Utiliser les objets (String et ArrayList)

Serge Rosmorduc – V. Aponte

CNAM-Paris

13 février 2018

Introduction

- Un programmeur utilise énormément de classes écrites par d'autres ;
- plusieurs milliers classes dans l'API standard de java ;
- très (très) nombreuses bibliothèques disponibles
- dans ce cours, nous allons apprendre à utiliser un objet.

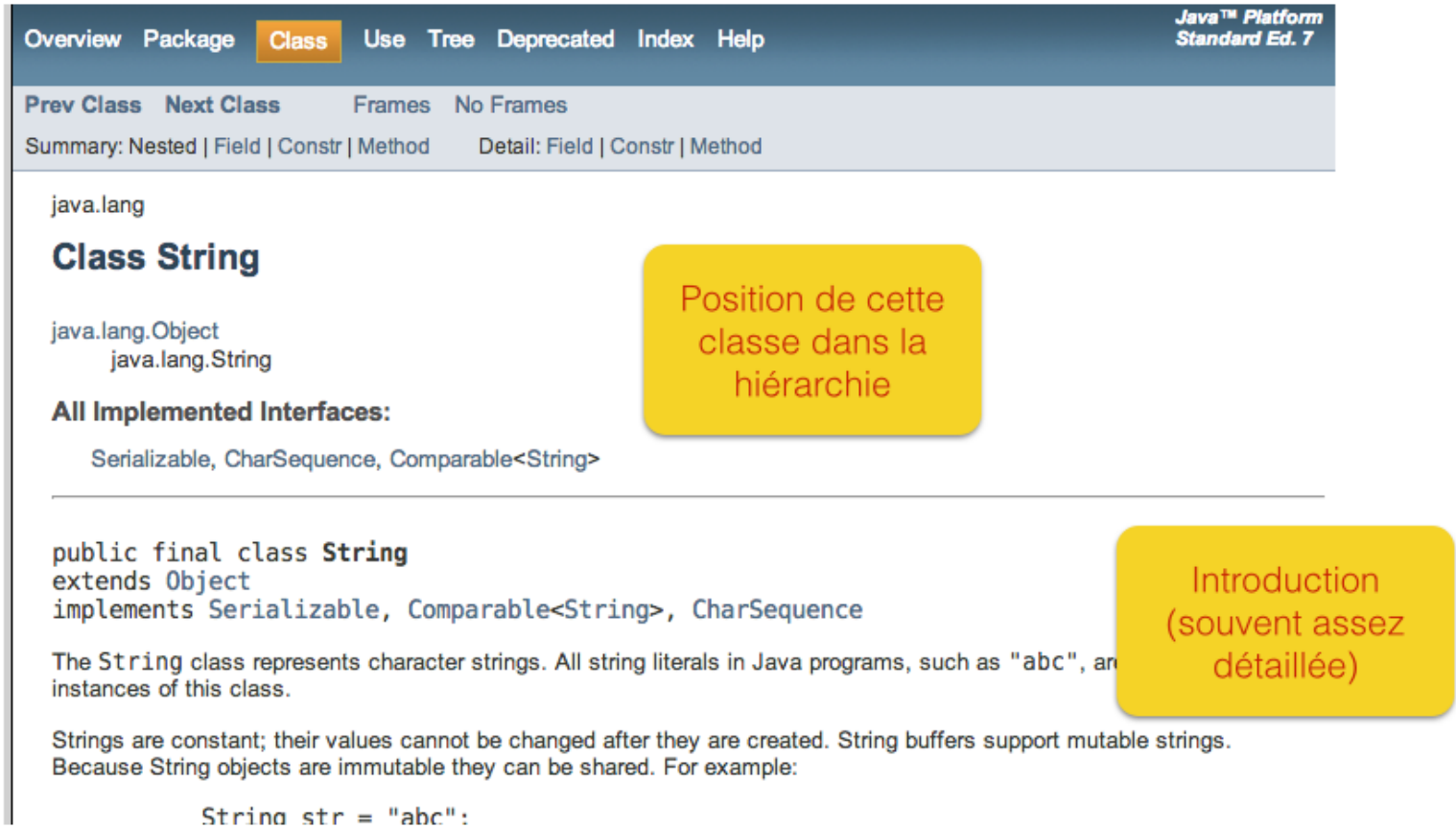
Utiliser une classe venue d'ailleurs . . .

