

Structure de contrôle & Chaîne de caractères

Rémi Forax

Structure de contrôle

Structure de contrôle en Java

Comme en C, séparation entre

- les expressions, 3, a + 2, 2 * Math.min(x)
- les instructions, if, while, for, etc

Java est rétro-compatible avec le C,
et possède quelque structures supplémentaires
mais pas de **goto**

If, ?:, for, while, do ... while

Les conditions des if, for, while doivent être des booleans

Par convention, on définit toujours un bloc même si il y a une seule instruction

```
if (foo == 4) {  
    System.out.println("ok !")  
}
```

Java est plus pointilleux

Java ne permet pas d'écrire des expressions qui ne font rien

```
var a = 3;  
a + 3; // ne compile pas
```

On ne peut pas non plus écrire du code après un return, break/continue, throw ou yield

```
...  
return 3;  
var a = 3; // ne compile pas
```

Initialisation des variables

Lorsqu'une variable est utilisée, elle doit être initialisée pour tous les chemins possibles

```
void m(int value) {  
    int result;  
    if (value < 10) {  
        result = 3;  
    }  
    return result; // ne compile pas car il y a un chemin  
}                // ou "result" n'est pas initialisée  
                // il faut ajouter un else avec un bloc
```

Exception

Exceptions

Les erreurs sont gérées hors du code principal

- Rend le code plus lisible
- Fournit une *stacktrace* pour le debug
- Rend le code plus efficace
(code assembleur plus compact)

Les exceptions doivent être gérées le plus tard possible (voire jamais)

Instruction throw

throw permet de jeter une exception

Elle remonte les appels de méthode en sens inverse

- jusqu'au main() et arrête la machine virtuelle
- jusqu'à une instruction pour reprendre sur l'erreur

Les exceptions sont des objets, il faut les allouer

```
throw new IllegalArgumentException("oops");
```

Exceptions usuelles

NullPointerException

La valeur est null (le message explique pourquoi)

IllegalArgumentException

La valeur de paramètre n'est pas permise

IllegalStateException

L'état de l'objet (valeur des champs) ne permet pas de faire l'opération

AssertionError

Heu, si je suis ici, il y a un problème, cela ne devrait pas être possible

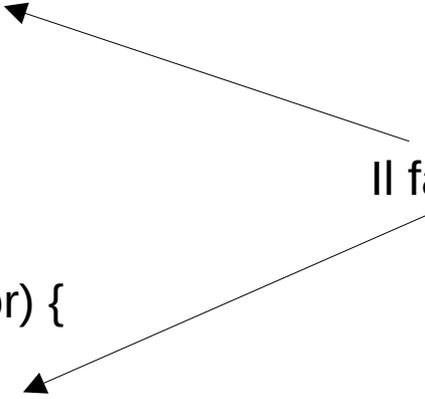
Exemple

On valide les paramètres pour éviter de créer des objets faux

```
public record Bus(String color, int weight) {  
    public Bus { // constructeur compact  
        if (color == null) {  
            throw new NPE("color == null");  
        }  
        if (weight <= 0) {  
            throw new IAE("weight <= 0");  
        }  
    }  
}
```

```
public Bus changeColor(String color) {  
    if (color == null) {  
        throw new NPE("color == null");  
    }  
    return new Bus(color, weight);  
}
```

Il faut faire les checks partout !



Exemple (suite)

```
public class BusTest {  
    public static void main(String[] args) {  
        var bus = new Bus("yellow", -30); // oups  
    }  
}
```

La machine virtuelle s'arrête et affiche la stacktrace

Exception in thread "main" java.lang.IllegalArgumentException:
weight <= 0

at Bus.<init>(Bus.java:7)

at BusTest.main(BusTest.java:3)

type de l'exception

méthode

fichier

numéro de la ligne

java.util.Objects

Contient des méthodes statiques que l'on utilise pour la validation

- Objects.requireNonNull(valeur, text)
- Objects.checkIndex(index, length)

```
import java.util.Objects;
```

```
public class Bus(String color, int weight) {  
    public Bus {  
        Objects.requireNonNull(color, "color is null");  
        if (weight <= 0) {  
            throw new IAE("weight <= 0");  
        }  
    }  
}
```

```
    public Bus changeColor(Color color) {  
        Objects.requireNonNull(color, "color is null");  
        return new Bus(color, weight);  
    }  
}
```

