Bonnes pratiques et types paramétrés

Arnaud Labourel arnaud.labourel@univ-amu.fr

4 avril 2022

Section 1

Bonnes pratiques de programmation

Pièges à éviter pour le nommage de variables/attributs

Vous ne devez pas avoir des variables avec des :

- noms en une lettre (même pour les indices) : `i`, `j`, ...
- noms numérotés : `a1`, `a2`, `a3`, ...
- abréviations ayant plusieurs interprétations : `rec`, `res`, ...
- noms ne donnant pas le sens précis : `temporary`, `result`, ...
- des noms trompeurs : par exemple un `accountList` doit être une `List` (et pas un `array` ou un autre type)
- types de l'objet au singulier pour une collection d'objet : une liste de personnes doit s'appeler `persons` et non `person`.
- noms imprononçables : `genymdhms`, ...

Règles de nommage (1/2)

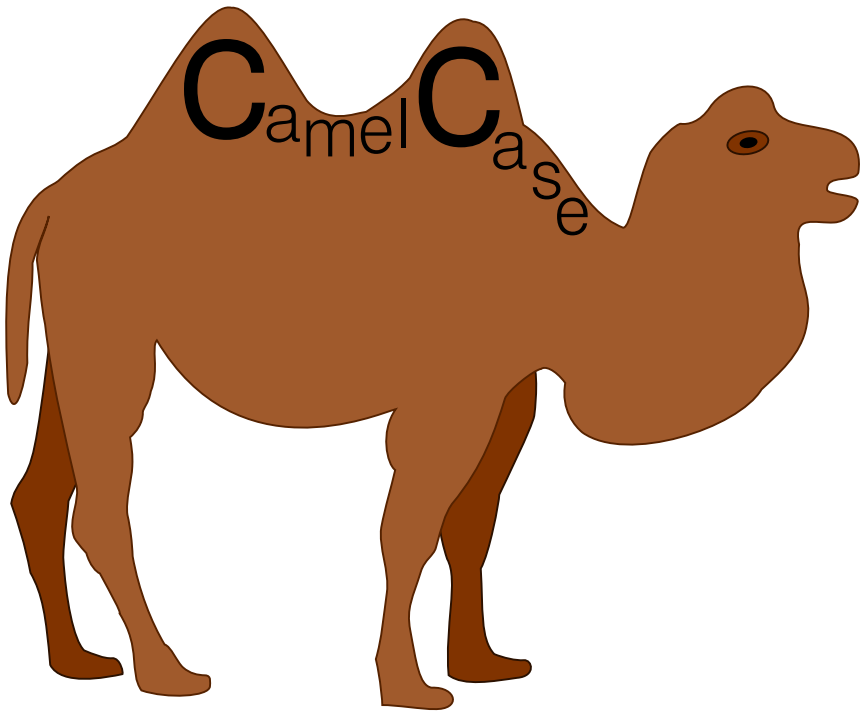
En anglais

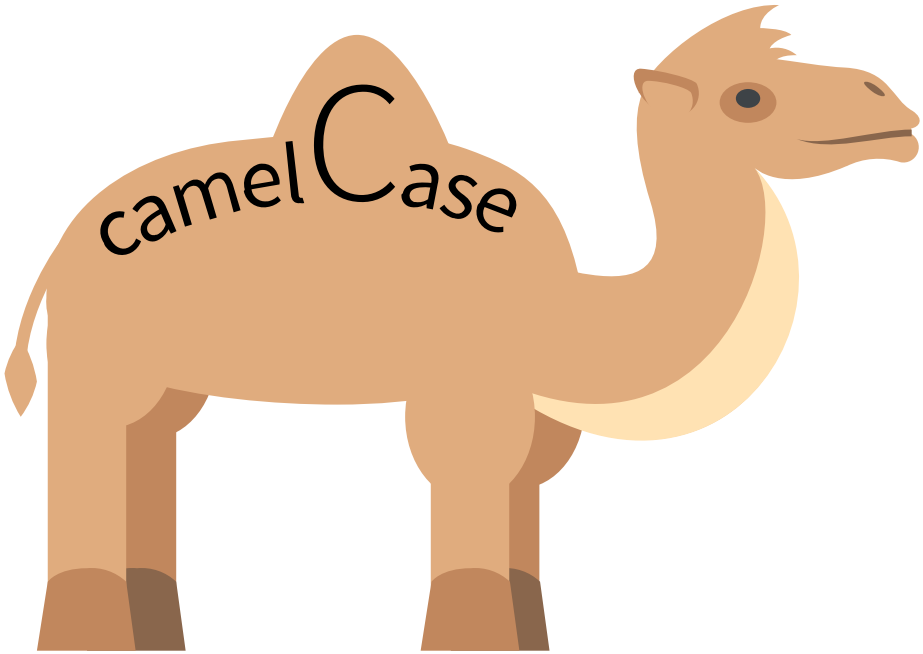
- Le **Java** et la quasi-totalité des langages de programmation utilise l'anglais (dans les mot-clés et les librairies standards).
  $\Rightarrow$ On doit programmer en anglais pour avoir la cohérence du code
- Utiliser l'anglais permet aussi d'augmenter le nombre de personnes pouvant lire le code et d'avoir de nombreux exemples existants pour s'inspirer.

