|  |  |
| --- | --- |
| |  | | --- | |  | |
| C34-VM Programmation Orientée Objet 1  **MODULE 03**  **Surcharge - constructeurs – String**  Automne 2023 |

Table des matières

[1. La surcharge de méthodes 3](#_Toc147250288)

[1.1 Comment choisi la bonne méthode 3](#_Toc147250289)

[1.2 La signature d’une méthode 4](#_Toc147250290)

[L03A\_Surcharge de méthodes (overloading) 4](#_Toc147250291)

[2. Les constructeurs 5](#_Toc147250292)

[2.1 Qu’est-ce qu’un constructeur 5](#_Toc147250293)

[2.2 Constructeur par défaut 6](#_Toc147250294)

[2.3 Utilité du constructeur 6](#_Toc147250295)

[2.4 Les paramètres des constructeurs 7](#_Toc147250296)

[2.5 Surcharge de constructeurs 8](#_Toc147250297)

[L03B\_Constructeur - Gym 8](#_Toc147250298)

[Retours sur le L03B 8](#_Toc147250299)

[L03C\_Compte bancaire 8](#_Toc147250300)

[3. La classe String 9](#_Toc147250301)

[Méthode length() 9](#_Toc147250302)

[Méthode charAt() 9](#_Toc147250303)

[Méthode indexOf() 9](#_Toc147250304)

[Méthode lastIndexOf() 9](#_Toc147250305)

[Méthode isEmpty() 10](#_Toc147250306)

[Méthodes toUpperCase() et toLowerCase 10](#_Toc147250307)

[Autres méthodes 10](#_Toc147250308)

[Mini-Exercice 11](#_Toc147250309)

[Démo StringFrame 11](#_Toc147250310)

[4. Les expressions régulières 12](#_Toc147250311)

[L03D\_Exercices sur les expressions régulières 14](#_Toc147250312)

[5. Les classes Pattern et Matcher 15](#_Toc147250313)

[5.1 Autres méthodes pour travailler avec une chaîne de caractères : 16](#_Toc147250314)

[6. Transtypage entre types prédéfinis et String 16](#_Toc147250315)

[6.1 Méthode toString() 16](#_Toc147250316)

[6.2 Méthode valueOf() de String 16](#_Toc147250317)

[6.3 Les classes StringBuffer et StringBuilder 17](#_Toc147250318)

[6.4 Conversions automatiques 17](#_Toc147250319)

[L03E\_Programme avec les expressions régulières 17](#_Toc147250320)

# La surcharge de méthodes

Java autorise l’utilisation de méthodes qui portent le même nom, mais effectuent des opérations différentes.

Exemple : deux appels d’une méthode **retirer**, du même objet, produisent un résultat différent.

|  |
| --- |
|  |
|  |

Ce comportement s’appelle **Surcharge** de méthodes.

C’est une forme de **polymorphisme**.

## Comment choisi la bonne méthode

Pour distinguer les 2 méthodes entres elles :

* Les **paramètres** entre les 2 méthodes doivent être **différents** en **type** et/ou en **quantité**.

Exemple : les paramètre de la fonction **Retirer** ne sont pas du même type :

|  |
| --- |
|  |

Note : une différence de type d’un seul paramètre aurait suffi pour permettre la surcharge.

## La signature d’une méthode

On analyse la syntaxe d’une méthode à partir d’un exemple :

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| public | String | retirer | (int n1, int n2) | { } |  |
| A | B | C | D | E |  |

Où :

A : modificateur d’accès

B : type du résultat retourné par la méthode, void si pas de retours

C : nom de la méthode DOIT COMMENCER PAR UNE MINUSCULE

D : Liste de paramètres, entre parenthèse, séparés par des virgules, précédés de leur type, si pas de paramètres, parenthèses vides requises

E : bloc de code compris entre accolades

On appelle la **signature** d’une méthode une expression qui identifie le nom de la méthode, le nombre de paramètres ainsi que le type des paramètres, dans une structure *nomDeMethode(type1, type2…)*.

