Introduction à la Programmation : Types "Avancés"

Laboratoire d'Intelligence Artificielle
Faculté I&C

J. Sam

Types de données : petit bilan

À ce stade du cours, la représentation des **données** se réduit aux types de base `int`, `double`, (`char`) et `boolean`.

Ils permettent de représenter, dans des **variables**, des concepts simples du monde modélisé dans le programme :
dimensions, sommes, tailles, expressions logiques, ...

Cependant, de nombreuses **données plus sophistiquées** ne se réduisent pas à un objet informatique élémentaire.

☞ un langage de programmation évolué doit donc fournir le moyen de **composer les types de base** pour construire des types plus complexes.

Support MOOC

Vidéo et Quiz de :
https://www.coursera.org/learn/initiation-programmation-java/home/week/4

https://www.coursera.org/learn/initiation-programmation-java/home/week/5

- ☞ Semaine 4 et 5
- ☞ Notes de cours et BOOC, semaine 4
- ☞ Notes de cours et BOOC, semaine 5

Pendant l'heure de cours

- Petit rappel des points importants
- Fiches résumé : chaînes de caractères, tableaux de taille fixe et tableaux dynamiques
- Approfondissements :
  - Affectation et comparaison de tableaux (35)
  - Comparaison de chaînes de caractères (51 à 53)
  - `ArrayList` : « autoboxing » et « unboxing » (70-71)
  - Le type `char`(5 à 11) (voir aussi l'annexe)

Le type `char`

Un `char` représente un caractère

- Le caractère s'écrit entre apostrophes
- un `char` contient exactement 1 caractère

```
char c1 = 'm';
char c2 = 'M';
char c3 = ' '; // espace
char c5 = '2';
```

Le type `char` (2)

Un **caractère précédé par un backslash** (\) a une signification spéciale :

- Caractère spécial non-imprimable :

```
char c5 = '\n';  // Saut de ligne
char c6 = '\t';  // tabulateur
```

- Caractère qui risque d'être mal interprété :

```
char c7 = '\'';  //apostrophe
char c8 = '\\'; //backslash
```

- Sinon, le backslash est erroné :

```
char c9 = '\a';
```

```
Error: Invalid escape character
```

La norme Unicode

Java utilise la norme unicode pour l'encodage des caractères.

Caractères Unicode et caratères Java :

- 1 `char` Java = 2 octets ($2^{16} = 65536$ combinaisons possibles de 0 et 1)
- 1 caractère Unicode peut nécessiter plus que 2 octets pour sa représentation (1,112,064 caractères possibles actuellement)
- dans la norme Unicode, la zone comprenant les codes allant de 0 à 65535 s'appelle la zone BMP (Basic Multilingual Plane) : suffisante dans la majorité des cas ;
- les caractères nécessitant des codes plus grands que 65535 sont dits "caractères supplémentaires" (voir les transparents de l'annexe)

  Ce qui suit est valable pour les caractères représentables dans la zone BMP.

Conversion entiers/caractères (BMP)

(BMP : Basic Multingual Plane)

Pour trouver l'Unicode d'un caractère :

- Conversion **char ⇒ int**

```
char c = 'A';
int i = (int)c;   //65
```

Pour trouver le caractère d'un Unicode :

- Conversion **int ⇒ char**

```
int i = 65;
char c = (char)i;   //'A'
```

Affichage de toute la zone BMP de la norme Unicode :

Note : Certains caractères ne sont pas affichables

```
for (int i = 0; i < 65535; i++) {
        System.out.println (i + " " + (char)i);
}
```


J. Sam

Arithmétique des caractères

Dans la norme Unicode, chaque caractère correspond à un `int` :

Que se passe-t'il si un `char` occupe la place d'un `int` dans une expression ?

☞ **il est automatiquement converti à son Unicode** (type `int`)

Exemple : Additions de `char`

`char` + `int` ⇒ `int` :

```
char c1 = 'a';                  // a (97)
int  i1 = c1 + 2;               // 99
int  i2 = (int)c1 + 2;          // 99
char c2 = (char)i1;             // c (99)
```

`char` + `char` ⇒ `int`

```
char c3 = '!';                  // ! (33)
char c4 = '#';                  // # (35)
int  i3 = c3 + c4;              // 68
int  i4 = (int)c3 + (int)c4;    // 68
char c5 = (char)i3;             // D (68)
```

Affichage de caractères

Puisque la conversion `char` ⇒ `int` est automatique :
Attention lors de l'affichage de plusieurs `char` de suite !

Exemple :

```
char c1 = '!';                                  // 33
char c2 = '#' ;                                 // 35
System.out.print(c1 + c2);                      // 68
System.out.print((c1 + c2));                    // 68
System.out.print((char)c1 + (char)c2);          // 68
System.out.print((char)(c1 + c2));              // D
System.out.print(c1 + "" + c2);                 // !#
```

Pas de `nextChar()` dans la classe `Scanner` !?

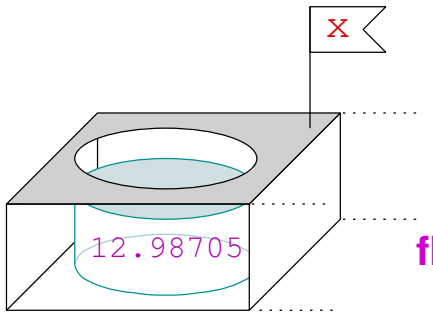
Pour récupérer un caractère (`char`) avec la classe `Scanner`, il faut faire :

```
// Lire la ligne qui contient un caractere
Scanner keyb = new Scanner(System.in);
String s = keyb.next();

// Prendre que le caractere
// c'est a dire le premier element de la String
char c = s.charAt(0);
```

