JAVA

S1 - ORIENTE OBJET

Pour un type **primitif**
- Indique la zone mémoire (sur la pile/*stack*) où se trouve la **valeur**

Nom variable
valeur

Pour un type **référence** (ex : String, tableau)
- La zone mémoire contient l'**adresse** de la zone mémoire (sur le tas/*heap*) contenant la valeur (indirection)

Un objet c'est un type référence.

Un langage orienté objet permet de créer ses propres types, liés au problème à résoudre.
Exemple : Ont créé un type étudiant, Produit, Partie, Vidéo…
Un type c'est : des valeurs possibles, ce qu'on peut en faire.

Apparu suite aux limites de la programmation procédurale (crise du logiciel).
Permet d'écrire des programmes plus lisibles, compacts, robustes.

Exemple : Sont tous les 2 des patates, ont les même attribut (yeux, nez, bouche) mais pas les mêmes instances = pas les mêmes valeurs pour leurs attribut (chaussure rouge et l'autre bleu).

Un programme peut créer et manipuler les objets. On peut envoyer un message pour changer l'attribut. Exemple : lui enlever les chaussures

Les nouveaux types qu'ont créé s'appelle une classe. Exemple : Video est une classe.

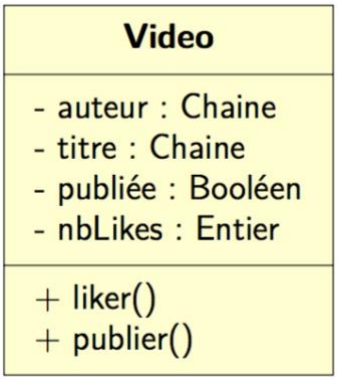
Un objet est une instance d'une classe (objet = instance)
- Construit à partir de la définition donnée par la classe ;
- Appartenant au type défini par la classe.

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

Décrire un objet dans une classe = cité tous ces attribue et les messages qu'on pourra lui envoyer.

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 ou méthodes ou constructeur



Caractéristique de la classe Vidéo :
Une vidéo :
- Possède un auteur (attribut)
- Possède un titre (attribut)
- Est publiée ou pas (attribut)
- A un certain nombre de "likes" (attribut)

On peut :
- La liker (méthode)
- La publier (méthode)

C'est un schéma ULM.
Le « - » devant les attributs c'est pour dire qu'il est privé. Et le « + » que c'est public.

JAVA

Une instance c'est une classe qui a des valeurs.

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()

Lorsque l'on définit une classe, les méthodes n'ont pas de « static »
Au sein de la classe on a 4 déclaration de « variable » ce sont des attributs. ≠ de variable.
S'écrivent « private String auteur ; », ça créé un attribut privé, donc on ne peut pas accéder à cet attribue dans un autre main. Le fait d'avoir des attributs privés et méthode public s'appelle l'encapsulation.

```
public class Video {

    private String auteur;
    private String titre ;
    private boolean publiée;
    private int nbLikes;

    public void liker () {
        nbLikes++;
    }

    public void publier () {
        publiée = true;
    }
}
```

Chaque méthode dans la classe peut accéder aux attribue de l'objet.

Instancier un objet c'est le construire en mémoire (on utilise l'obj).

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

Constructeur = initialiser des attributs, donner des valeurs aux attribut.
Ci-dessous on sait que c'est un constructeur car pas de type retour, a le même nom que la class.

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

Pour instancier :

- On utilise l'opérateur new
- On fournit les paramètres au constructeur

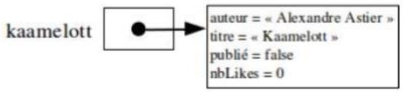
Exemple : Video kaamelott = new Video("Alexandre Astier", "Kaamelott");
Crée un nouvel objet kaamelott de type Video. Appelle le constructeur pour l'initialiser.
Objet nomDeObjet = new Objet(« param ») ;

Une classe est un type référence (comme les tableaux).
Exemple avec classe « Video ».

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

kaamelott ?

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



Int Type primitif on réserve emplacement taille ≈ 33
i = 5 le 5 est écrit sur la pille
s = « flop », l'emplacement « s » a une référence vers « flop ».

Le constructeur peut vérifier la validité des paramètres

```
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 "". ("" zone mémoire dans laquelle se trouve cette valeur)

Pour que le constructeur vérifie la validité, il faut ajouter des tests en début de constructeur.

```
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.

L'opérateur point permet d'appliquer une méthode à un objet. Son état change.
Exemple : kamelott.publier() ; il va publier la vidéo kamelott.

Impossible d'accéder aux attributs privés.

```
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.
Pour pouvoir afficher, il faut fournir des méthodes public.

Accesseur (getter)

Méthode donnant la valeur d'un attribut. Le nom de l'attribut commence par « get » ou « is » pour booléen. On rend les attributs publics.
Exemple : public String getAuteur(){return auteur ;} et pour un booleen isPubliee(){return publiee}.
On donne la valeur de auteur à getAuteur, pareil pour « publiee ».
Du coup, on peut l'utiliser dans un autre main, avant on ne pouvait pas : System.out.println(kaamelott.getNbLikes());

Mutateur (setter)

Méthode permettant de modifier l'état d'un objet (modifier un attribut)
Exemple : public void setTitre (String unTitre) { titre = unTitre;}
Le nom de l'attribut doit commencer par « set ».

