Domaine Sciences et Technologies
Master Bio-informatique : DLAD

Aix*Marseille université
Socialement engagée

Année 2023-24

Programmation avancée : Documentation
Code UE : SBICU05

Documentation en Java : Javadoc

Introduction

Dans tout développement informatique, il est souvent primordial de documenter correctement son code afin qu'il puisse être utilisable par d'autres développeurs. L'idée de la documentation est de détailler certains points (par exemple le comportement d'une méthode pour des cas spécifiques pour des valeurs précises de paramètres) et donner un résumé du fonctionnement des éléments du code de sorte qu'une personne puisse réutiliser les éléments documentés sans regarder le code (dont il n'a pas forcément l'accès). La documentation peut aussi être utile pour le développeur du code lui-même, car elle permet de donner des spécifications sur le comportement que doit avoir le code. Néanmoins, cette utilisation peut être contre-productive, car source de duplication d'informations. En effet, le comportement d'une méthode peut par exemple changer lorsque le développeur réécrit son code. Par conséquent, le développeur doit faire deux fois les changements (une fois dans son code et une autre fois dans sa documentation). C'est pour cela que généralement, il ne faut documenter son code que si cela a une utilité directe. Normalement, le code doit se suffire à lui-même, c'est-à-dire qu'il doit être directement lisible sans ajout de documentation ou de commentaires. Bien sûr, comme toutes les règles en développement logiciel, cette règle n'est pas absolue et autorise donc des exceptions.

Présentation de Javadoc

L'outil de base en Java pour documenter son code s'appelle Javadoc. La Javadoc est fournie avec le JDK afin de permettre la génération d'une documentation technique directement à partir du code source (et donc des fichiers `.java`). L'intérêt de ce système est de conserver dans le même fichier le code source et les éléments de la documentation qui lui sont associés. Cet outil utilise des commentaires dans un format spécifique qu'on va détailler. Il est notamment utilisé pour la génération de la documentation du JDK. Cette documentation contient :

- une description détaillée pour chaque classe et ses membres `public` et `protected`;
- un ensemble de listes (liste des classes, hiérarchie des classes, liste des éléments *deprecated* et un index général);
- des références croisées et une navigation entre ces différents éléments.

Les commentaires pour Javadoc suivent des règles précises. Le format de ces commentaires commence par `/**` et se termine par `*/`. Ils peuvent contenir un texte libre en HTML et des *tags* (mots-clés définis par Javadoc) précédés par `@`. Le commentaire peut être sur plusieurs lignes. Dans chaque ligne, les espaces qui précèdent le premier caractère `*` de la ligne du commentaire ainsi que le caractère lui-même sont ignorés. Ceci permet d'utiliser le caractère `*` pour aligner le contenu du commentaire.

Le format général de ces commentaires est le suivant :

```
/**
 * Description
 *
 * @tag1
 * @tag2
 */
```

Le texte du commentaire doit être au format HTML : les balises HTML peuvent donc être utilisées pour enrichir le formatage de la documentation. Les *tags* prédéfinis par Javadoc permettent de fournir des informations plus précises sur des composants particuliers de l'élément (auteurs, paramètres, valeurs de retour, ...). Ces *tags* sont définis pour un ou plusieurs types d'éléments. Il existe deux types de tags :

— *Inline tag* de la forme `{@tag}` et
— *Block tag* de la forme `@tag`.

Tags Javadoc

Il existe de nombreux *tags* définis par le format de la javadoc. Les principaux sont les suivants :

Tag	Description
`@author`	pour préciser l'auteur de la fonctionnalité
`@deprecated`	indique que l'attribut, la méthode ou la classe est dépréciée
`@return`	pour décrire la valeur de retour d'une méthode
`@param p`	pour décrire un paramètre `p` d'une méthode
`{@code literal}`	formate `literal` en code
`{@link reference}`	permet de créer un lien
`@exception e`	indique une exception `e` qui peut être levée par la méthode
`@since number`	permet de préciser depuis quelle version l'élément a été ajouté

Le *tag* `@author`

Le tag `@author` permet de préciser le ou les auteurs d'un élément. La syntaxe de ce tag est la suivante :

`@author nom des auteurs/autrices`

Exemple d'utilisation:

```
/**
 * @author Paul Calcul, Jean-Michel Bruitages
 */
```