RTFMEException

Les exceptions sont levées à cause

- d'erreurs de programmation

 - dans ce cas, il faut aller lire la javadoc

- d'évènements extérieurs

 - fichier pas présent, erreur de réseau, erreur de disque plein, etc

On apprendra plus tard à gérer ce second type d'erreur

Switch

Le switch du C

L'ancien switch existe toujours par retro-compatibilité

Attention à ne pas oublier les “**break**”s

```
int seats = ...
String type;
switch(seats) {
  case 1:
  case 2:
    type = "small";
    break;
  case 3:
  case 4:
  case 5: {
    System.out.println("debug");
    type = "medium";
    break;
  }
  default:
    type = "big";
}
System.out.println(type);
```

Le switch à flèche

Le switch de Java est mieux que celui du C

On utilise “->” et pas “:”, pas besoin de break, si plusieurs instructions, on doit utiliser un bloc

```
int seats = ...
String type;
switch(seats) {
    case 1, 2 -> type = "small";
    case 3, 4, 5 -> {
        System.out.println("debug");
        type = "medium";
    }
    default - > type = "big"; // obligatoire sinon erreur !
}
System.out.println(type);
```

Le switch expression

Un switch peut aussi envoyer une valeur

Lorsque l'on utilise un bloc, on sort avec **yield**

Le même exemple

```
int seats = ...  
var type = switch(seats) {  
  case 1, 2 -> "small";  
  case 3, 4, 5 -> {  
    System.out.println("debug");  
    yield "medium";  
  }  
  default - > "big"; // obligatoire sinon erreur !  
}; // <- ne m'oublié pas  
System.out.println(type);
```

Les chaînes de caractères

java.lang.String

Les chaînes de caractères en Java sont représentées par la classe String

La classe est **non-mutable**

- On peut envoyer une String en paramètre, on est sûr qu'elle ne sera pas modifiée
- Si on veut changer un des caractères, il faut recréer une String

La javadoc

← → ↻ 🔒 <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/lang/String.html> 📄 ☆ ⬇️ 🛡️ 2 🌐 ☰

OVERVIEW MODULE PACKAGE **CLASS** USE TREE DEPRECATED INDEX HELP Java SE 16 & JDK 16

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

SEARCH:

Module java.base
Package java.lang
Class String
java.lang.Object
 java.lang.String

All Implemented Interfaces:
Serializable, CharSequence, Comparable<String>, Constable, ConstantDesc

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

The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as 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:

Version de la documentation

Zone de recherche

String Littérale

Deux syntaxes

- String simple, entre *quotes*

"je suis une chaine constante"

- *Text block*, entre 3 quotes sur plusieurs lignes

''''''

je suis
une chaine
constante

''''''

Les caractères d'échappement sont les mêmes qu'en C (\n, \r, \t, etc)

Text Block

Chaîne de caractères multiligne

Définie une boîte autour du texte

```
String s = """"  
je suis  
une chaîne  
constante  
""",  
;
```

- Le triple quote au début doit être seul sur sa ligne
- Le triple quote à la fin indique les espaces avant chaque ligne
- Les retours à la ligne sont des `\n` même sous Windows

String littérale et mémoire

Les chaînes de caractères littérales, celles qui commencent par " ou "" sont stockées dans un dictionnaire/cache à l'exécution

```
String s = "hello";  
String s2 = "hello";  
s == s2 // true, même adresse en mémoire
```

Cela ne marche pas avec les autres chaînes de caractères

Représentation

String a une représentation interne différente de la représentation externe

- API (externe) utilise UTF-16, chaque caractère est un char (16 bits)
- Implantation (interne) utilise ISOLatin1 (8 bits) ou UTF-16 (16 bits) avec un flag qui dit quelle représentation est utilisée

Comme on est utilisateur de String et pas développeur, seule la représentation externe nous intéresse

equals/hashCode/toString

String possède les méthodes

- equals(), teste si deux chaînes sont égales en comparant les longueurs puis les caractères
- hashCode(), renvoie un entier “résumé” de la chaîne de caractères
- toString() renvoie this