```
Video kaamelott = new Video("Alexandre Astier", "Kaamelott");
System.out.println ( kaamelott.getTitre() );
kaamelott.setTitre("Kaamelott – Livre 1 – Tome II");
System.out.println ( kaamelott.getTitre() );
```

1ère ligne je fais appelle au constructeur.
2ème ligne elle va chercher la valeur de titre et l'affiche.
3ème ligne réserve le nom du nvx titre et le « mutateur » va dans l'objet et inclus dans titre, le nouveau titre.
4ème ligne affiche le nvx titre.

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;
}
```

Un objet reste toujours dans un état valide. Lorsque l'on est dans une classe on peut modifier une valeur sans setteur. (Pas bonne pratique)

JAVA

Chaque membre peut avoir comme visibilité :

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

Les attributs ont une valeur par défaut :

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

Si on ne met pas de constructeur, il y en a un par défaut. Qui n'a aucun paramètre et initialise à 0, si on ajoute un constructeur, celui par défaut n'existe plus.

Surcharge

On peut écrire plusieurs constructeurs = surcharge(overloading), doit avoir un nombre ou type de paramètres différents.

On peut aussi surcharger les méthodes :
Ici, dans la méthode « liker » on utilise l'autre « liker » qui a un paramètre.

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

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

this

« this » = une référence vers soi-même, permet à un objet de parler de lui-même. Lève une ambiguïté éventuelle.

Contexte méthode :

```
public void display(){
  System.out.println(x+ « , »+y) ; | System.out.println(this.x+ « , »+this.y) ;
}
```

Ici, c'est pour être sûre que lors de l'appelle de méthode, ce sois bien x et y de la variable. Exemple : p.display() ; pour être sûre que c'est bien ceux de « p ».
this.foo() pour être sûr d'utiliser le bon.

Contexte attribut :

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

Contexte constructeur :

Doit être la 1ère instruction. On l'utilise que s'il y a une surcharge de constructeur.
this(param1 ,param2, …)

JAVA

static

Une méthode sans « static » doit avoir un objet sur lequel agir, fais une action sur un objet. On l'appellera : « objet.methode(param) ; »
Avec « static », on l'apelle : « MaClasse.methode(param) ; » c'est juste du code que l'on veut exécuter, ne dépend pas d'objet.

Trois types de classes :

- Classe utilitaire (Math).
- Classe « objets » (String, Scanner).
- Classe mixte (utilitaire et objet dans 1 classe).

Un membre static :

- Fait référence à la classe et non à une instance.
- Est partagé par toutes les instances (éventuelles).

Attribut static :

- Existera pour la classe et toutes les instances, 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;
}
```

Méthode static :

- 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;
    }
}
```

Méthode pas static, agit sur 1 instance en particulier.

Static je ne connaît rien d'autre que ce que j'ai en paramètre.

Appel de méthodes :

- static : par le biais de la classe. Exemple : Math.sqrt(4) ;
- non-static : par une instancede la classe. kaamelott.liker() ;

import static

Je peux faire en sorte de n'écrire que « sqrt » ou « pow » en faisant
« import static java.lang.Math.sqrt ; import static java.lang.Math.pow ; » c'est un raccourcis d'écriture.

Encapsulation

C'est le 1er principe de l'orienté objet.
La cohérence de l'objet est assurée par la classe.

Le principe de garder les attribut privé et les méthode publique = encapsulation.

JAVA

S2

Exemple :

Déplacer les points :

```
public void move(double deltaX, double deltaY){
  this.x += deltaX ;
  this.y += deltaY ;
}
```

Calculer la distance :

```
public double distance(Point p){
  double distance = Math.sqrt(Math.pow(this.x-p.x, 2)+Math.pow(this.y-p.y, 2)) ;
  return distance
}
```

A moindre frais : surcharger la méthode et dans celle-ci, faire appel à l'autre méthode.

Méthode toString

Au-dessus de la méthode toString, écrire « @Override ».
@Override = réécrire. Annotation destiné aux compilateur, pour lui dire qu'on veut réécrire la méthode, afin que le compilateur vérifie qu'elle existe déjà et si elle est bien écrite.
Si on n'écrit pas le toString, lors de l'affichage du point il mettra le nom du package et des lettres et chiffres.

- 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.

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

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

S3

Grammaire

Grammaire (opt) = facultative
C'est intéressant pour pouvoir écrire un compilateur et décrire le langage.
La grammaire est décrite dans The Java Language Specification

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

- Grammaire lexicale : mot(token) doit être légal, (si le mot est valide)
- Grammaire syntaxique : séquence de mots doit être légale (un « ; » à la fin)
- Sémantique : le tout doit avoir un sens.

Exemple : El tahc tse rion problème de grammaire lexicale.
Le parapluie mange l'ascenseur problème de sémantique.

Fonctionnement d'une grammaire :

- Symbole de départ

- Règles de productions (productions)
- Symboles terminaux (token)

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

Exemple de fonctionnement de grammaire :

Un nombre décimal naturel
Nombre :
Chiffre
Chiffre Nombre
Chiffre : one of 0 1 2 3 4 5 6 7 8 9

Qu'est-ce qu'un nombre naturel ? C'est un nombre ou un chiffre. Qu'est-ce qu'un chiffre, c'est un de ces éléments : 0 1 2 3 4 5 6 7 8 9. Et qu'est-ce qu'un nombre ? c'est soit un chiffre, soit un chiffre suivit d'un nombre (qui peut lui même être un chiffre etc).

Grammaire lexicale

- Les symboles terminaux sont les caractères (tt les unicode)
- Les règles de production forment les mots (tokens), éléments d'entrée (input elements) de la grammaire syntaxique.

Unicode : bcp plus large que l'ASCII.
UTF 8, 16, 32 Utilise la taille la plus juste en fonction des caractère utilisé. S'il a besoin de 32, il utilise 32. Si on utilise UTF16 et qu'il a besoin de 32, il prend 32 mais il ne prend pas + petit.