Lire sa documentation ! Elle explique (dans l'idéal) :

- ce que *représente* un objet de la classe ;
- comment *créer* un objet de la classe :
- la liste des méthodes qui permettent de *manipuler* un objet d'une classe et définit son comportement.
- elle souvent écrite dans un format standard : la javadoc.

Lire la javadoc

Exemple: classe String



Overview Package Class Use Tree Deprecated Index Help

Java™ Platform Standard Ed. 7

Prev Class Next Class Frames No Frames

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.lang

Class String

java.lang.Object
java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

Position de cette classe dans la hiérarchie

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

Introduction (souvent assez détaillée)

The String class represents character strings. All string literals in Java programs, such as "abc", ar
instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings.
Because String objects are immutable they can be shared. For example:

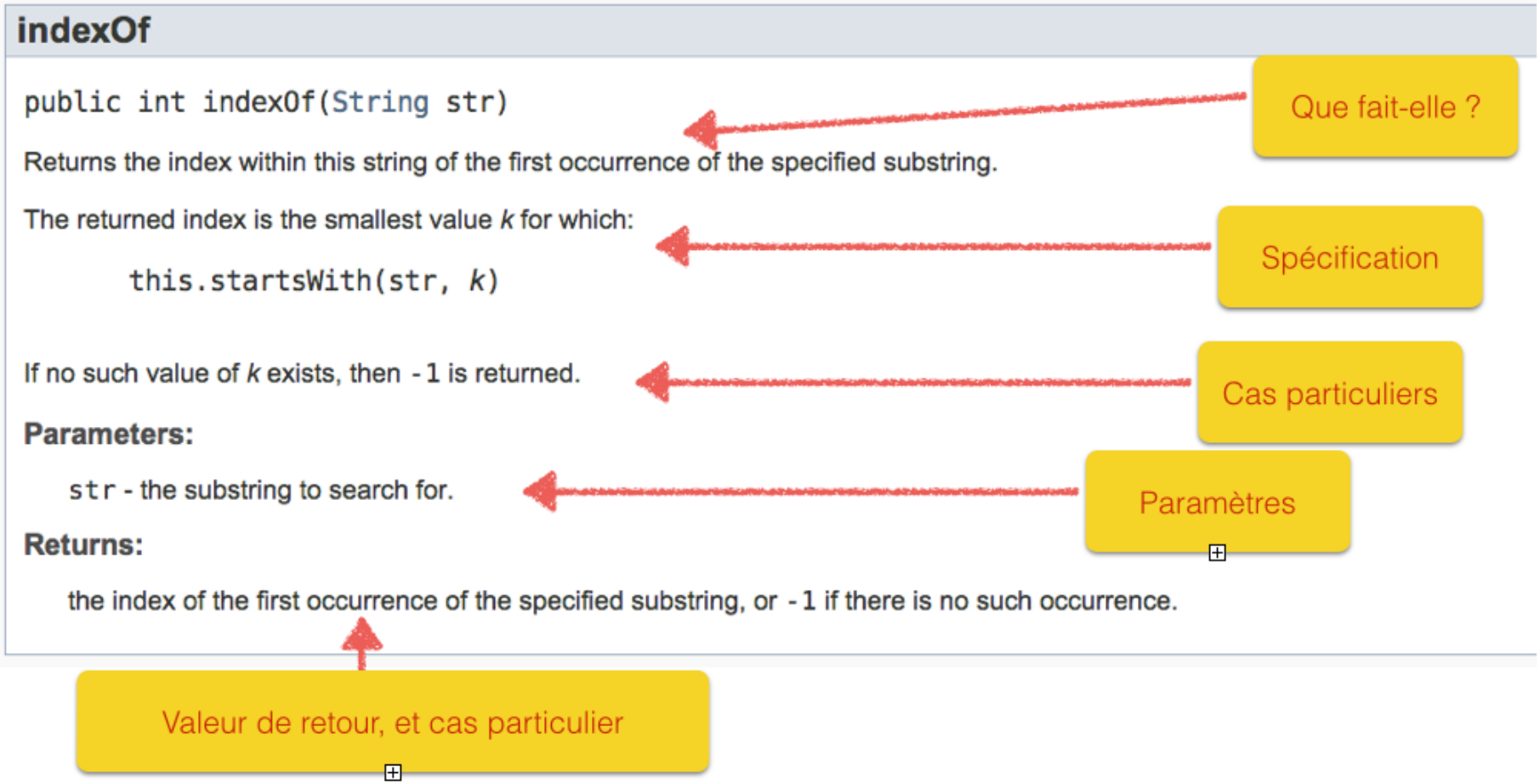
```
String str = "abc";
```

Index des méthodes et des champs

Methods

Modifier and Type	Method and Description
char	**charAt**(int index) Returns the char value at the specified index.
int	**codePointAt**(int index) Returns the character (Unicode code point) at the specified index.
int	**codePointBefore**(int index) Returns the character (Unicode code point) before the specified index.
int	**codePointCount**(int beginIndex, int endIndex) Returns the number of Unicode code points in the specified text range of this String.
int	**compareTo**(String anotherString) Compares two strings lexicographically.
int	**compareToIgnoreCase**(String str) Compares two strings lexicographically, ignoring case differences.
String	**concat**(String str) Concatenates the specified string to the end of this string.
boolean	**contains**(CharSequence s) Returns true if and only if this string contains the specified sequence of char values.
