Java Coding Style and Documentation Guidelines

Jim Henry
Started: 2/12/04
Current: 3/29/11

Introduction

Your program is (probably) not done when it works. You need to check it for good coding practices, good formatting, and good documentation. If you try to follow good practices *as you code*, you may not have too much to do - but you will usually have something remaining to improve your program.

Matters of program style and documentation are important. Much programmer effort is spent in reading other programmers' code and documentation when debugging, enhancing, or otherwise modifying code. (Some estimates place it as high as 60% to 70%.) Making this job easier is one of your duties as a programmer, and you must develop good habits as you learn.

Don't forget that *your own code* 6 months from now will look to you like *another programmer's code*. You won't remember much of it. But you will be thankful if you left yourself good, clean, clear, readable code, not some undocumented, messy, convoluted junk. It may be hard to imagine this happening, since no one (probably including you) looks at your class assignment programs after they are graded. But one day you will have to go back to modify or fix a program you wrote 6 months or a year before, and then - if you haven't written it well - you'll be sorry. Just remember that I told you so.

Write your code and your documentation for human readers. Understand that they *cannot* read your mind. If someone has to ask you "what does this method really do?" or "what does this variable represent?" you have probably failed to write clear code or to explain it well.

We *really* do *not* want to have to remind you of the things mentioned below. We have taken several hours to write up and supply these suggestions and guidelines to you, and we expect that you will take a significant amount of time to read them and to incorporate them into your code.

*Your program is **not** finished when it works.*
*Your program is not finished **until** you have addressed the points below.*

In general, prepare your source code for publication in a poetry magazine. It should look that good.

It's not just me that says this. Here's a famous guy:

> Using regular indentation, proper spelling for comments and identifiers, adequate lexical conventions -- a space before each opening parenthesis but not after -- does not make your task longer than ignoring these rules, but compounded over months of work and heaps of software produces a tremendous difference. Attention to such details, although not sufficient, is a necessary condition for quality software (and quality, the general theme of this book, is what defines software engineering).

--Bertrand Meyer
Read the rest in Object Oriented Software Construction, 2nd edition, p. 180

Coding Style and Practice

1. Don't write code that does nothing or is unnecessary.

Wrong	*Right*

`String s = "";  //no need to initialize`
 	
`...`
 	
`s = tf.getText();  // 1st use is assignment`
`if (s.equals("Jim")`
`  {`
`  .....`
	`String s;`
	
	`...`
	
	`s = tf.getText();`
	`if (s.equals("Jim")`
	`  {`
	`  .....`

If you write something and then later decide you don't need it, take it out.

2. Don't leave in code that is "doc'ed out" Remove it.

3. Keep code together that belongs together. Don't separate related code by many intervening lines of code. If you can see all the relevant code at one glance, you will be able to understand it more easily and more quickly and you won't make mistaken assumptions about things that might happen elsewhere.

Wrong - Ex 1	*Wrong - Ex 2*	*Right - Ex 1*	*Right - Ex 2*

`i = 1.07;`
`...`
`...`
`...`
	`void meth()`
	`  {`
	`  i = 0;`
	`  ...`
		`if (size > 2000)`
		`  i = 1.05;`
		`else`
		`  i = 1.07;`
			`void meth()`
			`  {`
			`  ... 20 lines`

```
// and after 20 more lines
if (size > 2000)
  i = 1.05;
```

```
...
// after 20 more lines
while (i < 10)
  {
  ...
  i++;
  }
}
```

```
i = 0;
while (i < 10)
  {
  ....
  i++;
  }
}
```

4. Don't ask two questions when one is enough.

Wrong

```
if (e.getSource() instance of Button)
  if ((String)(e.arg)).equals("Calculate"))
    // do something...
```

Right

```
if (e.getSource() == calcButton)
  // do something...
```

5. Do similar things in similar ways

Wrong

```
...
catch (SomeException e1)
  {
  tf.setText("error msg");   // 1
  showStatus("a message");   // 2
  inputOK = false;           // 3
  clearResults();            // 4
  }
catch (SomeOtherException e2)
  {
  showStatus("a message");   // 2
  tf.setText("error msg");   // 1
  clearResults();            // 4
  inputOK = false;           // 2
  }
```