Une accolade est un mot(token).

Ce que produit la grammaire lexicale

InputElement:
Comment
WhiteSpace
Token

Remarque : Les commentaires et espaces ne passent pas la phase suivante.

Ne prend pas en compte les commentaire et les espaces.

Les tokens du langage : Identifier, Keyword, Separator, Operator, Literal.

Les keyword : 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.

Separator : ( ) { } [ ] ; , ... @ ::
. Opérateur qui permet de faire appel de méthode ou attribut.
@ permet de tagger/annotations.
:: ecrire appel de méthode dans classe fonctionnel.

JAVA

Operator : = > < ! ~ ? : == <= >= != && || ++ -- + - * / & | ^ % << >> >>> += -= *= /= &= |= ^= %= <<= >>= >>>= ->
~ fais le complément à 1.
? opérateur ternaire.

Littéral : c'est la manière de représenter une quantité, une valeur.
IntegerLiteral FloatingPointLiteral BooleanLiteral CharacterLiteral StringLiteral NullLiteral
BooleanLiteral : true or false
NullLiteral : null
StringLiteral : entre 2 double coat.
CharacterLiteral : entre 2 coat.
07 : le 0 c'est pour dire que c'est un octal.

IntegerLiteral : Différence entre 7 et 7L : le nombre de type sur lequel il est codé. Le 7 sur 32bit et L est sur 64. On peut aussi écrire 7l. On peut écrire 1_000, c'est pareil que 1000, le underscore c'est pour la lisibilité. Mais on ne peut pas écrire 07 pour le décimal.
Pour le hexadecimal : on peut écrire 0x ou 0. On peut aussi mettre des underscores.
Binaire : 0b ou 0B.

FloatingPointLiteral : on peut utiliser la notation scientifique, du coup il faut mettre un e ou E. On peut aussi écrire 1. ou 0. on ne dois pas confondre avec le int. A la fin on peut mettre un f ou F ou d ou D. Par défaut un littéral est de type double.

Identifier : comment est-ce que je peut écrire un nom de variable, classe, méthode.
CamelCase comment par une majuscule. mixedCase : commence par une minuscule.
CamelCase : classe mixedCase : méthode, variable.
Un identifier ne commence pas par un chiffre. Ça peut être n'importe quel assemblage de mot à part un keyword, null ou un booleen. Même _ et $.

Grammaire syntaxique

Ne s'occupe de des tokens du langage.

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

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

- Les expressions
  - Représentent les « calculs » (1+1, fait intervenir operande et operateurs)
  - Ont une valeur et un type(1+2, la valeur c'est 3 et le type c'est int)
- Les instructions (for, while, if)
- Les expressions-instructions

S4 - EXPRESSION - INSTRUCTION

Expression

Une ExpressionStatment c'est un StatmentExpression avec un ;
Une expression a toujours une valeur et un type et ne termine pas par un ;.

JAVA

Assignation :

Donner une valeur a une variable.
A gauche un identifier ou un ArrayAccess puis un opérateur et une expression.
L'assignation est une expression, a un type et une valeur.

- a un type (celui de la variable)
- a une valeur (celle du left hand side) On en fait une instruction avec le ;

i=j=k=l=m=0 ; Le tout est une instruction mais si on prend sans le ; c'est une assignation.

Il existe d'autre opérateurs d'assignation = *= /= %= += -=

Ça n'est pas une assignation : (i+1) −= 2; car le membre de gauche DOIT être un identifier.

Pré/post in/décrémentation

--i ➔ pré-désincrémentation
i++ ➔ post-incrémentation
Doivent avoir une valeur, si on met le – devant ou derrière c'est ± égal.
Il faut écrire pas à pas la valeur de i et le résultat qui devra apparaître.
Regarder s'il y a une pré puis évaluer l'expression puis regarder s'il existe une post.

Post :

D'abord je donne ma valeur à l'expression(++) puis j'incrémente.

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

D'abord il affiche 1, puis il affiche 2.

Pré :

D'abord incrémenter et ensuite utiliser sa valeur dans l'expression.

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

Les 2 afficheront 2.

TOUJOURS évaluer les expression de la gauche vers la droite.

```
i = i++ + ++i;
```
6+8=14 i vaut 6 donc le 1er opérande = 6 maintenant i vaut 7, la 2ème opérande vaut 8

Appel de méthode

C'est aussi une expression, ça a donc un type (type de retour) et une valeur (la return).
Grammaire : MethodInvocation

Instanciation

Créé une instance d'une classe c'est aussi une expression type (celui de l'objet créé) valeur (la référence vers l'objet). Grammaire : ClassInstanceCreationExpression

Instructions

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

EmptyStatement : ne fais rien, juste un ; 2 ; c'est 2 instructions.
Block : un bloc d'instruction en accolade c'est une instruction.
Les choix : if et switch

Switch

Utilise les type; byte, char, short, int, Byte, Character, Short, Integer, String ou enum
Enum est par exemple automne, hiver, été, printemps.
Short, Byte, … : si je ne peux pas avoir un type primitif.
Utilisation d'un break pour sortir d'un switch

Les boucles

While, do-while, for

While : expression booléenne, l'expression DOIT être modifier dans Statement.

For : grammaire BasicForStatement EnhancedForStatement

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

ForInit :
- *StatementExpressionList*
- *LocalVariableDeclaration*

ForUpdate :
- *StatementExpressionList*

1 on initialise la boucle : ForInit.

