Why document?

- Civilization started with writing in 3100BC.
- We didn’t get any further since then. Let’s be frank.
  - We hate documentation, we love programming.
  - Documentation is boring.
  - But, fixing a program that is not properly documented is even worse.

Documentation and coding must go hand in hand

- Documentation and coding may diverge during the lifetime of a project.
  - Recall MUÇ:
    - “Try what is written in the document. If it does not work, try just the inverse.” (to Roy Küçükkateş and myself when SCSI termination did not work as explained in the document)
  - He was right, but I wish he was wrong.
    - There was documentation, but it was misleading.
  - There are tons of cases where there is no documentation.
    - Even you will not remember what you did after one month.

Commenting and documenting

- The problem: Documentation and code are not both simultaneously living entities
  - Bugs in the code are fixed. Since documentation is regarded as a separate phase, document remains unchanged.
  - Code and document need to be co-located and visually linked.
  - That is where javadoc comes into play.
What is javadoc?

- javadoc is a separate program that comes with every Java installation
- javadoc is a tool that processes your Java program to:
  - make list of all the classes, interfaces, methods, and variables, and
  - create an HTML page that shows the results.
- Since javadoc generates its output from your source file, it is always accurate.
- Of course only listing these structures is not enough. You should write good comments that complement these lists.
  - It is your responsibility to ensure these comments are also accurate.

Keep your comments up-to-date

- Your comments must be in sync with your code
  - Outdated comments are worse than having no comments
- You must describe the method interfaces before you code the methods (part of the design phase)
- Implement after you design; don’t code and then try to figure out what you did

Where to put comments

- javadoc comments must be immediately before:
  - a class
  - an interface
  - a constructor
  - a method
  - a field
- Anywhere else, javadoc comments will be ignored.

- javadoc can be set to display:
  - only public things (classes, methods, fields)
  - public and protected things
  - public, protected, and package things
  - everything, even private things
javadoc comment style

/**
 * This is where the text starts. The star lines up with the first star above; there is a space after each star. The first sentence is the most important: it becomes the "summary."
 * @tag these go at the end, after a blank line
 */

void myMethod() {
    // this lines up with the / in /**
}

Summary description

• Omit the subject in summary descriptions
• Write in the third-person narrative form.
  - Good: Reads the input and calls add method.
  - Not as good: Read the input and calls add method.
  - Bad: This method reads the input and calls add method.
  - Bad: Method readInput reads the input and calls add method.

Frequently used javadoc tags

<table>
<thead>
<tr>
<th>Tag &amp; Parameter</th>
<th>Usage</th>
<th>Applies to</th>
</tr>
</thead>
<tbody>
<tr>
<td>@author name</td>
<td>Describes an author.</td>
<td>Class, Interface</td>
</tr>
<tr>
<td>@version version</td>
<td>Provides version entry. Max one per Class or Interface.</td>
<td>Class, Interface</td>
</tr>
<tr>
<td>@since since-text</td>
<td>Describes since when this functionality has existed.</td>
<td>Class, Interface, Field, Method</td>
</tr>
<tr>
<td>@see reference</td>
<td>Provides a link to other element of documentation.</td>
<td>Class, Interface, Field, Method</td>
</tr>
<tr>
<td>@param name description</td>
<td>Describes a method parameter.</td>
<td>Method</td>
</tr>
<tr>
<td>@return description</td>
<td>Describes the return value.</td>
<td>Method</td>
</tr>
<tr>
<td>@exception classname description</td>
<td>Describes an exception that may be thrown from this method.</td>
<td>Method</td>
</tr>
<tr>
<td>@throws classname description</td>
<td></td>
<td>Method</td>
</tr>
<tr>
<td>@deprecated description</td>
<td>Describes an outdated method.</td>
<td>Method</td>
</tr>
<tr>
<td>@inheritDoc</td>
<td>Copies the description from the overridden method.</td>
<td>Overriding Method</td>
</tr>
<tr>
<td>@link reference</td>
<td>Link to other symbol.</td>
<td>Class, Interface, Field, Method</td>
</tr>
<tr>
<td>@value</td>
<td>Return the value of a static field.</td>
<td>Static Field</td>
</tr>
</tbody>
</table>

