Chapitre 5

Type de données Java et notion d'importation de bibliothèque

1. Notion d’importation de bibliothèques
2. Les types de données simples
3. Les types de données de référence
4. Un mot sur le passage de paramètres dans les méthodes

1 Notion d’importation de bibliothèques

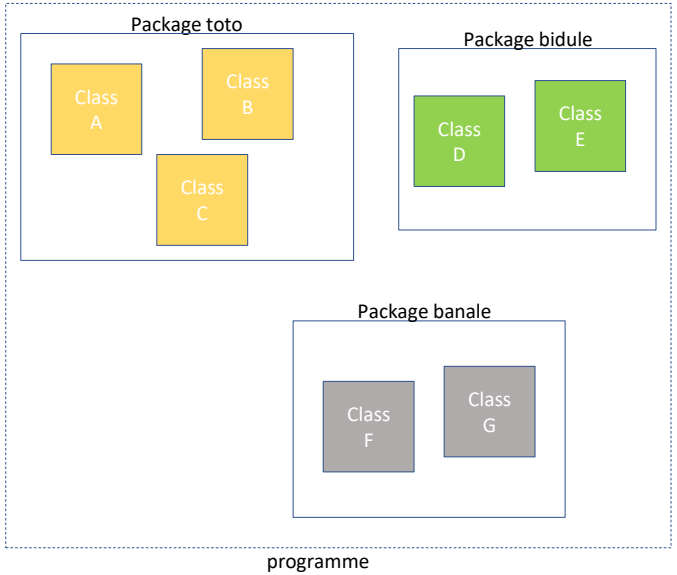
2 Les types de données simples

3 Les types de données de référence

4 Un mot sur le passage de paramètres dans les méthodes

Notion d'importation de bibliothèques

Définition



L'importation de bibliothèques est une conséquence de l'organisation d'un projet/programme en package**s**

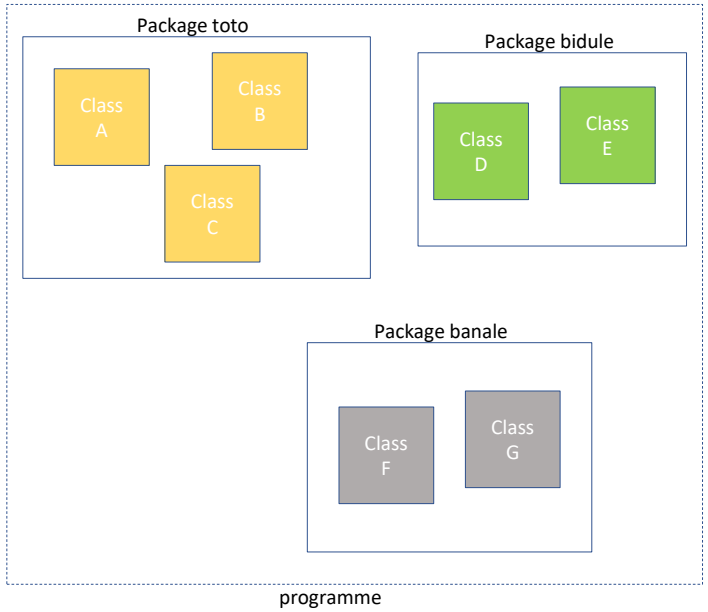
- Les classes regroupées dans un package, et qui fournissent des fonctionnalités utiles pour le reste du programme sont considérées comme une **bibliothèque** de fonctionnalités
  (package ⇔ Bibliothèque de classes)

- Toute classe **E** située dans un package bidule et qui a besoin d'utiliser une classe **A** logée dans un autre package toto, doit pouvoir l'**importer** pour la rendre visible et utilisable

- Java utilise le mot clé **import** pour répondre à ce besoin

Notion d'importation de bibliothèques

Exemple



- Importation ciblée :

```
package bidule;

import toto.A ;

public class E {

    A a = new A();
    //attributs
    //méthodes
}
```

Dans l'instruction import, on suffixe le nom de la bibliothèque avec la classe précise que l'on souhaite importer.
Ex : import toto.**A**
Seule la classe A de *toto* peut être utiliser dans E

- Importation exhaustive :

```
package bidule;

import toto.* ;

public class E {

    A a = new A();
    C c = new C();
    //attributs
    //méthodes
}
```