Notez que la signature n’inclue ni le type du retour, ni le nom des paramètres ni le bloc de code de la méthode.

Dans l’exemple précédent, les **signatures** des méthodes seraient :

* retirer(int, int)
* retirer(string, string)

En Java, pour permettre la **surcharge** :

* Les méthodes DOIVENT porter le même nom
* Les méthodes DOIVENT avoir une signature différente
* Les méthodes DOIVENT retourner le même type

### L03A\_Surcharge de méthodes (overloading)

Le laboratoire L03A permet de pratiquer la surcharge de méthodes.

# Les constructeurs

## Qu’est-ce qu’un constructeur

Lorsqu’on créé une instance d’une classe avec **new**, une méthode spéciale est utilisée.

NomDeLaClasse nomDObjet = **new** NomDeLaClasse();

On reconnait que c’est une méthode à l’utilisation de parenthèses - NomDeLaClasse();

Cette méthode :

* S’appelle un constructeur
* DOIT porter le même nom que la classe
* Peut avoir ou non des paramètres
* Ne peut pas avoir de valeur de retour

**Exemple - utilisation d’un constructeur**

|  |
| --- |
| Produit obj = new Produit(); |

**Exemple – définition d’un constructeur**

|  |
| --- |
| public class Produit {  String code;  String nom;  int qteVendue;   public Produit(){ *// Même nom que la classe, pas de valeur de retour* }  } |

## Constructeur par défaut

Jusqu’à maintenant, on n’a pas défini de constructeurs dans nos classes, mais on créé quand même des objets avec le mot clé **new**.

Si on ne définit pas de constructeur, le compilateur en ajoute/créé un automatiquement.

Ce constructeur, définit par le système, n’a pas de paramètres et s’appelle **constructeur par défaut**.

## Utilité du constructeur

On utilise le constructeur pour accomplir des tâches dès la création d’un objet.

**Exemple : donner une valeur à un champs dès la création de l’objet.**

|  |
| --- |
| public class Produit {  String code;  String nom;  int qteVendue;   public Produit(){ *// Même nom que la classe, pas de valeur de retour* this.qteVendue = 0; *// qteVendue initalisé à 0 à la création de l'objet* } } |

## Les paramètres des constructeurs

En tant que méthode, un constructeur peut recevoir des paramètres. Ces paramètres s’utilisent comme avec les autres méthodes.

**Exemple : création d’un objet avec assignation de valeurs aux champs**

|  |
| --- |
| public class Produit {  String code;  String nom;  int qteVendue;   public Produit(String code, String nom){  this.qteVendue = 0; *// qteVendue initalisé à 0* this.code = code; *// valeur de code fournie en paramètre* this.nom = nom; *// valeur de nom fournie en paramètres   // On peut aussi faire d'autres opérations dans le constructeur  // si ces opérations sont pertinentes à la création de l'objet* } } |
| Produit objProduit = new Produit("LJ888", "LaserJet88"); |

Note : on peut faire en une étape ce qu’on devait en 2 étapes ou plus, avant d’utiliser les constructeurs.

## Surcharge de constructeurs

Les constructeurs supportent la surcharge.

Les mêmes règles que pour les autres méthodes s’appliquent.

|  |
| --- |
| public class Produit {  String code;  String nom;  int qteVendue;   public Produit(String code, String nom){  this.qteVendue = 0; *// qteVendue initalisé à 0* this.code = code;  this.nom = nom;  }  public Produit(String code, String nom, int qteVendue){  *// Toutes les valeurs de* this.qteVendue = qteVendue; *// qteVendue reçue en paramètres* this.code = code;  this.nom = nom;  } } |
| Produit objProduit1 = new Produit("LJ888", "LaserJet88"); Produit objProduit2 = new Produit("LJ888", "LaserJet88", 4); |

Note :

* Pour utiliser la surcharge de constructeur ET utilisez le constructeur par défaut sans paramètres, il doit être défini explicitement.
* Entre autres pour cette raison, les meilleures pratiques suggèrent de toujours définir explicitement au moins un constructeur.