Une déclaration de variables

VariableDeclaratorList :
VariableDeclarator { , *VariableDeclarator*}

VariableDeclarator:
VariableDeclaratorId [= *VariableInitializer*]

Exemple

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

2 évaluation de l'expression

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 Evaluation de ForUpdate. Liste de StatmentExpression

JAVA

Foreach

On déclare une variable qui va recevoir chaque élément de la liste et le 2ème élément DOIT être itérable.

Permet de parcourir un tableau ou un Iterable. Exemple :

```
String [] toys = {"Hamm", "Slink", "Potato", "Woody", "Sarge", "Etch"}
for (String toy : toys) {
    // do something with toy
}
```

```
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.

←Avec un foreach

Ruptures

Étiquette

Toutes 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

On peut donner un nom a une instruction.

Break

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

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

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

On ne le voit que dans les boucle (while, for, …)

Exemples

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

Affichera les impaires

```
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 );
   }
```

N'affichera rien. Car on ne sort jamais de bcli

Récap des instructions

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

S5 - TABLEAU

Il faut déclarer le tableau,	int[] is ;
le créer	is = new int[3] ;
puis l'initialiser.	is[0]=4 ;

Se lit de droite à gauche is est un tableau d'entier (int[]is)
int [][] t est un tableau de tableau d'int. Donc un tableau de 2 dimensions.

Un tableau est un type de données
Les éléments d'un tableau peuvent être des tableaux.

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

On peut créer un tableau en donnant sa taille ou ces valeurs.
Le 1er : is2=new int[]{4,5,6} si j'utilise les {} je ne peux pas écrire le nbre d'élément.
Le 2ème : is = new int[3] ici on écrit le nbre d'élément.

ArrayCreationExpression :
`new` *TypeName Dims ArrayInitializer*

Dims :
`[ ]`
Dims `[ ]`

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

VariableInitializers :
VariableInitializer
VariableInitializers `,` *VariableInitializer*

VariableInitializer :
Expression
ArrayInitializer

Exemples

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

```
Video[] videos;                          Il y a 3 éléments dans le tab videos, dont null.
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};
```

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 fibo.length et aussi fibo[i].length

Création en donnant la taille :

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

DimExprs :
DimExpr
DimExprs DimExpr

DimExpr :
`[` *Expression* `]`

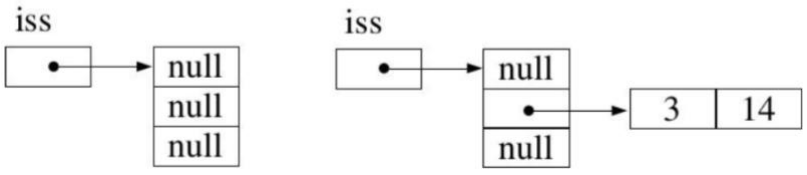
Dims :
`[ ]`
Dims `[ ]`

new int[2][3] tableau 2D de 2 lignes et 3 colonnes
new int[2][] tableau 2D de 2lignes mais on ne sait pas encore les colonnes.

Valeur par défaut (0, false, null)

Exemples

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



Parcours d'un tableau :

Possibilité de modification des éléments du tableau moss[i][j] = new MyObject(...)

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("");
}
```

Avec foreach ➔ for(:)
On ne peut pas parcourir colonne par colonne. Possibilité d'envoyer un message à un objet : mo.foo()

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("");
}
```

Arrays = classe utilitaires. Il n'y a que des méthodes static. Termine par un S.

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

- Copie très superficiel : copy = moss (référencer copy vers moss)
- Copie avec Arrays.copyOf()

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

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

- Copie en profondeur avec Arrays.copyOf

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

Chacune des lignes va être copié.

- 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];
    }
}
```

- 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]);
    }
}
```

S6 - LES COLLECTIONS

Ce sont des objets
Liste : dans une liste il y a un ordre (chaque élément à un n°) mais pas spécialement trié.
Un enssemble de permet pas les doublons

Collections

Représente un groupe d'objets, ses éléments.

- Tous les éléments sont de sont de même type.
- Certaines collections sont triées, d'autres ordonnées
- Certaines collections permettent les doublons

Liste

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

- La taille s'adapte à son contenu (≠ tableau)
- Les éléments ne sont pas nécessairement différents
- Ordonnés, pas nécessairement triés
- Possibilité d'ajouter, de supprimer, d'accéder, de remplacer un élément

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 : on doit spécifier le type E lors de la déclaration/création.
Generics : comme ça on peut l'utiliser pour n'importe quel type de valeurs, s'attend à un type référence.
List n'est pas une class, c'est une interface.

On écrit E mais ça peut être T ou autre.

Il faut stocker les valeurs !

JAVA

Interface

Le code qui définit un contrat s'appelle une interface.
Définir un contrat : se mettre d'accord pour écrire qqpart ce dont on a besoin ou ce qu'est cet objet. Il n'y a pas de code, juste les signatures. Dans API, <E>.

Exemple

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

Une classe peut implémenter une interface.

- L'indiquer via le mot clé implements
- Définir le code de toutes les méthodes

J'écris une classe qui va respecter le contrat de MyInterface.

Exemple

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

Une interface

- Définit un type de données OO
- 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
```

On peut déclarer un objet ou une interface, on ne peut pas instancier une interface car il n'y a pas d'interfaces dedans.
List x = new ArrayList

ArrayList

Derrière ArrayList, il y a une tableau. On doit spécifier que List sera de tel type car generic. <> veut dire qu'il est générique.

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"]
```

get

get(int) permet notamment le parcours
liste.get(i) permet d'avoir cette valeure

foreach

Une liste est Iterable.

```
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

LinkedList

La classe java.util.LinkedList est une classe qui implémente (aussi) List. Ne fonctionne pas avec des tableaux.

Exemple

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

Nombrils est de type List et son instance est de type LinkedList.