En "camel case"

Un nom composé de plusieurs mots n'utilise ni espace ni ponctuation, et sépare les mots en mettant en capitale la première lettre de chaque mot.
Exemples: `getElementsByTagName`, `ListArray`, ...

Règles de nommage (2/2)





CamelCase vs camelCase

- Les noms qui définissent un type (classes, interfaces, enum, record, ...) commence par un majuscule.
- Les autres noms (méthodes, variables, attributs, ...) commencent par une minuscule.

Nommage des méthodes : cas 1

Méthodes procédurales

Méthodes modifiant l'état de l'objet
$\Rightarrow$ groupe verbal à l'infinitif.

Exemples

- `boolean add(E element)`
- `E set(int index, E element)`
- `boolean removeAll(Collection<?> c)`

Nommage des méthodes : cas 2

Expressions non booléennes

Méthodes renvoyant une partie de l'état de l'objet ⇒ groupe nominal ou getter.

Exemples

- `int size()`
- `List<E> subList(int fromIndex, int toIndex)`
- `int hashcode()`
- `ListIterator<E> listIterator()`
- `E get(int index)`
- `Color getBackground()`
- `float getOpacity()`

Nommage des méthodes : cas 3

Expressions booléennes

Méthodes testant un prédicat sur l'objet $\Rightarrow$ groupe verbal au présent.

Exemples

- `boolean isEmpty()`
- `boolean contains(Object o)`
- `boolean equals(Object o)`

Nommage des méthodes : cas 4

Méthodes de conversion ⇒ utilisation du to

Exemples :

- `String toString()`
- `Object[] toArray()`

Les règles ne sont pas absolues mais juste des conventions qui peuvent avoir des exceptions.

En général le plus simple est de s'inspirer de l'existant : par exemple la **JDK** et de bien réfléchir lorsqu'on souhaite déroger aux règles.

Comment rendre le nommage des méthodes facile ?

En écrivant des méthodes courtes

De préférence une dizaine de ligne maximum.

Comment écrire des méthodes courtes

En extrayant le plus possible les partie du code d'une méthode à d'autres méthodes.

Conseils

- Réfléchir avant de coder au rôle de la méthode
- Se demander ce qui peut être confié à d'autres méthodes

Ne pas mentir

```
class User {
    private boolean authenticated;
    private String password;

    public boolean checkPassword(String password) {
        if (password.equals(this.password)) {
            authenticated = true;
            return true;
        }
        return false;
    }
}
```

La méthode authentifie l'utilisateur alors qu'elle ne devrait que vérifier la validité du mot de passe d'après son nom.

Section 2

Des principes pour bien programmer

Comment rendre le nommage des méthodes facile ?

En écrivant des méthodes courtes

De préférence une dizaine de ligne maximum.

Comment écrire des méthodes courtes

En extrayant le plus possible les partie du code d'une méthode à d'autres méthodes.