Dans l'instruction import, on suffixe le nom de la bibliothèque par une * (étoile)
Ex : import toto.*
Ce mode d'importation rend disponible toutes les classes de la bibliothèque.
On peut donc utiliser dans E, toutes les classes de *toto* : A, B et C

Notion d'importation de bibliothèques

Quelques bibliothèques Java

Java fournit plusieurs bibliothèques prêtes à l'emploi que nous serons amenés à importer dans nos programmes et donc voici quelques unes :

Bibliothèques / Packages&	Description
java.lang	Bibliothèque de base contenant les classes et fonctionnalités permettant d'écrire un programme simple avec la gestion des types de données. Java l'importe implicitement dans toute classe
java.util	Bibliothèque utilitaire contenant les classes et fonctionnalités permettant de manipuler les Collections, les streams, les traitements multithreads, etc
java.math	Contient les classes et fonctionnalités pour faire des calculs mathématiques : arrondis, racine carré, valeur absolue, génération de nombre aléatoire, etc
java.time	Contient les classes et fonctionnalités permettant de manipuler les dates et heures
java.io	Contient les classes et fonctionnalités pour manipuler les entrées/sorties (interaction entre un programme et un fichier/clavier), bref la gestion de flux de données
javax.swing	Contient des classes et fonctionnalités pour faire du graphisme
java.sql	Contient les classes et fonctionnalités pour la gestion des bases de données

1 Notion d’importation de bibliothèques

2 Les types de données simples

3 Les types de données de référence

4 Un mot sur le passage de paramètres dans les méthodes

Les types de données simples (1/5)

Types primitifs

Type	Description	Plage de valeurs	Taille en mémoire	Exemple de déclaration
boolean	Type binaire (0 ou 1)	true ou false	1 bit	boolean b = true
byte	Entier de très petite taille	-128 à 127 (-2^{7} à $2^{7}-1$)	8 bits (1 octet)	byte b = 72
short	Entier de taille moyenne	-32768 à 32767 (-2^{15} à $2^{15}-1$)	16 bits (2 octets)	short s = -12078
int	Entier	-2147483648 à 214483647 (-2^{31} à $2^{31}-1$)	32 bits (4 octets)	int i = 4038973
long	Entier de grande taille	-2^{63} à $2^{63}-1$	64 bits (8 octets)	long l = 12358769543
char	Caractère	a….z (\u0000 à \uffff)	16 bits (2 octets)	char c = 'e'
float	Nombre réel de petite taille	-2^{-149} à $(2-2^{-23})\cdot 2^{127}$	32 bits (4 octets)	float f = 12.583f
double	Nombre réel de grande taille	-2^{-1074} à $(2-2^{-52})\cdot 2^{1023}$	64 bits (8 octets)	double d = 986.1234587937d

Les types de données simples (2/5)

Les wrappers

Wrapper	Description	Type primitif associé	Exemple
Boolean	Classe enveloppe associé au type boolean	boolean	Boolean b = new Boolean(true)
Byte	Classe enveloppe associé au type byte	byte	Byte b = new Byte(72)
Short	Classe enveloppe associé au type short	short	Short s = new Short(-12078)
Integer	Classe enveloppe associé au type int	int	Integer i = new Integer(4038973)
Long	Classe enveloppe associé au type long	long	Long l = new Long(12358769543)
Character	Classe enveloppe associé au type char	char	Character c = new Character('e')
Float	Classe enveloppe associé au type float	float	Float f = new Float(12.583f)
Double	Classe enveloppe associé au type double	double	Double d = Double(986.1234587937d)

Les types de données simples (3/5)

Wrappers : explication

Wrappers ?