Polymorphisme

Le 2ème concept de l'OO.
Une instance et une variable peuvent être de types différents. C'est la « bonne méthode » qui sera exécutée. C'est la réécriture d'une méthode mais fonctionne différement.

- La variable est de type List
- L'instance est de type ArrayList → c'est la méthode de la classe ArrayList qui est exécutée

Bonne méthode : c'est la méthode de l'instance.
A l'exécution, c'est le code de l'instance, du type de l'instance qui va être exécuter.
A la compilation, vérifie que tout ce qu'on a écrit vérifie le type et à l'exécution, il vérifie l'instance.
Exemple avec nombrils, à la compilation, vérifie que tout est bien écrit pour la type List et à l'exécution, il exécute le code de l'instance LinkedList.

Types primitif et listes

Integer i= 6 sous-entends que le 6 est newInteger(6) un type référence est créé➔boxing
Enveloppe/Wrapper ce sont des classes qui englobe une valeur primitive dans un objet

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

Les conversions du type primitif vers son wrapper et vice versa sont automatiques
Boxing : mettre automatiquement un type primitif dans son wrapper.
Unboxing : list.get(0) peut être mis dans un int.

Les wrappers sont aussi des classes utilitaires.
C'est à dire que l'on peut instancier un objet
Il a aussi des méthodes statiques.

```
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 );
```

Pour pouvoir être trier, dans notre objet il doit y avoir qqchose pour trier, exemple : dire que le 5 est + grand que 4 etc…
L'objet doit être comparable. ➔ Comparable API. Il n'y a qu'une méthode qui compare tout seul un élément et un autres. this.compareTo()

S7 - HÉRITAGE

C'est le 3ème concept important de l'orienté objet

Principe

Lorsqu'un objet va hérité des membres d'un parent.
Mot clé : extends
Lorsqu'une classe hérite d'une autre

- Possède les mêmes attributs
  - Peut en ajouter
- Possède les mêmes méthodes
  - Peut en ajouter
  - Peut les réécrire, càd qu'on peut réécrire une même méthode (même attributs, tout pareil) mais qui fais autres choses.
- Les visibilités restent de mise (protected)

Si dans la classe parent les attributs sont privé, seul cette dernière peut y accéder. Donc même avec héritage, l'enfant peut ne pas avoir accès aux attributs.

Super

Attribut

super.x ➔Pour utiliser les membres d'un parent, afin d'être sûr d'utiliser le bon.

Constructeur

super(param, param2, …) ➔ je veux faire appel aux constructeur de mon parent(avec le mm nombre de paramètre)

Méthode

super.methode() ➔pour être sûr d'utiliser la méthode parent.

Utilisation de extends et de super.

```
public class ColoredPoint extends Point {
    private Color color;

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

On peut créer p1 en tant que Point et dire que c'est un point coloré.

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

}
```

Si on déclare un point coloré on ne peut pas instancier un point car un point n'est pas un point coloré.
Car p est un point et non un point coloré
p.display() ; affichera point coloré car on a mis new ColoredPoint.

```
Point p = new ColoredPoint(...); // OK
ColorPoint 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 fonctionnalité supplémentaires.
Si on peut dire que point coloré est un point alors c'est un héritage.
! pas pareil que « a un » !

Object

En Java, il y a un arbre d'héritage. Il y a une classe qui est parent de TOUTES les classe, c'est Object. Toutes les classes héritent 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

Ce equals prend en compte la référence.
Si 2 instances ont les mêmes valeurs mais pas la même référence, ils ne sont pas égaux.
On utilise equals afin qu'il vérifie que leur sémantique (valeur) est égale.

toString

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

equals

Permet de définir quand deux objets sont égaux, par défaut, retourne == cad qu'il prend en compte que la référence, réflexive, symétrique et transitive.
D'abord, si même référence ➔ true
Si on n'est pas de même type ➔ false
Cast : comme avec (int)Math.random() ça coupe
On peut réécrire equals.
Exemple : p1.equals(p2) si p2 null pas grave, on l'a gérer mais si p1 est null ➔ erreur

```
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;
}
```

getClass ou instanceof
rendre la classe ou la méthode final, afin d'empêcher de la réécrire (par l'enfant ou autre).
Si on réécrit equals ont DOIT réécrire hashCode

hashCode

Deux objets « equals » ⇒ même hashcode
hashcodes différents ⇒ objets différents (contraposée)
Objets différents ⇏ hashcodes différents (mais c'est mieux)

Parfois plus rapide de calculer un hashcode que de calculer l'égalité de chacun des attributs.
C'est pas facile de trouver un bon hashcode mais il existe des méthodes qu'il le font.

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

Clone

Méthode mal conçue dont l'usage est polémique, il est préférable d'utiliser un constructeur par copie (reçois un objet et retourne un objet).
Ne pas utiliser clone.
Constructeur par copie :