Conseils

- Réfléchir avant de coder au rôle de la méthode
- Se demander ce qui peut être confier à d'autres méthodes

Pourquoi découper et nommer ?

Quelle est la sémantique du texte suivant ?

L'animal vertébré vivipare caractérisé par la présence de mamelles, qui se nourrit en grignotant à l'aide de leurs deux paires d'incisives, et qui se multiplie avec une rapidité d'intensité forte, aux organes d'audition et d'équilibration à l'étendue supérieure à la moyenne dans le sens de la longueur, fait descendre par le gosier, postérieurement à une réduction en petites parcelles avec les dents, un être vivant appartenant au règne végétal cultivé pour l'usage culinaire, muni de pédicelles partant en faisceau depuis son pédoncule, et dont la partie souterraine permettant la fixation au sol est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

L'animal vertébré vivipare caractérisé par la présence de mamelles,
qui se nourrit en grignotant à l'aide de leurs deux paires d'incisives,
et qui se multiplie avec une rapidité d'intensité forte,
aux organes d'audition et d'équilibration
à l'étendue supérieure à la moyenne dans le sens de la longueur,
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
qui se nourrit en grignotant à l'aide de leurs deux paires d'incisives,
et qui se multiplie avec une rapidité d'intensité forte,
aux organes d'audition et d'équilibration
à l'étendue supérieure à la moyenne dans le sens de la longueur,
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
et qui se multiplie avec une rapidité d'intensité forte,
aux organes d'audition et d'équilibration
à l'étendue supérieure à la moyenne dans le sens de la longueur,
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
très prolifique
aux organes d'audition et d'équilibration
à l'étendue supérieure à la moyenne dans le sens de la longueur,
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
très prolifique
aux oreilles
à l'étendue supérieure à la moyenne dans le sens de la longueur,
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
très prolifique
aux oreilles
longues
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
très prolifique
aux oreilles
longues
avale
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
très prolifique
aux oreilles
longues
avale
après
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
très prolifique
aux oreilles
longues
avale
après
mâchage
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
très prolifique
aux oreilles
longues
avale
après
mâchage
une plante
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
très prolifique
aux oreilles
longues
avale
après
mâchage
une plante
potagère
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
très prolifique
aux oreilles
longues
avale
après
mâchage
une plante
potagère
ombellifère
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
très prolifique
aux oreilles
longues
avale
après
mâchage
une plante
potagère
ombellifère
à racine
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère
lagomorphe
très prolifique
aux oreilles
longues
avale
après
mâchage
une plante
potagère
ombellifère
à racine
rouge.

Pourquoi découper et nommer ?

Quelle est la sémantique du texte suivant ?

Le mammifère lagomorphe très prolifique aux longues oreilles avale après mâchage une plante potagère ombellifère à racine rouge.

Pourquoi découper et nommer ?

Le mammifère lagomorphe très prolifique aux longues oreilles
avale après mâchage
une plante potagère ombellifère à racine rouge.

Pourquoi découper et nommer ?

Le lapin
avale après mâchage
une plante potagère ombellifère à racine rouge.

Pourquoi découper et nommer ?

Le lapin
mange
une plante potagère ombellifère à racine rouge.

Pourquoi découper et nommer ?

Le lapin
mange
une carotte.

Pourquoi découper et nommer ?

Quelle est la sémantique du texte suivant ?

Le lapin mange une carotte.

Principes de programmation

Il existe de nombreux principes de programmation permettant de guider l'écriture de code :

- DOT : Do One Thing
- DRY : Don't Repeat Yourself
- KISS : Keep It Simple Stupid
- SRP : Single Responsibility Principle
- ...

Autres principes

Les 5 principes SOLID (principes de conception objet) que vous verrez en détail en L3 dans le cours de **Programmation et conception**

Do One Thing

Une fonction ne doit faire qu'**une seule chose**.

Pour cela, elle ne doit réaliser que des étapes de même niveau d'abstraction.

On décompose la fonction :

Pour faire la cuisine je dois (premier niveau d'abstraction) :

- choisir une recette;
- réunir les ingrédients;
- suivre la recette.

