

# INTRODUCTION AU LANGAGE (SUITE)



---

Moussa LO

UFR de Sciences Appliquées et de Technologie

Université Gaston Berger de Saint-Louis



# EXCEPTIONS

## *Les exceptions*

### Controle de saisie avant de calculer la factorielle d'un nombre :

```
public static void main(String arg[]){
    String s = null;
    do
        try {
            s = JOptionPane.showInputDialog("Entrez un nbre: ");
            if (s != null) {
                long n = Long.parseLong(s);
                long factN = factorielle(n);
                JOptionPane.showMessageDialog(null, n + "! =
"+ factN);
            }
        }
        catch(NumberFormatException ex){
            JOptionPane.showMessageDialog(null, s + " n'est pas
un entier");
        }
    ...
}
```

## *Les exceptions*

Controle de saisie avant de calculer la factorielle d'un nombre :

...

```
        finally {  
            reponse = JOptionPane.showConfirmDialog(null,  
"Voulez-vous quitter ?");  
        }  
        while (reponse == JOptionPane.NO_OPTION);  
    }
```

```
// methode factorielle  
public static long factorielle(long n){  
    if (n <= 1)  
        return 1;  
    return n * factorielle(n-1);  
}
```

## *Les exceptions*

- Une **exception** est un **signal** qui indique qu'un évènement exceptionnel comme une erreur est survenue *au cours de l'exécution d'un programme* :
  - **FileNotFoundException** survient lorsqu'on tente d'ouvrir un fichier inexistant
  - **IOException** survient lorsqu'on fait une lecture incorrecte, lorsqu'un fichier ne peut être fermé, etc.

## *Les exceptions*

- Java permet de gérer les erreurs par exception, i.e. de prévoir dans son application des blocs de code où sera traitée systématiquement, telle ou telle condition d'erreur.
- Une exception est un objet de la classe **java.lang.Exception**.
- Pour les gérer, on utilise l'**instruction *try***.

## *Les exceptions*

```
try {  
    // bloc d'instructions à protéger  
}  
  
catch ( Exception e) {  
    // traitement de l'exception  
}  
  
finally {  
    // à exécuter quelque soit la façon dont on sort du try  
}
```

## *Les exceptions*

- Un bloc d'instructions **try** permet de regrouper une ou plusieurs instructions de programme (appel de méthode, instructions de contrôle, etc.) où des exceptions peuvent se produire et être traitées.
- Un bloc **try** est suivi d'instructions *catch* et *finally*.



## *Les exceptions*

- **catch** spécifie le traitement associé aux exceptions, i.e ce que l'on fait si une exception se produit (**afficher un message d'erreur par exemple**).
- **finally** est exécuté inconditionnellement, quelque soit la façon dont on sort du bloc **try** (**pour fermer par exemple un fichier ouvert dans le bloc try**).

## *Les exceptions*

```
import java.io.FileReader;
public class lectureFichier {
    public static void main(String[] args) {
        try{
            FileReader f=new FileReader("fic.txt");
            System.out.println("Le fichier existe !");
            f.close();
            System.out.println("Le fichier a ete ferme !");
        }
        catch(FileNotFoundException fe) {
            System.out.println("Fichier inexistant !");
        }
        catch(IOException ie) {
            System.out.println("Erreur de fermeture !");
        }
    }
}
```

## Les exceptions

Chaque **méthode** susceptible de **générer une exception** qui lui est propre doit la déclarer dans son en-tête. On utilise l'instruction ***throws*** pour cela :

```
public class Pile {  
    final int taille_max = 100 ;  
    private int sommet ;  
    private int[ ] elements ;  
  
    public void empiler (int x) throws ErreurPilePleine {  
        if (sommet == taille_max)  
            throw new ErreurPilePleine()  
        else    elements[++sommet] = x ;  
    }  
}  
  
class ErreurPilePleine extends java.lang.Exception {  
    ErreurPilePleine(){ System.out.println("Pile Pleine");}  
}
```

## *La classe Scanner et InputMismatchException*

- **Scanner** génère une exception de type **InputMismatchException** en cas d'erreur de saisie. Cela permet de contrôler la saisie:

```
java.util.Scanner entree = new java.util.Scanner(System.in);
int x = 0;
for ( ; ;) {
    System.out.print("Donnez un entier x: ");
    try{
        x = entree.nextInt(); break; }
    catch (java.util.InputMismatchException ime) {
        entree.nextLine();
        System.out.print("Erreur - Recommencez"); }
} // for
System.out.print("x = " + x);
...
```



# JAVADOC

# JavaDoc : exemple

The screenshot shows a Netscape browser window displaying the JavaDoc website for the Java 2 Platform SE v1.4.1. The browser's address bar shows the URL "All Classes (Java 2 Platform SE v1.4.1)". The page has a blue header with the title "All Classes (Java 2 Platform SE v1.4.1) - Netscape". The main content area is titled "Java™ 2 Platform, Standard Edition, v 1.4.1 API Specification". It includes a navigation bar with links for "Overview", "Package", "Class", "Use", "Tree", "Deprecated", "Index", and "Help". The "Overview" link is selected. Below the navigation bar, there is a section titled "See:" with a link to "Description". The main content area features a table titled "Java 2 Platform Packages" with two columns: the package name and a description of its contents. The table lists several packages, including java.applet, java.awt, java.awt.color, java.awt.datatransfer, java.awt.dnd, java.awt.event, and java.awt.font. The left sidebar contains a list of "All Classes" and "Packages" with links to various classes and packages.