Les types primitifs ne sont pas des objets. Pour rester cohérent avec le principe stipulant que « le plus petit élément d'un programme doit être un **objet** », Java a défini dans la bibliothèque *java.lang* des *classes pour envelopper* chaque type de données primitif. Ces classes sont appelées Wrappers. C'est avec ces wrappers qu'il travaillera implicitement lorsqu'on déclare des types primitifs

On peut aussi les utiliser explicitement dans nos classes

Exemple :

```
public class Personne {

    private int age = 10;


    public int getAge(){ return this.age; }

}
```

```
public class Personne {

    private Integer age = new Integer(10);


    public Integer getAge(){ return this.age; }

}
```

Les types de données simples (4/5)

Fonctionnalités de wrappers

Les wrappers offrent différentes méthodes près à l'emploi et utiles lors des développements de programmes
Tous disposent à peu près des mêmes méthodes dont voici un exemple non exhaustif pour le cas du wrapper **Integer**

int intValue()	Retourne dans le type primitif *int* l'entier contenu dans l'objet de type Integer
double doubleValue()	Converti l'entier contenu dans l'objet de type Integer en équivalent de type double (nombre réel de grande taille)
long longValue()	Converti l'entier contenu dans l'objet Integer en équivalent de type *long* (nombre entier de grande taille)
int parseInt(String s)	Converti la chaine de caractères s et le retourne sous forme d'un entier primitif *int*
Integer valueOf(int i)	Converti un entier primitif et le retourne sous forme d'un objet de type Integer
Integer valueOf(String s)	Converti la chaine de caractères s et la retourne sous forme d'un objet de type Integer
Integer getInteger(String s)	Converti la chaine de caractères s et le retourne sous forme d'un objet de type Integer
String toString()	Transforme l'objet de type Integer courant en une chaine de caractères

Les types de données simples (5/5)

Wrappers/Autoboxing/Casting

Exemples utilisation des méthodes de wrappers :

- Integer ageW = new Integer(22);
  ==> int ageP = ageW.intValue(); ⇔ ageP = 22
- int in = Integer.parseInt("1675") ; ⇔ in = 1675
- Double salaire = new Double(1368.56);
  ==> long res = salaire.longValue(); ⇔ res = 1368

AutoBoxing/Unboxing :

Java sait implicitement convertir une valeur primitive en son type wrapper (et inversement) lorsqu'on réalise une opération d'affectation

Integer a = 12; ==> Integer a = new Integer(12);
Float f = 17.78f; ==> Float f = new Float(17.78f);
→ Autoboxing

Double d = new Double(1368.56d);
double dbl = d; → Unboxing

Casting de types primitifs :

Le *cast* est le fait de forcer le compilateur à considérer une variable comme étant d'un type qui n'est pas le type déclaré au départ

Deux types de casting :

- Cast automatique
  Toute variable ayant un type de taille inférieure au nouveau type où elle est affectée sera automatiquement « castée » par Java
  byte < short < char < int < long < float < double
  Exemple : byte b = 99; ==> short sh = b; ==> char c = sh, etc

- Cast manuel
  Toute variable ayant un type de taille supérieure au nouveau type où elle est affectée doit explicitement être « castée » par le développeur. Si casting impossible, une erreur sera levée
  double > float > long > int > char > short > byte
  Exemple : double d = 17.78d; ==> long a = (long) d ==> a = 17

1 Notion d’importation de bibliothèques

2 Les types de données simples

3 Les types de données de référence

4 Un mot sur le passage de paramètres dans les méthodes

Les types de référence (1/6)

Les types de référence représente essentiellement des familles objets créée à partir :

- ❖ d'une Classe
- ❖ d'une Interface
- ❖ d'une String : chaines de caractères
- ❖ d'un tableau de primitifs
- ❖ d'une Enumération
- ❖ d'une Collection

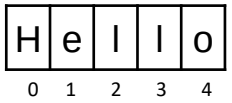
Les types de référence (2/6)

String ?

- Représente le type de données dédié aux chaines de caractères et chaines alphanumériques. Java considère une String comme un tableau de caractères dont l'indexation commence à la position 0 (et non 1)

- 2 modes de déclaration de variables de type String :
  => par affectation direct d'une chaine à la variable, ou par utilisation de l'opérateur *new*

  Exemples : - String var1 = null; //chaîne nulle
  - String var3 = ""; //chaîne vide
  - String var2 = "Hello";
  - String var4 = new String("Hello");



