PREMIERS PAS EN JAVA

CONTEXTE GÉNÉRAL

UN PEU D'HISTOIRE

Java a été développé dans les années 1990 par des ingénieurs de Sun Microsystems insatisfaits du langage C++, pour les systèmes embarqués :

- Gestion de la mémoire souvent source d'erreur (pointeurs)
- Nécessité de nettoyer soi-même la mémoire en désallouant explicitement les objets : pas de ramasse-miettes (garbage collector en anglais)
- Nom original : oak (chêne)

- 1995 : présentation officielle sous le nom de Java
- Exécution de code dans des pages web au moyen “d’applets”
- Nécessité d’avoir du code portable, compatible avec n’importe quel OS
- Utilisation de la JVM

- Depuis 2000 : sortie d’une version majeure de Java tous les deux ans
- 2009 : rachat de Sun Microsystems par Oracle
- Employé largement dans le développement des premières App. Android
- Langage le plus “populaire” selon l’indice Tiobe…

PRINCIPES DE JAVA

Compilation du code une fois pour toute !

- Fichiers sources : " *.java "
- Après compilation : " *.class " :
  - bytecode **indépendant** du système d'exploitation
  - Destiné à être exécuté par la **JVM** (Java Virtual Machine)
- Une fonction **principale** pour l'exécution : **main**

JAVA : LES PROMESSES

- Déploiement **facile** des applications :
  - Le bytecode généré fonctionne sur tout OS,
  - Une seule compilation / archivage avant la distribution
- Pas de gestion compliquée de la mémoire
- Paradigme objet facile à appréhender (pas d’héritage multiple)

OBJECTIFS DE CETTE SESSION :

- Les impératifs dans un fichier Java
- Introduction aux différents **types** en Java
- Introduction aux opérations de base
- Introduction aux instructions de **flow control**
- Introduction à la fonction main

CRÉER UN FICHIER JAVA

RETOUR SUR LE “HELLO WORLD”

Fichier **Main.java**:

```java
import library.Message;

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

Fichier **Main.java**:

```java
import library.Message;

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

- Correspondance nom de classe / nom de fichier
- Premières lignes : les importations de bibliothèques
- `public class Main`: on crée une classe appelée `Main``

DÉFINITION D'UNE MÉTHODE

```
public static void main(String[] args) {
    System.out.println(Message.content);
}
```

- `public static` : indique certaines propriétés de la méthode
- `void` : type de retour (obligatoire)
- `main` : nom de la fonction
- `String[] args` : type et nom des arguments

```
TYPE_DE_RETOUR nom(TYPE1 arg1, TYPE2, arg2,...){
  Instruction1 ;
  Instruction2 ;
  ...
}
```

- Corps de la fonction entre les accolades
- Une fonction ne peut retourner qu'un seul objet
- ou rien (type `void`)

```
System.out.println(Message.content)
```

- `System.out.println` : instruction pour afficher quelque chose dans le terminal
- `Message.content` : argument de la fonction (ici une chaîne de caractères)
- `System.out.println` peut prendre n'importe quoi comme argument pour l'afficher dans la console !

QUELQUES RÈGLES SUR LES INSTRUCTIONS :

```
Instruction ; // Ceci est un commentaire
/*
Ceci est un bloc de commentaires
Qui peut s'étendre sur plusieurs lignes
*/
Instruction_suivante ;
```

- Se termine toujours par un point-virgule
- Les commentaires sont avec les symboles `//` (commentaire simple) ou `/* ...*/` (bloc)

LES TYPES JAVA

TYPES DE BASE

Java est un langage fortement typé. Presque tous les types sont objets.