Pour choisir une recette, je dois (deuxième niveau d'abstraction):

- réfléchir à ce que j'ai envie de manger;
- chercher sur marmiton.

Mauvaise approche

```
void cook(){
    // On choisit la recette
    Food wantToEat = thinkAboutFood();
    Recipe recipe = lookOnMarmiton(wantToEat);
    // On réunit les ingrédients
    openFridge();
    for(Ingredient ingredient :
        recipe.getFreshIngredients()){
        takeInFrige(ingredient);
    }
    closeFridge();
    openCupboard();
    ...
    // On suit la recette
    ...
}
```

Bonne approche

```
void cook(){
    Recipe recipe = chooseRecipe();
    gatherIngredients(recipe);
    followRecipe(recipe);
}

Recipe chooseRecipe(){
    Food wantToEat = thinkAboutFood();
    Recipe recipe = lookOnMarmiton(wantToEat);
    return recipe;
}

...
```

Do One Thing

Do One Thing

Une variable/méthode/classe doit n'avoir qu'une seule signification.

Une variable/méthode/classe **ne** doit **pas** avoir :

- un sens dans une circonstance et autre sens dans un domaine différent,
- ou bien avoir deux sens en même temps.

Le principe **Do One thing** est fortement lié aux principes suivants :

- **Don't Repeat Yourself :** il ne faut pas se répéter car la répétition produit de l'inconsistance et des erreurs.
- **Once and Only Once :** chaque comportement ne doit être défini qu'une fois.

Do one Thing pour les méthodes

Une méthode ne doit avoir qu'une **responsabilité = raison de changer**.

Comment déceler une méthode ne respectant pas DOT

- Une méthode ayant un "and" dans le nom ⇒ `doOneThingAndAnotherThing`
- Une méthode peut réalisant une action complexe sans appeler d'autres méthodes.
- Une méthode trop longue (plusieurs dizaines de lignes de code)
- un niveau d'indentation élevé : par exemple un `if` dans un `for` dans un `for`

Exemple de méthode ayant trop de responsabilités

```
int countAndPrintSquares(List<Rectangle> rectangles) {
  int countSquare = 0;
  for (Rectangle rectangle : rectangles) {
    if (rectangle.width == rectangle.height) {
      System.out.println(rectangle);
      countSquare++;
    }
  }
  return countSquare;
}
```

La méthode doit :

- tester si un rectangle est un carré
- afficher les carrés
- compter les carrés

Chaque méthode a une responsabilité

```
boolean isSquare(){ return this.width == this.height; }
static int printSquares(List<Rectangle> rectangles) {
    for (Rectangle rectangle : rectangles) {
        if (rectangle.isSquare()) {
            System.out.println(rectangle);
        }
    }
}
static void countSquares(List<Rectangle> rectangles) {
    for (Rectangle rectangle : rectangles) {
        if (rectangle.isSquare()) squareCount++;
    }
    return squareCount;
}
```

Autre exemple

```
boolean removeFirstMinimumValue(List<Integer> list) {
    if (list.isEmpty()) { return false; }
    int position = 0;
    for (int i = 0; i < list.size(); i++) {
        if (list.get(position) > list.get(i)) {
            position = i;
        }
    }
    list.remove(position);
    return true;
}
```

Correction (1/2)

```
int findMinimumValuePosition(List<Integer> list) {
    if (list.isEmpty()) {
        throw new IllegalArgumentException();
    }
    int position = 0;
    for (int i = 0; i < list.size(); i++) {
        if (list.get(position) > list.get(i)) {
            position = i;
        }
    }
    return position;
}
```

Correction (2/2)

```
boolean removeFirstMinimumValue(List<Integer> list) {
    if (list.isEmpty()) {
        return false;
    }
    int position = findMinimumValuePosition(list);
    list.remove(position);
    return true;
}
```

Single Responsability Principle (SRP)

Principe SRP

Une classe ne doit avoir qu'une **responsabilité = raison de changer**

Les effets néfastes des responsabilités multiples

- Difficulté à nommer car la classe est trop complexe
- Difficulté à réutiliser le code car seulement une des responsabilités est réutilisable
- Difficulté à mettre à jour le code car changer l'implémentation d'une partie impacte les autres parties

Pourquoi SRP ?