- Quelques fonctions applicables sur une String :
  - char charAt(int index) : retourne le caractère situé à la position index de la chaine. Ex : var2.charAt(1) ==> e
  - String concat(String str) : concatène str en queue de la chaîne courante et retourne le résultat. Ex : var2.concat(" World") ==> Hello World
  - boolean contains(CharSequence str) : retourne true ou false si la sous-chaine str est trouvé dans la chaine courante. Ex : var2.contains("el") == > true
  - boolean endsWith(String suffix) : si la chaine courant se termine par la chaine suffix, retourne true. Sinon, retourne false. Ex : var2.endsWith("ti") == > false
  - boolean StarsWith(String prefix) : si la chaine courant se commence par la chaine prefix, retourne true. Sinon, retourne false
  - String substring(int beginIndex) : retourne la sous-chaine de la chaine courante commençant à l'index beginIndex
  - String substring (int beginIndex, int endIndex) : retourne la sous-chaine de la chaine courante commençant à l'index *beginIndex* et se terminant à l'index *endIndex*-1
  - String toLowerCase() / String toUpperCase() : converti la chaine courante en minuscule / converti la chaine courante en majuscule
  - int length() : retourne le nombre de caractères présents dans la chaine courante

Les types de référence (3/6)

Les tableaux de données primitifs ?

Java traite les tableaux de données créés à partir de types primitifs, comme des objets et donc par référence

Exemple :

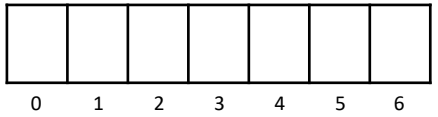
```
int[] t1 = new int[7]
short[] t2 = new short[58]
```

→ Tableaux à 1 dimension

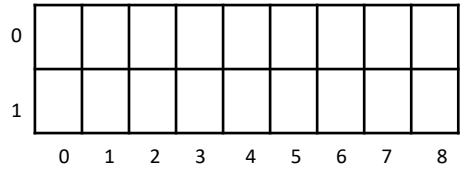
```
float[][] t3 = new float[6][9]
long[][] t4 = new long[2][9]
```

→ Tableaux à 2 dimension

t1 :



t4 :



- L'indexation des cases d'un tableau commence à **0** (et non 1)
- A la déclaration d'un tableau primitif, Java crée implicitement une classe qui sera le type de l'objet découlant
- L'accès à la taille d'un tableau primitif se fait par l'appel de l'attribut length. Ex : t1.length == 7
- L'accès à un élément du tableau se fait via l'utilisation de son/ses index. Ex : int a = t1[3]; long b = t4[1][5]
- Possibilité de définir des tableaux multidimensionnels supérieurs à 2. Ex : double[][][] d = new double[n][m][p]

Les types de référence (4/6)

Les énumérations ?

Type de données particulier correspondant à un ensemble fini de valeurs qui le caractérise. Chaque valeur est une constante et est donc immuable. A une variable de ce type, on affecte toujours exclusivement une des valeurs de l'ensemble

Exemple : la **Civilité,** le **statut matrimonial** peuvent être modélisés en Java par une énumération

- Syntaxe de déclaration :
- Exemple affectation de variables :
- 2 exemples de méthodes :

```
public enum Civilite {

   MADEMOISELLE, MADAME, MONSIEUR

}
```

```
public enum StatutMatrimonial {

   CELIBATAIRE, PACSE, MARIE, DIVORCE

}
```

```
Civilite mr = Civilite.MONSIEUR;

Civilite[] civilites = Civilite.values();
```

```
StatutMatrimonial statut = StatutMatrimonial.MARIE;

int  ordre  = statut.ordinal()     ==> ordre == 2
```

(On peut réaliser des opérations un peu plus complexes avec les énumérations, mais nous ne l'aborderons pas dans ce cours)

Les types de référence (5/6)

Collections ? (1/2)

Classes basées sur la *généricité,* définient dans la bibliothèque *java.util*, et qui fournissent des fonctionnalités permettant de stocker et gérer des ensembles d'objets. Prosaïquement, on peut simplement voir une collection comme un tableau d'objets disposant de caractéristiques particulières de par les traitements qu'elle offre sur les éléments stockés

Les collections Java de base : ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap, HashTable, Properties, ArrayDeque, Stack, CopyOnWriteArrayList, ConcurrentHashMap, CopyOnWriteArraySet

Caractéristiques de 3 collections les plus usités dans les programmes :

- **ArrayList :** collection d'éléments ordonnés par leurs index et qui accepte les doublons
- **HashSet :** collection d'éléments non ordonnés qui n'accepte pas les doublons
- **HashMap :** collection sous la forme d'une association de paires clé/valeur