On teste des Strings avec equals() pas avec ==

Méthodes de `java.lang.String`

Méthodes les plus courantes

`s.length()`

- obtenir la taille

`s.charAt(index)`

- obtenir le caractère à l'index (commence à 0)

`s.repeat(times)`

- répète la même chaîne "times" fois

`s1.compareTo(s2)`

- <0 si s1 plus petit que s2, >0 si s1 plus grand que s2, == 0 si égales

`s.startsWith(prefix), s.endsWith(suffix)`

- est-ce que cela commence par un préfixe, finit par un suffixe

Extraire une sous chaine

`s.indexOf(char)/lastIndexOf(char)`

- index du caractère par le début/la fin ou -1 si pas trouvé

`s.substring(end)`

- équivalent à `s.substring(0, end)`

`s.substring(start, end)`

- extrait la sous-chaine entre start et end (non compris)
 - Contrairement à Python, les valeurs négatives ne sont pas permises

split/join

`s.split(regex)` découpe une chaîne de caractères en un tableau suivant une expression régulière

```
String[] array = "foo,bar".split(",")  
array[0] // foo  
array[1] // bar
```

`String.join(delimiter, elements)` agrège les éléments d'un tableau avec un délimiteur

```
String joined = String.join("-", array);  
joined // foo-bar
```

Majuscule/Minuscule

L'alphabet turk a 2 'i' minuscule et 2 'I' majuscule

```
"IDEA".toLowerCase(Locale.Turkish) // idea
```

```
"idea".equals("idea") // false, le 'i' n'a pas de point au dessus
```

Et mettre en majuscule/minuscule dépend de la Locale
(c'est ce qui dit la norme Unicode)

```
string.toLowerCase() est équivalent à  
string.toLowerCase(Locale.getLocale())
```

cf https://en.wikipedia.org/wiki/Dotted_and_dotless_I

La javadoc de toLowerCase()

toLowerCase

```
public String toLowerCase()
```

Converts all of the characters in this String to lower case using the rules of the default locale. This is equivalent to calling `toLowerCase(Locale.getDefault())`.

Note: This method is locale sensitive, and may produce unexpected results if used for strings that are intended to be interpreted locale independently. Examples are programming language identifiers, protocol keys, and HTML tags. For instance, `"TITLE".toLowerCase()` in a Turkish locale returns `"t\u0131tle"`, where `'\u0131'` is the LATIN SMALL LETTER DOTLESS I character. To obtain correct results for locale insensitive strings, use `toLowerCase(Locale.ROOT)`.

Returns:

the String, converted to lowercase.

See Also:

`toLowerCase(Locale)`

Majuscule / Minuscule

Il ne faut pas oublier d'indiquer `Locale.ROOT`

- `toUpperCase(Locale.ROOT)`
- `toLowerCase(Locale.ROOT)`

Pour comparer des Strings indépendamment de la casse

- `string1.equalsIgnoreCase(string2)`

Conversion String <-> type primitif

Parsing

Convertir une String en un type primitif

Plein de méthodes statiques

- `Boolean.parseBoolean(text)`
 - Convertit une chaîne de caractères en boolean
- `Integer.parseInt(text)`
 - Convertit une chaîne de caractères en int
- `Double.parseDouble(text)`
 - Convertit une chaîne de caractères en double

Attention à ne pas confondre `Boolean` et `boolean`, le premier est une classe contenant les méthodes de conversion, le second est le type primitif

Convertir en String

2 façons

Méthodes statiques

- `Boolean.toString(boolean value)`
- `Integer.toString(int value)`
- `Double.toString(double value)`

