How to Write Good Comments

Write for your audience

- Program documentation is for programmers, not end users
- There are ***two*** groups of programmers, and they need different kinds of documentation
  - Some programmers need to *use* your code
    - Do *not* explain to them *how* your code works--they don't care and don't want to know
    - Tell them what your methods do, how to call them, and what they return
    - Javadoc is the best way to document your code for users
  - Other programmers need to maintain and enhance your code
    - They need to know how your code works
    - Use *internal comments* for these programmers
- When you work on your program, you are in *both* groups
  - Document as though you will have forgotten everything by tomorrow!

Internal comments

- Use internal comments to:
  - Explain the use of temporary variables
  - Label closing braces in deeply nested statements, or when many lines are between the open and close braces
    - `while (i != j) { … … … … … … } // end while`
  - Explain complex or confusing code
  - Explain what the next section of code does
- *Never* repeat the code!
  - `count = count + 1;  // add one to count`

Good code requires few comments

- Use internal comments to:
  - Explain the use of temporary variables
    - Better: Give them self-explanatory names
  - Label closing braces in deeply nested statements, or when many lines are between the open and close braces
    - Better: Don’t nest statements that deeply
    - Better: Keep your methods short
  - Explain complex or confusing code
    - Better: Rewrite the code
    - If it’s complex or confusing, it’s probably buggy as well
  - Explain what the next section of the method does
    - Better: Make it a method with a self-explanatory name

Good uses of internal comments

- Use internal comments:
  - If you *really* can't make the code simple and obvious
  - To reference a published algorithm
  - To mark places in the code that need work
    - Eclipse provides three tags for this purpose (you can add more):
      - TODO – Means: This code still needs to be written
      - FIXME -- Means: This code has bugs
      - XXX -- Means: I need to think about this some more
      - To see these, choose Window --> Show View --> Tasks
  - To indicate an intentional flow-through in a switch statement
  - To temporarily comment out code (Eclipse: control-/)

javadoc

- javadoc is a separate program that comes with every Java installation
- javadoc reads your program, makes lists of all the classes, interfaces, methods, and variables, and creates HTML pages displaying its results
  - This means javadoc's generated documentation is always accurate
- You can write special documentation ("doc") comments
  - Your doc comments are integrated into javadoc's HTML page
  - It's your job to ensure these are also accurate
- Javadoc's output is very professional looking
  - This makes *you* look good
  - It also helps keep your manager from imposing bizarre documentation standards

javadoc

- Always use doc comments to describe the API, the Application Programming Interface
  - Describe all the classes, interfaces, fields, constructors, and methods that are available for use
- javadoc can be set to display:
  - only public elements
  - public and protected elements
  - public, protected, and package elements
  - everything--that is, public, protected, package, and private elements
- Remember, doc comments *for the programmer who uses your classes*
  - Anything you want to make available outside the class should be documented
  - It is a good idea to describe, for your own use, private elements as well

Contracts

“The primary purpose for documentation comments is to define a *programming contract* between a *client* and a supplier of a *service*. The documentation associated with a method should describe all aspects of behavior on which a caller of that method can rely and should not attempt to describe implementation details.”

--The Elements of Java Style

by Allan Vermeulen and 6 others

javadoc is a *contract*

- In the “real world,” almost all programming is done in teams
  - Your Javadoc is a contract between you and the other members of your team
    - It specifies what you expect from them (parameters and preconditions)
    - It specifies what you promise to give them in return
  - Do not be overly generous!
    - Provide what is really needed, but...
    - Remember that anything you provide, you are stuck with debugging, maintaining, and updating
    - Providing too much can really hamper your ability to replace it with something better someday

Know 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!*
  - Plus, they look silly

javadoc comment style

- Use this format for all doc comments:

```
/**
 * This is where the text starts. The asterisk lines
 * up with the first asterisk above; there is a space
 * after each asterisk. The first sentence is the most
 * important: it becomes the “summary.”
 *
 * @param x Describe the first parameter (don’t say its type).
 * @param y Describe the first parameter (don’t say its type).
 * @return Tell what value is being returned (don’t say its type).
 */
public String myMethod(int x, int y) {  // p lines up with the / in /**
```

HTML in doc comments

- Doc comments are written in HTML
- In a doc comment, you *must* replace:

  `<` with `&lt;`   `>` with `&gt;`   `&` with `&amp;`

  ...because these characters are special in HTML
- Other things you may use:

  `<i>...</i>` to make something italic
  - Example: `This case should <i>never</i> occur!`

  `<b>...</b>` to make something boldface

  `<p>` to start a new paragraph
- Other types of comments are *not* in HTML

Identifiers in doc comments

- Wrap keywords and the names of variables and methods with <code> . . . </code> tags
- Example:

```
/**
 * Sets the <code>programIsRunning</code> flag
 * to <code>false<code>, thus causing
 * <code>run()</code> to end the Thread
 * doing the animation.
 */
```

