UD06: Code documentation

1. Introduction

Documenting a program's code is adding enough information to explain what it does, point by point, so that not only do computers know what to do, but humans also understand what they are doing and why.

Because between what a program has to do and how it does it there is an impressive distance: all the hours that the programmer has spent developing a solution and writing it in the corresponding language for the computer to run blindly.

Documenting a program is not only an act of good work of the programmer for that of leaving the work finished. It is also a need that is only appreciated in its due magnitude when there are errors to repair or the program must be extended with new capabilities or adapted to a new scenario. There are two rules that should never be forgotten:

1. all programs have errors and discovering them is only a matter of time and that the program succeeds and is used frequently
2. all programs undergo modifications throughout their lives, at least all those that are successful

A program, if successful, will probably be modified in the future by the person who coded it or by another programmer. Thinking about this code review is why it is important that the program is understood: to be able to repair and modify it.

2. What needs to be documented?

We must add explanations to everything that is not obvious.

Do not repeat what you do, but explain why it is done

And that translates into:

- What is a class in charge of? a package?
- what does a method do?
- what is the expected use of a method?
- What is a variable or attribute used for?
- what is the expected use of an attribute?
- what algorithm are we using? Where did we get it?
- what limitations does the algorithm have? What are the limitations of the implementation?
- what should be improved... if there was time?

3. Types of comments

In Java we have three notations to introduce comments:

One-line comments:

- They start with the characters "//" and end with the line
- It is used to document code that we do not need to appear in the external documentation (the one generated by javadoc). these types of comments are used even when the comment takes up several lines, each of which begins with "//"

Multi-line comments:

- They begin with the characters "`/*`" and end with the characters "`*/`".
- Often used to remove code. It is common that obsolete code does not want it to disappear and we keep it "just in case". So that it does not run, it is commented.
  (In English it is usually called "comment out")

Javadoc comments:

- They begin with the characters "`/**`", can be extended along several lines (which probably begin with the character "*") and end with the characters "`*/`".
- They are used to generate external documentation.

4. When should I put a comment?

1. Always, at the beginning of each class
2. Always, at the beginning of each method
3. Always, before each class variable (static attributes, constants).
4. At the beginning of a code snippet that is not obvious.
5. When the program does something "weird"

Note: When a program is modified, the comments must be modified at the same time, other than
that the comments end up referring to an algorithm that we no longer use.

5. JavaDoc

The Java development package includes a tool, javadoc, to generate a set of web pages from the code files. This tool takes into consideration some feedback to generate a well-presented documentation of classes and class components (variables and methods).

Although javadoc does not help the understanding of the details of code, it does help the understanding of the architecture of the solution, which is not little. Javadoc is said to focus on the interface (API - Application Programming Interface) of Java classes and packages.

Javadoc highlights some comments, of which it requires a special syntax. They should start with "`/**`" and end with "`*/`", including a description and some special tags:

```
/**
* Descriptive part.
* Which may consist of several sentences or paragraphs.
*
* @label specific label text
*/
```

These special comments must appear just before the declaration of a class, attribute, or method in the same source code. The following sections detail the tags that javadoc interprets and then converts into documentation.

As a general rule, it should be noted that the first sentence (the text up to the first point) will receive a prominent treatment, so it must provide a concise and forceful explanation of the documented element. The other sentences will go into detail.