**Java™ 2 Platform Std. Ed. v1.4.1**

[All Classes](#)

**Packages**

[java.applet](#)

[java.awt](#)

[java.awt.color](#)

**All Classes**

[AbstractAction](#)

[AbstractBorder](#)

[AbstractButton](#)

[AbstractCellEditor](#)

[AbstractCollection](#)

[AbstractColorChooserPanel](#)

[AbstractDocument](#)

[AbstractDocument.Attribute](#)

[AbstractDocument.Content](#)

[AbstractDocument.Element](#)

[AbstractInterruptibleChannel](#)

[AbstractLayoutCache](#)

[AbstractLayoutCache.Node](#)

[AbstractList](#)

[AbstractListModel](#)

[AbstractMap](#)

[AbstractMethodError](#)

[AbstractPreferences](#)

[AbstractSelectableChannel](#)

[AbstractSelectionKey](#)

**Overview** Package Class Use [Tree](#) [Deprecated](#) [Index](#) [Help](#)

PREV NEXT [FRAMES](#) [NO FRAMES](#)

## Java™ 2 Platform, Standard Edition, v 1.4.1 API Specification

This document is the API specification for the Java 2 Platform, Standard Edition, version 1.4.1.

See: [Description](#)

### Java 2 Platform Packages

<a href="#">java.applet</a>	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
<a href="#">java.awt</a>	Contains all of the classes for creating user interfaces and for painting graphics and images.
<a href="#">java.awt.color</a>	Provides classes for color spaces.
<a href="#">java.awt.datatransfer</a>	Provides interfaces and classes for transferring data between and within applications.
<a href="#">java.awt.dnd</a>	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
<a href="#">java.awt.event</a>	Provides interfaces and classes for dealing with different types of events fired by AWT components.
<a href="#">java.awt.font</a>	Provides classes and interface relating to fonts.

Document : Terminé (1.893 s)

# JavaDoc : exemple

The screenshot shows a Netscape browser window titled "String (Java 2 Platform SE v1.4.1) - Netscape". The address bar shows "String (Java 2 Platform SE v1.4.1)". The left sidebar contains a navigation menu for "Java™ 2 Platform Std. Ed. v1.4.1", including links for "All Classes", "Packages" (java.applet, java.awt, java.awt.color), and a list of "All Classes" (AbstractAction, AbstractBorder, AbstractButton, AbstractCellEditor, AbstractCollection, AbstractColorChooserPanel, AbstractDocument, AbstractDocument.Attribute, AbstractDocument.Content, AbstractDocument.Element, AbstractInterruptibleChannel, AbstractLayoutCache, AbstractLayoutCache.Node, AbstractList, AbstractListModel, AbstractMap, AbstractMethodError, AbstractPreferences, AbstractSelectableChannel, AbstractSelectionKey). The main content area displays the "compareTo" method signature: `public int compareTo(String anotherString)`. Below the signature is a detailed description of the method's behavior, comparing two strings lexicographically based on Unicode values. It explains that the result is a negative integer if the current string precedes the argument, a positive integer if it follows, and zero if they are equal. It also defines lexicographic ordering and provides the formula for the result: `this.charAt(k) - anotherString.charAt(k)` for differing characters and `this.length() - anotherString.length()` for strings of different lengths. The "Parameters" section lists "anotherString" as the String to be compared. The "Returns" section describes the integer result based on lexicographic comparison. The "Throws" section lists "NullPointerException" if the argument is null.

**String (Java 2 Platform SE v1.4.1) - Netscape**

Fichier Edition Afficher Aller à Signets Outils Fenêtre Aide

Messagerie Accueil My Netscape.fr Recherche Shopping Signets

String (Java 2 Platform SE v1.4.1)

**Java™ 2 Platform Std. Ed. v1.4.1**

[All Classes](#)

**Packages**

[java.applet](#)

[java.awt](#)

[java.awt.color](#)

**All Classes**

[AbstractAction](#)

[AbstractBorder](#)

[AbstractButton](#)

[AbstractCellEditor](#)

[AbstractCollection](#)

[AbstractColorChooserPanel](#)

[AbstractDocument](#)

[AbstractDocument.Attribute](#)

[AbstractDocument.Content](#)

[AbstractDocument.Element](#)

[AbstractInterruptibleChannel](#)

[AbstractLayoutCache](#)

[AbstractLayoutCache.Node](#)

[AbstractList](#)

[AbstractListModel](#)

[AbstractMap](#)

[AbstractMethodError](#)

[AbstractPreferences](#)

[AbstractSelectableChannel](#)

[AbstractSelectionKey](#)

**compareTo**

```
public int compareTo(String anotherString)
```

Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this `String` object is compared lexicographically to the character sequence represented by the argument string. The result is a negative integer if this `String` object lexicographically precedes the argument string. The result is a positive integer if this `String` object lexicographically follows the argument string. The result is zero if the strings are equal; `compareTo` returns 0 exactly when the `equals(Object)` method would return true.