boolean	**contentEquals**(CharSequence cs) Compares this string to the specified CharSequence.
boolean	**contentEquals**(StringBuffer sb)

détail sur une méthode



Documentation d'une méthode

- **Signature de la méthode** : nom de la méthode, type de retour, type des paramètres ;
- courte description de ce que fait la méthode ;
- spécification de la méthode ;
- description des paramètres ;
- description de la valeur retournée ;
- description des exceptions levées par la méthode.

Description d'une classe

La javadoc explique :

- quels sont les *invariants* des objets : quelles propriétés sont vraies sur les objets une fois créés.
- comment *créer un objet* ;
- comment le modifier ;
- comment connaître son état.
- comment utiliser l'objet.

Création et initialisation de variables objet

Les types objet correspondent à des référence :

- référence = pointeur ;
- une variable de type référence déclarée et non initialisée contient l'adresse `null` :

```
String s;     // s contient null
s.length(); // provoque une erreur
```

- si une variable contient `null` :
  - l'objet référencé par la variable n'existe pas ;
  - impossible d'invoquer des méthodes sur cette variable ;
  - invocation sur `null` $\Rightarrow$ erreur exécution

Avant d'utiliser une variable objet

On doit l'initialiser avec un objet qui aura été crée en mémoire.

Affectation entre variables référence

- Possible à condition que les types de ces variables soient compatibles.
  Ex : entre deux tableaux de int, entre deux Strings, etc.

- Quelle est le résultat d'une telle affectation ?

```
String s1, s2, s3;
s1 = "ab";
s2 = "cdef";
s1 = s2;
```

⇒ On copie le contenu d'une variable dans l'autre.
⇒ Ce contenu est **une adresse**.

Création d'objet

Plusieurs possibilités :

- créer l'objet directement avec new et un constructeur ;
- faire créer l'objet par une méthode statique de la classe de l'objet ;
- faire créer l'objet par une méthode d'un autre objet.

