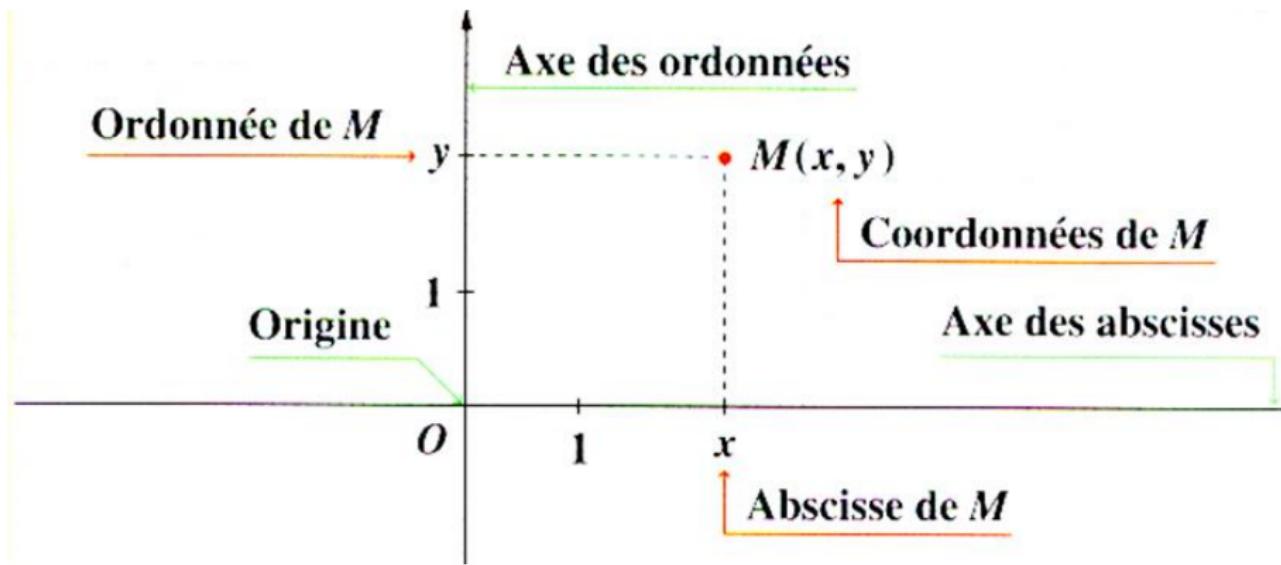


Séance 2

Orienté objet
(étude de cas)