Les types de référence (6/6)

Collections : déclaration et fonctionnalités (2/2)

ArrayList

Exemples de déclaration :

ArrayList<E> a = new ArrayList<>(); OU List<E> a = new ArrayList<>();

où E correspond à un type de référence (classe, Interface, Enumération, Wrapper, String)

Quelques fonctions applicables sur un objet ArrayList :

- *boolean add(E e)* : ajoute l'élément e à une ArrayList et retourne true ou false si l'opération s'est terminée avec succès ou pas
- *boolean add(int index, E e)* : ajoute l'élément e à la case numéro *index* d'une ArrayList. Ecrasera la donnée existante
- *boolean addAll(Collection c)* : ajoute tous les éléments de la collection c dans l'ArrayList en cours
- *E get(int index)* : retourne l'élément situé à la case numéro *index*
- *boolean contains(E e)* : détermine si l'élément e est présent dans l'objet ArrayList en répondant par true ou false
- *E remove(E e)* : supprime le premier élément e trouvé et le retourne
- *boolean remove(int index)* : supprime l'élément situé à la case de position *index*
- *int size()* : retourne le nombre d'éléments présents dans l'ArrayList

HashSet

Exemples de déclaration :

HashSet<E> a = new HashSet<>(); OU Set<E> a = new HashSet<>();

où E correspond à un type de référence (classe, Interface, Enumération, Wrapper)

Quelques fonctions applicables sur un objet HashSet :

- *boolean add(E e)* : ajoute l'élément e à une HashSet et retourne true ou false si l'opération s'est bien déroulée ou pas. Si e est déjà présent, retourne false, car n'accepte pas les doublons
- *boolean addAll(Collection c)* : ajoute tous les éléments de la collection c dans le HashSet en cours. Les doublons seront éliminés
- *boolean isEmpty()* : vérifie si l'objet HashSet est vide ou pas
- *boolean contains(E e)* : détermine si l'élément e est présent dans l'objet ArrayList en répondant par true ou false
- *E remove(E e)* : supprime le premier élément e trouvé et le retourne
- *int size()* : retourne le nombre d'éléments présents dans le HashSet

HashMap

Exemples de déclaration :

HashMap<K, E> a = new HashMap<>();
OU Map<K, E> a = new HashMap<>();

où K, E correspondent à un type de référence (classe, Interface, Enumération, Wrapper, String)

Quelques fonctions applicables sur un objet HashMap :

- *boolean put(K key, E e)* : ajoute la paire (key, e) dans la HashMap. e est identifié dans la HashMap par la clé unique *key*. Si la HashMap contient déjà un élément avec cette clé, celui-ci sera remplacé par e
- *E get(Key key)* : retourne l'élément associé à la clé *key*
- *Set<K> keySet()* : retourne dans une HashSet l'ensemble des clés présents dans la HashMap
- *Collection<E> values()* : retourne dans une collection l'ensemble des objets de type E stockés dans la HashMap
- *E remove(K key)* : supprime l'élément associé à la clé *key* et le retourne
- *int size()* : retourne le nombre d'éléments présents dans la HashMap

(ces collections disposent de beaucoup plus de méthodes que celles exposées ici…)

1 Notion d’importation de bibliothèques

2 Les types de données simples

3 Les types de données de référence

4 Un mot sur le passage de paramètres dans les méthodes

Passage de paramètres aux méthodes

Rappel :

<ModificateurVisibilité> <TypeDeDonnéesRetour> **<NomMéthode>** (**<ListeDesParamètres>**) ;

Exemples :
public void methodeX(int i, char c);
private void methodeY(Double d, Integer in)
protected float methodeZ(UneClasse c)

- Les paramètres d'une méthode sont la liste des variables contenant ses données d'entrée
- Lorsqu'on définit des méthodes contenants des paramètres d'entrées, des effets de bord peuvent survenir en fonction de leur typage et il convient de les avoir en mémoire :

- Java effectue par défaut le passage de paramètres par valeur (par copie) pour tous les types (primitif ou référence)