Utiliser le '+' avec la chaîne vide

`"" + value`

C'est la façon idiomatique

Switch sur les Strings

Switch sur les Strings

On peut faire un switch sur des String

```
int computeTax(String vehicle) {  
    return switch(vehicle) {  
        case "bicycle" -> 10;  
        case "car", "sedan" -> 20;  
        case "bus" -> 40;  
        default -> throw new IAE("unknown " + vehicle);  
    };  
}
```

le switch est fait sur les valeurs de hashCode()
+ un equals() pour vérifier

Le compilateur génère

```
int computeTax(String vehicle) {  
    var index = switch(vehicle.hashCode()) {  
        case -117759745 -> vehicle.equals("bicycle")? 0: -1;  
        case 98260, -> vehicle.equals("car")? 1: -1;  
        case 109313023 -> vehicle.equals("sedan") ? 1: -1;  
        case 97920 -> vehicle.equals("bus") ? 2: -1;  
        default -> -1;  
    };  
    return switch(index) {  
        case 0 - > 10;  
        case 1 - > 20;  
        case 2 - > 40;  
        default - > throw new IAE(...);  
    };  
}
```

Concaténation

Opérateur +

Si on a une String à gauche ou à droite d'un +, le compilateur fait une concaténation et renvoie la String résultante

```
var s = "hello";  
s + 3 // hello3  
3 + s // 3hello
```

1 concaténation =>

1 allocation d'une nouvelle String

Concaténation et toString()

La concaténation est utilisée fréquemment à l'intérieur des toString()

```
public record Pet(String type, int age) {  
    @Override  
    public String toString() {  
        return type + " " + age;  
    }  
}
```

...

```
var cat = new Pet("flerken", 120);  
cat.toString() // flerken 120
```

Multiple opérateur +

Java est un peu malin, si on a plusieurs + dans une même expression, il fait une seule allocation

Pour

```
return type + " " + age;
```

Algo en deux passes,

- On calcule la taille: `type.length() + 1 + taille(age)`
- On alloue une String de cette taille
- On copie les contenus

Opérateur + et boucle

Java n'est pas malin si il y a des boucles

```
String join(String[] array) {  
    var result = "";  
    for(var s: array) {  
        result = result + ":" + s; // alloue 1 nouvelle String  
    }  
    return result;  
}
```

Allocation à chaque tour de boucle (ahhhh)

- Et l'algo est faux ...

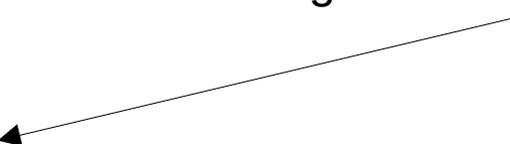
java.lang.StringBuilder

Buffer extensible de caractères

Évite d'avoir trop d'allocations de String intermédiaires

```
String join(String[] array) {  
    var builder = new StringBuilder();  
    for(var s: array) {  
        builder.append(s);  
        builder.append(":");  
    }  
    return builder.toString();  
}
```

demande le contenu d'un
StringBuilder sous forme de String



Algo est toujours faux, mais différemment

Chainage des append()

Les méthodes `StringBuilder.append()` renvoient pas `void` mais un `StringBuilder` (en fait, lui-même)

```
public class StringBuilder {  
    public StringBuilder append(String s) {  
        ...  
        return this;  
    }  
    // pareil pour les autres méthodes append  
}
```

On peut écrire `builder.append(...).append(...)`

Chainage des appends

En utilisant le chainage des appends

```
String join(String[] array) {  
    var builder = new StringBuilder();  
    for(var s: array) {  
        builder.append(s).append(":");  
    }  
    return builder.toString();  
}
```

Algo toujours faux ...

Pas de + dans les append()

Si on fait un + dans les append(), on ajoute une allocation dans le append()

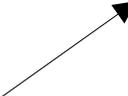
```
String join(String[] array) {  
    var builder = new StringBuilder();  
    for(var s: array) {  
        builder.append(s + ":"); // ahhhhh  
    }  
    return builder.toString();  
}
```

Et l'algo ?