Orienté objet - Étude de cas - Point

Représentons un point dans le plan



Orienté objet - Étude de cas - Point

Premier projet

Point
- x : Réel - y : Réel
+ display() + getX() : double + getY() : double

Orienté objet - Étude de cas - Point

Ajout d'un **constructeur**

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

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

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

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

Grammaire

Principe Fonctionnement Lexicale Syntaxique

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

Java Language Specification Introduction

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 11 Edition

James Gosling

Bill Joy

Guy Steele

Gilad Bracha

Alex Buckley

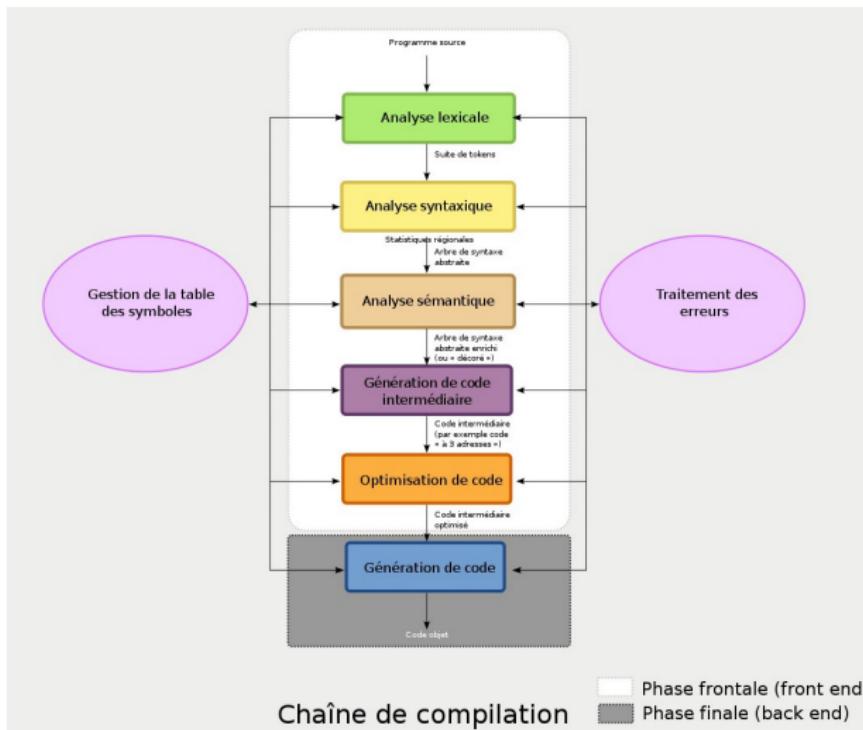
Daniel Smith

Grammaire - Fonctionnement

Une grammaire est une description finie de l'infinie 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



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

*Nombr*e :

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 È

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.

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 : one of *Identifier Keyword Separator Operator Literal*

Keyword : 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)

Separator : one of

() { } [] ; , @ ::

Operator : one of

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

Grammaire - Literal

Literal:

IntegerLiteral
FloatingPointLiteral
BooleanLiteral
CharacterLiteral
StringLiteral
NullLiteral

BooleanLitteral: one of
 true false

NullLiteral:
 null

Pour les autres littéraux, consulter la spécification . . .
[jls8] p 25-38(45-58)

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 [jls8] p 462(482)
- ▶ Les **instructions**
- ▶ Les **expressions-instructions**



Séance 4

Instruction-expression Les instructions

Expression-instruction

Assignment
Incrémantion
Appel de méthode
Instanciation

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



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++;` \approx `++i;` \approx `i+=1;` \approx `i=i+1;`

`++i` et `i++` sont des expressions

- ▶ Elles ont une valeur
- ▶ Mais elles ont aussi un effet (l'incrémantation)
- ▶ 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

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

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("auteur", "titre")
new Point(2,3)
```

Instructions

Revue des troupes . . .

Crédit photo

Les instructions

Statement :

EmptyStatement
Block
LabeledStatement
BreakStatement
ContinueStatement
ExpressionStatement
IfThenStatement
IfThenElseStatement
SwitchStatement
WhileStatement
DoStatement
ForStatement
ReturnStatement
AssertStatement
SynchronizedStatement
ThrowStatement
TryStatement

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);  
}
```



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();
```

Et de ceci (*dangling else*) ?

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

Switch

SwitchStatement:

`switch (Expression) SwitchBlock`

SwitchBlock:

`{ {SwitchBlockStatementGroup} {SwitchLabel} }`

SwitchBlockStatementGroup:

`SwitchLabels BlockStatements`

SwitchLabel:

`case ConstantExpression :
case EnumConstantName :
default :`

-
- ▶ L'expression est de type ; **byte**, **char**, **short**, **int**, **Byte**, **Character**, **Short**, **Integer**, **String** ou **enum**

Switch

Utilisation d'un **break** pour sortir d'un **switch**

Exemple

