CS100 Fall 1998 Notes on Java Programming Style

David Gries

A goal of CS100 is for you to learn to write programs that are not only correct but also understandable. These guidelines should help you toward that goal. We ask that you follow these guidelines when writing programs in this course. They will give you a good basis for developing a style of your own as you become a more experienced programmer.

This handout includes guidelines for Java constructs that will be covered later in the semester. Skim these sections now and read them more carefully later when the topics are discussed.

Table of contents

Reasons for following a disciplined programming style

Your computer program should be readable, because if it is not readable, the chances of it being correct are slim. Moreover, if your program is unreadable, it will be difficult and time-consuming to find and correct the errors in it. So,

> The major reason for using a disciplined style of programming is that it will save you time whenever you have to read your program or debug it.

Your program should be readable by others, not just you. Consider a program you will write in this course. If the grader has trouble understanding it, they is not likely to give you bad a grade. The more understandable, the simpler your program appears to a grader, the better the grade they will be willing to give you.

Outside this course, making programs readable by others becomes even more important. Most programs live a long time and require "maintenance" --changes to adapt to new and different requirements, upgrades in other software, new hardware, etc. And the author of the program is quite likely not going to be around when the maintenance is required; someone else must read the program and understand it enough to update it successfully.

Even the programs you write for yourself should be readable; if not, four weeks after finishing it you will not remember it enough to make changes simply.

Thus, simply for your own sake and for the sake of others, it makes sense to develop programming habits that lend themselves to writing readable, understandable, and correct, programs.

Part of these habits concern simple, syntactical measures like indenting program parts properly and using a few conventions for names of variables, methods, etc. The more important part concerns recording enough information in comments for the reader to understand how a program is designed and why.

A computer program is the result many design decisions. These decisions --why this variable was introduced, what that function does, etc.-- are often not reflected in the final Java code, which consists of low-level, detailed declarations and statements. However, the higher-level design *must* be understood if a programmer is to modify the program successfully. Trying to understand decisions that are not recorded in comments in the code is tedious, error-prone, and aggravating --but all too common.

So, it will be to your advantage to instill in yourself some disciplined programming habits, right from the beginning, like

- Write a precise definition (in a comment) of a variable as you write its declaration
- Update the definition of the variable (the comment) when you decide to change its meaning
- Write a precise specification for a method (which says what it does) as you write the method's heading --don't wait until after the method is completely written or until the program is debugged.
- Change the specification for a method whenever you make changes to the body of the method that make the body not fit the specification.

Preparing all the comments after the program is finished is bad for three reasons: (0) the comments were not of use to you when developing and debugging the program, so you took more time than was necessary. (1) The comments are harder to write after the program is finished, because it is difficult to remember the meaning of all the variables and methods. (2) It is quite likely that you won't write the comments --even programmers with the best intentions don't spend much time filling in comments after the fact, because there are too many other interesting things to do.

You will find that writing good comments as you write a program will help you clarify your ideas and write better, correct code sooner. If you can write down clearly what your program is doing you are more likely to have a good understanding of the problem and your program is more likely to be correct. Time spent on careful thinking and writing is more than repaid in time saved during testing and debugging.

Return to table of contents

Conventions for names

Good naming conventions can help you and any reader of a program understand the program easily; bad naming conventions or no naming conventions can lead to confusion and misunderstanding. Some people advocate using very long names that define what the named entity (a method, field, local variable, etc.) represents. However, in general, this is not feasible. A good definition for an entity may require three lines of explanation or more ---how can you have such a long name? On the other hand, extremely short names --one or two characters long-- don't help at all. Thus, some in-between measures have to be adopted. There are contexts where a short name is best and contexts where a longer name is best.

Remember that a name can rarely be used to give a complete, precise definition of the entity it names, and a complete definition should always be given where the entity is defined.

The people developing Java programs have developed some conventions for identifiers, which help one readily see what the identifier represents (to some extent). You can see these conventions at work by looking at the classes in java.awt (Abstract Window Toolkit). Below, we state the conventions, with a few additions.

Package names: e.g.AnimalStory

A package is a set of classes that have been grouped together. The name of a package is usually something that identifies its contents. Generally speaking, a package name should begin with a capital letter, and all succeeding words in the name should start with a capital letter.

Class names: e.g. FilterInputStream, LivingMammals

Since a class represents a set of possible objects, each of which is an instance of the class, a class name should generally be a noun phrase that identifies the possible objects. A class names should begin with a capital letter, and all successive words in the class name should be capitalized.