This is the definition of lexicographic ordering. If two strings are different, then either they have different characters at some index that is a valid index for both strings, or their lengths are different, or both. If they have different characters at one or more index positions, let *k* be the smallest such index; then the string whose character at position *k* has the smaller value, as determined by using the < operator, lexicographically precedes the other string. In this case, `compareTo` returns the difference of the two character values at position *k* in the two string -- that is, the value:

```
this.charAt(k) - anotherString.charAt(k)
```

If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string. In this case, `compareTo` returns the difference of the lengths of the strings -- that is, the value:

```
this.length() - anotherString.length()
```

**Parameters:**

anotherString - the `String` to be compared.

**Returns:**

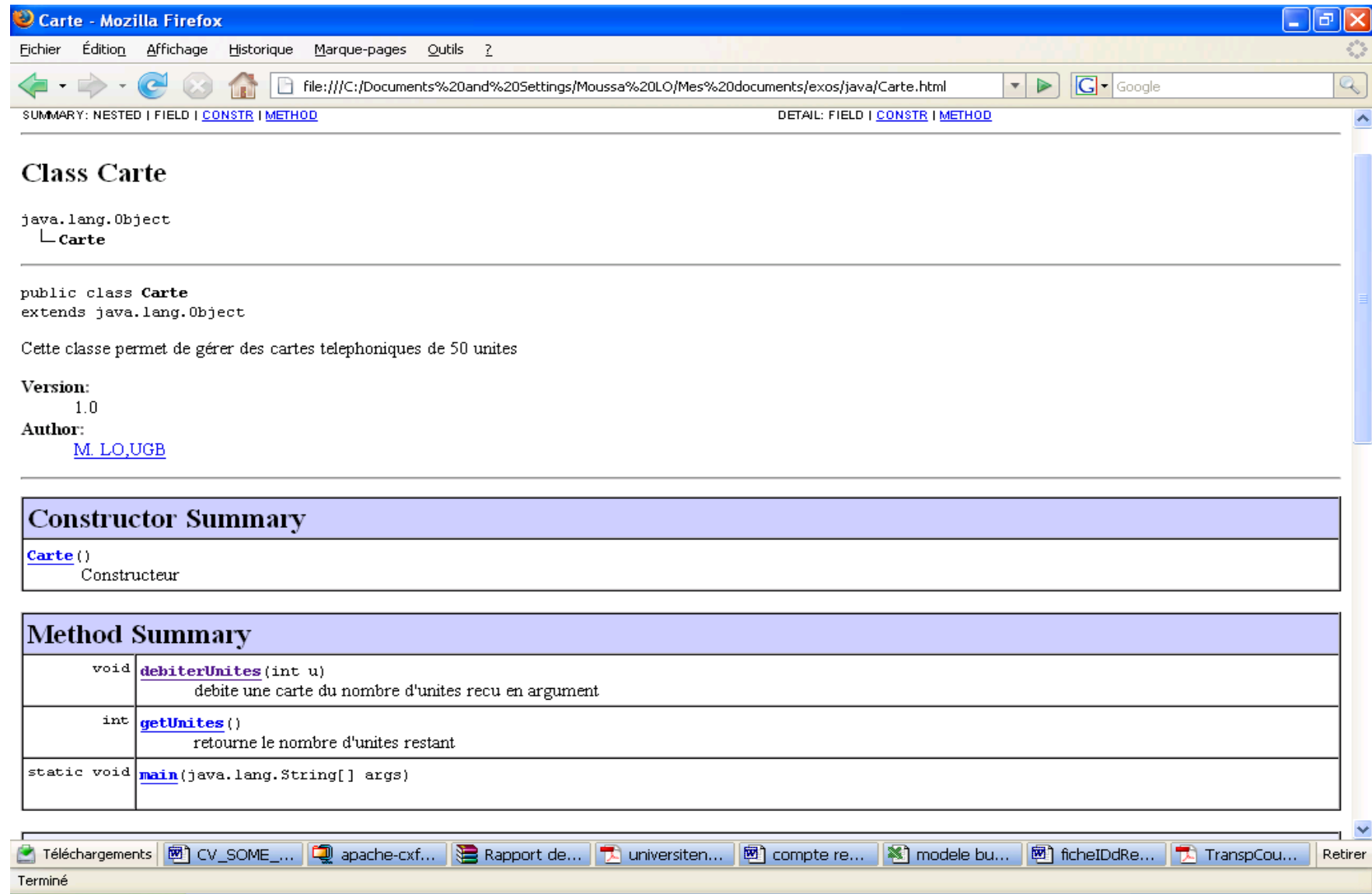
the value 0 if the argument string is equal to this string; a value less than 0 if this string is lexicographically less than the string argument; and a value greater than 0 if this string is lexicographically greater than the string argument.