```
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

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)
```

equals

Ici, elle est static donc ne porte pas sur un objet. Gère le null.

deepEquals

On peut voir l'égalité de manière profonde (si des tableaux qui contiennent des tableau) pour voir que TOUS sont égaux.

requireNonNull

Le 1er remplace le 2ème.

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

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

hash

Objects.hash(param1, toutNosParam, …)

En java on ne peut hériter que d'une seule class et d'implémenter plusieurs interfaces.
Interface et default méthode, se renseigner.

S8 – ÉNUÉRATION, EXCEPTION

Énumération

Une énumération est un ensemble fixe et petit de valeurs sémantiquement liées.
Exemple : saison : Printemps - Été - Automne - Hiver
En Java :

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

Public enum Saison{
PRINTEMPS, ETE, AUTOMNE, HIVER ;
}

ClassDeclaration:
    NormalClassDeclaration
    EnumDeclaration

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

EnumBody = {et tout ce qu'il y a entre} = le corps de la classe.

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 on peut aussi en ajouter
    - A un constructeur par défaut
- Avec des fonctionnalités en plus :
    - Conversion automatique vers une chaine (il y a déjà un toString)
    - Peut apparaitre dans un switch
    - Fournit un tableau des valeurs de l'énumération

Type de var dans switch : caractère, entier, string et enum.
On peut mettre un main dans public enum Saison{}
Accéder à une saison particulière : Saison.PRINTEMPS
On peut le mettre dans un tableau : Saison[] nom = Saison.values() ;
Utiliser for : avec un tableau.

Ajout d'attribut, dans la classe il suffit de mettre le private type nom ;
On est obligé de mettre le constructeur en privé car on ne peut pas changer élément liste.
Le constructeur est « appelé » a chaque éléments de la liste.

```
public enum Saison{
        PRINTEMPS(21,3), …
          private LocalDate dateDebut ;

          private Saison(int jour, int mois){          //constructeur
          LocalDate now = LocalDate.now() ;
          This.dateDebut = LocalDate.of(now.getYear(), mois, jour)
          }
}
```

Avant, il n'y avait pas enum.
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;
```

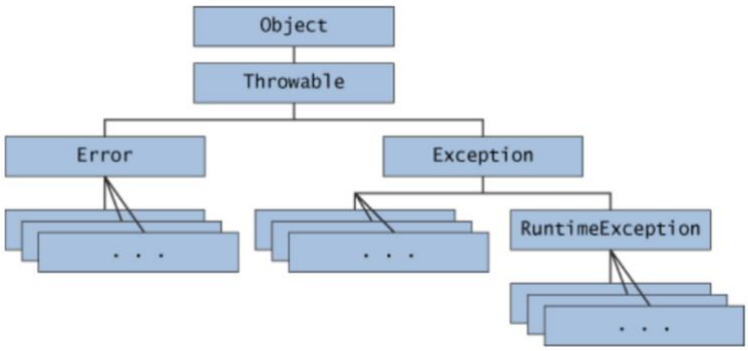
Exception

Rappels

Lancer exception
throw new Exception(« raison »)
S'il y a l'exception, le code après ne sera pas exécuté.

Attraper une exception
try{
code pouvant lancer un exception
}catch(Exception e){gestion de l'exception}

On peut créer des exceptions.



NullPointerException, IllegalArgumentException, … hérite de RuntimeException
Exception contrôlé, , Throwable, tous les objets « jetables », c'est ceux dont on sait faire un « throw ».

- 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, pas obligé de traiter, se fait souvent à l'exécution.

Exceptions contrôlées (par le compilateur) : le compilateur vérifie qu'on traite l'exception, on est obligé de la traiter. (c'est le rectangle à gauche de RuntimeException), il faut préciser qu'une telle exception est lancée, utilisation de throws. Doit être gérée (try catch) ou être relancée (throws)

throws pour dire qu'elle est susceptible de lancer cette exception.

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

Une exception contrôlée doit : être gérée (try catch) ou être relancée (throws)

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

Créer ces propres exceptions

MyException est une sous-classe de Exception, utilisation du mot clé extends, le constructeur fait appel au constructeur parent via le mot clé super.

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

On peut faire « extends Exception » ou « extends RuntimeException », …

(re)catch

Une exception est un objet.

- méthode getMessage() e.getMessage() ;
- méthode printStackTrace () Pour avoir le même message que si on ne met rien(cad toutes les lignes).

Un try peut avoir plusieurs catchs.

- Plusieurs catchs

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

- Catch multiple

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

- Try with resources

Certains objets Java sont closeable (qu'on peut fermer, fichier, flux, …)

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

BuffferedReader : permet de lire un flux (objet)
Lorsque l'on a écrit le = new BufferedReader(new FileReader(path)) on a ouvert le flux.

On ne l'a pas fermé car ici, c'est fait pour qu'a la fin il tente de le fermer direct
Avec tout ça, on a tenté d'ouvrir le fichier, le catch, attrape si le fichier c'est ouvert mais n'a pas su lire. C'est pour ça qu'on ne doit pas le fermer car pas ouvert. Mais entre dans le catch aussi si lorsqu'il est ouvert, il n'arrive pas à le fermer.
Si au début il n'a pas su ouvrir le fichier, il ne fais rien d'autre.

S9 - LE CODAGE DES FICHIERS, TROUVER SON CHEMIN, ENTREES-SORTIES

Pouvoir faire communiquer notre programme avec des fichiers.

Le codage des fichiers

Coder l'information de manière binaire ou textuelle

- Binaire - représentation mémoire
- Texte - utilisation d'une suite de caractères (ouvrir avec éditeur de texte, il n'y a pas de mise en forme, écrire en gras, mettre un titre, …)

Binaire

hexdump (permet de voir les byte): à droite il met le contenu du fichier en hexa et à gauche les numéros de ligne (n° des byte)

```
0000000 0010
0000001
```

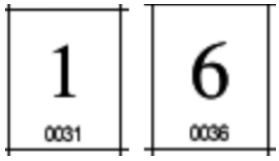
- le fichier fait 1 byte
- la valeur stockée est 16 = 0x10

Textuel

```
0000000 3136
0000002
```

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

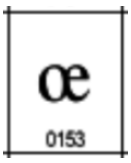
Table Unicode à droite (ASCII fait partit d'Unicode)