Right

```
...
catch (SomeException e1)
  {
  tf.setText("error msg");   // 1
  showStatus("a message");   // 2
  inputOK = false;           // 3
  clearResults();            // 4
  }
catch (SomeOtherException e2)
  {
  tf.setText("different msg");         // 1
  showStatus("a message");   // 2
  inputOK = false;           // 3
  clearResults();            // 4
  }
```

6. Avoid duplicate code in different places. See if you can put it in a method and *call* it from multiple places.

7. Don't create lots of objects when one will do.

Wrong - a new DecimalFormat object is created *every time* action() is called

```
class Ap extends JApplet
  {
  ......

  public void actionPerformed(ActionEvent e)
    {
    DecimalFormat df =
        new DecimalFormat("#.00");

    ...
    // use df here ...
    .....
    }
```

Right - only one DecimalFormat object as an object data member

```
class Ap extends JApplet
  {
  DecimalFormat df =
      new DecimalFormat("#.00");

  public void actionPerformed(ActionEvent e)
    {
    .....
    // use df here ...
    .....
    }
```

8. Write code that tells you what it affects. If you write a method that alters object variables, have it return a value or pass the objects to be altered (even though you do have global access to the variables) so that

- we can see what is potentially altered
- the code could be used in a different place where the variable names might be different.

There may be exceptions to this rule, but only make exceptions for a good reason.

Wrong

```
calculateResults();
```

This method actually uses an array of values and sets the values of total, grandTotal, and tax - but we can't tell that without looking at the method's code.

Also, let's say part of its job is to add up array elements - like *calcTotal()* to the right here - but it will *only* be able to work with an array using whatever name is coded into it. To use this method in *another* place in your program, using *a different* array, we'd

Right

```
total      = calcTotal(valueArray);
tax        = calcTax(total, taxRate);
grandTotal = calcGrndTotal(total, tax);

OR ...

calculateResults(valueArray, resultsObj);
```

(The second method assumes that all the answers will be stored as data members in resultsObj.)

have to rename something - and that's a mess. (Do you see why?)	

9. In general, a method should not calculate and also print or display a result. Use one method to do the calculation and another to print it. Sometimes you may want the result and not print until later. Sometimes you may want to display the result via *System.out.println()* and other times you might want to display in a TextField. Sometimes you might only want to print one of several things or even not print at all. In all of these cases, a method that calculates-and-prints will not work for you since you can't stop it from doing the whole thing every time. So keep them separate.

Wrong	*Right*

`doCalculations();`
 	
or even
 	
`doCalculationsAndPrintThem();`
 	
(Here results are computed and possibly stored; also they are printed. There's no way to know any of this without looking at the method's code (except by looking at the name of the second version). Also, assuming that it calculates and prints a total, tax, and grand total, there's no way to have it print just one or two of them, or to print it in a different order.)
	`total = calcTotal(valueArray);`
	`tax = calcTax(total, taxRate);`
	`grandTotal = calcGrndTotal(total, tax);`
	
	then
	
	`System.out.println("Total is " + total);`
	
	or
	
	`totalTextField.setText(total.toString());`
	
	(Here we can print *what* we want, *how* we want.)

10. Write methods that do one thing only. Don't write methods that *should* be named *doThisAndThat()* if you really name them accurately. That is, your methods should do just *one* thing or task (handle an event, calculate an answer (or a set of closely related answers), paint a picture, run an animation, send a TCP/IP message, clear some TextFields, etc.) Point 9 was another specific example of this principle.

11. Write short to medium-sized methods. Any method that exceeds a page in length should be a candidate for breaking up into smaller methods. This is not a absolute rule - some methods by their nature are big. But the bigger a method, the more difficult to understand, usually. So consider breaking big methods up into smaller ones.

12. Don't use magic numbers. A magic number is a numeric literal that appears in source code with no explanation of it's meaning in the context of the problem. You look at it and say "what's that about?" Instead, use symbolic constants with meaningful names. In Java, implement these as final and capitalize them as in standard C usage for #defines.

Wrong	*Right*

`total = amt * 1.0625;`
 	
`total = amt * 1.07;`
	`total = amt * TAX_RATE;`
	
	`total = amt * JOB_SURCHARGE;`

13. Write clear code. In the following example, it is not at all clear what adding 1 to *wasteCost* means. In fact, a percentage (like .05) is being turned into 1.05 so that (later) a new value can be calculated that is 5% more than the original (i.e. 1.05% of the original).