Création d'objet

constructeur

Exemple :

```
char[] t = {'a', 'b'};
String s1 = new String();
String s2 =  new String(t);
ArrayList<String> lst = new  ArrayList<String>();
```

- s1 pointe vers un objet contenant la chaîne vide de caractères.
- s2 pointe vers un objet contenant de caractères "ab"
- lst pointe vers un objet contenant la liste vide.

Ces appels n'échouent pas :

```
s1.length(); // renvoie 0
lst.add(s2); // ajout
```

Création d'objet

Méthode retournant un objet

Exemple :

```
String s1 = "ab";
String up = s1.toUpperCase();
```

- ”ab” crée un objet en mémoire (tout comme new).
- s1 fait référence à cet objet.
- up fait référence à un autre objet contenant la chaîne ”AB”

1. Les objets String

Caractéristiques

```
String s1 = "ab";
String s2 ="";
```

- On peut créer des chaînes via des constantes : ”ab” crée un objet en mémoire (tout comme new).
- On peut créer des chaînes vides (taille 0) $\Rightarrow$ s2 contient bien un objet et non pas NULL !
- On ne peut pas mofidier les chaînes crées en mémoire : pas d’ajout ou modification de caractères internes.
- Caractères accessibles par leur indice (partant de 0)

Ce qu'on peut faire

```
String s1 = "ab";
char c = s1.charAt(0);
```

- les créer, affecter, passer en paramètre, envoyer en résultat.
- appliquer des méthodes de la classe String pour :
  - obtenir la taille
  - obtenir le caractère à une position i (en partant de 0)
  - créer de nouvelles chaînes en partant de celle sur laquelle la méthode s'applique ;
  - beaucoup d'autres méthodes

Helper class (ou classes utilitaires)

- classes comportant des méthodes statiques, travaillant souvent sur des instances d'une autre classe ;
- normalement rares dans un programme bien conçu (ça n'est pas de l'objet) ;
- assez utilisées en java, pour des fonctionalités "transverses" :
  - classe `Arrays` : fournit des fonctionalités sur les tableaux ;
  - classe `Collections`, travaille sur les collections en général (voir plus tard) ;
  - classes `Integer, Double, Character...` : ont des instances, mais fournissent aussi des méthodes pour manipuler les types de base.
- à partir de java 8, devraient devenir moins fréquentes (pour les nouvelles classes).

Classes associées au types primitifs

- int, double, char, boolean ne sont pas des classes ;
- où mettre les méthodes qui travaillent sur ces types (exemple : méthode qui dit si un char est une majuscule) ?
- parfois, on a vraiment besoin d'un objet. Par exemple, dans une `ArrayList`, les éléments sont des objets.
- comment faire quand on a vraiment besoin d'un objet, mais que les données sont d'un type primitif ?

Classes associées au types primitifs

- Solution : on asssocie à chaque type primitif une classe.
  - char → Character ; double → Double ; int → Integer ; boolean → Boolean ;
  - ces classes comportent de très nombreuses méthodes utilitaires ;
  - On peut utiliser ces classes pour ≪ encapsuler ≫ des données primitives ; les enrober dans un objet.

    ```
    Character c= new Character('a');
    ```

  - java sait – plus ou moins – le faire automatiquement. On peut écrire : Character c= 'a' ; // autoboxing !

et

```
Integer id= new Integer(3);
int i= id;
```

Les objets `ArrayList<T>`

La classe `ArrayList<T>`

Classe prédéfinie en Java semblable à un tableau :

`ArrayList<T>`

- classe générique de Java :
  - les éléments doivent tous être de même type T ;
  - mais ce type peut être quelconque, mais doit être un type objet ;
  - on doit juste préciser ce type à la déclaration.
- fonctionne comme un tableau de taille *automatiquement* variable
  - on peut y ajouter autant d'éléments qu'on veut,
  - on peut les accéder/modifier selon leur position (en partant de 0).