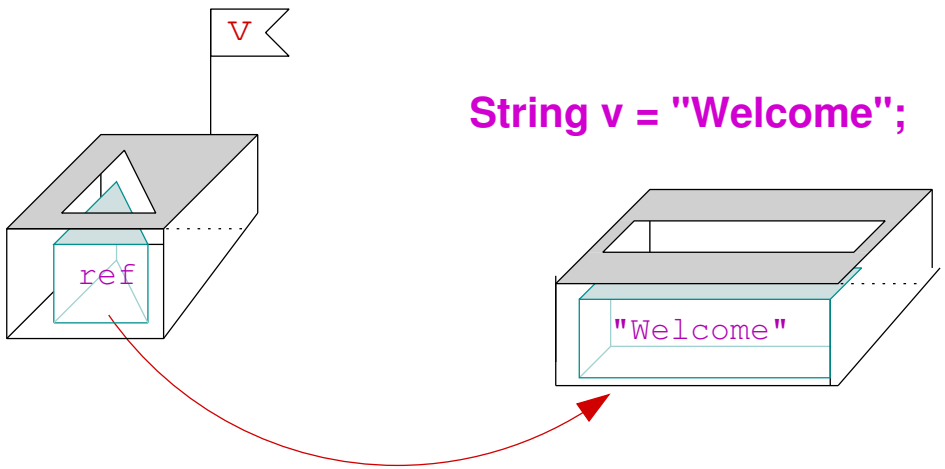
Types de base et types évolués

- Toute variable de type de base stocke directement **une valeur** :



float x = 12.98705;

- Toute variable de type évolué, comme les tableaux ou les chaînes de caractères (`String`) que vous allez voir dans ce cours, stocke **une référence** (adresse) vers une valeur :



String v = "Welcome";

Types de base et types évolués (2)

- Toute variable de type de base stocke directement **une valeur**
- Toute variable de type évolué stocke **une référence** vers une valeur

☞ **Attention** : ceci a une très grande incidence sur la sémantique des opérateurs `=` et `==` en Java !

- `double x = 13.5` veut dire *"J'affecte à x la valeur 13.5"*
- `String s = "coucou"` veut dire *"J'affecte à s une référence à la chaîne de caractères coucou"*

En clair, si `v1` et `v2` sont de type évolué :

- `v1 = v2` affecte l'adresse de `v2` à la variable `v1`
- `v1 == v2` compare l'adresse de `v2` avec celle de `v1`
- `System.out.println(v1);` affiche l'adresse de `v1` (dans le cas général)

Nous y reviendrons . . .

Petit exemple introductif

Vous programmez un jeu en ligne

Exercice : afficher les scores des joueurs

Score	écart à la moyenne
1000	-1860
1500	-1360
2490	-370
6450	3590
...	...

Vous êtes modeste et ne prévoyez pas plus de ... deux joueurs

Solution avec les moyens actuels

```
class Scores {
   public static void main(String [] args) {
      Scanner keyb = new Scanner(System.in);

      // Lecture des donnees et calculs
      System.out.println ("Score Joueur 1:");
      int score1 = keyb.nextInt();
      System.out.println ("Score Joueur 2:");
      int score2 = keyb.nextInt();
      // Calcul de la moyenne
      int moyenne = (score1 + score2) / 2;

      // Affichages
      System.out.println("Score " + " Ecart Moyenne");
      System.out.println(score1 + " "
                         + (score1 - moyenne));
      System.out.println(score2 + " "
                         + (score2 - moyenne));
   }
}
```

Solution avec les moyens actuels (2)

Votre jeu a du succès .. vous voilà obligé de faire le calcul pour 200 joueurs

...adapter le code précédent revient à déclarer **200 variables** !

```
System.out.println ("Score Joueur 1:");
int score1 = keyb.nextInt();
System.out.println ("Score Joueur 2:");
int score2 = keyb.nextInt();
...
System.out.println ("Score Joueur 200:");
int score200 = keyb.nextInt();
int moyenne = (score1 + score2 + ...
                          + score200) / 200;
```

Evidemment, cette solution n'est pas viable !

Il nous faut une **structure de données** !

Les tableaux en Java

En Java, on utilise :

		taille initiale connue *a priori* ? non	taille initiale connue *a priori* ? oui
taille pouvant varier lors de l'utilisation du tableau ?	oui	`ArrayList`	`ArrayList`
	non	tableaux de taille fixe	tableaux de taille fixe

☞ Commençons par les tableaux de tailles fixe

Tableaux

Un tableau est une **structure de données** qui :

- regroupe `n` valeurs du même type
- donne le même nom aux `n` valeurs
- énumère les valeurs de `[0]` à `[n-1]`
- Les `n` valeurs sont les **éléments** du tableau

Exemple : Tableau `scores` contenant 4 `int`

1000	1500	2490	6450
scores[0]	scores[1]	scores[2]	scores[3]

Déclaration d'un tableau

Syntaxe générale : Type des éléments + crochets `[]`.

```
int[] scores;
```

Note : Java autorise la syntaxe équivalente suivante :

```
int scores[];
```

Les crochets indiquent que la variable peut contenir plusieurs éléments du type spécifié

Il existe **deux techniques pour initialiser** les éléments :

1. Dans l'instruction de déclaration
2. Dans des instructions séparées

Initialisation d'un tableau (1)

Si l'on connaît les valeurs de tous les éléments lors de la déclaration du tableau

☞ une seule instruction de **déclaration-initialisation**

1. Déclarer le type du tableau
2. Indiquer les éléments entre accolades
3. Séparer les éléments par des virgules

Exemple : avec des constantes :