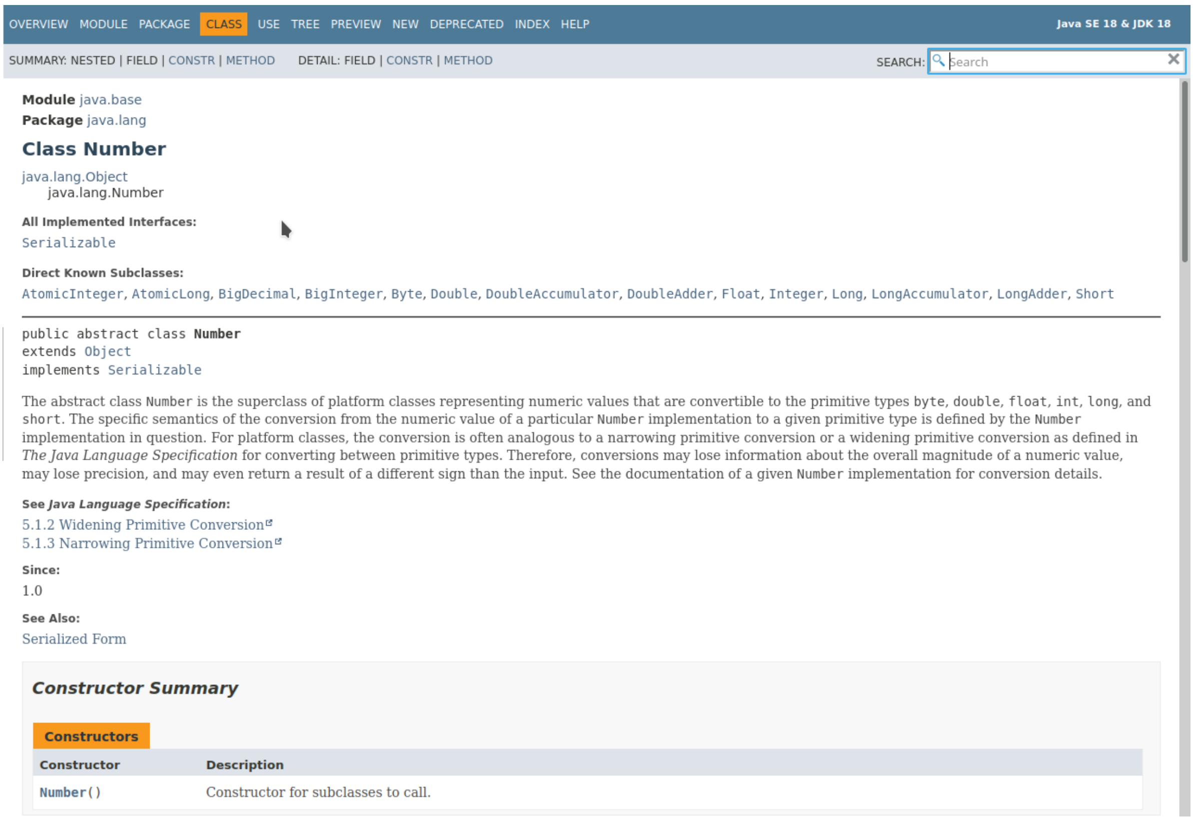
Label	Where to use	Objective
`@autor` name	Classes, interfaces,	Indicate the author of the code. A label is put on each author
`@version` VersionIdentifier	Classes, interfaces,	Information about version
`@since`	Classes, methods	From which version it is. Ex: from JDK 1.1
`@deprecated`	Classes, methods	To indicate that something should not be used anymore, it has become obsolete, even if it is maintained for compatibility. It is usually accompanied by what is to be used in your place.

Label	Where to use	Objective
`@see` ClassName	Classes, interfaces, methods, and attributes.	You will put the address to connect to this class in the documentation
`@see` ClassName#MethodName	Classes, interfaces, methods, and attributes.	You will put the address to connect with this method in the documentation.
`@return` description	Methods	To describe the values returned by each method and its type.
`@exception` name description	Methods	Exceptions that the method can raise. A label is put on each possible exception. They are usually sorted alphabetically.
`@param` name description	Methods	To describe the parameters, their use and their type. A label is put for each parameter

6. JavaDoc generation in NetBeans

Javadoc is a tool that processes documentation declarations and comments in a set of java source files. From that information it generates a set of html pages that describe the classes, methods, fields...