Code in doc comments

- Wrap code with <pre>…</pre> tags.
  - Preformatted text is shown in a monospaced font (all letters the same width, like Courier), and keeps your original formatting (indentation and newlines)
  - Preformatted text is also good for ASCII “drawings”

```
<pre>
       NW   N   NE
          \ | /
       W — + — E
          / | \
       SW   S   SE</pre>
```

Tags in doc comments

- **U**se the standard ordering for javadoc tags
  - In class and interface descriptions, use:

    @author *your name*
    @version *a version number or date*

    - ***Use the @author tag in your assignments!!!***
  - In method descriptions, use:

    @param *p A description of parameter p.*
    @return *A description of the value returned (unless the method returns void).*
    @exception *e Describe any thrown exception.*

Keep comments up to date

- Keep comments accurate
  - An incorrect comment is worse than no comment!
  - Any time you change the code, check whether you need to change the comment
- Write the doc comments before you write the code
  - It’s better to *decide what to do, then do it*

    than it is to

    do something, then try to figure out what you did

Document nearly everything

- If it’s available outside the class, document it!
- If it’s private to the class, it’s still a good idea to document it

- The class itself should be documented
  - In other words: Tell what your program does!
  - You would be surprised how quickly you can forget what the program does

this object

- Use the word "this" rather than "the" when referring to instances of the current class.
- In Java, this is a keyword that refers to the instance of this class that is responding to the message (that is, the instance that is executing the method)
- Hence, this *object* has an especially clear meaning in comments
- Example: Decides which direction this frog should move. (As a comment in the Frog class)

Parentheses

- **C and C++ programmers, pay attention!**
- Do not add parentheses to a method or constructor name unless you want to specify a particular signature!
- If, in a comment, you refer to `turn( )`, you are implying that `turn` is a method with *no* parameters
  - If that's what you meant, fine
  - If that's *not* what you meant, say `turn` instead
- Why is this different from C and C++?
  - In C, method overloading is not allowed
  - C++ programming is strongly rooted in C

The first sentence is special

- If your doc comment is more than one sentence long:
  - The *first sentence* should *summarize* the purpose of the element (class, method, etc.)
  - This first sentence should make sense when read alone
  - Javadoc uses the first sentence by itself, as a summary
  - Javadoc puts summaries near the top of each HTML page, with a link to the complete doc comment further down the page

Rules for writing summaries

- For methods, omit the subject and write in the third-person narrative form
  - Good: Fin<u>ds</u> the first blank in the string.
  - Not as good: Fin<u>d</u> the first blank in the string.
  - Bad: This method finds the first blank in the string.
  - Worse: Method findBlank(String s) finds the first blank in the string.

Include examples

- Include examples if they are helpful.
  - Most methods should be simple enough not to need examples
  - Sometimes an example is the best way to explain something

Input and output conditions

- Document preconditions, postconditions, and invariant conditions.
- A precondition is something that must be true beforehand in order to use your method
  - Example: The piece must be moveable
- A postcondition is something that your method makes true
  - Example: The piece is not against an edge
- An invariant is something that must *always* be true about an object
  - Example: The piece is in a valid row and column

Bugs and missing features

- Document known problems
  - What? Admit my code isn't perfect?
    - That might lower my grade, or get me in trouble with my boss!
    - But it will be worse if they discover it themselves
  - Be kind to the poor user, struggling to find the bug in her code, when the bug is really in yours

Who cares?

- Aren’t we supposed to be learning how to program in Java, not a bunch of stupid “style rules”?

Or in other words:

- What do we care what our teachers and prospective employers think?

Aren’t these just arbitrary conventions?

- All these rules have good reasons, but some rules are more important than others
  - Keep comments and code in sync
    - This rule is *important*
  - Write in the third person narrative form
    - That’s “just” ordinary good writing style
- Good documentation is *essential* in writing, debugging, and maintaining a large program
  - It even helps in small programs

When do you add comments?

- 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
- Write the comments *before* you write the code.

Vocabulary I

- Preformatted text: HTML text that maintains your indentation and spacing
- Monospaced font: One in which all the letters (and usually other characters) have the same width
- Signature of a method: The information needed to distinguish one method from another

Vocabulary II

- Precondition: A condition that must be true before a method (or other block of code) if it is to work properly
- Postcondition: A condition that is made true by executing a method (or other block of code)
- Invariant: A condition that must always be true of an object.
- 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.

The End

It should be noted that no ethically-trained software engineer would ever consent to write a `DestroyBaghdad` procedure. Basic professional ethics would instead require him to write a `DestroyCity` procedure, to which `Baghdad` could be given as a parameter.

--Nathaniel S Borenstein