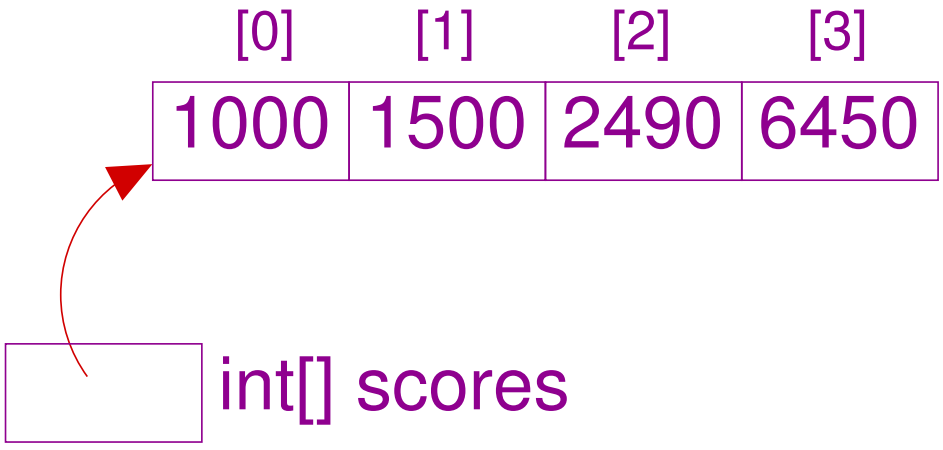
```
int[] scores = {1000, 1500, 2490, 6450};
```

Exemple : avec des expressions :

```
int[] scores = {(2500 * 10),(4200 * 10)};
```

Situation en mémoire

Important : Un tableau n'est pas de type de base, il est donc manipulé via une **référence** !



On dit que la variable `scores` **référence** (ou pointe vers) un tableau de `int`.

La variable `scores` contient une adresse : l'emplacement du tableau en mémoire !

Initialisation d'un tableau (2)

Dans le cas général, on ne connaît pas les valeurs de tous les éléments lors de la déclaration du tableau

On utilise alors plusieurs instructions pour **déclarer et initialiser** :

1. Déclarer le type du tableau
2. Construire le tableau avec :
   `new type [ taille ]`
3. remplir le tableau élément par élément

La **déclaration-construction** d'un tableau peut se faire avec :

- deux instructions distinctes :

```
int[] scores;          // declaration
scores = new int[2]; // construction
```

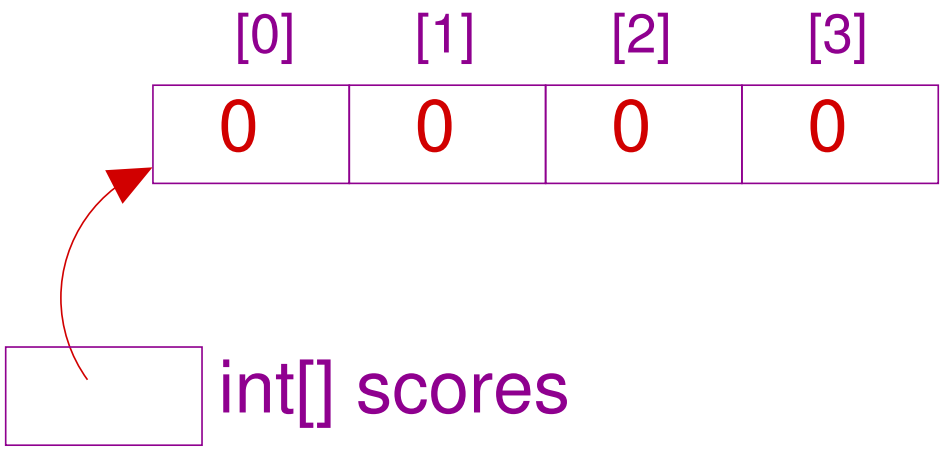
- une seule instruction :

```
int[] scores = new int[2]; // declaration-
                           // -construction
```

Valeurs par défaut

Chaque élément d'un tableau reçoit une **valeur par défaut** lors de la construction avec `new`

int	0
double	0.0
boolean	false
char	'\u0000'
(objet quelconque)	(null)

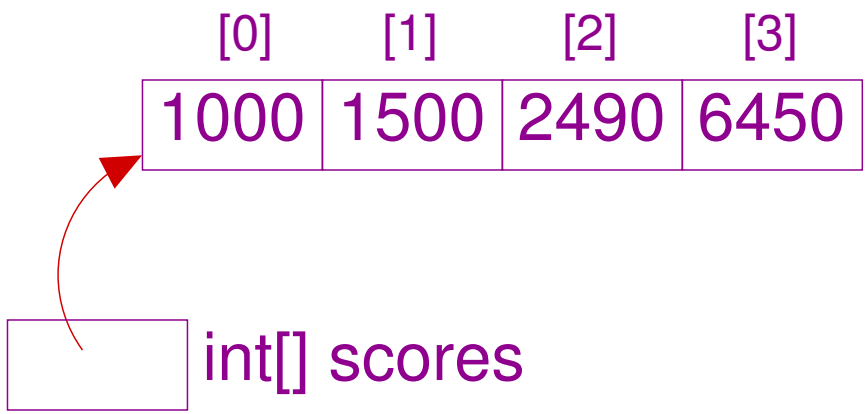


Initialisation du tableau (3)

Une fois le tableau déclaré et construit, il faut le **remplir élément par élément** :

```
int[] scores = new int[4];
scores[0] = 1000;
scores[1] = 1500;
scores[2] = 2490;
scores[3] = 6450;
```

Situation en mémoire : Comme pour le 1er exemple d'initialisation



Tableaux : Affichage

Le code suivant :

```
double[] t1 = {1.1, 2.2, 3.4} ;
System.out.println(t1);
```

affiche la référence au tableau `t1`, donc une adresse.

Si l'on veut faire afficher les entrées du tableau référencé par `t1`, il faut prévoir une boucle (on verra comment faire un peu plus loin) !

Exemple revisité (1)

```
System.out.print ("Donnez le nombre de joueurs:");
int n = keyb.nextInt();
if (n > 0) {
   int moyenne = 0;
   int scores [] = new int[n];

    // Lecture des scores
   for (int i = 0; i < n; i++) {
      System.out.println ("Score Joueur " + i + " :");
      scores[i] = keyb.nextInt();
      moyenne += scores[i];
   }

   moyenne /= n; // calcul de la moyenne

    // Affichages
   System.out.println(" Score " + " Ecart Moyenne");
   for (int i = 0; i < n ; i++) {
      System.out.println(scores[i] + " "
                         + (scores[i] - moyenne));
   }
}
```

Exemple revisité (2)

Avec un tableau, même nombre de "variables" mais

beaucoup moins d'identificateurs !