```
switch( nbFois ) {  
    case 3:  
        System.out.println ("Bonjour !");  
    case 2:  
        System.out.println ("Bonjour !");  
    case 1:  
        System.out.println ("Bonjour !");  
    default:  
}
```



Les boucles

while
do-while
for

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*

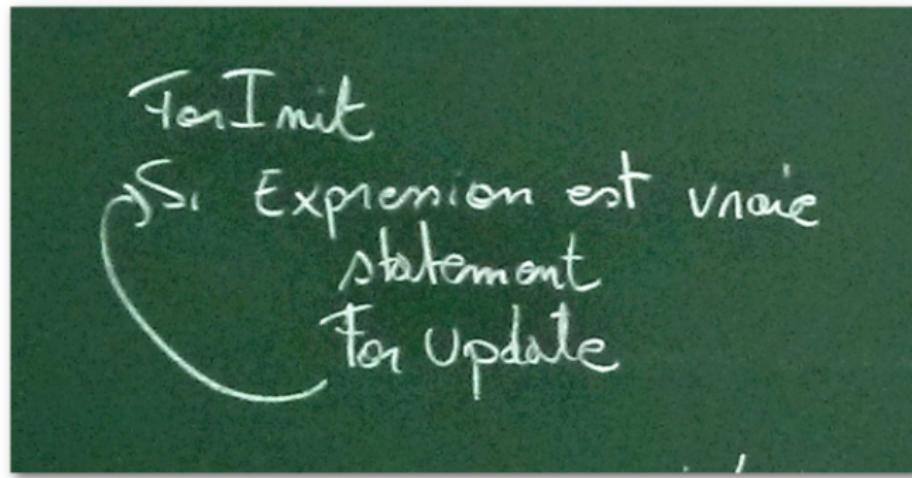
Évaluation de l'expression représentant le **test**

- ▶ 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", "Slinky", "Potato", "Woody", "Sarge", "Etch",
    "Lenny", "Squeeze", "Wheezy", "Jessie", "Stretch", "Buster"}
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

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 );  
}
```



Credit photo

```
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;  
    }  
}
```

~
~
~
~
~
~
~

Les instructions

Statement :

EmptyStatement

Block

LabeledStatement

BreakStatement

ContinueStatement

ExpressionStatement

IfThenStatement

IfThenElseStatement

SwitchStatement

WhileStatement

DoStatement

ForStatement

ReturnStatement

ThrowStatement

TryStatement

AssertStatement

SynchronizedStatement



Séance 5

Tableaux
(le retour)

Retour sur 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

Tableaux de tableaux

- ▶ Un tableau est un type de données
- ▶ Les éléments d'un tableau peuvent être des tableaux...
 ... de tableaux de tableaux de tableaux de tableaux de tableaux
- ▶ **Exemple :** `int [][] t`

Tableau - Cration

ArrayCreationExpression :

new TypeName Dims ArrayInitializer
new TypeName DimExprs Dims_(opt)