**Throws:**

[NullPointerException](#) - if anotherString is null.

Document : Terminé (0.781 s)

# JavaDoc : exemple



The screenshot shows a Mozilla Firefox browser window with the title 'Carte - Mozilla Firefox'. The address bar displays the file path: `file:///C:/Documents%20and%20Settings/Moussa%20LO/Mes%20documents/exos/java/Carte.html`. The page content is a JavaDoc document for the `Carte` class.

**Class Carte**

`java.lang.Object`  
└─ `Carte`

---

`public class Carte`  
`extends java.lang.Object`

Cette classe permet de gérer des cartes telefoniques de 50 unites

**Version:**  
1.0

**Author:**  
[M. LO,UGB](#)

---

**Constructor Summary**

<a href="#">Carte()</a>
Constructeur

---

**Method Summary**

void	<a href="#">debiterUnites</a> (int u) debite une carte du nombre d'unites reçu en argument
int	<a href="#">getUnites</a> () retourne le nombre d'unites restant
static void	<a href="#">main</a> (java.lang.String[] args)

The browser's taskbar at the bottom shows several open applications: 'Téléchargements', 'CV\_SOME...', 'apache-cxf...', 'Rapport de...', 'universiten...', 'compte re...', 'modele bu...', 'ficheIDdRe...', 'TranspCou...', and 'Retirer'. The status bar at the very bottom indicates 'Terminé'.



# JavaDoc

- Outil fourni avec le JDK pour permettre la génération d'une documentation technique à partir du code source.
- La documentation générée est par défaut en HTML.
- Il est possible de changer le format de la doc. (pour, par exemple, générer du RTF ou du XML).
- *Exemple :*

```
/**
```

```
Ceci est un commentaire javadoc
```

```
*/
```

## *Fichiers HTML JavaDoc (1/2)*

La documentation générée contient les fichiers suivants :

- un *fichier HTML par classe* ou *interface* qui contient le détail chaque élément de la classe ou interface.
- un *fichier HTML par package* qui contient un résumé du contenu du package.
- ...

## *Fichiers HTML JavaDoc (2/2)*

Ainsi que des fichiers HTML tels que :

- *overview-summary.html*
- *overview-tree.html*
- *deprecated-list.html*
- *overview-frame.html*
- *all-classe.html*
- *package-summary.html* (pour chaque package)
- *package-frame.html* (pour chaque package)
- *package-tree.html* (pour chaque package)

## *Tags JavaDoc*

- **Tags prédéfinis** pour fournir des **infos précises** sur des *éléments particuliers* de l'application :  
auteur (*@author*), paramètres (*@param*), valeurs de retour (*@return*), etc.
- Ces tags commencent tous par @.

## *Tags JavaDoc : exemple*

```
/**
```

```
 * Cette classe permet de gérer des palindromes
```

```
 * @version 1.0
```

```
 * @author <a href="mailto:lom@ugb.sn">M.  
          LO,UGB</a>
```

```
 */
```

## *Qqs tags JavaDoc prédéfinis*

<i>Tag</i>	<i>Rôle</i>	<i>Elt concerné</i>
<b>@author</b>	préciser l'auteur	classe et interface
<b>@version</b>	préciser le n° de version	classe et interface
<b>@param</b>	préciser un paramètre	constructeur et méthode
<b>@return</b>	préciser la valeur de retour	méthode
<b>@exception</b>	préciser une exception qui peut être levée	méthode
<b>{@link}</b>	insérer un lien vers un élément de la doc dans n'importe quel texte	package, classe, interface, méthode, champ

## *JavaDoc : tag @author*

- Permet de préciser le nom du ou des auteurs du code.
- Doit être utilisé uniquement pour un élément de type classe ou interface.
- *Syntaxe :*

**@author nom\_de\_l\_auteur**

## *JavaDoc : tag @author*

- Génère une entrée **Author**: avec le nom de l'auteur dans la documentation.
- Par défaut, ce tag **n'est pas pris en compte** par javadoc.
- Pour qu'il soit pris en compte il faut utiliser l'option **-author**.
- Pour préciser plusieurs noms, il faut :
  - ✓ les séparer par une virgule
  - ✓ ou utiliser plusieurs tags chacun contenant un nom.