À l’exception de 8 types de base :

- **byte, short, int, long**
- **float, double**
- **boolean**
- **char**

LES DIFFÉRENTS ENTIERS

- **byte** : entier signé sur 8 bits $[-128, 127]$
- **short** : entier signé sur 16 bits $[-32768, 32767]$
- **int** : entier signé sur 32 bits $[-2^{31}, 2^{31}-1]$
- **long** : entier signé sur 64 bits $[-2^{63}, 2^{63}-1]$

LES ENTIERS LITTÉRAUX

- Par défaut des **int** (ex. : 1, 2, 3,…)
- **byte**, **short**, **int** et **long** peuvent être initialisés avec un entier littéral
- Les **long** avec des valeurs supérieures à 2^{31} peuvent être initialisés avec des littéraux finissant par “L” (ex. : 1000000000L)
- Possibilité d’ajouter des underscores "_" pour la lisibilité (ex. : 1_100)

LES DIFFÉRENTES BASES D'ENTIERS

```
// Le nombre 26 en décimal
int decVal = 26;
//  Le nombre 26, en hexadécimal
int hexVal = 0x1a;
// Le nombre 26, en binaire
int binVal = 0b11010;
```

LES FLOTTANTS

- **float** : Nombres flottants simple précision codés sur 32 bits
- **double** : Nombres flottants double précision codés sur 64 bits.

Exemples :

```
double a = 1 ;
double b = 1.0 ;
double c = 1.3e3 ;
```

ATTENTION AUX EXPRESSIONS LITTÉRALES

À tester dans BlueJ…

```
double a = 3 ;
double b = 2 ;
double c = a/b ;

double d = 3/2 ; // Que se passe-t-il ?
```

LES CARACTÈRES

- **char** : caractère unicode codé sur 16 bits
- Va de *u0000* à *uFFFF*

Exemple :

```
  char a = 0 ;
  char capitalC = 'C' ;
  (int)capitalC  // Conversion de capitalC en entier (code
unicode)
```

LES OBJETS

- Tous les autres types sont des objets
- ex. : Chaînes de caractères, tableaux, listes,…
- Un objet : des attributs et des méthodes : on y accède à l’aide de l’opérateur “ . ”
- On crée une nouvelle instance d’un objet grâce au mot clé new

LE PENDANT OBJET DES ENTIERS ET FLOTTANTS

Il s'agit des classes **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**

```
Integer a ;
a = new Integer(3) ;
```

- On déclare une variable `a` de type `Integer`
- On crée une nouvelle instance grâce à `new` puis le constructeur.
- Ex. : doc Java Integer

LES TABLEAUX

Les tableaux (**array**) permettent de stocker un nombre **connu** d'objets en mémoire. Ils peuvent être de n'importe quel type.

Déclaration :

```
TYPE[] tableau ; // TYPE peut être n'importe quoi (int, double ou une classe)
```

Initialisation :

```
tableau = new TYPE[N] ; // N est un entier
```

Accès :

```
tableau[i] ; // i < N
```

Une autre façon d’initialiser un tableau lorsqu’on connaît les éléments :

```
int[] tableauInt= {3,4,5,6} ;
```

Les tableaux peuvent être modifiés directement :

```
tableauInt[0] = 0 ;
```

Comment connaître la taille d’un tableau ?

- Rappel : le type d’un tableau d’entier est int[] : la taille n’est pas mentionnée.
- **length** : attribut qui stocke la longueur du tableau. Accessible de façon publique. Il s’agit d’une valeur immuable pour un tableau.

```
System.out.println(tableauInt.length) ; // afficher la
taille dans le terminal
```

Il est possible de créer des tableaux sur plusieurs dimensions. Il s'agit alors d'un tableau de tableau.

```
int[][] tableau2dim = {{1 , 2 , 3},{4 , 5 , 6}} ;
```

Accès aux dimensions :

- Première dimension **tableau2dim.length**?
- Deuxième dimension **tableau2dim[1][0]** ?