Method names: e.g. drawOval, fillOval, toString, length

A method that is a command to do something should be given by a verb phrase that gives some indication of what the method does. (But this name should *never* be used in place of a precise comment for the method!) A method that implements a function --that returns a value-- should be a phrase that describes the value. For a method that both executes some task and returns a value, use common sense in creating its name.

A class name should begin with a small letter, but all successive words in the class name should be capitalized.

Names of parameters of methods: e.g. x, y

The precise meaning of a parameter, and any restrictions on it, should be given in the comment of the heading of the method. Therefore, particularly if the method is fairly short, it is wise to give parameters short names.

For example, in the two method headings given below, the first is preferable because it is shorter and easier to understand. Moreover, the body of the method of the first method will also be shorter and far easier to understand and manipulate.

```
//Draw an ellipse that fits exactly within the rectangle whose
//upper left corner is at position (x,y), whose width is w, and
//whose height is h. Use the current color to draw the ellipse.
void drawOval(int x, int y, int w, int h) {

//Draw an ellipse that fits exactly within the rectangle whose
//upper left corner is at position (xCoordinate,yCoordinate)
//whose width is width, and whose height is height. Use the
//current color to draw the ellipse.
void drawOval(int xCoordinate,int yCoordinate,
              int width, int height)
```

A name like *theLoopCounter* or *firstNumber* instead of *k* or *x* causes clutter.

A parameter used as a "flag" should be named for what the flag represents, like *noMorePizza*, rather than simply *flag*. Avoid generic names like *count* and *value*; instead, describe the items being counted or the value stored in the variable.

Field names: e.g. size, xCoordinate, noLines

A field name, or instance variable as they are sometimes called, contains information that helps describe the "state" of the object in which it occurs. A field name should be a noun phrase that describes the information the field contains. (However, the field still needs a (more) precise comment that describes it.) All words in a field name, *except the first*, should be capitalized.

Local-variable names: e.g. size, xCoordinate, noLines

A local variable of a method contains information that helps describe the "state" of the method during its execution.. A field name should be a noun phrase that describes the information the field contains. However, the field still needs a (more) precise comment that describes it. All words in a field name, *except the first*, should be capitalized.

If the body of a method is short, or the places in which a local variable is used is fairly short, then a short, one-or-two letter name can be used for the local variable (see also the conventions for parameter names). A name like *theLoopCounter* or *firstNumber* instead of *k* or *x* causes clutter. If the local variable is used only in a short context, and if it is suitably defined with a comment at its place of declaration, then use the short name.

A variable used as a "flag" should be named for what the flag represents, like *noMorePizza*, rather than simply *flag*. Avoid generic names like *count* and *value*; instead, describe the items being counted or the value stored in the variable.

Accessor names: e.g. textLength, setTextLength

An accessor method for a field returns the value in the field or stores a value in the field. Two examples will illustrate a convention for naming such accessor methods. To reference a field named *textLength*, use a method *textLength*() --the method has the same name and has no parameters-- or use a method *setTextLength*(). To store a value in the field, use a method *setTextLength*().

Constant names: e.g. PI, ANDY'S_IQ

In Java, a constant is simply a field of a class that has been declared final. In order to distinguish a constant from an instance variable or local variable, one could use only capital letters in the names of constants and also separate adjacent words in a constant name with an underscore.

Return to table of contents

Indentation

Indentation is used to make the structure of a program clear. The basic rule is:

Substatements of a statement or declaration should be indented.

For example, the body of a method, the if-part and then-part of a conditional statement, and the body of a method should be indented.

There are several methods for placing curly braces that delimit statements, none of which is liked by all and all of which are disliked by some. Below, we illustrate three methods, with pros and cons explained to the right.

```
(0)  if (x < y)        Pro: It is easy to match up braces
     {                 Con: Too many lines are used, and the
         x = y;             number of lines is a scarce re-
         y = 0;             source on the monitor.
     }                 Con: The substatement {x= y; y= 0;} has
     else                   not been indented, because the
     {                      braces are part of the substatement.
         x = 0;             Therefore, the basic rule concerning
         y = y/2;           indentation has not been followed.
     }

(1)  if (x < y) {      Pro: There are no wasted lines.
         x = y;        Pro: The opening brace is out of the way
         y = 0;             and the closing brace indicates

     } else {               nicely the end of a substatement
         x = 0;        Con: It is difficult to match up be-
         y = y/2;           ginning and end braces.
         }
```

```
(2)  if (x < y)          Pro: There are no wasted lines.
         {x = y;         Pro: The basic indentation rule is followed.
         y = 0;          Pro: It is easy to match up beginning and
         }                    end braces and easy to see where a
     else                     substatement ends.
         {x = 0;         Con: The first statement of a substatement
         y = y/2;             indented differently from the rest.
         }
```