Wrong	*Right*

`wasteCost +=1;`	`final double WASTE_COST = 1.05;`
	
	or
	
	`// convert a percent to 100 + that percent;`
	`// e.g. .05 -> 1.05 for later use`
	`wasteCost += 1;`

14. When possible, within a method, check for errors first and then do the regular work. Don't do a whole lot of setup work and then find an error that forces you to exit or return. Make the error handling visible early in the method and avoid unnecessary work that the program would do if an error were found later.

15. Declare variables with only the scope they require. If you have a variable that is only used in one method and whose value does not need to be retained between calls, declare it locally. Don't clutter up an object with the variable.

Wrong

```
public class A
  {
  int width, vGap hGap;
  ..
  public void paint(Graphics g)
    {
    Dimension d;

    d = size();
    vGap = d.width/8;
    hGap = d.height/8;
    g.drawRect(x1 + hGap,
               y1 + vGap,
               d.width  - 2*hGap,
               d.height - 2*vGap);
    }
```

Right

```
public class A
  {
  ..
  public void paint(Graphics g)
    {
    Dimension d;
    int width, vGap hGap;

    d = size();
    vGap = d.width/8;
    hGap = d.height/8;
    g.drawRect(x1 + hGap,
               y1 + vGap,
               d.width  - 2*hGap,
               d.height - 2*vGap);
    }
```

16. Don't Repeat Yourself. The DRY principle: every piece of knowledge must have a single, unambiguous, authoritative representation within a system. (Hunt and Thomas p. 27.) Don't write code so that some piece of information is stored in two different places. If you change one, you'll have to change the other. It isn't a question of *whether* you'll remember: it's a question of *when* you'll forget.

```
JCheckbox colorCb = new JCheckbox("Color");
boolean inColor = false;

if (colorCb.isSelected())
  inColor = true;

....

if (inColor)
  ....

if (e.getSource() == someButton)
  {
  inColor = false;
  colorCb.setSelected(false);
  }
```

Here the colorCb itself knows if it is checked or not (and we can find out via *isSelected()*) but the program has an additional variable that keeps track of this, too. It's easy to imagine that the boolean might get out of synch with the actual state of the Checkbox if you forget to include a line of code somewhere.

```
JCheckbox colorCb = new JCheckbox("Color");

if (colorCb.isSelected())
  ....

if (e.getSource() == someButton)
  colorCb.setSelected(false);
```

Here the code is simpler and more reliable because the information about the Checkbox state is kept in one place (in the object itself) and is always set and "get"ed from there.

Here's a somewhat subtle example:

wrong

```
class Line
  {
  Point start;
  Point end;
  double length;
  }
```

Here, length is defined by the start and the end Points. Change one and the length changes, but you might forget to store the new value into length. So now length is wrong.

right

```
class Line
  {
  Point start;
  Point end;
  double length()
    {
    //calc dist based on start and end
    return dist;
    };
  }
```

Here, length is a calculated value. The distance calculation will depend at any moment on the current values of start and end, so it will always be correct. The length information is stored in one place: the end points of the line.

17. Handle Exceptions - don't ignore them. At least write a line to the console reporting the condition.

wrong

1) Here the compiler forces you to catch the exception, but you just ignore it anyway. The rest of your program is unlikely to work, but it might not be evident until later when the program tries to create i/o streams.

```
try
  {
  s = new Socket(....)
  }
catch (IOException ioe)
  {
  }
```

right

1) Here the error is reported and the program is terminated immediately so the programmer can find out why the Socket creation failed and fix it.

```
try
  {
  s = new Socket(....)
  }
catch (IOException ioe)
  {
  System.out.println(
    "Socket creation failed" +
    ioe.getMessage());
```

(Gosling calls this "the dreaded curly-curly")

2) nextToken() can throw a NoSuchToken exception. If you don't catch it, you won't know what happened but your program will terminate. (You might be able to see the error on the system console.)

```
StringTokenizer stk =
  new StringTokenizer();

...
String s = stk.nextToken();
```

```
  System.exit(0);
  }
```

2) Here, the exception is just reported. In other cases, it might make sense to ignore it; in still others, it might cause a flag to be set to exit a loop that processes all tokens.

```
StringTokenizer stk =
  new StringTokenizer();

...
try
  {
  String s = stk.nextToken();
  }
catch (NoSuchTokenException nste)
  {
  System.out.println("no token..");
  }
```

18. A blank is *not* a String with no characters.

Check this out:

```
String s;
JTextField tf;
...
// clear the TextField
tf.setText(" ");
...
// now the user enters some text into tf by typing... let's say "Jim" is entered and
// the program gets it:
s = tf.getText();
// and now the program tests it:
if (s.equals("Jim") ...
```

and guess what? The test fails. The string returned from *getText()* is " Jim" - not "Jim" (with a leading space).