Nombre d'éléments d'un tableau

Pour connaître la **taille d'un tableau** :

- `nomTableau.length`

Exemple :

```
int[] scores = {1000, 1500, 2490, 6450};
System.out.println(scores.length); // 4
boolean[] bs = {true, false};
System.out.println(bs.length);     // 2
```

Attention ! `length` donne le **nombre possible** d'éléments. Le remplissage effectif du tableau n'a pas d'importance !

Exemple :

```
int[] scores;          // declaration
scores = new int[2]; // construction
```

Tableaux partiellement remplis

Dans certains cas, il est intéressant de créer un tableau de taille fixe et de le remplir ou de l'utiliser seulement partiellement.

Exemples :

- la taille exacte du tableau n'est pas connue lors de l'écriture du programme
- on aurait besoin d'un tableau de différentes tailles suivant l'exécution du programme

Dans ces deux cas, on **surestime** la taille réelle du tableau (un peu de gaspillage mémoire) et on utilise seulement une partie de celui-ci.

Tableaux partiellement remplis (2)

```
int tailleMaximale = 100;
int[] scores = new int[tailleMaximale]
```

Pour savoir jusqu'où le tableau peut être lu (sinon, on accèderait une valeur non affectée), il faut maintenir soi-même une variable `nombreUtilise`.
Si le tableau `scores` est rempli de 4 entiers : `1,4,2,8` alors il faudra que `nombreUtilise` soit 4.
De cette façon, on parcourera le tableau `scores` en faisant :

```
// pas i < scores.length !!
for (int i=0; i < nombreUtilise; i++)
        System.out.println(scores[i]); }
```

Si on parcourait le tableau jusqu'à `scores.length`, on lirait des valeurs non initialisées (toutes les valeurs `scores[i]` où `i > nombreUtilise`!)

Erreurs courantes avec les tableaux

1. Problème d'indice
2. Accès avant la construction du tableau
3. Accès à un élément non initialisé (objets, valeur `null`)
4. Confusion syntaxique tableau/objet

Les deux derniers types d'erreurs seront exposés après introduction de la notion d'objet

Erreur : problème d'indice

Attention !

- L'indice est toujours un **`int`**
- Il faut respecter les bornes du tableau :
  - ☞ Toujours énumération de `[0]` à `[n-1]`

Exemples :

```
int[] entiers = new int[250];
entiers[1.0] = 1; // erreur type
entiers[-13] = 2; // erreur borne
entiers[250] = 4; // erreur borne
```

Erreur : accès avant construction

Il est impossible en Java d'accéder à un élément si le tableau n'a pas encore été construit.

La construction se fait :

- Soit en indiquant les valeurs directement dans l'instruction de déclaration
- Soit en spécifiant la taille avec `new + type + taille`

Exemple :

```
int[] entiers1 = {1, 2, 3}; // Methode 1
entiers1[0] = 4;            // OK
int[] entiers2;             // Methode 2
// entiers2 = new int[10];  // construction oubli'ee
entiers2[0] = 4;            // erreur
```

Tableaux : sémantique de l'opérateur =

```
// Les tableaux a et b pointent vers deux emplacements
// differents en memoire
int[] a = new int[10]; // tableau de 10 entiers
int[] b = new int[10]; // tableau de 10 entiers

for (i=0; i<a.length; i++) {
   a[i]=i; // remplissage du tableau point'e par a
}
b=a; // operateur =  (affectation)
System.out.println("a[2] vaut " + a[2] + " et b[2] vaut " + b[2]);
a[2]=42;
System.out.println("a[2] vaut " + a[2] + " et b[2] vaut " + b[2]);
```

ce qui affiche :

a[2] vaut 2 et b[2] vaut 2
a[2] vaut 42 et b[2] vaut 42

Les deux tableaux `a` et `b`, après l'affectation `b=a;` pointent vers le même emplacement mémoire ⇒ En changeant `a[2]` on change alors implicitement `b[2]` et inversément !

Le tableau créé pour `b` : `int[] b = new int[10];` n'est donc jamais rempli ni utilisé !

Tableaux : utilisation (rare) de l'opérateur =

A moins de vouloir deux noms de variables pour le même tableau, il n'y a pas d'intérêt à vouloir assigner un tableau un à autre

☞ l'utilisation de l'opérateur = pour les tableaux est donc rare !

Pour avoir deux tableaux distincts `a` et `b` qui ont les mêmes valeurs (c'est à dire faire une copie de `a` dans `b`) il aurait fallu utiliser :

```
for (int i=0; i < a.length; i++) {
        b[i] = a[i];
}
```

Attention : il faut que `b.length` $\geq$ `a.length` !

Tableaux : sémantique de l'opérateur ==

L'opérateur `a == b` teste si les variables `a` et `b` pointent vers le même emplacement mémoire.

⇒ ce qui est donc le cas lors de l'affectation `b = a;`

L'opérateur `a==b` **ne teste pas** l'égalité des valeurs contenues dans les tableaux pointés par `a` et `b` !

Pour vérifier l'égalité de contenu des tableaux, il faut écrire explicitement les tests :

```
if (a == null || b == null || a.length != b.length) {
        System.out.println("contenus differents ou nuls");
      }
      else {
         int i = 0;
         while (i < a.length && (a[i] == b[i])){
            ++i;
         }
         if (i >= a.length)
            System.out.println("contenus identiques");
         else
            System.out.println("contenus differents");
      }
```

Tableaux à plusieurs dimensions

Comment déclarer un tableau à plusieurs dimensions ?

☞ On ajoute simplement un niveau de [ ] de plus :

C'est en fait un tableau de tableaux...