Déclaration et création de `ArrayList<T>`

- T doit être un type objet (pas de type primitif) ;
- on doit préciser T à la déclaration :
  `ArrayList<Integer>`
- et on doit créer la liste en mémoire

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

- la liste crée ici est vide.
- Important : on doit déclarer en début de fichier,

```
import java.util.ArrayList;
public class ...
```

Quelques méthodes

Ici, `T` désigne un type objet quelconque.

- `int size()` : renvoie la taille de la liste ;
- `boolean isEmpty()` : teste si la liste est vide ;
- `T get(int i)` : renvoie le contenu de la case numéro `i`.
- `add(T el)` : ajoute l'élément `el` à la *fin* de la liste.
- `remove(int i)` supprime l'élément qui est dans la case `i` ;
- `set(int i, T el)` : remplace la valeur dans la case i par `el`. `i` doit être inférieur à la taille de la liste.

Immutabilité

Un objet est immuable s'il est impossible de le modifier après création.

Intérêt :

- sémantique simple : l'objet représente toujours la même *valeur* ;
- on peut utiliser l'objet sans risque. Il ne sera pas modifié sans qu'on en soit prévenu.

Exemples :

- String est immuable ;
- ArrayList est mutable (on peut modifier la liste en ajoutant, enlevant ou modifiant des composantes).

Exemple : caisse enregistreuse

Une caisse enregistreuse

- On peut éditer le ticket de caisse d'un client en y ajoutant des produits ou en les supprimant.
- le ticket est représenté par 2 `ArrayList.` : le 1er contient les noms des produits achetés ; le 2ème, leurs prix.

```
import java.util.ArrayList;

public class Caisse {
 public static void main(String args[]) {
   ArrayList<String> nomsArticles= new ArrayList<String>();
   ArrayList<Double> prixArticles= new ArrayList<Double>();
```

Menu + option 1

```
boolean fin= false;
while (! fin) {
    Terminal.ecrireStringln("votre choix : ");
    Terminal.ecrireStringln("1: ajouter un article");
    Terminal.ecrireStringln("2: supprimer un article");
    Terminal.ecrireStringln("3: terminer et afficher le tot
    int rep= Terminal.lireInt();
    if (rep == 1) { // ajout d'un produit
      Terminal.ecrireString("nom de l'article :");
      String nom= Terminal.lireString();
      Terminal.ecrireString("prix de l'article :");
      double prix= Terminal.lireDouble();
      nomsArticles.add(nom);
      prixArticles.add(prix);
      afficherTicket(nomsArticles, prixArticles);
```

Options 2 et 3

```
  } else if (rep == 2) {
      // suppression du produit dans la case i.
      afficherTicket(nomsArticles, prixArticles);
      Terminal.ecrireString("numéro de l'article à enlever"
      int i= Terminal.lireInt();
      // on supprime le produit dans les deux ArrayList.
      nomsArticles.remove(i);
      prixArticles.remove(i);
      Terminal.ecrireStringln("Ticket après suppression");
      afficherTicket(nomsArticles, prixArticles);
  } else { fin = true;  }
}
```

Solde total

```
  Terminal.ecrireStringln("Ticket␣final");
  afficherTicket(nomsArticles, prixArticles);
  // calcul du prix
  double total= 0;
  for (int i= 0; i < prixArticles.size(); i++) {
    total= total + prixArticles.get(i);
  }
  Terminal.ecrireStringln("Total␣" + total);
}
```

Affichage d'un ticket

```
public static void afficherTicket(ArrayList<String> noms,
  ArrayList<Double> prixArticles) {
  for (int i= 0; i < noms.size(); i++) {
    Terminal.ecrireStringln(i+ ". "+
        noms.get(i)+" prix: "+ prixArticles.get(i));
   }
}
```