Il faut insérer le séparateur (le “:”) sauf si c’est le premier tour de boucle

```
String join(String[] array) {  
    var builder = new StringBuilder();  
    for(var i = 0; i < array.length; i++) {  
        if (i != 0) {  
            builder.append(":");  
        }  
        builder.append(s);  
    }  
    return builder.toString();  
}
```

Il existe une solution sans “if”
dans la boucle



Mais le code n’ est pas très beau :(

Algo plus joli

Ne pas faire de test en déclarant une variable correspondant au séparateur

```
String join(String[] array) {  
    var builder = new StringBuilder();  
    var separator = "";  
    for(var s: array) {  
        builder.append(separator).append(s);  
        separator = ":";  
    }  
    return builder.toString();  
}
```

Formatting

Utiliser le format de printf

Attention, comparativement à la concaténation, ces méthodes sont super lentes (comme printf en C)

L'équivalent de sprintf

avec une méthode statique

- `String.format(String format, Object... objects)`

avec une méthode d'instance

- `format.formatted(Object... objects)`

Un `PrintStream` a aussi une méthode `printf()`

```
System.out.printf("hello %s\n", name);
```

Expression régulière

Some people, when confronted with a problem, think:
“I know, I’ll use regular expressions.” Now they have two problems.
— Jamie Zawinski

Pattern et Matcher

`java.util.regex.Pattern` représente un automate créé à partir d'une expression régulière

```
Pattern.compile("a+")
```

`java.util.regex.Matcher` parcourt l'automate

créé sur un texte: `pattern.matcher("aaaaa")`

3 sémantiques

- `matcher.matches()` // reconnaît tout le texte
- `matcher.looksAt()` // reconnaît le début du texte
- `matcher.find()` // reconnaît une partie du texte

Composer des regex

Plusieurs sortes qui sont combinables

- mot: hello
- répétition: +, *, ?
- conjonction/disjonction: a+b+, foo|bar
- caractères: [abc], [a-z]
- classes: \d, \p{Digit}, \p{Alpha}
- capture: (a+)

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/regex/Pattern.html>

Examples

```
var pattern = Pattern.compile("a+");  
pattern.matcher("aaa").matches() // true
```

```
var pattern = Pattern.compile("[0-9]+\\. [0-9]*");  
pattern.matcher("12.5").matches() // true
```

```
var pattern = Pattern.compile("\\d+\\. \\d*");  
pattern.matcher("24.").matches() // true
```

```
var pattern = Pattern.compile("-?[0-9]+");  
pattern.matcher("-14").matches() // true
```

```
var pattern = Pattern.compile("[A-Za-z][A-Za-z_0-9]*");  
pattern.matcher("old_regex3").matches() // true
```

```
var pattern = Pattern.compile("\\p{Alpha}\\w*");  
pattern.matcher("old_regex3").matches() // true
```

matches() vs lookingAt()

matches() reconnaît le texte complet,
lookingAt() regarde que le début du texte

```
var pattern = Pattern.compile("a+");  
pattern.matcher("aaa").matches() // true
```

```
var pattern = Pattern.compile("a+");  
pattern.matcher("aaab").matches() // false
```

```
var pattern = Pattern.compile("a+");  
pattern.matcher("aaab").lookingAt() // true
```

find()

matcher.start()/end()/group() permet d'extraire les valeurs

```
var text = """"  
The quick brown fox jumps over the lazy dog  
""",  
var pattern = Pattern.compile("the|fox", Pattern.CASE_INSENSITIVE);  
var matcher = pattern.matcher(text);  
while(matcher.find()) {  
    var start = matcher.start();  
    var end = matcher.end();  
    var group = matcher.group();  
    System.out.println(start + " " + end + " " + group);  
} // 0 3 The  
  // 16 19 fox  
  // 31 34 the
```

Groupe de capture

Les parenthèses permettent d'indiquer des parties de regex que l'on veut extraire (Attention: les groupes commencent à 1)

```
var pattern = Pattern.compile("(\\d+)/ (\\d+)/ (\\d+)");
var matcher = pattern.matcher("2015/7/23");
if (matcher.matches()) {
    for(var i = 1; i <= matcher.groupCount(); i++) {
        var start = matcher.start(i);
        var end = matcher.end(i);
        var group = matcher.group(i);
        System.out.println(start + " " + end + " " + group);
    }
} // 0 4 2015
// 5 6 7
// 7 9 23
```

En résumé ...

En résumé

On doit initialiser les variables locales pour tous les chemins

Java a un switch avec des flèches

On lève une exception en cas d'erreur avec
throw new NPE, IAE, ISE ou AssertionError

""

hello

"" et "hello" sont des chaînes constantes

On doit comparer les Strings avec equals()

On utilise pas le + des Strings dans les boucles