Exemples :

```
double [][] statistiques =
        new double[nb_cantons][nb_communes];

int [][] scores =
        new int[nb_joueurs][nb_parties];
```

`scores[i]` est un tableau de `nb_parties` entiers
☞ `scores` est bien un tableau de tableaux.

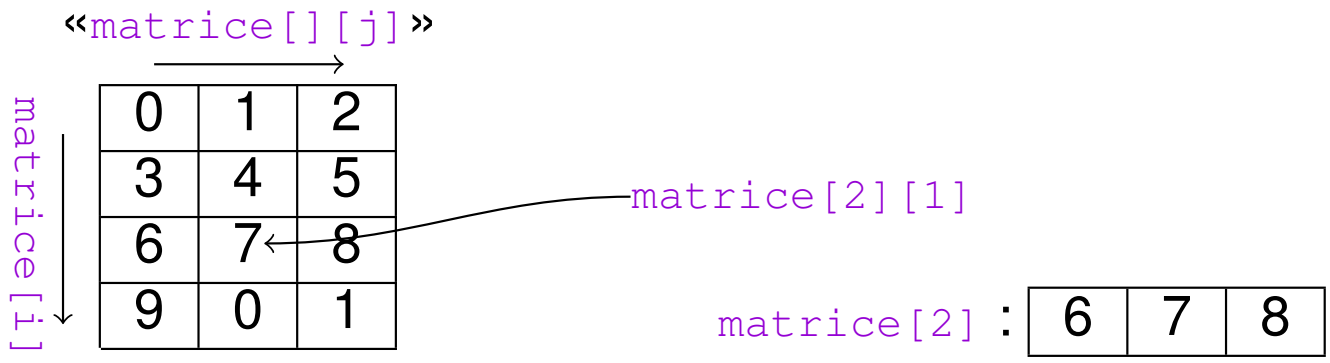
En faisant une analogie avec les mathématiques, un tableau à une dimension représente donc un vecteur, un tableau à deux dimensions une matrice et un tableau de plus de deux dimensions un tenseur.

Tableaux à plusieurs dimensions

Les tableaux multidimensionnels peuvent également être initialisés lors de leur déclaration.
Il faut bien sûr spécifier autant de valeurs que les dimensions et ceci pour chacune des dimensions.

Exemple :

```
int [][] matrice = {
        { 0, 1, 2 },
        { 3, 4, 5 },
        { 6, 7, 8 },
        { 9, 0, 1 }
    };
```



Déclaration-initialisation (1)

Cas 1 : On connaît tous les éléments lors de la déclaration

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

Accès aux éléments de la 1ère dimension :

- Type `int[]`
- `y[0]`, `y[1]` et `y[2]`

Accès aux éléments de la 2ème dimension :

- Type `int`
- `y[0][0]` `y[0][1]` (1er tableau)
- `y[1][0]` `y[1][1]` (2ème tableau)
- `y[2][0]` `y[2][1]` (3ème tableau)

Déclaration-initialisation (2)

Cas 2 : On ne connaît pas tous les éléments lors de la déclaration

```
int[][] y = new int[3][2];

// remplissage a la main

y[0][0] = 1;
y[0][1] = 2;

y[1][0] = 3;
y[1][1] = 4;

y[2][0] = 5;
y[2][1] = 6;
```

Parcours

Le moyen le plus naturel de parcourir un tableau multidimensionnel consiste à utiliser des boucles `for` imbriquées :

1. 1ère boucle : fait varier le 1er indice
2. 2ème boucle : fait varier le 2ème indice

Exemple :

```
for (int i = 0; i < y.length; i++)
   for (int j = 0; j < y[i].length; j++)
      System.out.println (y[i][j]);
```

Exemple : Variante (tous les éléments de la 1ère dimension ayant la même taille)

```
System.out.println(y[0].length); // 2
System.out.println(y[1].length); // 2
System.out.println(y[2].length); // 2
for (int i = 0; i < y.length; i++)
   for (int j = 0; j < y[0].length; j++)
```

`for` pour des collections

Depuis la version 5.0, Java permet de parcourir facilement des collections de données au moyen d'une nouvelle forme d'itération.

Cette forme d'itération s'applique notamment aux tableaux.

Syntaxe générale :

```
for (type variable : collection)
  Instructions
```

A chaque pas de l'itération `variable` prend la valeur de l'élément suivant dans la collection `collection`.

`type` est le type des éléments de la collection

`Instructions` est soit une instruction élémentaire soit un bloc d'instructions.

Attention : cette tournure ne permet pas d'affecter des valeurs au tableau !

for pour des collections (2)

C'est un moyen très pratique pour afficher les éléments d'un tableau par exemple :

```
for (int variable : tab)
        System.out.println(variable);
```

Ce code est équivalent à :

```
for (int i=0; i < tab.length; i++)
        System.out.println(tab[i]);
```

`for` pour des collections (3)

Attention, une itération sur ensemble de valeurs (`for` pour les collections) :

- ne permet pas de modifier le contenu du tableau
- ne permet d'itérer que sur un seul tableau à la fois : il n'est pas possible de traverser en une passe deux tableaux pour les comparer par exemple
- ne permet l'accès qu'à un seul élément : on ne peut pas par exemple comparer un élément du tableau et son suivant
- itère d'un pas en avant seulement.

Les tableaux

Déclaration : `type[ ] identificateur;`
Construction : `new type[taille]`
Initialisation avec éléments connus :

$$\texttt{type[ ] identificateur=\{val}_1\texttt{, ... , val}_{taille}\texttt{\};}$$

Accès aux éléments : `tab[i]` `i` entre **0** et **taille-1**

Tableaux multidimensionnels (exemple avec 2 dimensions) :
Déclaration : `type[ ][ ] identificateur;`
Construction : `new type[taille1][taille2];`
Accès : `tab[i][j];`

Itération particulière pour parcourir un tableau :

```
for (type variable : collection)
     Instructions
```

Le type `String`

`"Bonjour tout le monde !"`

Les chaînes de caractères Java sont définies par le type **String**.
(En toute rigueur, ce n'est pas un type comme les types élémentaires mais une classe).