➔ Exemple : soit un variable *p* primitive ou de référence. Si *p* est passé en paramètre à une méthode *m*, alors *m* ne peut modifier sa valeur par *simple affectation*. Java protège les paramètres en effectuant implicitement une copie de ceux-ci afin que les méthodes ne travaillent qu'avec leurs copies afin de limiter les impacts hors de celle-ci

- Petit bmol : si le paramètre est un objet issu d'un type de référence, alors c'est sa vraie référence qui est passée

➔ Si le type du paramètre est une classe, ou une collection alors la modification explicite de ses attributs ou de ses éléments dans le corps de la méthode aura pour effet de modifier effectivement l'objet initial

Passage de paramètres aux méthodes

Illustration du passage de paramètre par valeur

```
public class E {
  private String nom;

  public E(String nom) {
    this.nom = nom;
  }

  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }
}
```

```
public class Exemple {

   private short varS = 10;
   private E varE = new E("Max");

   public short setVarS(short s) {
     s = (short) (this.varS +  5);
     return (short) s;
   }

   public void setVarE(E e) {
     e = this.varE;
   }

   public static void main(String args[]) {
     Exemple ex = new Exemple();

A  short s = 27;
   short rs = ex.setVarS(s);
   System.out.println(" rs = " + rs + " # valeur de s = " + s);

B  E e = new E("Brad");
   ex.setVarE(e);
   System.out.println(" e.nom = " + e.getNom());

   }

}
```

Ici nous avons 2 classes **E** et **Exemple** (qui est la classe principale) dans notre programme

La classe Exemple déclare une primitive *varS* de type *short* et une instance *varE* de la classe E.

La classe Exemple dispose d'une méthode *setVarS()* qui essaie d'affecter à son paramètre *s* (de type short) le résultat du calcul *this.varS + 5* ; et d'une autre méthode *setVarE()* qui affecte l'instance *varE* à son paramètre *e*.

L'exécution du programme donnera le résultat suivant :

```
rs = 15 # valeur de s = 27
e.nom = Brad
```

Observation :

- Dans le bloc **A**, la variable *s* appartenant à la méthode *main()* de valeur initiale 27, n'a pu être modifiée par la méthode *setVarS()*.
- Idem pour le bloc **B**. L'instance *e* appartenant à la méthode *main()* n'a pu être modifiée par *setVarE()*, car elle essaie par simple affectation d'écraser l'objet *e* initial

Passage de paramètres aux méthodes

Illustration du passage de paramètre par référence

```java
public class E {
  private String nom;

  public E(String nom) {
    this.nom = nom;
  }

  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }
}
```

```java
import java.util.ArrayList;

public class Exemple {

  public void setVarR(ArrayList<Integer> p) {
    //l'élément situé à l'index 0 de p se voit affecter la valeur 19
    p.add(0, 19);
  }

  public void setVarE(E e) {
    //l'attribut de l'objet e est modifié par la valeur Max
    e.setNom("Max");
  }

  public static void main(String args[]) {
    Exemple ex = new Exemple();

    ArrayList<Integer> varR = new ArrayList<>();
    varR.add(63);
    varR.add(102);
    varR.add(711);
    ex.setVarR(varR);
    System.out.println(" varR[0] = " + varR.get(0));

    E e = new E("Brad");
    ex.setVarE(e);
    System.out.println(" e.nom = " + e.getNom());

  }

}
```

Ici nous avons 2 classes **E** et **Exemple** (qui est la classe principale) dans notre programme

La classe Exemple dispose d'une méthode *setVarR()* qui modifie l'élément situé à l'index 0 (le premier élément) de la collection *p* de type ArrayList passé en entrée, par la valeur 19; et d'une autre méthode *setVarE()* qui modifie l'attribut *nom* de l'objet *e* de type E passé en paramètre

L'exécution du programme donnera le résultat suivant :

```
varR[0] = 19
e.nom = Max
```

Observation :

- Dans le bloc **A**, la variable *varR* appartenant à la méthode *main()* est de valeur initiale *[63, 102, 711].* On constate que l'appel de la méthode *setVarR()* modifie effectivement la case 0 de « 63 » à « 19 » et donc VarR vaudra *[19, 102, 711]* après appelle de la méthode
- Idem pour le bloc **B**. L'instance *e* appartenant à la méthode *main()* s'est vu effectivement modifiée la valeur de l'attribut *nom* de « Brad » à « Max »


SUCCESS