- Facilite le nommage et documentation
- Facilite l'écriture des tests
- Facilite la réutilisation des classes

Section 3

Types paramétrés

Stack d'Object

Supposons que nous ayons la classe suivante :

```
public class Stack {
    private Object[] stack = new Object[100];
    private int size = 0;
    public void push(Object object) {
        stack[size] = object; size++;
}
    public Object pop() {
        size--;
        Object object = stack[size];
        stack[size]=null; // Pour le Garbage Collector.
        return object;
    }
}
```

Problème de Stack d'Object

Nous rencontrons le problème suivant :

```
Stack stack = new Stack();
String string = "truc";
stack.push(string);
string = (String)stack.pop();
// Transtypage (cast) obligatoire !
```

Nous avons également le problème suivant :

```
Stack stack = new Stack();
Integer intValue = new Integer(2);
stack.push(intValue);
String string = (String)stack.pop();
// Erreur à l'exécution
```

La solution : types paramétrés

Par conséquent, on souhaiterait pouvoir préciser le type des éléments :

```
Stack<String> stack = new Stack<String>();
String string = "truc";
stack.push(string); // Le paramètre doit être un String.
String string = stack.pop(); // retourne un String.
```

Java nous permet de définir une classe Stack qui prend en paramètre un type. Ce type paramétré va pouvoir être utilisé dans les signatures des méthodes et lors de la définition des champs de la classe.

Lors de la compilation, Java va utiliser le type paramétré pour effectuer :

- des vérifications de type ;
- des transtypages automatiques ;
- des opérations d'emballage ou de déballage de valeurs.

Définition de classes paramétrées

La nouvelle version de la classe Stack :

```
public class Stack<T> {
    private Object[] stack = new Object[100];
    private int size = 0;
    public void push(T element) {
        stack[size] = element;
        size++;
    }
    public T pop() {
        size--;
        T element = (T)stack[size];
        stack[size] = null;
        return element;
    }
}
```

Emballage et déballage

Les types primitifs ne sont pas des classes :

Dans le cas d'un `int`, on doit utiliser la classe d'emballage (wrapper class) Integer :

Interdit : ~~`Stack<int> stack = new Stack<int>();`~~

Autorisé :

```
Stack<Integer> stack = new Stack<Integer>();
int intValue = 2;
Integer integer = Integer.valueOf(intValue);
// → emballage du int dans un Integer.
stack.push(integer);
Integer otherInteger = stack.pop();
int otherIntValue = otherInteger.intValue();
// → déballage du int présent dans le Integer.
```

Types primitifs

type	classe d'emballage	taille	valeurs possibles
byte	Byte	8 bits	-128 à 127
short	Short	16 bits	-32768 à 32767
int	Integer	32 bits	-2^{31} à $2^{31}-1$
long	Long	64 bits	-2^{63} à $2^{63}-1$
float	Float	32 bits	
double	Double	64 bits	
char	Character	16 bits	caractère unicode
boolean	Boolean	non définie	`false` ou `true`

Classes d'emballage

La classe `Number` sert de base pour toutes les classes d'emballage.

Elle contient les méthodes suivantes :

- `public int intValue()`
- `public long longValue()`
- `public float floatValue()`
- `public double doubleValue()`
- `public byte byteValue()`
- `public short shortValue()`

Classes d'emballage

Les classes d'emballage étendent `Number` :

- `Byte` → `public static Byte valueOf(byte b)`
- `Short` → `public static Short valueOf(short s)`
- `Integer` → `public static Integer valueOf(int i)`
- `Long` → `public static Long valueOf(long l)`
- `Byte` → `public static Byte valueOf(byte b)`

Ils existent des constructeurs mais ils sont dépréciés (et donc pas à utiliser).