Syntaxe : déclaration d'une chaîne de caractères

```
String identificateur;
```

L'initialisation peut se faire en affectant à la variable un littéral de type `String` (exemple : `"Bonjour"`)

Exemples : Déclarations et initialisation

```
String unNom;
String citation="Why use Windows when there are doors?";
```

Notes : nous verrons plus tard qu'il est possible de construire un `String` différement, en utilisant la notion d'objet.

J. Sam

String : Sémantique des opérateurs = et ==

- Comme pour les tableaux, une variable de type String contient une **référence** (vers une chaîne de caractères). La sémantique des opérateurs = et == est donc la même que pour les tableaux :

```
String chaine = "";     // chaine pointe vers ""
String chaine2 = "foo";// chaine2 pointe vers "foo"
chaine = chaine2 ;      // chaine et chaine2
                        // pointent vers "foo"
chaine == chaine2;      // retourne true
```

- Les litteraux de type String occupent une zone mémoire unique :

```
String chaine1 = "foo";  // chaine1 pointe vers
                         // le litteral "foo"
String chaine2 = "foo" ; // chaine2 pointe vers
                         // le litteral "foo"
chaine1 == chaine2;      // true : chaine1 et
                         // chaine2 contiennent
                         //la meme adresse
```

String : Affichage

```
String chaine = "Welcome";
System.out.print(chaine);
```

Puisque la variable `chaine` contient une référence à la zone mémoire contenant la chaîne `"Welcome"`, il est raisonnable de penser que ce code affiche une adresse (comme pour les tableaux de manière générale) : **ce n'est pas le cas !**

☞ Le code précédent affiche `Welcome`

Les raisons de ce comportement particulier des `String` en tant qu'entités de type évolué seront expliquées lorsque nous aborderons la POO.

La concaténation

La **concaténation** de chaînes est effectuée par l'opérateur `+`.

`chaine1 + chaine2` correspond à une **nouvelle chaîne** associée à la valeur littérale constituée de la concaténation des valeurs littérales de `chaine1` et de `chaine2`.

Combinaisons possibles : `String + type_de_base`, `type_de_base + String`, `String + String`, où `String` correspond à une variable littérale de type `String`, et `type_de_base` à une variable ou une valeur littérale de l'un des types de base (`char`, `boolean`, `double`, `int`..).

Les concaténations de la forme `String+char` (resp. `char+String`) constituent donc un moyen très pratique pour ajouter des caractères à la fin (resp. au début) d'une chaîne.

La concaténation : exemples

- Constitution du nom complet à partir du nom de famille et du prénom :

```
String nom;
String prenom;
nom = nom + ' ' + prenom;
```

- Ajout d'un 's' final au pluriel :

```
String reponse="solution";
//...
if (n > 1) {
   reponse = reponse + 's';
}
```

Important ! La concaténation **ne modifie jamais** les chaînes concaténées. Elle effectue une **copie** de ces chaînes dans une autre zone en mémoire.

(Non-)Egalité de `Strings`

Les opérateurs suivants :

`==`	égalité
`!=`	non-égalité

testent si deux variables `String` **font référence** (ou non) à la même zone mémoire (occupée par une chaîne de caractères).

Ceci est le cas lorsque les variables de types `String` ont été initialisées au moyen de **littéraux**

Exemple : utilisation de l'opérateur `!=`

```
while (reponse != "oui") ....;
```

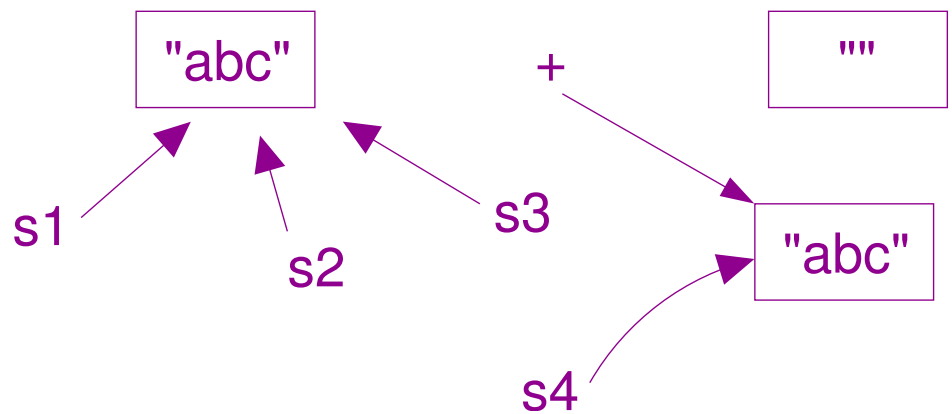
(voir aussi le tranparent 53)

(Non-)Egalité de Strings (2)

Exemple :

```
String s1 = "abc";     // s1 pointe vers le litteral "abc"
String s2 = "abc";     // idem (donc meme zone memoire que s1)
String s3 = s2;        // s3 stocke la meme adresse que s2
String s4 = s1 + "";   // s4 contient l'adresse d'une nouvelle
                       // chaine
                       // (construite par concatenation)
System.out.println ((s1==s2) && (s2==s3)); //affiche true
System.out.println(s4);                    // affiche abc
System.out.println ((s1==s4));             //affiche false
```

Situation en mémoire :



Comment faire pour comparer lexicographiquement deux chaînes ?

☞ **Traitement spécifique** aux String

Littéraux introduits par l'utilisateur

Un littéral introduit par l'utilisateur suite à une instruction de lecture n'est pas dans le pool des littéraux

Pour qu'il y soit, il faut l'y mettre explicitement au moyen de `intern`

Exemple

```
Scanner s = new Scanner(System.in);
String response;
do {
   response = s.next();
   //on met le litteral lu dans le pool
   response = response.intern();
   System.out.println("Read: " + response);
   // sans le intern, la boucle ne pourrait
   // pas s'arreter!
} while (response != "oui");
```

Traitements spécifiques aux chaînes

Certains traitements sont **propres aux** `String`. Ils s'utilisent avec la syntaxe particulière suivante :

```
nomDeChaine.nomDeTraitement(param1, param2, ...);
```

Ces traitements s'appellent des **méthodes** en Java. Vous en comprendrez la syntaxe dès le cours sur la Programmation Orientée Objet.