HTML in doc comments

- Doc comments can be written in HTML
- You must replace
  - "<" with "&lt;"
  - ">" with "&gt;"
  - "&" with "&amp;"
- Other things you may use:
  - <i>...</i> for italics
  - <b>...</b> for boldface
- Avoid <h1> <h2>
- Inside the comment block, use <p> to separate paragraphs and javadoc predefined tags to define specific elements

Method signatures

- Fully describe the signature for each method
- The signatures distinguish the methods from each other
  - The signature includes the number, order, and types of the parameters
- Use a @param tag to describe each parameter
  - @param tags should be in the correct order
  - Don't mention the parameter type; javadoc does that for you
  - Use @return tag to describe the result (unless it's void)

Specifying identifiers and monospaced text

- Enclose identifiers, keywords, and constants in <code>... </code>
- Enclose (preformatted) monospaced text in <pre>... </pre> to preserve indentation and newlines.

this

- Use “this” rather than “the” when referring to instances of the current class.
- In Java, “this” is a keyword that refers to the instance of the current class (i.e., the instance that is executing the method)
Other stuff

• Provide examples when necessary
• Document known defects and deficiencies
  - It is worse if the reader discovers it

Warning for C/C++ programmers

• Don’t append parentheses to a method (or constructor) unless you really mean a call with no parameters.

Basic rules about programming

• There is always time at the start of a project
• There is never time at the end of a project
• Remember the 90/90 rule:
  - The first 90% of a project takes the first 90% of the time; the remaining 10% of the project takes the remaining 90% of the time.
• Do it right the first time.

When to write the comments

• So, describe the programming interface before you write the code.
Example #1

/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute [{@link URL}]. The name
 * argument is a specifier that is relative to the url argument.
 * This method always returns immediately, whether or not the
 * image exists.
 * @author Sun
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url argument
 * @return the image at the specified URL
 * @see Image
 */
public Image getImage(URL url, String name) {
    ...
}

Source: http://www.cse.ohio-state.edu/~paolo/teaching/421/lectures/lecture07.pdf

Example #2

/**
 * Validates a chess move. Use {@link #doMove(int, int, int, int)} to move a piece.
 * @param theFromFile file from which a piece is being moved
 * @param theFromRank rank from which a piece is being moved
 * @param theToFile file to which a piece is being moved
 * @param theToRank rank to which a piece is being moved
 * @return true if the move is valid, otherwise false
 */
boolean isValidMove(int theFromFile, int theFromRank, int theToFile, int theToRank) {
    ...
}

/**
 * Moves a chess piece.
 * @see java.math.RoundingMode
 */
boolean doMove(int theFromFile, int theFromRank, int theToFile, int theToRank) {
    ...
}

Example #3 1/3

/**
 * @author Tuna Tugcu
 * @version 1.0
 */
import java.util.*;
/*
 * Defines a class that inputs two integers and prints the sum.
 */
public class Add {
    ...
}

Example #3 2/3

/**
 * Reads the input and calls <code>add</code> method.
 * Also displays the outcome <i>one by one</i>.
 * @param args These are the command line parameters. In fact, they
 * are not checked at all.
 */
public static void main(String[] args) {
    Scanner console = new Scanner(System.in);
    int a,b,c;
    // Read two integers into a and b.
    a = console.nextInt();
    b = console.nextInt();
    // Call the add method and get the return value.
    c = add(a,b);
    // Display the input parameters and the result.
    System.out.println("a\n b\n c");
}

Source: http://www.cse.ohio-state.edu/~paolo/teaching/421/lectures/lecture07.pdf

CMPE160: Introduction to Object Oriented Programming
### Example #3

```java
/**
 * Performs the actual arithmetic operation. As simple as that.
 * @param x This is the first parameter
 * @param y This is the second parameter
 * @return The sum of <code>x</code> and <code>y</code>
 */
public static int add(int x, int y) {
    return x+y;
}
```

### Alternative: Doxygen

- Javadoc-like comment tags and formatting
  - comment block with description and tags
  - author, param, return, etc.
- Supports multiple programming languages
  - C/C++, Java, C#, PHP, Python,…
  - Comment syntax language dependent
- Supports multiple output formats
  - html, rtf, pdf, latex, man, xml,…
  - Documentation text less html-ized
- Better support for design-by-contract
  - Has built-in tags for @pre, @post, @invar