On peut crer 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};
```

```
Video [] videos;  
videos = new Video[] {  
    new Video("Alexandre Astier", "Kaamelott"),  
    new Video("Mickaël Launay", "Dimensions idéales terrain foot"),  
    null  
};
```

Dans une déclaration, version simplifiée permise

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

Tableau - Crédit

Exemples

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

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

Tableau - Crédit

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

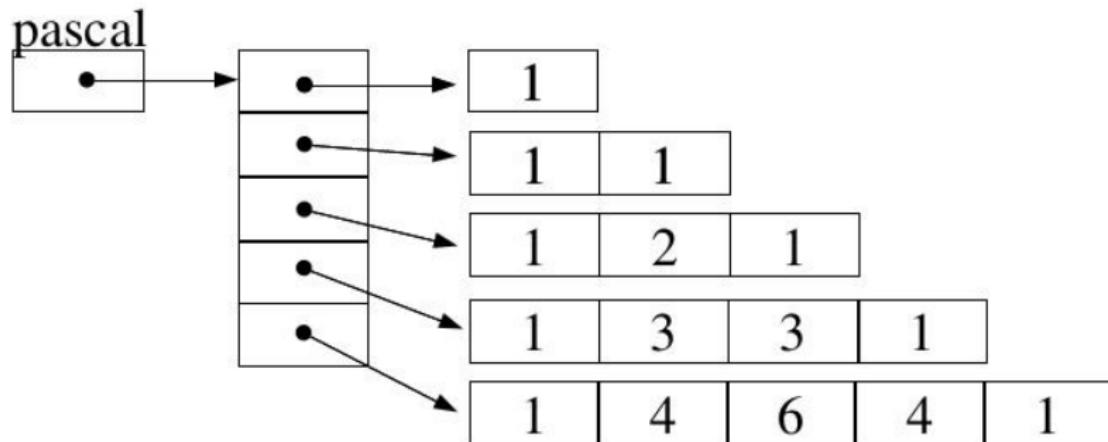


Tableau - Création

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 []

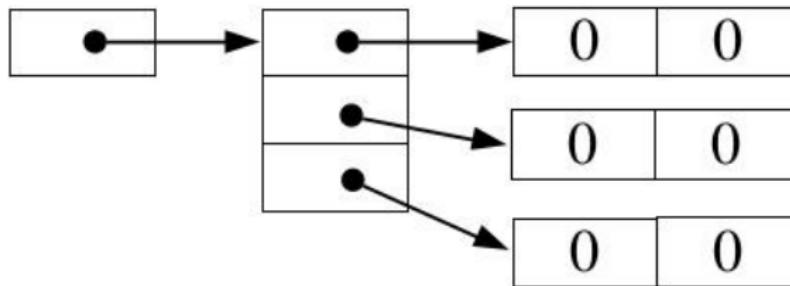
- ▶ **Expression** est un **int**, positif, peut être nul

Tableau - Crédit

Exemples

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

iss



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

Tableau - Crédit

Exemples

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

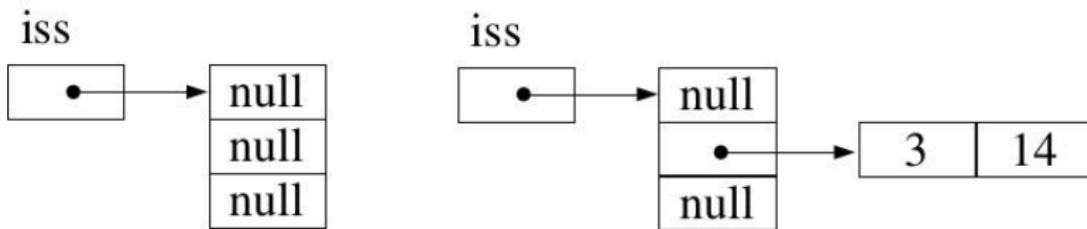


Tableau - Création

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

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

Copie de tableau



ÉCOLE FRANÇAISE DU XIX^{ME} SIECLE
Paysage. Sur les bateaux de pêche.
Huile sur toile. 65x81 cm.



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 vieilles 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]);
    }
}
```

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

Les collections

Collection

List - Interface

ArrayList - LinkedList

Polymorphisme

Wrapper

Collections

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 <code>java.util.List</code>)

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 MyInterface {  
    public void foo(int i);  
    public boolean isBar(char c);  
}
```

Interface

Une classe peut **implémenter** une interface.

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

Exemple

```
public class MyClass implements MyInterface {  
    public void foo(int i) {  
        // do something interesting  
    }  
    public boolean isBar(char c) {  
        return true; // or false or 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

```
MyObject o = new MyObject();      // OK
MyObject o = new MyInterface();   // Non
MyInterface o = new MyObject();   // OK
MyInterface o = new MyInterface(); // Non
```