### L03B\_Constructeur - Gym

Le laboratoire L03B permet de pratiquer la création de constructeurs dans une classe.

### Retours sur le L03B

### L03C\_Compte bancaire

Le laboratoire L03C permet de pratiquer la modélisation d’une classe avec des champs des constructeurs et des méthodes.

# La classe String

Le traitement de texte est une action fréquente en programmation.

Le langage Java répond à ce besoin avec la classe **String**, une classe de chaine de caractère qui a des comportements ressemblants à des types primitifs, mais qui offre plusieurs méthodes prédéfinies.

Un objet String est en réalité un tableau (array) de caractères.

Chaque caractère à une position dans le tableau, en commençant par 0.

Cette position s’appelle **l’index** du caractère.

### Méthode length()

Trouve le nombre de caractères dans un objet String.

|  |
| --- |
| String phrase = "Voici une c  haine de texte de plusieurs caractères";  int nombreCar = phrase.length(); |

### Méthode charAt()

Trouve le caractère à une position donnée (un index donné).

|  |
| --- |
| String phrase = "Voici une chaine de texte de plusieurs caractères";  char troisiemeCaractere = phrase.charAt(2);  //3ième caractère => index 2 pcq commence à 0, c’est un i |

### Méthode indexOf()

Par défaut cherche la première occurrence à partir du début (index 0).

|  |
| --- |
| String phrase = "Voici une chaine de texte de plusieurs caractères";  int positionDeC = phrase.indexOf('c');  //Position du premier c de la phrase est 3 |

### Méthode lastIndexOf()

Trouve l’index de la dernière occurrence d’un caractère dans la chaine.

|  |
| --- |
| String phrase = "Voici une chaine de texte de plusieurs caractères";  int positionDeC = phrase.lastIndexOf('c');  //Position du dernier c de la phrase est 43 |

### Méthode isEmpty()

Retourne **true** (vrai) si la chaine de caractère est vide et **false** (faux) il elle n’est pas vide.

Chaine de caractère vide : ""

Pas une chaine de caractère vide : " "

|  |
| --- |
| String phrase = "Voici une chaine de texte de plusieurs caractères";  bool phraseEstVide = phrase.isEmpty(); |

Ou souvent :

|  |
| --- |
| **String phrase = "";   if (phrase.isEmpty()) {  System.*out*.println("Champ vide, svp entrez du texte");  }  else {  *//Traitement du texte ici* }** |

### Méthodes toUpperCase() et toLowerCase

Modifie une chaine de caractères pour que tous les caractères soient minuscules (LowerCase) ou majuscule (UpperCase).

Ces méthodes sont un excellent exemple d’une modification qu’on ne pourrait pas faire avec un type primitif : on modifie directement le contenu de l’objet, ci qui n’est pas permis avec un type primitif.

|  |
| --- |
| **String phrase = "Voici une chaine de texte de plusieurs caractères";**  **System.*out*.println(phrase.toUpperCase()); System.*out*.println(phrase.toLowerCase());** |

### Autres méthodes

Plusieurs autres méthodes sont disponibles pour la classe String.

### Mini-Exercice

1. Dans IntelliJ IDEA, dans n’importe quel projet, créer un objet de type String et utilisez l’autocomplétion pour trouver :

|  |  |
| --- | --- |
| La méthode de String qui permet de comparer une chaine avec une autre chaine | String. equals() |
| La méthode de String qui permet de concaténer 1 chaine à une autre chaine. | String.concat() |

1. Cherchez en ligne et trouvez :

|  |  |
| --- | --- |
| La méthode de String qui permet de remplacer toutes les occurrences d’un caractère par un autre caractère dans une chaine | String.replace() |
| La méthode de String qui permet de séparer une chaine de caractères en plusieurs chaines, basé sur un caractère séparateur | String.split() |

### Démo StringFrame

Démonstration des méthodes de String utilisées dans un programme d’analyse de texte.