Here is an example of the documentation generated:


OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP
Java SE 18 & JDK 18
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD
SEARCH: Search

Module java.base
Package java.lang

Class Number

java.lang.Object
    java.lang.Number

All Implemented Interfaces:
`Serializable`

Direct Known Subclasses:
`AtomicInteger`, `AtomicLong`, `BigDecimal`, `BigInteger`, `Byte`, `Double`, `DoubleAccumulator`, `DoubleAdder`, `Float`, `Integer`, `Long`, `LongAccumulator`, `LongAdder`, `Short`

```
public abstract class Number
extends Object
implements Serializable
```

The abstract class `Number` is the superclass of platform classes representing numeric values that are convertible to the primitive types `byte`, `double`, `float`, `int`, `long`, and `short`. The specific semantics of the conversion from the numeric value of a particular `Number` implementation to a given primitive type is defined by the `Number` implementation in question. For platform classes, the conversion is often analogous to a narrowing primitive conversion or a widening primitive conversion as defined in *The Java Language Specification* for converting between primitive types. Therefore, conversions may lose information about the overall magnitude of a numeric value, may lose precision, and may even return a result of a different sign than the input. See the documentation of a given `Number` implementation for conversion details.

See *Java Language Specification*:
5.1.2 Widening Primitive Conversion
5.1.3 Narrowing Primitive Conversion

Since:
1.0

See Also:
Serialized Form

Constructor Summary

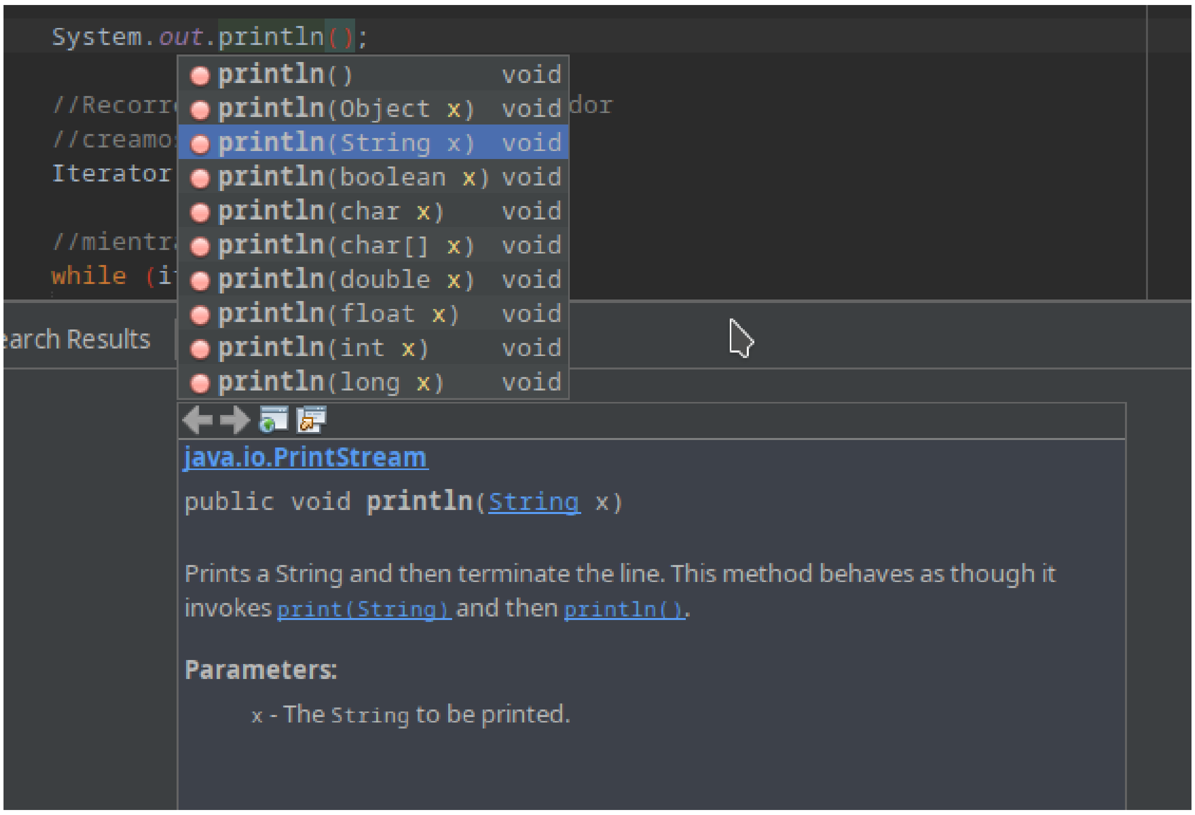
Constructors

Constructor	Description
`Number()`	Constructor for subclasses to call.