## *JavaDoc : tag @author*

```
package ugb.sat.essai;
```

```
// import
```

```
/**
```

```
 * description de la classe
```

```
 *
```

```
 * @version 1.0
```

```
 * @author <a href="mailto:lom@ugb.sn">M LO, UGB</a>
```

```
 *
```

```
 */
```

## *JavaDoc : @param*

- Permet de fournir des informations sur les paramètres.
- Doit être utilisé uniquement pour un élément de type constructeur ou méthode.
- *Syntaxe :*

**@param** **nom\_parametre** **description\_parametre**

*Exemple :* **@param** **tva** **taux de TVA**

## *JavaDoc : @param*

- Génère une ligne dans la section **Parameters**: avec son nom et sa description dans la documentation.
- La description peut tenir sur plusieurs lignes.
- Il faut utiliser un tag **@param** pour chaque paramètre en respectant l'ordre des paramètres dans la signature.

## *JavaDoc : tag @return*

- Permet de préciser la valeur de retour.
- Doit être utilisé uniquement pour un élément de type méthode qui renvoie une valeur.
- *Syntaxe :*

**@return description\_retour**

- Ce tag génère une ligne dans la section **Returns:** avec sa description dans la documentation.
- La description peut tenir sur plusieurs lignes.

## *JavaDoc : exemple*

```
/**
 * permet de comparer deux entiers
 * @param x    premier entier
 * @param y    deuxieme entier
 * @return     true si les 2 entiers sont
egaux et false sinon
 */
public boolean comparer (int x, int y) {
    ...
}
```

## *JavaDoc : tag @exception*

- Permet de fournir des informations sur une exception qui peut être levée.

- *Syntaxe :*

**@exception** **nom\_classe** **description\_exception**

- Il faut utiliser un tag **@exception** pour chaque exception déclarée dans la signature de la méthode.
- Le tag **@throws** est équivalent au tag **@exception**

## *JavaDoc : exemple*

```
/**
 * permet d'empiler un nouvel element
 * @param x    entier à empiler
 * @exception ErreurPilePleine si pile pleine
 */
public void empiler (int x) throws ErreurPilePleine {
    if (sommet == taille_max)
        throw new ErreurPilePleine()
    else    elements[++sommet] = x ;
}
```

## *JavaDoc : tag @version*

- Permet de préciser la version d'un élément.
- Doit être utilisé uniquement pour un élément de type classe ou interface.
- *Syntaxe :*

**@version description\_de\_la\_version**

- Génère une entrée Version: avec la description de la version dans la documentation.
- Par défaut, ce tag **n'est pas pris en compte** par javadoc.
- il faut utiliser l'option **-version**.



# JavaDoc : exemple

```
/**  
 * Cette classe permet de gérer des cartes telephoniques de 50 unites  
 * @version 1.0  
 * @author <a href="mailto:lom@ugb.sn">M. LO,UGB</a>  
 */
```

```
public class Carte  
{
```

```
    private int unites;
```

```
    /**
```

```
     * Constructeur
```

```
     */
```

```
    public Carte(){ this.unites = 5000; }
```

```
    /**
```

```
     * debite une carte du nombre d'unites reçu en argument
```

```
     * @param u  nombre d'unites a debiter
```