# Les expressions régulières

Une expression régulière (regex) est une séquence de caractères qui décrit un motif/modèle de caractères à rechercher dans une chaine ou comparer avec une chaine.

**Par exemple** :

Soit **String phrase = "Voici une chaine de texte de plusieurs caractères";**

alors **phrase.contains("ex")**

retournera **true** parce que cette suite de caractères est présente dans la chaine.

Ici on a utilisé la méthode **contains** avec une suite de caractères spécifiques.

L’expression régulière (le motif) ici est la lettre **e** suivi de la lettre **x**.

Les expressions régulières permettent de créer des modèles plus complexes.

**Par exemple** :

Dans une expression régulière, des caractères entre crochets **[ ]** signifient **“n’importe quel des caractères mentionnés”.**

Alors dans l’exemple ci-dessous :

**unChaine.equals("saut[ea]");**

La comparaison sera vraie pour **“saute”** ou **“sauta”** mais pas pour aucune autre combinaison.

Une des applications les plus répandues des expressions régulières est la validation de champs texte qui doivent respecter un format précis :

* Un code postal (au canada) doit être de format **lettre**-**chiffre**-**lettre**-**espace**-**chiffre**-**lettre**-**chiffre**
* Un numéro de téléphone doit avoir 10 caractères et seulement des chiffres
* Etc.

Java inclus les classes du package **java.util.regex** pour fournir des méthodes pour les expressions régulières.

Utiliser : **import java.util.regex.\***

Voici quelques-uns des modèles utilisables dans les expressions régulières en Java.

|  |  |
| --- | --- |
| **Modèles ( pour un caractère ou un nombre )** | |
| [abc] | a, b, ou c |
| [^abc] | Tous les caractères sauf a, b, ou c (négation) |
| [a-zA-Z] | a à z, ou A à Z, ( ensemble ) |
| [a-d[m-p]] | a à d, ou m à p: [a-dm-p] (union) |
| [a-z&&[def]] | d, e, ou f (intersection)  "d", "e" et "f" sont les seuls caractères qui satisfont cette expression régulière. |
| [a-z&&[^bc]] | a à z, sauf b et c: [ad-z] (soustraction) |
| [a-z&&[^m-p]] | a à z, mais pas m à p: [a-lq-z] (soustraction) |

|  |  |
| --- | --- |
| **Modèles généraux ( pour un caractère )** | |
| . | Tout caractère |
| \d | Un chiffre: [0-9] |
| \D | Tout sauf un chiffre: [^0-9] |
| \s | Un caractère blanc: [ \t\n\x0B\f\r] |
| \S | Tout sauf un caractère blanc: [^\s] |
| \w | Un caractère ( lettre ou chiffre ): [a-zA-Z\_0-9] |
| \W | Tout sauf un caractère ( lettre ou chiffre ): [^\w] |

|  |  |
| --- | --- |
| **Multiplicateurs** | |
| X? | X : 0 ou 1 fois seulement |
| X\* | X: 0 ou plusieurs fois |
| X+ | X: 1 ou plusieurs fois |
| X{n} | X: exactement n fois |
| X{n,} | X: au moins n fois |
| X{n,m} | X: au moins n fois et au plus m fois |

L'utilisation de ^ et $ garantit que la correspondance doit couvrir toute la chaîne, en s'assurant qu'il n'y a pas de caractères supplémentaires avant ou après le motif spécifié. En d'autres termes, avec ^ et $, vous indiquez que la chaîne doit correspondre entièrement au modèle et ne peut pas simplement contenir une partie du modèle au milieu d'autres caractères. C'est couramment utilisé pour définir des règles strictes pour le format des chaînes.