LA CLASSE STRING

La classe **String** permet de manipuler les chaînes de caractère.

```
String chaine = "Hello World"; // Déclaration et
initialisation
```

Méthodes utiles…

- *int* **length**() : donne la longueur de la chaîne
- *char* **charAt**(*int* i) : retourne la caractère à l'emplacement i
- *String* **concat**(*String* str) : retourne une **nouvelle** chaîne correspondant à la concaténation de l'instance en cours et de l'argument str.
- Opérateur **+** : permet de concaténer deux chaînes

Les chaînes de caractères Java sont immuables :

- Il n’existe aucun moyen de modifier le contenu d’une chaîne.
- L’opération **chaine = "Hello World bis"** revient à placer une **nouvelle** chaîne de caractères dans la variable chaine.
- Autre exemple : la méthode **concat** ne modifie pas la chaîne initiale.

Exemples

```
String s1 = " Hello " ;
char c = s1.charAt(0) // C vaut 'H'

String s2 = s1.concat("World"); // Hello World

String s3 = s1 + " World " ;  // Hello World

String s4 =  " World ".replace("Wor","Bo") ; // Bold
```

LES INSTRUCTIONS DE BASE EN JAVA

DÉCLARATION / AFFECTATION

- Déclaration : fournir le nom du type et de la variable

```
String s ;
```

- Affectation : avec le signe =

```
s = "Hello" ;
s = new String("Hello"); // Création de l'objet String
avec new
```

- Combinaison déclaration/affectation :

```
String s = "Hello";
```

OPÉRATIONS MATHÉMATIQUES

- Opérations mathématiques de base **+ , - , * , /** (avec les priorités mathématiques habituelles)
- Opérations d'incrémentation/décrémentation **++/--**

```
i ++ ; // Comparer avec ++ i ;
```

- Reste de la division entière **%**

```
int i = 11 % 3 ; // i vaut 2
```

TESTS ET LOGIQUE BOOLÉENNE

- Test d'égalité == ou de non égalité !=

```
if (i==3) ...
```

- Tests comparatifs <=, <, >= et >

```
if (i>=3) ...
```

- Opération de négation !

```
Boolean a = !true ; // a = false donc...
```

- Le ET et OU logique : && et ||

```
Boolean b = true && false ; // false !
```

OPÉRATEUR TERNAIRE ?

Si la condition vaut **true**, alors on retourne val1, sinon on retourne val2.

```
condition ? val1 : val2 ;
```

Exemple

```
int note = 15 ;
char grade = (note >= 16) ? 'A':'B';
```

LE CONTRÔLE DE FLUX EN JAVA

Il s’agit ici de définir l’ordre d’exécution des instructions.

Par défaut, au sein d’une méthode, les instructions sont exécutées **les une après les autres**.

LE MOT-CLÉ RETURN

Le mot-clé **return** permet d'interrompre définitivement l'exécution d'une méthode et de retourner la valeur précisée après le mot clé.

```
int renvoie1(){
  return 1;
}
```

Ce mot clé est obligatoire pour les méthodes qui retournent autre chose que *void*

LES INSTRUCTIONS IF/ELSE

```
instruct1 ;
if (x == 4)
{
  instruct2 ;
}else
{
  instruct3 ;
}
instruct4 ;
```

- Si $x = 4$, on aura instruct1 → instruct2→ instruct4
- Si $x \neq 4$, on aura instruct1 → instruct3 → instruct4
- On peut également avoir **if** sans **else**

LES BOUCLES FOR

```
for (initialisation ; conditionFin ; increment){
  instructions;
}
instructionsSuivante ;
```

- Réaliser des opérations un nombre défini de fois
- Parcourir un tableau / une liste
- Une fois la condition de fin réalisée, **instructionsSuivante** est exécutée

```
for (int i=0 ; i < = 10 ; i++){
  System.out.println("On affiche le nombre "+i);
}
System.out.println("On a compté jusqu'à 10")


for (int i=10 ; i > = 10 ; i--){
  System.out.println("On affiche le nombre "+i);
}
System.out.println("Fin du compte à rebours !")


for ( ; ;){
  // Boucle infinie
}
```

LES BOUCLES WHILE

```
while (expressionTest) {
  instructions;
}
instructionsSuivantes
```