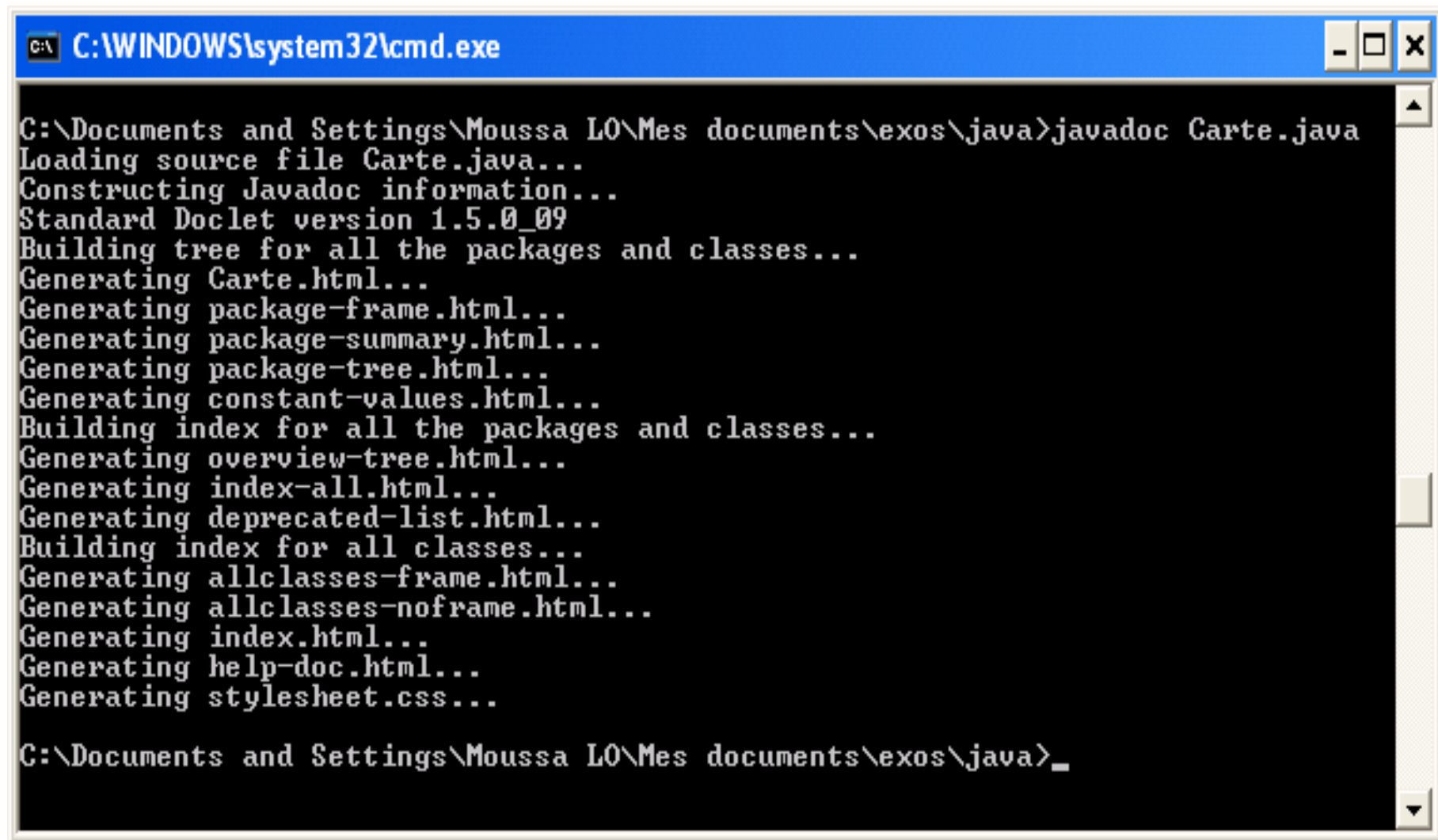
```
     */
```

```
    public void debiterUnites(int u){  
        this.unites -= u;
```

```
    }
```

```
    ...
```

## *JavaDoc : exemple*



```
C:\WINDOWS\system32\cmd.exe

C:\Documents and Settings\Moussa LO\Mes documents\exos\java>javadoc Carte.java
Loading source file Carte.java...
Constructing Javadoc information...
Standard Doclet version 1.5.0_09
Building tree for all the packages and classes...
Generating Carte.html...
Generating package-frame.html...
Generating package-summary.html...
Generating package-tree.html...
Generating constant-values.html...
Building index for all the packages and classes...
Generating overview-tree.html...
Generating index-all.html...
Generating deprecated-list.html...
Building index for all classes...
Generating allclasses-frame.html...
Generating allclasses-noframe.html...
Generating index.html...
Generating help-doc.html...
Generating stylesheet.css...

C:\Documents and Settings\Moussa LO\Mes documents\exos\java>_
```

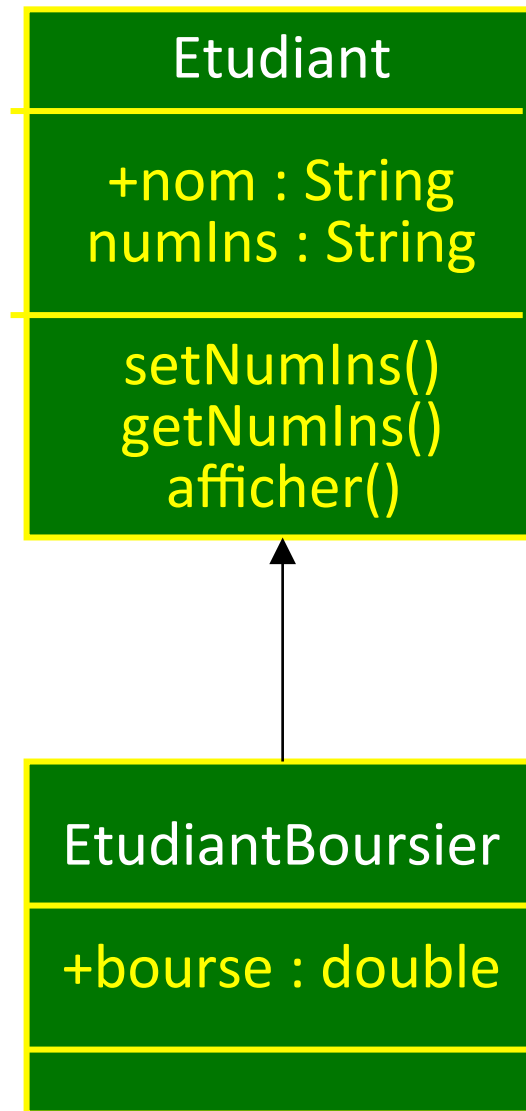


# HERITAGE, Classes abstraites, Interfaces

## *L'héritage*

- L'héritage permet de définir une nouvelle classe à partir d'une (ou plusieurs) classe(s) existante(s).
- Les classes sont souvent organisées en hiérarchies; toutes les classes Java héritent de la classe **java.lang.Object**.
- Java n'autorise que l'héritage simple.
- L'héritage multiple est “remplacé” par la notion d'**interface**.