19. A null String is not a blank OR a String with no characters.

Consider each of the following and understand how they are different:

1. a reference to a String object that does not exist. A null reference. There is no object.

```
String s;
```

2. a String that contains no characters. The object exists. The String's length is 0

```
String s1 = "";
```

3. a String that contains 1 character - a blank. The object exists. The String's length is 1

```
String s2 = " ";
```

Documentation and Program Style

The most important principles of documentation are that it must be

- accurate
- clear
- consistent
- readable
- *correct!*

Also:

- Do not lie in your documentation.
- Tell the complete truth, not just part of it.
- Don't say a method does something when it doesn't.
- Do not copy documentation from one program to another and then "change it later".
- Use technical terms properly.

*Wrong documentation is **worse** than none!*

You should provide reasonable documentation in your programs including AT LEAST

- an explanation of the program's *overall* function and other important notes (at the top of the listing). This should focus on *what* the program does, not on *how* it does it. If special programming techniques are used, you can include a section called *Program Technical Notes* or explain these in the body of the program.
- The author(s) name(s) and the current date (at the top of the listing). Make a habit of changing the date (if necessary) at the end of *every* code editing session. (When you find 3 version of a program 6 months after you write it, and wonder which one is the "final" one, you will understand why you do this.)
- a short description of each method's purpose with documentation of arguments in many cases.
- line or section documentation of any unusual or obscure code.
- self-documenting variable and method names.
- any time you make a change to a working program (fix a bug, add a feature, change an implementation), make a note of the change and the date in the program's opening documentation box. In this way, you will have a log or record of changes to the program over its history. You don't need to do this for programs you hand in, but it's a good idea for the future.

When in doubt, ask yourself: *if I were reading this code for the first time, what would I like and need to know?* Wipe your mind clean of what you have just written and pretend you've never seen it before. This will usually tell you what you should do.

We are **very fussy** about this. After you get something to work, ask if this is the best/right way to do it. If there is a better way, do it that way.

An Interesting Viewpoint on Documentation Writing is actually my first and strongest line of defense for debugging code. If I can't write a simple comment explaining the code, then it's time to start redesigning and recoding. Almost all of the bugs that survive are typos (e.g. = instead of ==) and the few design bugs that survive are ones where I never really was able to write a simple comment. --Ronald Bourret on the xml-dev mailing list

Here are some other guidelines:

1. Name variables with clear and readable names. If you do this, you don't need to document their meanings when you declare them *and in addition* their meanings will be clear later in the code so you won't have to explain them every time you use them. Don't use misleading names.

Wrong	*Right*

`int rgCst;  //cost of a sq yard of carpet`
 	
`c = tCst * 10.5; //cost = total Cost * 105%`
 	
`getYards = sqFt/9;  // there's no "getting"`
	`int rugCostPerSqYd;`
	
	`totalCost = carpetCost * JOB_SURCHARGE;`
	
	`yards = sqFt/9;`

Other naming suggestions:

- the more global a name is, the more descriptive it should be since they can be referenced from lots of places and need to remind the programmer of what they are. For example, an object variable might be named *studentCount*, but a local loop counter variable might just be named *num* or *numStudents*, but *numberOfStudents* may be more than necessary.
- function names should be based on verbs: *calculateCost()*, *getTime()*. Data should be nouns: *cost*, *time*, *pointA*
- boolean variables should be named so that their value makes clear sense in a condition: *if (checkValid(c))* isn't clear because we don't know if the "check" returns true or false if c is ok; better to use *if (isValid(c))*

1a. Name Classes with meaningful names.

Wrong	*Right*

`class MyApplet ...`
 	
`class LeftPanel ...`
 	
`class XCanvas ...`
	`class MathEditorApplet ...`
	
	`class DesignDisplayPanel ...`
	
	`class WormAnimCanvas ...`

2. Don't tell us what we already know.

Wrong	*Right*

`/*****************************************`
`void init() is called once when the applet is`
`first loaded and is not called again.`
`*****************************************/`
	`/*************************************`
	`void init() sets up GUI components and`
	`loads questions from a text file.`
	`*************************************/`

```
or

total = amt + tax; //calc total = amount + tax

or

// Create a new Panel
p1 = new Panel();
```

```
or

total = amt + tax;

or

p1 = new Panel();
```

3. Declare variables in a neat and consistent manner. Note in the *Right* example that compile-time initialize variables are grouped together before the non-initialized variables.