- Réaliser des opérations tant qu'une condition est réalisée
- Si **expressionTest** vaut **false** lors de sa première évaluation, on passe à **instructionsSuivantes**
- **while(true)** → boucle infinie
- Une fois que **expressionTest** est faux, **instructionsSuivantes** est exécutée

LES BOUCLES DO WHILE

```
do {
  instructions;
}while (expressionTest);
instructionsSuivantes ;
```

- Similaire à **while**
- Mais garantie que le bloc **instructions** est exécuté **au moins une fois**.
- Une fois que **expressionTest** est faux, **instructionsSuivante** est exécutée

LE MOT CLÉ BREAK

Permet de sortir d'un bloc d'instruction **for**, **while** ou **do while** prématurément et d'exécuter les instructions suivantes.

```
String chaine = "Hello World" ;
// Recherche de la présence du caractère 'W'

boolean wPresent = false ;
for (int i = 0 ; i< chaine.length() ; i++){
  if (chaine.charAt(i) == 'W'){
    wPresent = true ;
    break ; // Il n'est plus utile de continuer le for
  }
}
```

LE MOT CLÉ CONTINUE

Permet de “**sauter**” l’itération courante d’un bloc d’instruction **for**, **while** ou **do while**.

```
String chaine = "Hello world" ;
// Comptage du nombre de 'l'

int nb = 0 ;
for (int i = 0 ; i< chaine.length() ; i++){
    if (chaine.charAt(i)!='l')
      continue ; // On passe à i+1

    // On traite le caractère
    nb++;
}
```

L’INSTRUCTION SWITCH

Permet de placer le contrôle de flux à un endroit spécifique en fonction de la valeur d’une variable parmi un ensemble donné :

```
switch(variable){
  case valeur1 : instr1 ; instr2 ; //...
  case valeur2 : instr3 ; instr4 ; //...
  case valeur3 : instr5 ; instr6 ; //...
}
```

Dès qu’une des conditions est vérifiée, le code exécute **toutes les instructions suivantes**

Si variable = valeur2 alors instr3 → instr4 → instr5 → instr6

Penser à l’instruction **break** et à l’instruction **default**

switch(variable){ case valeur1 : instr1 ; instr2 ; break //… case valeur2 : instr3 ; instr4 ; break //… case valeur3 : instr5 ; instr6 ; break //… … default: instrDefault; }

LES EXCEPTIONS

BUT

- Lors de la compilation, le compilateur vérifie surtout le respects de **contrats** (signature des fonctions).
  - ex. : on passe les bons types en arguments
- Il ne vérifie pas la cohérence des valeurs
- Le but des exceptions est de “signaler” les problèmes
- Représente un cas particulier de contrôle de flux.

EXEMPLE

```
public class Main {
  public static void main(String[] args) {
      System.out.println(1/0);
  }
}
```

On n’a pas l’affichage voulu mais :

```
Exception in thread "main" java.lang.ArithmeticException: /
by zero at Main.main(Main.java:3)
```

QUE SE PASSE-T-IL ?

- Je réalise une division par zéro avec des entiers
- Il s’agit d’une erreur arithmétique détectée par la fonction qui encode la division des entiers
- Cette fonction **lève une exception** et la transmet à la méthode appelante (ici **main**)
- **main** reçoit cette exception. Par défaut, l’exécution de **main** s’arrête et le texte descriptif de l’exception s’affiche dans la console

PRINCIPE EXCEPTIONS ?

- C’est une mesure de protection pour éviter des comportements non prévus
- Lorsqu’une exception est levée, la méthode appelante la reçoit. Si elle n’est pas traitée, l’exécution de la méthode est interrompue et l’exception remontre à la méthode appelante
- Si l’exception n’est toujours pas traitée par la méthode principale (**main**), alors le programme s’arrête et le contenu de la pile d’exécution est affichée (on voit toutes les fonctions appelées)

COMMENT TRAITER LES EXCEPTIONS ?