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 polymorphe 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 );
```

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

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

Exemple :

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

Pour aller plus loin



Quelques questions pour aller plus loin ...

- ▶ Peut-on mélanger un tableau ?
- ▶ Dans un code qui contient :
`ArrayList<String> l = new ArrayList<>();`
est-ce qu'on peut remplacer `ArrayList` par
`LinkedList` ?



Séance 7

Héritage

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, elle

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

Héritage - Principe

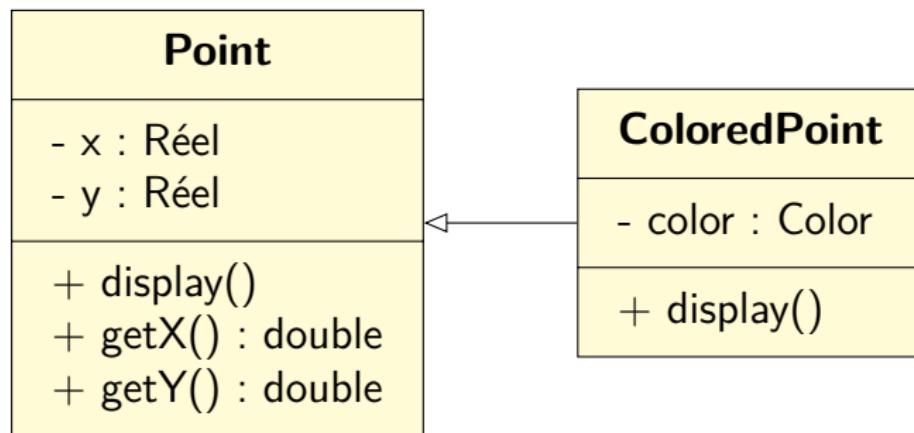
Exemple

Reprenez la classe Point ...

Point	
- x :	Réel
- y :	Réel
+ display()	
+ getX()	: double
+ getY()	: double
...	

Héritage - Principe

... et définissons la classe ColoredPoint



super

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()**

```
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

- ▶ Le compilateur utilise le type déclaré
- ▶ La machine virtuelle utilise le 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 <code>toString()</code>	représentation textuelle
boolean <code>equals(Object o)</code>	égalité sémantique
int <code>hashCode()</code>	<i>hash</i> associé à chaque objet
Object <code>clone()</code>	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

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 this)) 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**

Object

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

- ▶ deux objets « equals » \Rightarrow même hashcode
- ▶ hashcodes différents \Rightarrow objets différents
(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**

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);

// replace
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);  
}
```

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

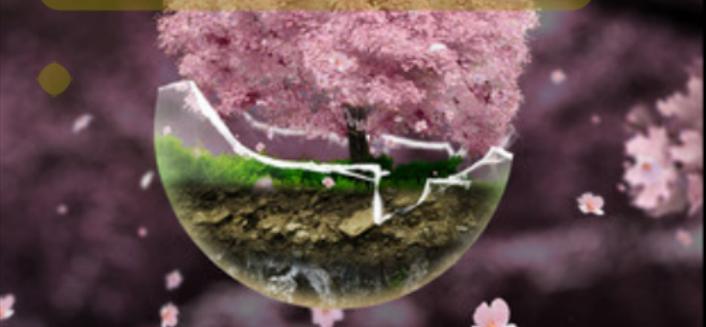


Séance 8

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 Saison {  
    PRINTEMPS, ÉTÉ, AUTOMNE, HIVER;  
}
```

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 chaîne ;
 - peut apparaître 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 SAISON_ÉTÉ = 1;  
final int SAISON_AUTOMNE = 2,  
final int SAISON_HIVER = 3,  
final int SAISON_PRINTEMPS = 4;
```

Retour sur les exceptions



Credit photo

Retour sur les exceptions

Rappels

Lancer une exception

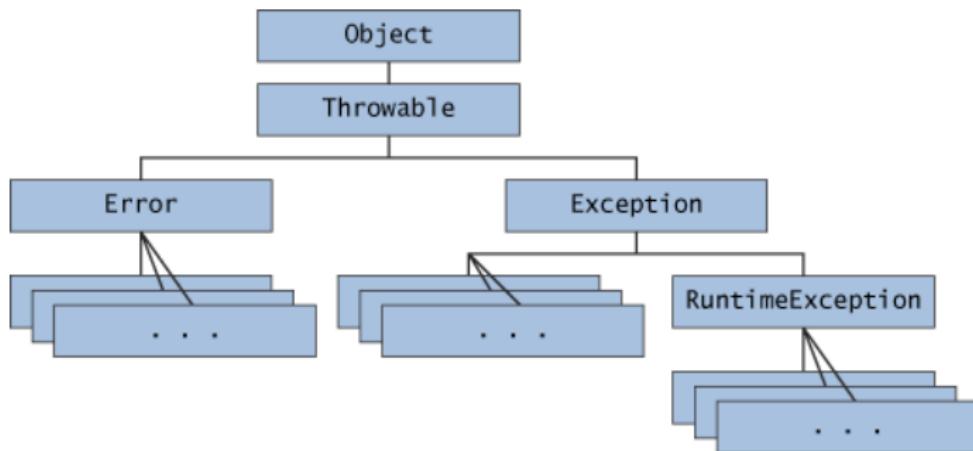
```
throw new IllegalArgumentException("Raison");
```

Attraper une exception