**\*\*\* attention en Java, \ est un caractère d’échappement. Pour utiliser les modèles de caractères, on doit donc utiliser \\.**

On constitue donc une String composée des différents symboles ci-dessus (l’expression régulière) et on compare la chaîne à vérifier.

La méthode **matches** du type string est une autre méthode qui permet de valider une expression régulière.

EX.

String modele = "..\\d" ;

System.out.println ("nana".matches(modele)); 🡪 false

String modele = "..\\d" ;

System.out.println ("na9".matches(modele)); 🡪 true

NOTES:

* La méthode **matches** de la classe String permet de comparer une chaine avec une expression régulière.
* Cet exemple illustre bien que l’expression régulière est une chaine de caractères en soi.

### L03D\_Exercices sur les expressions régulières

Le laboratoire L03D permet de pratiquer l’utilisation des expressions régulières.

# Les classes Pattern et Matcher

Les classes **Pattern** et **Matcher** sont 2 classes du package java.util.regex qui servent à manipuler des expressions régulières.

La méthode **matches** de la classe String imite en tous points celle de la classe Pattern. En effet, on peut également créer des expressions régulières en créant un objet Pattern. Cela permet d’accéder à de nouvelles méthodes…

**Les principales classes du package javax.util.regex :**

* Pattern :
* La classe Pattern représente une version compilée d’une expression régulière
* Elle n’a pas de constructeur; on crée un objet Pattern à l’aide de la méthode statique compile :

Pattern p = Pattern.compile ( ”\\d{5}”);

* Matcher :
* La classe Matcher permet d’obtenir un objet qui interprétera le Pattern (l’expression régulière) et d’y appliquer des méthodes
* Comme Pattern, on ne peut pas créer un objet Matcher avec un constructeur, on doit utiliser la méthode matcher de la classe Pattern :

Matcher m = p.matcher(“12345666633”);

* méthodes à utiliser sur le Matcher : matches, find, reset

m.matches() 🡪 faux car 12345666633 ne correspond pas à 5 chiffres

m.find() 🡪 vrai car tu peux trouver 5 chiffres dans 12345666633

m.find() 🡪vrai car tu peux trouver 5 chiffres à la suite de l’autre 5 chiffres

m.find() 🡪false car il ne reste qu’un chiffre

m.reset() 🡪 retourne au début

* PatternSyntaxException
* Lancée lorsque la syntaxe de l’expression régulière n’est pas correcte

## Autres méthodes pour travailler avec une chaîne de caractères :

* méthode split de la classe String :

String[] tab= "asdf4pa4osi".split("\\d");

for ( int i = 0; i < tab.length; i++ )

System.out.println(tab[i]);

* classe Scanner ( package java.util )

* permet de « scanner » tout un fichier plutôt que seulement une String
* permet de retourner tout type prédéfini plutôt que seulement des Strings
* délimiteur par défaut : un caractère blanc ( espace, \r, \n, d'autres ) mais on peut utiliser une expression régulière à la place

# Transtypage entre types prédéfinis et String

Java est un langage fortement typé : obligatoire d’identifier le type d’une variable.

Pour convertir un type à un autre, on peut utiliser les méthodes de certaines classes, mais Java effectuera automatiquement certaine conversion.

Voici quelques-unes des façons de convertir des types :

## Méthode toString()

Plusieurs méthodes mettent en place la méthode toString() qui convertit une valeur numérique en String.

Integer nombre = 5;

String texte = nombre.toString();

## Méthode valueOf() de String

|  |
| --- |
| Integer nombre = 5;  String texte = String.valueOf(nombre); |

## Les classes StringBuffer et StringBuilder

Ces classes offrent une série de méthodes conçues spécifiquement pour construire des chaines de caractères.

## Conversions automatiques

Dans certains cas, Java convertira automatiquement le type.

|  |
| --- |
| String texte = "Expo" + 69;  Produira la chaine de texte “"Expo69"; |

### L03E\_Programme avec les expressions régulières

Le laboratoire L03E permet de pratiquer l’utilisation des expressions régulières dans un programme.