Le type `String`

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`

Exemples : Déclaration et initialisation

```
String unNom;
String message = "Bonjour tout le monde !";
```

Le type `char`

Les caractères (constituants d'une chaîne) peuvent aussi se représenter en tant que tels :

☞ le type `char`

- Le caractère s'écrit entre guillemets simples
- 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 :

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

- Caractère qui risque d'être mal interprété :

```
char c7 = '\'';  //guillemet simple
char c8 = '\\'; //backslash
```

- Sinon, le « backslash » est erroné :

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

`Error: Invalid escape character`

`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
```

`String` : Sémantique des opérateurs = et ==

Les litteraux de type `String` occupent une zone mémoire unique

☞ « Pool » des littéraux

```
String chaine1 = "foo";  // chaine1 pointe vers le litteral "foo"
String chaine2 = "foo" ; // chaine2 pointe vers le litteral "foo"
if (chaine1 == chaine2 ) // true : chaine1 et chaine2 contiennent la même
                         //adresse
```

`String` : Affichage

Qu'affiche le code suivant ?

```
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`

☞ Pour les `String` l'affichage est défini de sorte à prendre en compte la référence pointée plutôt que la référence elle-même. C'est une exception.

Concaténation

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

Exemple : constitution du nom complet à partir du nom de famille et du prénom :

```
String nom;
String prenom;
....
nom = nom + " " + prenom;
```

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

Concaténation

Les combinaisons suivantes sont possibles pour la concaténation de deux chaînes :

`String + String`

`String + typeDeBase` `typeDeBase + String`

où `String` correspond à une variable ou une valeur littérale de type `String`, et `typeDeBase` à une variable ou une valeur littérale de l'un des types de base (`char`, `boolean`, `double`, `int` etc.).

Exemple revisité (avec `char`) :

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

Concaténation

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

De même la concaténation `char+String` permet l'ajout d'un caractère en début de chaîne.
Exemple : ajout d'un 's' final au pluriel :

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

(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**

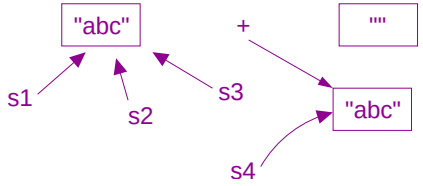
Exemple : utilisation de l'opérateur `!=`

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

(Non-)Egalité de `Strings` (2)

```
String s1 = "abc";    // s1 pointe vers le littéral "abc"
String s2 = "abc";    // idem (donc même zone mémoire que s1)
String s3 = s2;       // s3 stocke la même adresse que s2
String s4 = s1 + "";  // s4 contient l'adresse d'une nouvelle chaîne
                      // (construite par concaténation)
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 les contenus référencés** plutôt que les références ?

☞ **Traitement spécifique** aux `String`

Comparaison de `String`

`chaine1.equals(chaine2)` teste si les chaînes de caractères référencées par `chaine1` et `chaine2` sont constituées des mêmes caractères

```
String s1 = "abc";
String s2 = "aBc";
String s4 = s1 + "";

System.out.println(s1.equals(s4)); // true
System.out.println(s1.equals(s2)); // false
```

(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**

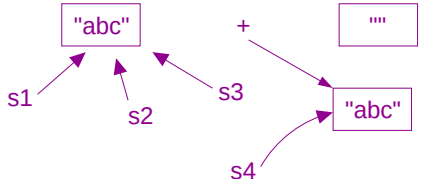
Exemple : utilisation de l'opérateur `!=`

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

(Non-)Egalité de `Strings` (2)

```
String s1 = "abc";     // s1 pointe vers le littéral "abc"
String s2 = "abc";     // idem (donc même zone mémoire que s1)
String s3 = s2;        // s3 stocke la même adresse que s2
String s4 = s1 + "";   // s4 contient l'adresse d'une nouvelle chaîne
                       // (construite par concaténation)
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 les contenus référencés** plutôt que les références ?