Le *tag* `@deprecated`

Le tag `@deprecated` permet de préciser qu'un élément ne devrait plus être utilisé même s'il fonctionne toujours : il permet donc de donner des précisions sur un élément déprécié (*deprecated*). La syntaxe de ce tag est la suivante :

`@deprecated texte`

Il est recommandé de préciser depuis quelle version l'élément est déprécié et de fournir dans le texte libre une description de la solution de remplacement, si elle existe, ainsi qu'un lien vers une entité de substitution. Exemple d'utilisation pour le constructeur de `Float` :

```
1      /**
2       * Constructs a newly allocated {@code Float} object that
3       * represents the floating-point value of type {@code float}
4       * represented by the string. The string is converted to a
5       * {@code float} value as if by the {@code valueOf} method.
6       *
7       * @param   s   a string to be converted to a {@code Float}.
8       * @throws      NumberFormatException if the string does not contain a
9       *              parsable number.
10      *
11      * @deprecated
12      * It is rarely appropriate to use this constructor.
13      * Use {@link #parseFloat(String)} to convert a string to a
14      * {@code float} primitive, or use {@link #valueOf(String)}
15      * to convert a string to a {@code Float} object.
16      */
17     public Float(String s) throws NumberFormatException {
18         /* ... */
19     }
```

On peut voir dans l'exemple ci-dessus que le constructeur de `Float` est déprécié (donc normalement son utilisation ne sera plus possible dans une version futures). La documentation conseille d'utiliser d'autres méthodes à la place du constructeur.

Le *tag* `@throws`

Le *tag* `throws` permet de documenter une exception levée par la méthode ou le constructeur décrit par le commentaire. Il est possible d'ajouter du texte afin de décrire les conditions de levée de l'exception. La syntaxe d'utilisation de ce *tag* est la suivante :

`@throws nom_exception description`

Le *tag* est suivi du nom de l'exception puis d'une courte description des raisons de la levée de cette dernière. Il faut utiliser autant de *tag* `@throws` qu'il y a d'exceptions. Ce tag doit être utilisé uniquement pour une méthode ou un constructeur. Exemple d'utilisation pour la méthode `add` de `List` :

```
1      /**
2       * Appends the specified element to the end of this list (optional
3       * operation).
4       *
5       * <p>Lists that support this operation may place limitations on what
6       * elements may be added to this list.  In particular, some
7       * lists will refuse to add null elements, and others will impose
8       * restrictions on the type of elements that may be added.  List
9       * classes should clearly specify in their documentation any restrictions
10      * on what elements may be added.
11      *
```

```
12        * @param e element to be appended to this list
13        * @return {@code true} (as specified by {@link Collection#add})
14        * @throws UnsupportedOperationException if the {@code add} operation
15        *         is not supported by this list
16        * @throws ClassCastException if the class of the specified element
17        *         prevents it from being added to this list
18        * @throws NullPointerException if the specified element is null and this
19        *         list does not permit null elements
20        * @throws IllegalArgumentException if some property of this element
21        *         prevents it from being added to this list
22        */
23       void add(int index, E element);
```

Dans l'exemple ci-dessous, il y a la description des différentes exceptions pouvant être levées par la méthode `add` de `List`

Le tag `@param`

Le tag `@param` permet de documenter un paramètre d'une méthode ou d'un constructeur. La syntaxe de ce tag est la suivante :

`@param name texte de description`

Ce *tag* est suivi du nom du paramètre (sans le type) puis d'une courte description de ce dernier. Il faut utiliser autant de *tag* `@param` que de paramètres dans la signature de l'entité concernée. Par convention les paramètres doivent être décrits dans leur ordre dans la signature de la méthode ou du constructeur. Exemple d'utilisation pour la méthode `of` de `List` :

```
1    /**
2     * Returns an unmodifiable list containing nine elements.
3     *
4     * See <a href="#unmodifiable">Unmodifiable Lists</a> for details.
5     *
6     * @param <E> the {@code List}'s element type
7     * @param e1 the first element
8     * @param e2 the second element
9     * @param e3 the third element
10    * @param e4 the fourth element
11    * @param e5 the fifth element
12    * @param e6 the sixth element
13    * @param e7 the seventh element
14    * @param e8 the eighth element
15    * @param e9 the ninth element
16    * @return a {@code List} containing the specified elements
17    * @throws NullPointerException if an element is {@code null}
18    *
19    * @since 9
20    */
21   static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9){
22     /* ... */
23   }
```

On peut observer que dans l'exemple ci-dessus les 9 paramètres sont bien documentés dans l'ordre de la signature

de la méthode.