As you can see, the java documentation itself is created with the javadoc tool (the capture is the java API documentation). Some of the great advantages of this documentation is that:

- it is navigable: having a web format, you can click on the classes and navigate. In general it is more comfortable than a printed document because it avoids having to search pages
- it is generated automatically: it costs less to update, it does not generate formatting problems, it can be centralized...
- it is very easy to create: the programmer has to write, but with basically the same comments that he already adds to the code. The special javadoc language is very, very simple.

Most current IDEs have the API integrated into the help system of the environment, so while we write the code we are already shown related information:



```
System.out.println();

//Recorr
//creamo
Iterator

//mientr
while (i
```

java.io.PrintStream

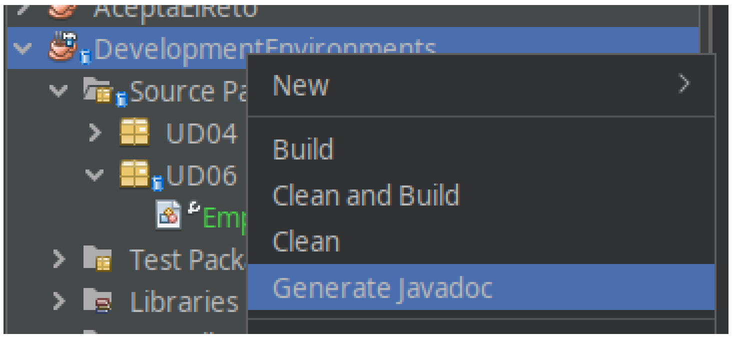
public void println(String x)

Prints a String and then terminate the line. This method behaves as though it invokes print(String) and then println().

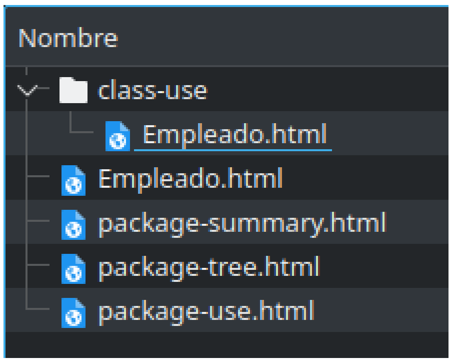
Parameters:

x - The String to be printed.

You could generate Javadoc right-clicking on your project and selecting `Generate Javadoc`:



This would create a structure folder inside your project folder `projectFolder\dist\javadoc\packageName\` with all the files needed to show documentation.



6.1. Comment Editing

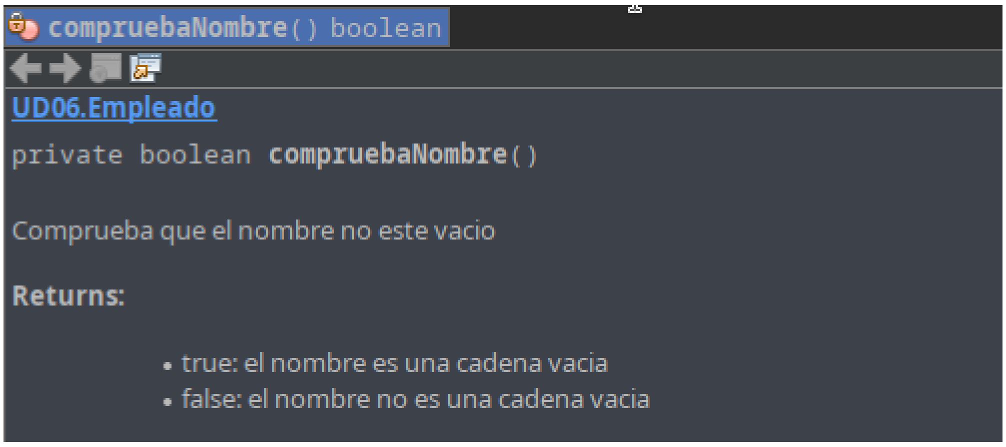
Comments written by the programmer are considered as text. However it is also optionally possible to write the comments using html internally. In this way we can create a presentation of the documentation more visual and comfortable. Obviously for this, the programmer must be familiar with the syntax of some basic html tags.