Il faut essayer !

```
public class Main {
  public static void main(String[] args) {
      try{
        System.out.println(1/0);
      }catch(Exception e){
        System.out.println("Ce n'est pas très bien !");
      }
  }
}
```

```
catch(Exception e)
```

- **J’attrappe** l’exception **e**
- **Exception** est une classe Java. **e** est donc un objet avec des méthodes et attributs

Attraper une exception spécifique

```
public class Main {
  public static void main(String[] args) {
      try{
        System.out.println(1/0);
      }catch(ArithmeticException e){
        System.out.println("Ce n'est pas très bien !");
      }
  }
}
```

On ne traite que les exceptions de type

ArithmeticException

Traiter plusieurs exceptions

```
public class Main {
  public static void main(String[] args) {
      try{
        System.out.println(1/0);
      }catch(ArithmeticException e){
        System.out.println("C'est une exception arithm!");
      }catch(Exception e){
        System.out.println("C'est une autre exception !");
      }
  }
}
```

DEUX MÉTHODES INTÉRESSANTES SUR LES EXCEPTIONS

- *String* getMessage() : permet de récupérer le message
- *void* printStackTrace() : permet d’afficher la pile d’exécution

LE MOT-CLÉ “THROWS”

- Par défaut, les exceptions sont vérifiées par le compilateur
- Si votre méthode est susceptible de ne pas traiter un type d’exception et de le renvoyer, il faut en principe le déclarer dans la signature.

```
public static void main(String[] args) throws
ArithmeticException {
  System.out.println(1/0);
}
```

- Certaines exception étant très communes, les déclarer via **throws** est superflu. C'est le cas de *ArithmeticException*
- Pour d'autres exceptions (ex. *IOException*), il faut les déclarer via **throws** sous peine d'erreur de compilation.
- Liste des exceptions par défaut

LEVER UNE EXCEPTION

Utiliser le mot clé **throw**

```
public void printAge(int i){
  if (i<0){
    throw(new IllegalArgumentException("Age >0 !"))
  }else{
    System.out.println("Vous avez "+i+" ans !");
  }
}
```

CRÉER UNE EXCEPTION PERSONNALISÉE

- Exemple : type **SaisieErroneeException**
- Dans le fichier **SaisieErroneeException.java**

```
public class SaisieErroneeException extends
Exception {

  public SaisieErroneeException() {
    super();
  }

  public SaisieErroneeException(String s) {
    super(s);
  }
}
```

```
public void printAge(int i) throws SaisieErroneeException{
  if (i<0){
    throw(new SaisieErroneeException("Age >0 !"))
  }else{
    System.out.println("Vous avez "+i+" ans !");
  }
}
```

LA STRUCTURE D'UNE APPLICATION JAVA

LA FONCTION MAIN

Il est possible “d’exécuter” une classe si et seulement si celle-ci contient une fonction **main** dont la signature est la suivante :

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

- **public** : la méthode est publique (peut être appelée depuis une autre classe)
- **static** : la méthode est statique. C’est une méthode de classe qui ne nécessite pas d’instance de l’objet.
- **void** : la méthode ne retourne rien.
- **String**[] **args** : l’argument de main est un tableau de String.

POURQUOI STRING[] ARGS?

- Identification des arguments lors d'une commande textuelle dans un terminal

```
$ ls -l *.java      (unix)
$ dir *.java        (windows)
```

Liste tous les fichiers avec l'extension java et les présente sous forme de liste

RÉCUPÉRER LES ARGUMENTS

```
$ ls -l *.java
```

- Exemple avec la ligne de commande unix
  - **ls** : nom de la commande
  - **-l** : premier argument
  - ***.java** : deuxième argument
- Du point de vue de Main :
  - args[0] = "-l"
  - args[1] = "*.java"

RÈGLE D'USAGE POUR MAIN

En général, il est préférable de définir une classe particulière qui contient le Main. Il est rarement approprié de définir cette fonction dans une classe normale.

POUR EXÉCUTER

Se mettre dans le répertoire contenant le fichier .class (ex. MainClass.class) issu de la compilation à l'aide de la commande **cd**.

```
$ java MainClass argument1 argument2....
```