Exemples (où `chaineXX` est une variable de type `String`) :

- `chaine1.equals(chaine2)` teste si deux chaînes sont constituées des mêmes caractères
- `chaine1.compareTo(chaine2)` compare l'ordre lexicographique des deux chaînes
- `chaine.length()` renvoie la taille (i.e. le nombre de caractères) de `chaine`. (**Attention !** il y a une paire de parenthèses, ça n'est pas comme pour les tableaux !)

J. Sam

equals

Teste si deux chaînes sont **constituées des mêmes caractères** :

```
String s1 = "abc";
String s2 = "abc";
String s3 = "aBc";
System.out.println ((s1.equals(s2))); // true
System.out.println ((s1.equals(s3))); // false
```

compareTo

Soient `s1` et `s2` deux `String`s à comparer lexicographiquement (ordre de la norme Unicode).

Attention !

- L'ordre alphabétique des ordinateurs ne respecte pas les lettres accentuées
- chiffres $<$ majuscules $<$ minuscules : `"Z" <"a"`, `"1java" < "java"`

Note : référez-vous au programme d'énumération des caractères Unicode pour voir l'ordre alphabétique considéré par vos programmes Java.

compareTo (2)

Soient `s1` et `s2` deux `String`s à comparer lexicographiquement (ordre de la norme Unicode).
Utilisation de la méthode :

```
int i = s1.compareTo(s2);
```

- `i = 0` si `s1.equals(s2)`
- `i < 0` si lexicographiquement `s1 < s2`
- `i > 0` si lexicographiquement `s1 > s2`

String s1	String s2	s1.compareTo(s2)
abc	def	-3
abc	Abc	32
abc	abcdef	-3
abc	123	48

Note : La signification exacte de la valeur retournée est sans intérêt

Les chars d'un String

Il est souvent nécessaire d'**accéder à un caractère précis** dans un `String`.

Note : Les caractères sont numérotés comme les éléments d'un tableau (à partir de 0)

- L'instruction `chaine.charAt(index)` retourne le caractère occupant la position `index` dans la `String chaine`
- L'instruction `chaine.indexOf(caractere)` retourne la position de la première occurence du `char caractere` dans la `String chaine` (et -1 si caractere n'est pas dans `chaine` !)

Exemple :

```
String s1 = "abcmbx";
int longueur = s1.length();          // 6
char c1 = s1.charAt(0);              // a
char c2 = s1.charAt(longueur-1);     // x
int i = s1.indexOf('b');             // 1
```

Les chars d'un String (2)

Exercice : qu'affichera le programme suivant :

```
String essai = "essai";
String test = "";
for (int i=1; i <= 3; i++) {
        test = test + essai.charAt(6-2*i);
        test = essai.charAt(i) + test;
}
System.out.println(test);
```

replace et substring

- `chaine.replace(char1,char2)` : remplace chaque occurence de `char1` par `char2` dans `chaine`.

  Exemple :

  ```
  String exemple ="abracadabra";
  exemple.replace('a','*');
  ```

  construit la chaîne `"*br*c*d*br*"`.

  `exemple` vaut toujours `"abracadabra"`.

- `chaine.substring(position1, position2)` : retourne la sous-chaîne comprise entres les indices de `position1` (compris) et `position2` (non-compris)

  Exemple : `String exemple="anticonstitutionnel";` `exemple.substring(4,16);` construit la chaîne `"constitution"`.

Les chaînes de caractères

Déclaration/initialisation : `String identificateur="valeur";`

Affectation :
```
chaine1 = chaine2;
chaine1 = "valeur";
```

Concaténation :
```
chaine1 = chaine2 + chaine3;
chaine1 = chaine2 + "valeur";
chaine1 = chaine2 + type_de_base;
```

Accès au (`i`+1)-ème caractère : `chaine.charAt(i)` ;

Fonctions spécifiques :

taille :	`chaine.length()`
comparaison alphabétique :	`chaine.equals(chaine2)`
	`chaine1.compareTo(chaine2)`
remplacement :	`chaine.replace(char1,char2)`
recherche :	`chaine.indexOf(char)`
extraction	`chaine.substring(indice1, indice2)`

Il ne s'agit là que d'un tout petit échantillon (consultez l'API de `String` pour aller plus loin)

Important : Le type `String` est immutable : tous les traitements qui changent le contenu de la chaîne de caractère (concatenation, par exemple) **créent une copie** de la chaîne. . . à un endroit en mémoire différent, donc !

Tableaux dynamiques

Un **tableau dynamique**, est une *collection* de données homogènes, dont *le nombre peut changer* au cours du déroulement du programme, par exemple lorsqu'on ajoute ou retire des éléments au/du tableau.

Les tableaux dynamiques sont définis en Java par le biais du type

```
ArrayList
```

Pour les utiliser, il faut tout d'abord importer les définitions associées à l'aide de la directive suivante :

```
import java.util.ArrayList;
```

à placer en tout début de fichier

Déclaration d'un tableau dynamique

Une variable correspondant à un tableau dynamique se déclare de la façon suivante :

```
ArrayList<type> identificateur;
```

où *identificateur* est le nom du tableau et *type* correspond au type des éléments du tableau.

Le type des éléments doit nécessairement correspondre à un **type évolué**.

Exemple :

```
ArrayList<String> tableau;
```

Initialisation d'un tableau dynamique

Un tableau dynamique initialement vide (sans aucun élément) s'initialise comme suit :

```
ArrayList<type> identificateur= new ArrayList<type>();
```

où *identificateur* est le nom du tableau et *type* correspond au type des éléments du tableau.