☞ **Traitement spécifique** aux `String`

Comparaison de `String`

`chaine1.equals(chaine2)` teste si les chaînes de caractères référencées par `chaine1` et `chaine2` sont constituées des mêmes caractères

```
String s1 = "abc";
String s2 = "aBc";
String s4 = s1 + "";

System.out.println(s1.equals(s4)); // true
System.out.println(s1.equals(s2)); // false
```

Les `char` d'un `String`

- L'instruction `chaine.charAt(index)` donne le caractère occupant la position `index` dans la `String chaine`
- L'instruction `chaine.indexOf(caractere)` donne la position de la première occurence du `char caractere` dans la `String chaine`, et -1 si `caractere` n'est pas dans `chaine`.
- `chaine1.length()` donne la taille (c'est-à-dire le nombre de caractères) de `chaine1`. **Attention :** il y a une paire de parenthèses ; différent des tableaux !

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

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);
```

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 caractère
Scanner keyb = new Scanner(System.in);
String s = keyb.nextLine();

// Prendre comme caractère le premier élément de la String
char c = s.charAt(0);
```

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.nextLine();
   //on met le littéral lu dans le pool
   response = response.intern();
   System.out.println("Read: " + response);
   // sans le intern, la boucle ne s'arrête pas!
} while (response != "oui");
```

Traitements spécifiques aux chaînes

Nous avons vu que certains traitements sont **spécifiques** aux `String`.

Ils s'utilisent en fait tous avec la syntaxe particulière suivante :

`nomDeChaine.nomDeTraitement(arg1, arg2 ...);`

Ces traitements s'appellent des **méthodes** en Java.

☞ Ils produisent toujours une nouvelle chaîne de caractères

replace

`chaine.replace(char1, char2)` : construit une nouvelle chaîne valant `chaine` où `char1` est remplacé par `char2`.

Exemple :

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

construit la nouvelle chaîne `"*br*c*d*br*"`.
`exemple` vaut toujours `"abracadabra"`.

substring

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

Exemple :

```
String exemple = "anticonstitutionnel";
String racineMot = exemple.substring(4,16);
```

construit la nouvelle chaîne `"constitution"`.

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

Les `ArrayList`

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;

class ArrayListExemple {
 public static void main(String[] args){
   ArrayList<String> liste = new ArrayList<String>();

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

   for(String v : liste) {
    System.out.print(v + " ");
   }

   System.out.println(liste.get(1));

   liste.set(0, "premier");

   }
  }
}
```

Que faire pour des 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**

Exemple

Ecrivons un programme qui (ré)initialise un tableau dynamique d'entiers en les demandant à l'utilisateur, qui peut

- ajouter des nombres strictement positifs au tableau
- recommencer au début en entrant *0*
- effacer le dernier élément en entrant un nombre négatif

```
Saisie de 3 valeurs :
Entrez la valeur 0 : 5
Entrez la valeur 1 : 2
Entrez la valeur 2 : 0
Entrez la valeur 0 : 7
Entrez la valeur 1 : 2
Entrez la valeur 2 : -4
Entrez la valeur 1 : 4
Entrez la valeur 2 : 12

-> 7 4 12
```

Exemple

Ecrivons un programme qui (ré)initialise un tableau dynamique d'entiers en les demandant à l'utilisateur, qui peut

- ajouter des nombres strictement positifs au tableau
- recommencer au début en entrant *0*
- effacer le dernier élément en entrant un nombre négatif

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

System.out.println("Donnez la taille voulue : ");
int taille = scanner.nextInt();
System.out.println("Saisie de " + taille + " valeurs :");
while (vect.size() < taille) {
  System.out.println("Entrez la valeur " + vect.size() + " : ");
  int val = scanner.nextInt();
  if ((val < 0) && (!vect.isEmpty())) { vect.remove(vect.size() - 1); }
  else if (val == 0)                  { vect.clear();                 }
  else if (val > 0)                   { vect.add(val);                }
}
```

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);

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

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