It doesn’t matter which style you use, as long as you use it consistently throughout a program. When working on a program written by someone else, match their style

Below, we illustrate two ways to place the curly braces surrounding thebody of a method. There are others. Choose one and use it consistently within a program.

```
(0) // Specification of method
    void drawLine(int x1, int y1, int x2, int y2) {
        statement 1;
        statement 2;
    }

(1) // Specification of method
    void drawLine(int x1, int y1, int x2, int y2)
        {statement 1;
        statement 2;
        }
```

Return totable of contents

Statement-comments

Just as the sentences of an essay are grouped in paragraphs, so the sequence of statements of the body of a method should be grouped into logical units. Often, the clarity of the program is enhanced by preceding a logical unit by a comment that explains what it does. This comment serves as the specification for the logical unit; it should say precisely what the logical unit does. .

The comment for such a logical unit is called a "statement-comment". It should be written as a command to do something. Here is an example.

```
    // Truthify x >= y by swapping x and y if needed.
         if (x < y)
            {int tmp = x;
            x = y;
            y = tmp;
            }
```

The comment should explain what the group of statements does, not how it does it. Thus, it serves the same purpose as the specification of a method: it allows one to skip the reading of the statements of the logical unit and just read the comment. With suitable statement-comments in the body of a method, one can read the method at several "levels of abstraction", which helps one scan a program qickyl to find a section of current interest, much like on scans section and subsection headings in an article or book. But this purpose is served only if statement-comment are precise.

Statement comments must be complete. The comment

```
    // Test for valid input
```

is not adequate. What happens if the input is valid? What if it isn’t --is an error message written or is some flag set? Without this information, one must read the statements for which this statement-comment is a specification, and the wole purpose of the statement comment is lost.

Placement of statement-comments. In the example above, and in the following example, the statements specified by a statement comment are indented. In a program with several levels of statement-comments, this indentation is useful in clarifying the structure of a program. In fact, with several levels of statement-comments, even judicious use of blank lines cannot eliminate all ambiguity concerning what statement belongs to what statement-comment. The program fragment given below illustrates this.

```
// Truthify the definition of t --return false if not possible
     // Eliminate whitespace from the beginning and end of t
         while (t.length() != 0 && isWhitespace(t.charAt(0)))
           t= t.substring(1);

     // If t is empty, print an error message and return
         if (t.length() == 0)
             {...
             return false;
             }

     if (containsCapitals(t))
        {...
        }

 // Store the French translation of t in tFrench
     ...
```

At the highest level, the program fragment consists of two statements: (0) Truthify the definition *t* and (1) Store the French translation of *t* in *tFrench*. The statement "Truthify the definition of t is implemented in three steps, two of which are themselves statement-comments. Thus, this program fragment has three levels of abstraction.

In this example, note how reliance on the definition of t in the statement-comment "Truthify the definition of t" allows the statement-comment to be short but precise. Of course, a suitable definition for t must appear at its declaration.)

Many people prefer not to indent substatements of a statement-comment and to rely on blank lines to separate the end of the substatements of a statement-comments. Below, we show the above program fragment in this style. Note the extra comment to help the reader see the end of the substatements for "truthify the definition of t". This style is has its problems. If you decide to use this style, then program in such a way that statement-comments do not appear within statement-comments, so that such awkwardnesses do not arise.

```
// Truthify the definition of t --return false if not possible

// Eliminate whitespace from the beginning and end of t
while (t.length() != 0 && isWhitespace(t.charAt(0)))
   t= t.substring(1);

// If t is empty, print an error message and return
if (t.length() == 0)
   {...
  return false;
   }

if (containsCapitals(t))
   {...
```

```
        }

    // (End of Truthify the definition of t)

    // Store the French translation of t in tFrench
     ...
```

Return totable of contents

Specification of methods (procedures and functions)