## *L'héritage simple : exemple*



## *L'héritage simple : exemple*

```
public class EtudiantBoursier extends Etudiant {  
  
    public double bourse;  
  
    public EtudiantBoursier(String nom, double bourse) {  
        super(nom); //appel au constructeur Etudiant(String)  
        this.bourse = bourse;  
    }  
    public EtudiantBoursier(String nom) {  
        this(nom, null); //appel au constructeur précédant  
    }  
  
    public void afficher() { // redefinition  
        System.out.println("etudiant :"+ this.nom);  
        System.out.println("bourse :"+ this.bourse);  
    }  
}
```

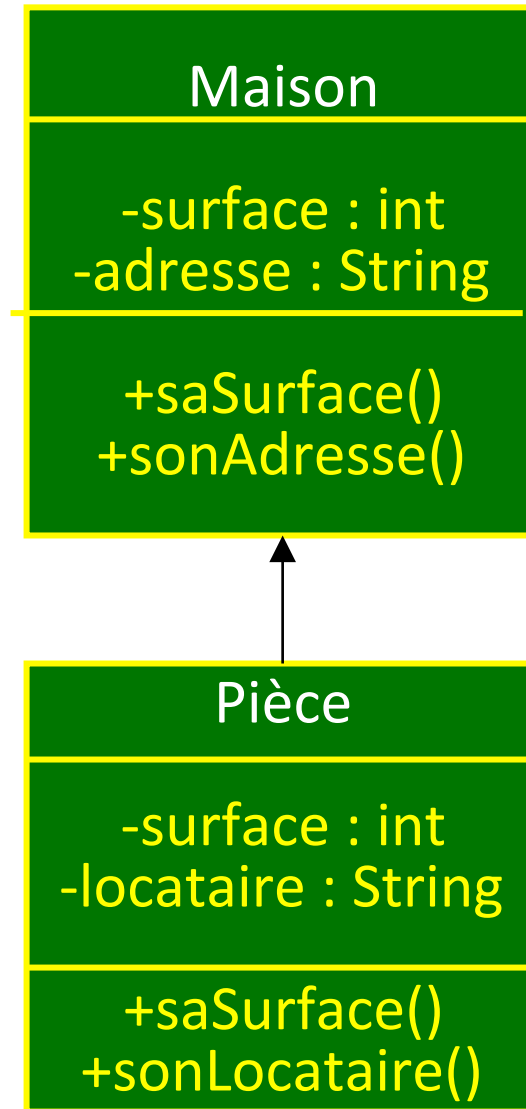
## *La pseudo-variable super*

- En faisant hériter une classe d'une autre, on peut définir une méthode dont l'identificateur est le même que celui de sa classe mère.

➔ masquage de la méthode de la classe mère.

- La pseudo-variable **super** permet d'accéder à la méthode masquée de la classe mère.

## *La pseudo-variable super : exemple*





## La classe Object

- Toutes les classes héritent de la classe Object qui contient certaines propriétés et méthodes intéressantes permettant, par exemple :
  - ✓ Connaître la classe d'un objet : **getClass()**
  - ✓ Comparer des objets : **equals()**
  - ✓ Afficher des objets : **toString()**

- Exemple :

```
public class Fraction {  
    public int num, den ;  
    public Fraction (int n, int d) { this.num=n; this.den=d;}  
    public String toString() {  
        return this.num + "/" + this.den;  
    }  
}
```

## *Exemple d'héritage : compte et compte d'épargne*

(1/3)

```
class Compte {  
  
    private String id;  
    private float solde;  
  
    public Compte (String id, float depot){  
        this.id = id;  
        this.solde = depot;  
    }  
    public String getId (){  
        return this.id;  
    }  
    public float getSolde (){  
        return this.solde;  
    }  
}
```

## *Exemple d'héritage : compte et compte d'épargne (2/3)*

```
class CompteEpargne extends Compte {  
    private float taux;  
    private int annees;  
    public CompteEpargne (String id, float depot, float taux){  
        super (id, depot);  
        this.taux = taux;  
    }  
    public void setAnnees (int annees){  
        if (annees >= 0) this.annees = annees;  
    }  
    public int getAnnees() { return this.annees; }  
    public float getTaux() { return this.taux; }  
    public float getSolde () {  
        float solde = super.getSolde();  
        for (int i=0; i<this.annees; i++) solde *= 1 + this.taux;  
        return solde;  
    }  
}
```