Le tag `@return`

Le tag `@return` permet de fournir une description de la valeur de retour d'une méthode qui en possède une. La syntaxe de ce tag est la suivante :

```
@return description de la valeur de retour
```

Il ne peut y avoir qu'un seul tag `@return` par commentaire : il doit être utilisé uniquement pour un élément de type méthode qui renvoie une valeur.

Exemple d'utilisation pour la méthode `size` de `List` :

```
1      /**
2       * Returns the number of elements in this list. If this list contains
3       * more than {@code Integer.MAX_VALUE} elements, returns
4       * {@code Integer.MAX_VALUE}.
5       *
6       * @return the number of elements in this list
7       */
8      int size();
```

Le tag `{@link}`

Ce *tag inline* permet de créer un lien vers un autre élément de la documentation. La syntaxe de ce tag est la suivante : `{@link package.class#membre texte}`

Le *tag* `{@code}`

Ce *tag inline* permet d'afficher un texte dans des tags `<code> ... </code>` qui ne sera pas interprété comme de l'HTML. La syntaxe de ce tag est la suivante : `{@code texte}`

Le *tag* `@since`

Le *tag* `@since` permet de préciser un numéro de version de la classe ou de l'interface à partir de laquelle l'élément décrit est disponible. Ce *tag* peut être utilisé avec tous les éléments. La syntaxe de ce *tag* est la suivante :

```
@since texte
```

Exemple : `@since 2.0`

Documentation des *packages*

En plus de la documentation contenue dans les fichiers de classes et d'interfaces, il est possible de rajouter dans chaque package un fichier `package-info.java` qui sera utilisé par Javadoc pour produire la documentation du *package*. Par exemple, le fichier `package-info.java` du *package* `applet` d'`awt` est le suivant :

```
1  /**
2   * Provides the classes necessary to create an
3   * applet and the classes an applet uses
```

```
 4   * to communicate with its applet context.
 5   * <p>
 6   * The applet framework involves two entities:
 7   * the applet and the applet context.
 8   * An applet is an embeddable window (see the
 9   * {@link java.awt.Panel} class) with a few extra
10   * methods that the applet context can use to
11   * initialize, start, and stop the applet.
12   *
13   * @since 1.0
14   * @see java.awt
15   */
16  package java.lang.applet;
```

Génération de documentation

Pour générer la documentation

Pour générer la documentation, il faut donc invoquer l'outil Javadoc. Javadoc recrée à chaque appel la totalité de la documentation. Par défaut, Javadoc la documentation au format HTML, mais il est possible de générer la documentation dans d'autres formats comme du RTF ou du XML.

Pour appeler l'outil Javadoc, on peut passer par le terminal en appelant la commande `javadoc package1 package2 ...` où `package1`, `package2`, ... sont les noms des *packages* dont on souhaite générer la documentation. Sous IntelliJ IDEA, il faut passer par le menu `Tools` puis `generate Javadoc` en choisissant le répertoire dans lequel vous souhaitez sauvegarder la documentation. Vous pouvez aussi utiliser le moteur de production `gradle` soit par le terminal en appelant la commande `gradle javadoc` à la racine du projet ou bien en passant par le menu dédié d'IntelliJ IDEA (situé à droite) avec la tâche `javadoc` dans `documentation`. Si vous utilisez Gradle pour générer la documentation, celle-ci sera placée dans le répertoire `build/doc/javadoc` du projet.

La génération de la documentation crée de nombreux fichiers et des répertoires pour structurer la documentation au format HTML. Les fichiers les plus importants sont :

— Un fichier HTML par classe ou interface qui contient le détail de chaque élément de la classe ou de l'interface. Le fichier a le nom de la classe/interface et se trouve dans le répertoire de la classe ou de l'interface.
— Un fichier HTML par package qui contient un résumé du contenu du package. Le fichier se trouve dans le répertoire du package et porte le nom `package-summary.html`.
— Un fichier `index.html` qui est la page principale de la documentation