JAVA

UTF8

oe n'est pas dans la table ASCII
Le fichier est codé en UTF8




```
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 ?

UTF32 prend + de place, pas intéressant pour les anglais car ils n'utilise pas bcp d'accents. Chaque caractère sur 32bits.

Dans le 1er byte que je vais lire, il y a une information pour dire combien de byte par caractère. Si c'est 4bytes alors ➔ 1111

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	11101xxx 10xxxxxx 10xxxxxx	E4 à E7	
U+8000 à U+BFFF	1110x0xx 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	11110x1x 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)	

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

Binaire = 1 5 3 chacun en 4bits.

Trouver son chemin

Relatif : à partir d'où je suis.
Absolue : à partir de la racine.

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

Le séparateur (delimiter) est différent en fonction du filesystem
Certains systèmes de fichiers autorisent la notion de lien symbolique (symbolic link)

Path(s)

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, …

C'est une interface donc on ne peut pas l'instancier.

La classe Paths est une classe utilitaire permettant de créer un path (une Factory).
Factory : sert à construre des choses, fournir des objets.

Exemple

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

path1 : créé le fichier foo.
path2 : getProperty permet d'utiliser une méthode qui créé un path. user.home est une variable d'environnement qui donne le répertoire home de l'user.
Les fichier du répertoire tmp sont toujours supprimer au reboot.
Variable d'environnement : variable que l'OS connaît.

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);
    }
}
```

Path fournit des méthodes pour interroger un chemin
Soit la déclaration (dans un contexte linux)

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

toString −→ /home/alice/foo
getFileName −→ foo
getParent −→ /home/alice
getRoot −→ /

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
```

Files

La classe Files est une classe utilitaire du package NIO.2 (new I O). Différentes méthodes : createTempFile(String, String)
Files est aussi une Factory.

Entrées-sorties bas niveau

Types de fichiers

En Java, les fichiers sont des flux (stream) : uniformes, non structurés, binaire ou texte.
Un fichier se lit dans l'ordre byte par byte, parfois faut relire plusieurs fois 1 fichier.

Vue d'ensemble

Java fournit des classes de bas niveau pour lire / écrire des byte / char.
Dans les flux binaires on parle de (File)InputStream et de (File)OutputStream. On lit des bytes.
En texte, on parle de FileReader et de FileWriter. Là on lit des caractères.
En java, on est en UTF16.

Lecture binaire

Créé un fichier FILE et ouvre le fichier en lecture seule, donc le fichier doit exister.
IOException = exception spécial.
in.read() va lire des bytes.
Lorsqu'il est en fin de fichier, il retourne -1. Tant que ce n'est pas le dernier byte.

```
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());
}
```

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

Assignation = StatmentExpression

ON NE VA JAMAIS VOIR UN CLOSE DANS TRY, JAVA LE FAIS POUR NOUS, LE FLUX SERA FERME.

lire avec ce code un fichier contenant 16

interpréter le résultat obtenu

49 54 10

49 c'est le 1, 54 c'est le 6 et le 10 c'est le saut à la ligne.

Écriture binaire

int InputStream.read() et OutputStream.write(int) traitent des bytes alors que leur paramètre est un int. Pourquoi ? car si je veux définir ma fin de fichier par -1.

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

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 »

```
Public class TexteWrite{
Public static void main(String[] args){
Try(Writer out = Files.newBufferedWriter(Paths.get("file3"),
StandardOpenOption.CREATE)){
Out.write("Hello world\n");
}catch(IOException e){Syste.err.println(e.getMessage());}
}

Public class TextRead{
Try(Reader in = Files.newBufferedReader(Paths.get(«file3 »))){
Int c;
While((c=in.read())!= -1){
Sout((char)c+ ' ')
}
}catch(IOException e){
e.printStackTrace();
}
}
```

S10 – ENTRÉES SORTIES (HAUT NIVEAU) LES FICHIERS (HAUT NIVEAU)

Flux englobants

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

Flux englobant, aura en paramètre un flux de base et va l'utiliser pour faire un boulot de + hauts niveaux.

Données primitives

Pour écrire des données primitives dans un fichier binaire on se base sur la classe DataOutputStream. (OutputStream ➔ bytes par bytes)
Pour le construire j'ai besoin du flux de base

```
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());
}
```

Ça offre des méthodes supplémentaire. Comme writeBoolean, …

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

```
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());
}
```

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

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 (Exemple : Strin s ; s.matches([oOnN]{1}) retourne boolean)

Exemple : une plaque d'immatriculation : 1(le chiffre) 3lettres 3chiffres

Intervalle : [1-5]
Exactement 2 fois un chiffre : \d{2}

Au moins une fois : +
Tout ce qui est chiffre : \d
0 ou 1 : *
O ou N minuscule ou majuscule mais que 1 caractère : [oOnN]{1}
Intervalle non-fini, de 5 à infini : {5 -}
Vérifier un pattern coute chère, bcp de test.

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());
}
```

Au lieu de system.in on peut mettre d'autre classe comme Paths.get()

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

Out ➔ attribut.
Out est un print writer qui est connecté à la sortie standard.
System ➔ une classe
Printf c'est un print formaté, il ne va pas à la ligne, %s = chaine de caractère
printf(« Hello %s\n », pseudo) pseudo c'est la chaine de caractère.
%d = entier
%f = nbre à virgule flottante.

```
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());
}
```

%04d ➔ pour écrire les chiffres avec 4 caractères et s'il fais moins de 4, il met des 0.
%4.2f un flottant de 4 chiffrés avant la virgule et 2 après la virgule.
Writer permet d'écrire caractère par caractère.

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

Console

Existe mais on ne peut pas y avoir accès de n'importe quelle terminale.
Permet de rassembler Scanner.in et System.out.