## Exemple d'héritage : compte et compte d'épargne (3/3)

```
class CalculInterets{
    public static void main(String arg[]){
        Compte compte1 = new Compte("A01", 100000f);
        CompteEpargne compte2 = new CompteEpargne("E99", 100000f,
0.1f);
        compte2.setAnnees(5);

        Compte c;
        String s = "L'argent qui dort ne rapporte rien:";
        c = compte1;
        s += "\n solde du compte n° " + c.getId() + ":" + c.getSolde();

        c = compte2;
        s += "\n solde du compte n° " + c.getId() + ":" + c.getSolde();
        javax.swing.JOptionPane.showMessageDialog(null,s);
    }
}
```

## *Les classes abstraites (1/2)*

- Elles permettent de définir l'interface des méthodes.
- Elles contiennent au moins une **méthode abstraite**.
- *Exemple :*

```
abstract class FormeGeometrique {  
    String nom;  
    abstract double perimetre();  
    abstract double surface();  
    public getNom() { return nom; }  
}
```

## *Les classes abstraites (2/2)*

- Une **méthode abstraite** est précédée du mot-clé **abstract** et ne possède pas de corps.
- Elles ne peuvent pas être instanciées.
- Elles doivent être dérivées en sous-classes fournissant une implémentation à toutes les méthodes abstraites.
- Elles sont définies par l'introduction du mot-clé **abstract**.

## *Les interfaces : généralités (1/2)*

- Une interface Java contient une liste de méthodes abstraites que doit implémenter une classe pour rendre un service.
- **Classes abstraites** dont l'instanciation serait sans intérêt. Elles ne peuvent donc pas être instanciées.

- *Exemple :*

```
public interface FormeGeometrique {  
    public double perimetre();  
    public double surface();  
}
```

## *Les interfaces : généralités (2/2)*

- Lorsqu'une classe implémente une interface, elle doit implémenter toutes les méthodes définies dans l'interface.
- Une classe peut implémenter **une ou plusieurs** interfaces.
- *Exemple :*

```
public class Rectangle implements FormeGeometrique {  
    public double larg, long;  
    ...  
    public double perimetre() {return 2*(larg+long);}  
    public double surface() {return larg*long;}  
}
```

- Une interface peut hériter d'autres interfaces (héritage simple).



## *Les interfaces : déclaration*

- Une interface Java se déclare comme une classe en faisant précéder son identificateur du mot-clé **interface**.

```
public interface UneInterface {  
    // déclaration des champs et des méthodes  
}
```

- Tous les champs sont des constantes dont les modificateurs sont implicitement *public static final*
- Toutes les méthodes sont publiques et non implémentées; elles utilisent implicitement des modificateurs *public abstract* et sont suivies d'un ;.

## *Les interfaces : implémentation*

- Une interface **doit être implémentée** par une ou plusieurs classes.
- Une classe peut implémenter une ou plusieurs interfaces.
- Lorsqu'une classe implémente une interface, elle doit implémenter **toutes les méthodes** définies dans l'interface.

```
class UneClasse implements UneInterface {
```

```
    // champs et méthodes de la classe UneClasse
```

```
    // et implémentation des méthodes de l'interface UneInterface
```

```
}
```

## *Les interfaces : exemple*

```
public interface FormeGeometrique {  
    public double perimetre();  
    public double surface();  
}
```

```
public class Rectangle implements FormeGeometrique {  
    public double larg, long;  
    public Rectangle(double long, double larg) {  
        this.long=long; this.larg=larg;  
    }  
    public double perimetre() {return 2*(larg+long);}  
    public double surface() {return larg*long;}  
}
```

## *Les interfaces : exemple*

```
public class Carre implements FormeGeometrique {
    public double cote;
    public Carre(double cote) {this.cote = cote;}
    public double perimetre() {return 4*cote;}
    public double surface() {return cote*cote;}
}

public class Cercle implements FormeGeometrique {
    public final static double PI = 3.1416;
    public double rayon;
    public Cercle (double rayon) {this.rayon = rayon;}
    public double perimetre() {return 2*PI*rayon;}
    public double surface() {return PI*rayon*rayon;}
    public void afficher() {
        System.out.println("cercle de rayon"+rayon); }
}
```

## *Les interfaces : exemple*

```
public class Figures {  
    public static void main (String[] args) {  
        FormeGeometrique[] formes;  
        formes = new FormeGeometrique[3];  
        formes[0] = new Cercle(10);  
        formes[1] = new Carre(4);  
        formes[2] = new Rectangle(5,8);  
        for (int i=0;i<formes.length;i++){  
            System.out.print("surface de la forme"+i+" : ");  
            System.out.println(formes[i].surface());  
        }  
    }  
}
```