Exemple :

```
ArrayList<String> tableau = new ArrayList<String>();
```

Méthodes spécifiques

Un certain nombre d'opérations sont **directement attachées** au type `ArrayList`.

L'utilisation de ces opérations spécifiques se fait avec la syntaxe suivante :

```
nomDeTableau.nomDeMethode(arg1, arg2, ...);
```

Exemple :

```
ArrayList<String> prenoms = new ArrayList<String>();

System.out.println(prenoms.size()); // affiche 0
```

Méthodes spécifiques

Quelques fonctions disponibles pour un tableau dynamique nommé `tableau`, de type `ArrayList<type>` :

`tableau.size()` : renvoie la taille de `tableau` (un entier)

`tableau.get(i)` : renvoie l'élément à l'indice `i` dans le tableau (`i` est un entier compris entre 0 et `tableau.size() - 1`)

`tableau.set(i, valeur)` : affecte `valeur` à la case `i` du tableau (cette case doit avoir été créée au préalable)

Méthodes spécifiques

Quelques fonctions disponibles pour un tableau dynamique nommé `tableau`, de type `ArrayList<type>` :

`tableau.isEmpty()` : détermine si `tableau` est vide ou non (`boolean`).

`tableau.clear()` : supprime tous les éléments de `tableau` (et le transforme donc en un tableau vide). Pas de (type de) retour.

Méthodes spécifiques

Quelques fonctions disponibles pour un tableau dynamique nommé `tableau`, de type `ArrayList<type>` :

`tableau.remove(i)` : supprime l'élément d'indice `i`

`tableau.add(valeur)` : ajoute le nouvel élément `valeur` à la fin de `tableau`. Pas de retour.

Exemple de quelques manipulations de base

```
import java.util.ArrayList; // pour pouvoir les utiliser

class ArrayListBase {
   public static void main(String[] args){
         // un tableau dynamique vide
         // (contenu : pas de type elementaire)
         ArrayList<String> liste = new ArrayList<String>();

         liste.add("un");
         liste.add("deux");

         // affiche :  un deux
         for (String v : liste){
            System.out.print(v + " ");
         }
                  System.out.println();
         //affiche l'entree d'indice 1 -> deux
         System.out.println(liste.get(1));
         // modifie l'entree d'indice 0
         liste.set(0, "premier");
         // affiche  premier deux
         for (String v : liste){
            System.out.print(v + " ");
         }
      }
}
```

Conversions entre types de bases et types évolués

Un `ArrayList` ne peut contenir que des données de type évolué.

- ☞ Que faire pour les types de base ?

En Java, à chaque type de base correspond un type évolué prédéfini :

- `Integer` est le type évolué correspondant à `int`
- `Double` est le type évolué correspondant à `double`
- etc.

- ☞ Utiles dans certains contextes (typiquement les `ArrayList`)
- ☞ La conversion du type de base au type évolué **se fait automatiquement**

Comparaison d'éléments

Attention : les éléments d'un tableau dynamique sont toujours des **références**

☞ Comparaison au moyen de `equals`

```
ArrayList<Integer> tab = new ArrayList<Integer>();

tab.add(2000);
tab.add(2000);

// affiche false:
System.out.println(tab.get(0) == tab.get(1));

// affiche true:
System.out.println((tab.get(0)).equals(tab.get(1)));
```

Tableaux dynamiques

Déclaration/initialisation (tableau vide) :

```
ArrayList<type> identificateur = new ArrayList<type>();
```

Fonctions spécifiques :

taille :	`tableau.size()`
ajout d'une valeur à la fin du tableau :	`tableau.add(valeur)`
suppression de la valeur d'indice `i` :	`tableau.remove(i)`
accès au ième élément :	`tableau.get(i)`
modification de la ième valeur :	`tableau.set(i,valeur)` (la case `i` doit exister)
tester si le tableau est vide :	`tableau.isEmpty()`
vider le tableau	`tableau.clear()`

Consultez l'API de `ArrayList` pour aller plus loin

Important : Un `ArrayList` ne peut contenir que des données de types évolués (objets)

Pour préparer le prochain cours

- Vidéos et quiz du MOOC semaine 6 :
  - Fonctions : introduction [12 :00]
  - Fonctions : appel [8 :17]
  - Fonctions : passage des arguments [16 :12]
  - Fonctions : entête [16 :12]
  - Fonctions : définition [16 :42]
  - Fonctions : surcharge [5 :19]
  - Fonctions : méthodologie [5 :40]

- Le prochain cours :
  - de 14h15 à 15h : résumé et quelques approfondissements

Annexe : Java et la norme Unicode (1)

Convention d'interprétation :

- Les caractères de la zone Unicode BMP sont représentés au moyen d'**un `char` Java**
- Les caractères supplémentaires sont représentés au moyen de **deux `char` Java**

Correspondance `char` / `int`

- Dans la zone BMP, un `char` Java est directement substituable au code Unicode correspondant (un code Unicode `U+xxxx` a pour équivalent Java `'\uxxxx'`, où `xxxx` est un nombre en base hexadécimale)
  - `char c = 'A'` et `char c = '\u0041'` sont deux notations équivalentes en Java (car le code Unicode de `'A'` est `U+0041`)
  - comme la valeur de `0041` en décimal est `65`, il existe une correspondance entre `'A'` et son code entier `65` (comme nous l'avons vue en début de cours)

Annexe : Java et la norme Unicode (2)

Terminologie :

- une caractère Unicode = un “codepoint”
- un `char` Java = un “codeunit”
  - ☞ un “codepoint” est consitué de un “codeunit” dans la zone BMP et de deux “codeunits” pour les caractères supplémentaires
- Par exemple, un caractère gothique (`ahsa`) a pour code Unicode `U+10330` (plus grand que 65535), il sera donc représenté de façon interne par une paire de `char` java (“codeunits”) : `\uD800,\uDF30`

J. Sam