```
try {  
    // code pouvant lancer une exception  
} catch (IllegalArgumentException e){  
    // gestion de l'erreur  
}
```

Retour sur les exceptions

Hiérarchie des exceptions



Source

Retour sur 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

Retour sur les exceptions

Exceptions **contrôlées** par le compilateur

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

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

Retour sur les exceptions

Une **exception contrôlée** doit :

- être gérée (**try catch**)

```
try {  
    myMethod();  
} catch (FileNotFoundException ex) {  
    // gérer l'exception  
}
```

- ou être relancée (**throws**)

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

Peut-on créer ses propres exceptions ?

Retour sur les exceptions

Créer une exception

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

- ▶ **MyException** est une sous-classe de **Exception**
- ▶ 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 = "monfichier";
try (BufferedReader br = new BufferedReader(new FileReader(path))) {
    return br.readLine();
} catch (NoSuchFileException e){
    // traitement de l'absence du fichier
}
```

Inférence de type



Crédit photo

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 suffisamment 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<>(...);
```

- ➊ redondant. Répétition inutile (depuis JDK8)
- ➋ *good*
- ➌ *good*
- ➍ 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

Le codage des fichiers
Trouver son chemin
Entrées-sorties

Le codage des fichiers

Binaire
Textuel

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

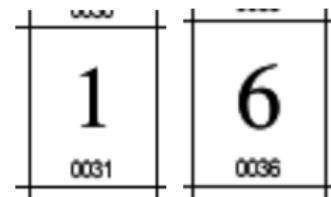
- ▶ le fichier fait 1 byte
- ▶ la valeur stockée est $16 = 0x10$

Le codage des fichiers

texte

0000000 3136

0000002

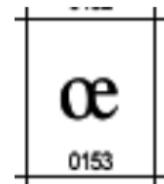


- ▶ le fichier fait 2 bytes
- ▶ les valeurs stockées sont les codes unicodes de 1 et 6

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	0xxxxxx	00 à 7F	1 octet codant 1 à 7 bits
U+0080 à U+07FF	110xxxx 10xxxxxx	C2 à DF	2 octets codant 8 à 11 bits
U+0800 à U+0FFF	1110xxxx 101xxxxx 10xxxxxx	E0 (le 2 ^e 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	1110110x 10xxxxxx 10xxxxxx	EC	
U+D000 à U+D7FF	11101101 10xxxxxx 10xxxxxx	ED (le 2 ^e 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 2 ^e 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+FFFFFF	1111001x 10xxxxxx 10xxxxxx 10xxxxxx	F2 à F3	
U+100000 à U+10FFFF	11110100 1000xxxx 10xxxxxx 10xxxxxx	F4 (le 2 ^e 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	11000101 10010011
Représentation hexa UTF-8	c5 93



Trouver son chemin

Principe
Path(s)
Files

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

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 `newInputStream`, `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

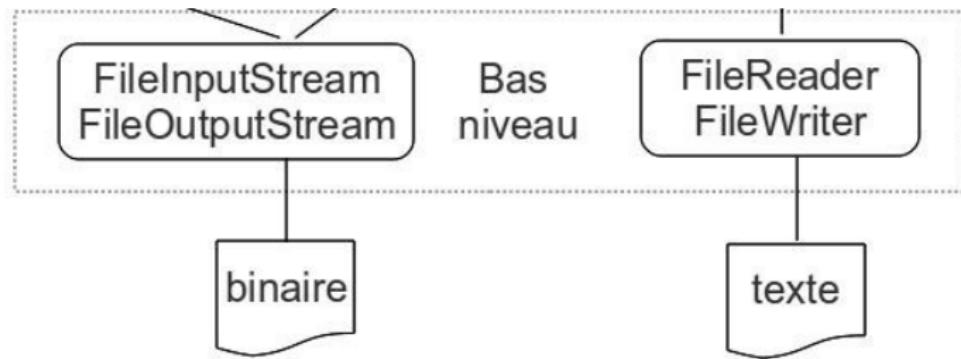
I/O bas niveau - Types de fichiers

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 »



Séance 10

Entrées-sorties
(haut niveau)

Les fichiers (haut niveau)

Flux englobants

Données primitives

Expressions régulières

Scanner

Flux standards

Console

Sérialisation

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 **binnaire** 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());
}
```