Every method should be preceded by a comment giving its specification. This specification, together with the heading of the method, which gives the number and types (classes) of the parameters and the type (class) of the result, should provide all of the information needed to use the method --and no more. It should describe any restrictions on the parameters and what the method does, not how it does it. One should never have to look at the body of a method to understand how to use it. The specification comment should describe the parameters of the method. The specifications can usually be written in few sentences. Here is an example.

```
    // Print on System.in the most frequently occurring temperature
    // in t[0..c-1]. If there is more than one possibility, print
    // the least temperature.
    void findCommon (Temperature [] t, int c)
```

Unfortunately, it is more typical to find a comment like the following (if any comment is provided at all).

```
    // Find most frequent temperature
    void findCommon (Temperature [] t, int c)
```

This specification fails to say what part of array t is to be included in finding the most frequent temperature. It also fails to say where to print, which will be a problem if there is more than one possibility. The only way for the user to find out is to look at the body of the method (if it is available), and that should not be necessary.

For a function --i.e. a method that returns a value-- it is often easiest to simply describe the value returned, using the word "yield":

```
   // yield distance between points (x1,y1) and (x2,y2)
       double dist (int x1, int y1, int x2, int y2)
```

The Elements of Style, a famous little book on writing style by Cornell Professors W. Strunk, Jr., and E.B. White, contains several rules that are useful for programming as well as writing. Among them are:

> Omit needless words.
>
> Use the active voice.

Follow these rules when writing specifications of methods. For example, don’t write the specification ‘‘This function searches list x for a value y and ...’’ or ‘‘Function isIn searches list x for a value y ...’’. Such specifications are too wordy and are not commands but descriptions. Instead, say the following.

```
    // Yield "y is in list x"
    boolean isIn(int y, List x)
```

Return to table of contents

Definitions of variables (field names, parameters, local variables)

Every significant variable and data structure needs a precise and complete definition, which provides whatever information is needed to understand the variable The most useful information is often the *invariant properties* of the data: facts that are always true except, perhaps, momentarily, when several related variables are being updated.

> **Important hint:** Write the definition of a set of variables when you first conceive of using them, and type the definitions as comments into the program when you type in the declarations of the variables. If you later decide to change the meanings of the variables, change the definitions *before* you change the statement that use the variables.

Here is an example of a definition for two variables *i* and *currentItem* --they are defined in a single comment because they are related.

```
// 0 <= i <= currentItem < numberItems and i is the smallest value
// such that item i’s price is at most item currentItem’s price.
```

The definition for a boolean variable is usually best presented as the value of some English (or mathematical) statement. For example, the following two definitions are equivalent, but the first is shorter.

```
boolean b; // = " user selected menu item File | Quit"

boolean b; // true if the user selected menu item File | Quit;
           // false otherwise
```

The more precise a definition, the better. Comments like "flag for loop" or "index into b" are useless; they only say how the variable is used, but not what it means.

Related variables should be declared and described together. For example, the definition of a table should describe not only the array that holds the data but also the integer variable that contains the number of items currently in the table. In the example below, for utmost clarity tabs are used to line up identifier names and to line up the comments.

```
final int maxTemp = 150; // max number of temperature readings.
       int nTemp   = 0;   // Temperature readings are in
                           // temp[0..nTemp-1], where
 double temps   = new double{maxTemp];// 0 <= nTemp < maxTemp
```

If the comment is too long to fit nicely on the right, then put it above the declarations and indent the declarations:

```
// Temperature readings are in temp[0..nTemp-1], where
// 0 <= nTemp <= maxTemp and maxTemp is the maximum number of
// temperature readings
      final int maxTemp = 150;
      int       nTemp   = 0;
      double    temps   = new double[maxTemp]
```

Return to table of contents

Specifying classes

In Java, typically, each class is given in a separate file. The beginning of the file should contain a comment that explaine the what the class is for. Often, one also puts here information concerning the author, the date of last modification, and so on. Here is an example.

```
//An object of class Auto represents a car.
```

```
      //Author: John Doe.
      //Date of last modification: 25 August 1998
      public static class Car;
```

Return to table of contents

Acknowledgements

The ideas in this missive originated in the structured-programming movement of the 1970's. They are every bit as applicable today. Many examples and ideas were taken from old Cornell CS100 and and CS211 handouts, originating in the work of Richard Conway and David Gries (see their 1973 text on *An Introduction to Programming, using PL/C*). Hal Perkins has a hand in writing this.

Return to table of contents

Back to Top
Back to CS100B Home