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

```
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, ' ');
```

String name = console.readLine(« Enter name : ») ➔ fait l'affichage et enregistre direct la valeur dans la variable.
console.format(« Your name is %s », name) ➔ le \dans l'image est faux.
char[] password = console.readPassword(« Password : ») ; ➔ on ne voit pas ce qu'on écrit, comme sur linux, de plus il stocke dans tableau, intérêt ➔ après on peut faire Arrays.fill(password,' ') ➔ pour effacer en RAM le mdp au cas ou un autre processus voudrait y accéder.

Sérialisation

Lorsque l'on doit insérer un objet dans un flux.
Transformer un objet en une séquence de bytes et de la reconstruire
Sérialisable c'est une classe.
Pour pouvoir sauvegarder l'état d'un objet à un moment.

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
- Une classe doit être la même à l'écriture et à la lecture (serialVersionUID)

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

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

S11 – UN PEU DE FONCTIONNEL

Expression lambda

= raccourcis d'écriture.

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
```

JAVA

Étude de cas 1 : Trier

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);
```

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

On doit démontrer quand une vidéo est + grande qu'une autre. Si ce comparable retourne un entier positif ➔ + grand sinon + petit. Ici, une vidéo est + grande si auteur, niveau de l'alphabet est le + premier. Du coup on peut le trier. ToIgnoreCase : ne fait pas attention aux majuscules et minuscules. v1.compareTo(v2)

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

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

Si je fais Collections.sort(ListeVideos) ➔ il va les trier suivant l'ordre de compareTo d'avant.
Du coup il faut fournir un comparator, n'a qu'une méthode : compare(T o1, T o2)

Afin de comparer des vidéos en fonction du nombre de likes :

```
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

Il est possible de créer le comparator directement dans le trie.
Lourd si la classe n'est utilisée qu'une seule fois. La solution ?
Une classe anonyme (à usage unique)

```
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

Permet de remplacer une class interne anonyme si cette classe n'a qu'une seule méthode.

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

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

Étude de cas 2 : For each/ itérer une collection

On déclare une variable qui va recevoir chaque élément de la liste et le 2ème élément DOIT être itérable.

Supposons qu'on veuille liker toutes les vidéos d'une liste.

On peut utiliser un for each

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

Nouveauté Java 8 : une méthode forEach

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

Qu'on peut raccourcir en methode 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 );
}
```

Permet de diviser le travail.

Étude de cas 3 : Filtrer une collection

Supposons qu'on veuille ne garder d'une liste de vidéos que les plus likées.
Je veux garder les vidéos qui on minimum le nombre de like « limite ».
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 ;
}
```

En Java 8 : on peut passer par Stream

Stream lorsque l'on va vouloir effectuer une série d'opération sur une collection. Le compilateur va analyser le travail a faire puis fais ce qu'il a a faire.

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

Collect permet de convertir le stream en liste.

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

Stream

Le stream Java est très puissant
Étape 1 : Création avec un .stream
Étape 2 : Opérations intermédiaires : Manipulations

- transforment le stream
- peuvent être chainées

Étape 3 : Opération finale : Réduction

- produit autre chose qu'un stream Expl : à la fin transformer le stream en liste
- une seule permise

On doit toujours faire les 3 étapes.
Map filter reduce
Map : apd d'un flux, je veux abtenir un autre stream mais de même type. Exemple : apd d'un flux de vidéos, je veux récupérer qu'un flux d'auteur.

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);
```

Un stream est infini, ce flux est ouvert à un moment et temps que je ne fais pas de reduce sur mon flux, ça reste ouvert.

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 : appliquer une méthode sur chaque éléments, là je créé un stream de stream.

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

Predicate, un type j'associe à un booleén
Une fonction a un type j'associe un autre type.

Étape 3 - Opérations terminales

Opération terminale :

- forEach : déjà vu (équivalent du for each)
- collect : déjà vu (convertit le stream en liste)
- any/all/noneMatch : teste le stream

```
boolean nonLiké = videos.stream().anyMatch( x->x.getNbLikes()<100 );
```

- findFirst /Any : cherche un élément

```
Video nonLiké = videos.stream().findAny();
```

- count : compte le nombre d'éléments

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

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

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

- max/min : chercher le maximum/minimum

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

mapToInt retourne un entier

Stream

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

Le temps

"10/05/2012" peut être JJ/MM/AAAA ou MM/JJ/AAAA, Fuseau horaire, Heure d'hiver, quelle année somme nous.

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

Un temps s'écrit sous la norme ISO8601 ➔ yyyy-mm-ddThh :mm :ss[TZ]
TZ = time zone, zone géographique.

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
```

- 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 on ne peut pas modifier) : 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);
```

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);
```

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);
```

Instant

Représente un temps « machine » (nbre qui représente un temps qui est unique, en nbre de seconde.)

- 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");
```

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 ();
```

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());
```

JAVA

Mise en page

De nombreuses possibilités de mettre en page un temps
DateTimeFormatter : comment je veux écrire la date.
Pour écrire la date de différente manière, format.

```
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)));
```

Calendriers

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

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

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.
StringJoiner

StringJoiner sj = new StringJoinder(« . « , » {», » }»)
Tant qu'il y a qqchose afficher entre les guillemet

Lire TOUTE ITERATION1 MINIMUM PROJET JAVA

Classe abstraite = objet qu'on ne peut pas implémenter, polymorphisme : exemple j'ai la classe Animal avec une méthode avancer() , puis j'ai une classe lievre qui hérite d'animal, il a aussi avancer mais il avance plus vite donc avancée(){avancée+=5}. Animal est abstrait car je ne peux pas faire = new Animal() mais je peux faire new Lievre().