On the comments, it is also important to note that the main purpose of the documentation is to make it useful. The documentation of an API is used to learn (or remember) what and how a set of classes work. Therefore, when documenting it is sought that the information is practical, we must tell what is important and the more summarized it is the better.

HTML comments:

```
[...]
//Metodos privados
    /**
     * Comprueba que el nombre no este vacio
     * @return <ul>
     *  <li>true: el nombre es una cadena vacia</li>
     *  <li>false: el nombre no es una cadena vacia</li>
     *  </ul>
     */
    private boolean compruebaNombre(){
        if(nombre.equals("")){
            return false;
        }
        return true;
    }
[...]
```

Will be shown like this:



compruebaNombre() boolean

UD06.Empleado

private boolean **compruebaNombre**()

Comprueba que el nombre no este vacio

Returns:

- true: el nombre es una cadena vacia
- false: el nombre no es una cadena vacia

7. JavaDoc Sample

```java
/**
 * Clase Empleado
 *
 * Contiene informacion de cada empleado
 *
 * @author Fernando
 * @version 1.0
 */
public class Empleado {

    //Atributos

    /**
     * Nombre del empleado
     */
    private String nombre;
    /**
     * Apellido del empleado
     */
    private String apellido;
    /**
     * Edad del empleado
     */
    private int edad;
    /**
     * Salario del empleado
     */
    private double salario;

    //Metodos publicos

    /**
     * Suma un plus al salario del empleado si el empleado tiene mas de 40 años
     * @param sueldoPlus
     * @return <ul>
     *          <li>true: se suma el plus al sueldo</li>
     *          <li>false: no se suma el plus al sueldo</li>
     *          </ul>
     */
    public boolean plus (double sueldoPlus){
        boolean aumento=false;
        if (edad>40 && compruebaNombre()){
            salario+=sueldoPlus;
            aumento=true;
        }
        return aumento;
    }

    //Metodos privados
    /**
     * Comprueba que el nombre no este vacio
     * @return <ul>
     *  <li>true: el nombre es una cadena vacia</li>
     *  <li>false: el nombre no es una cadena vacia</li>
     *  </ul>
     */
    private boolean compruebaNombre(){
        if(nombre.equals("")){
            return false;
        }
        return true;
    }

    //Constructores
    /**
     * Constructor por defecto
     */
    public Empleado(){
        this ("", "", 0, 0);
    }

    /**
     * Constructor con 4 parametros
     * @param nombre nombre del empleado
     * @param apellido nombre del empleado
     * @param edad edad del empleado
     * @param salario salario del empleado
     */
    public Empleado(String nombre, String apellido, int edad, double salario){
        this.nombre=nombre;
        this.apellido=apellido;
        this.edad=edad;
        this.salario=salario;
    }
}
```

8. Information sources

- Wikipedia
- Uml Diagrams
- Code&Coke (Fernando Valdeón)
- Apuntes IES El Grao (Mª Isabel Barquilla?)
- Apuntes IOC (Marcel García)
- Apuntes José Luis Comesaña
- Apuntes IES Luis Vélez de Guevara 17-18 (José Antonio Muñoz Jiménez)
- Oracle provides material on how to document program interfaces: (How to write doc comments for the javadoc tool)