La classe Character

Les classes d'emballage ne contiennent pas que des méthodes liées aux instances :

- `public static Byte valueOf(byte b)`
- `public static char charValue()`
- `public static boolean isLowerCase(char ch)`
- `public static boolean isUpperCase(char ch)`
- `public static boolean isDigit(char ch)`
- `public static boolean isLetter(char ch)`
- `public static boolean isLetterOrDigit(char ch)`
- `public static char toLowerCase(char ch)`
- `public static char toUpperCase(char ch)`
- `public static char toTitleCase(char ch)`

Emballage et déballage automatique

Depuis Java 5, il existe l'emballage et le déballage automatique :

```
Stack<Integer> stack = new Stack<Integer>();
int intValue = 2;
stack.push(intValue);
// → emballage automatique du int dans un Integer.
int otherIntValue = stack.pop();
// → déballage automatique du int.
```

Attention

Il est important de noter que des allocations sont effectuées lors des emballages sans que des `new` soient présents dans le code.

Plusieurs paramètres de types

```
public class Pair<A, B> {
    public A first;
    public B second;
    public Pair(A first, B second) {
        this.first = first;
        this.second = second;
    }
    public static <A, B> Pair<A,B>
            makePair(A first, B second) {
        return new Pair<A,B>(first, second);
    }
}
```

Utilisation d'une classe avec plusieurs paramètres de types

```
public class Main {

    public static void main(String[] args) {
        Pair<String,Integer> pair =
            Pair.makePair("tot",12);
        System.out.println(pair);
    }
}
```

Section 4

Interfaces génériques

Exemple d'interfaces génériques en Java

- `Comparable<T>` : objets qu'on peut comparer à des objets de type `T`.
- `List<T>` : liste d'objets de type `T`.
- `Stack<E>` : pile d'objet de type `T`.
- `Iterable<T>` : collection d'objet de type `T` qu'on peut parcourir avec un boucle.

Rappel : types paramétrés/génériques

Types dont la définition contient un autre type.

Définition de Comparable<T>

```
public interface Comparable<T>{
    /**
     * Compares this object with the specified object for
     * order. Returns a negative integer, zero, or a
     * positive integer as this object is less than,
     * equal to, or greater than the specified object.
     * @param other the objet to be compared
     * @return a negative integer, zero, or a positive
     * integer as this object is less than, equal to, or
     * greater than the specified object.
     */
    int compareTo(T other);
}
```

Utilisation de Comparable<T>

```
// Utilisation typique

public class MyOrderedClass
    implements Comparable<MyOrderedClass>{

    int compareTo(MyOrderedClass other){
        // ...
    }
}
```

Sert à définir une relation d'ordre entre les objets d'une classe (par exemple pour les trier).

Utilisation de Comparable<T> pour la classe Student

```
public class Student
    implements Comparable<Student>{

    public final string lastName;
    public final in idNumber;
    //...
    public int compareTo(Student other){
        return lastname.compareTo(other.lastName);
    }
}

List<Student> students; //...
Collections.sort(students);
// Tri par nom de famille
```

Utilisation de Comparable<T> pour la classe Student

```
public class Student
    implements Comparable<Student>{

    public final string lastName;
    public final in idNumber;
    //...
    public int compareTo(Student other){
        return idNumber - other.idNumber;
    }
}

Student[] students; //...
Arrays.sort(students);
// Tri par numéro d'étudiant
```

Interface Iterable

```
public interface Iterable<T>{
    Iterator<T> iterator();
    void forEach(Consumer <? super T> action);
    Spliterator<T> spliterator();
}
```

- Définition complexe
- Utilisation facile et utile

Utilisation d'Iterable

Utilisation : si une classe implémente `Iterable<T>`, ces instances contiennent une collection d'objets de type `T`.

On peut parcourir les objets de la collection à l'aide d'une boucle `for`.

```
public class Order{
    Iterable<Items> items;
    //...
    public void printAllItems(){
        for (Item item : this.items){
            System.out.println(item.toString());
        }
    }
}
```

Interface Collection

```
public interface Collection<T> extends Iterable<T>{
    boolean add(T element);
    boolean contains(T element);
    boolean isEmpty();
    boolean remove(Object o);
    //...
}
```

Mot-clé extends

Quand une interface “fille” étend une interface “mère”, elle hérite de toutes les méthodes de sa “mère”.
Une classe implémentant la classe “fille” doit donc définir les méthodes des deux interfaces.