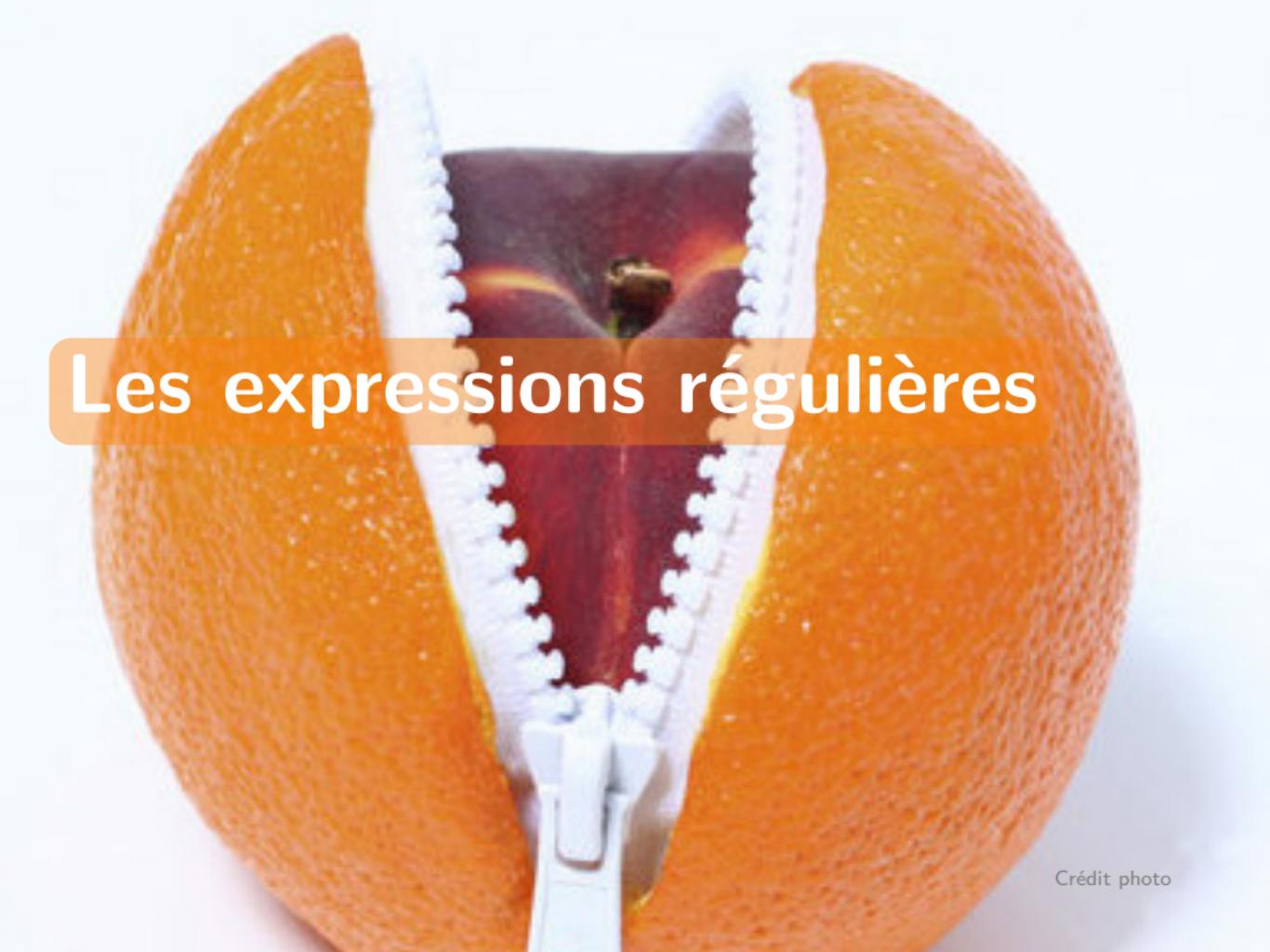
I/O haut niveau - Données primitives

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 chaîne 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`

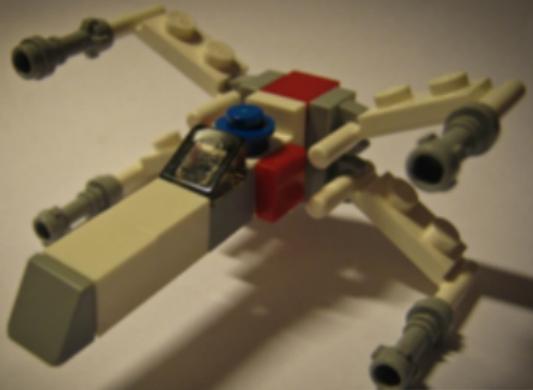
Exercice : Détailler `System.out.println()`

I/O haut niveau - Console

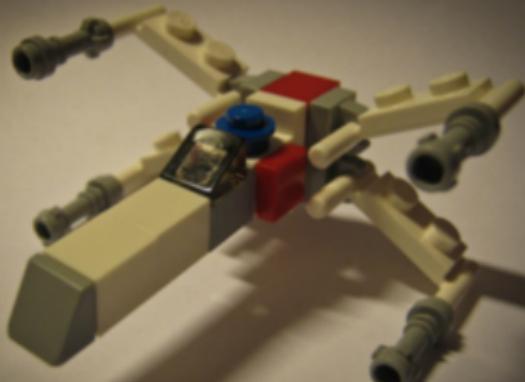
Pour des entrées sorties *via* la console. La classe **Console** peut se substituer à **Scanner** et à des **System.out**

Exemple

```
Console console = System.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 11

Un peu de fonctionnel...
Le temps



Un peu de fonctionnel

Expression lambda

Étude de cas 1 : Trier

Étude de cas 2 : For each

Étude de cas 3 : Filtrer

Récapitulons

Le lambda calcul

Java 8 a introduit de la **programmation fonctionnelle**

Expression λ (lambda)

```
x -> x*x
```

```
(a,b) -> a>b
```

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

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.compareTolgnoreCase(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 qu'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 qu'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() );  
}
```

Qu'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 qu'on veuille ne garder d'une liste de vidéos que les plus likées

En Java classique, on écrirait

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

Via un stream

Java 8 : utilisation de Stream.

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

Un stream parallèle

À nouveau on peut paralléliser

On peut **paralléliser** l'exécution

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

Récapitulons. . .



Credit 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 - Crée 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 élément

```
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 élément

```
// 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

Vous poursuivrez l'exploration de la programmation fonctionnelle en DEV₃ et DEV₄.

Références

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

Stream - Exercices



Pour aller plus loin...

- ➊ Lire la documentation de [Comparator](#) et voir comment indiquer que l'on veut inverser l'ordre d'un tri.
- ➋ 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... et au delà

- 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...

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

Le temps

LocalDate

LocalTime

LocalDateTime

Instant

ZonedDateTime

ZonedDateTime

Mise en page

Calendriers

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 (**immutable**) : 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
 - 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 - Zoneld

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

- Relatif à GMT ou lié à un endroit

```
Zoneld plus1 = Zoneld.of("GMT+1");
Zoneld bxl = Zoneld.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

Et pour conclure ce cours... Lisez !

« Qui nous devenons dépend
de ce que nous lisons
après que tous les professeurs
aient fini avec nous.
De toutes, la plus grande université
est une collection de livres. »

Thomas Carlyle (1795 - 1881)

Crédits

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

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 