Wrong

```
double baseCost, discount_cutoff = 250,
tax, totalCost, mimimumPurchase = 15.00,
something, another;
int numTransactions = 0, i, j;
```

Right

```
double numTransactions = 0,
       discountCutoff  = 250.00,
       minumumPurchase = 15.00;
       baseCost,
       tax,
       totalCost,
       something,
       another;

int    numTransactions = 0,
       i, j;
```

4. Don't break up code with documentation more than necessary. Document *before* a section of code, not *in* it whenever possible. In the *Wrong* example below, it's a little hard to see the code in among all the documentation.

Wrong

```
// calculate the specific gravity of
// the refrigerant
sg = wt/vol;
// now multiply that by the result of the
//gizmo matrix and store into temp
temp = getGizmo(matr);
// return the calculated result
return temp;
```

Right

```
// 1. calculate the specific gravity of the
//    refrigerant
// 2. multiply that by the result of the
//    gizmo matrix
//    and store into temp
// 3. return result

sg   = wt/vol;
temp = getGizmo(matr);
return temp;
```

5. Indent consistently. The precise style of indentation is less important than the following the style consistantly. Indent 2 - 4 spaces per level.

Wrong - three different styles for a simple if

```
if (x < 10) break;

if (y > 10) {
  do something;
  do somethingelse;
  }

if (j == 10)
  {
  do something;
  do somethingelse:
}
```

Right - one style for all

```
if (x < 10)
  break;

if (y > 10)
  {
  do something;
  do somethingelse;
  }

if (j == 10)
  {
  do something;
  do somethingelse:
  }
```

However, there is reason to prefer the style on the right. According to McConnell (see References below) the *Fundamental Theorem of Formatting* is: good visual layout shows the logical structure of the program. Indenting a block should show that the block is a logical subordinate of something, for example:

- the body of a loop
- a decision structure alternative
- the body of a method
- the contents of a class

The syntactical marks which delimit the block (the { and }) that is, which show the beginning and the end of this subordinate logical block therefore most appropriately belong with the block, indented the same way that it is.

Consider the following abstract layouts (adapted from McConnell, pp. 418, 419):

```
A  ********************** {
B     *********
C     *********
D     }
```

This saves1 line of space by placing the opening brace on statement A. However, the opening block delimiter is *not* part of whatever statement A is and so does not logically belong there. A variant of this, obtained by moving the closing brace in statement D

	to the left 2 spaces makes things even worse (see the example below).
A ********** B { C ******* D ******* E }	Here, is statement B subordinate to statement A? It doesn't look like part of statement A and it doesn't look like it's subordinate to it either. Same idea for E
A ********** B { C ********* D ********* E }	Statements C an D are indented as if they were subordinate to statement B. They are not. It is also more complex than it needs to be - there are two levels of indenting but only one level of logical subordination.
A ********** B { C ********** D ********** E }	This version correctly places the {} at the same logical level as the block of code they delimit. There is one level of indenting and one level of logical subordination.

6. Use white space.

The spacebar is not wired to an electric shock machine. Hitting the space bar does not cost money. It is free. Same for the <Enter> key. Use them

- to separate arguments in method calls
- to separate declared variables
- to separate operands and operators (usually)
- to separate logical sections of code

But don't separate *every* line of code. If the white line is to have a meaning (i.e. this is something new here - a break, a logical difference from the code above) then using it too much will rob it of any meaning.

Wrong	*Right*

`// no sep args with commas`
`stdDeviation = std(x,squareOfx,n);`
 	
`// no sep var declrs`
`int x,y,z,num,gizmo,somethingelse;`
 	
`// no sep operands`
`x=-b+sqrt(b*b-4*a*c);`
 	
`// no sep logical lines`
`tf1=new TextField();`
`tf2=new TextField();`
`cbg=new CheckboxGroup();`
`cbRed=new Checkbox("red",cbg,true);`
`cbGreen=new Checkbox("green",cbg,false);`
`cbBlue=new Checkbox("blue",cbg,false);`
	`// sep args with commas`
	`stdDeviation = std(x, squareOfx, n);`
	
	`// sep var declrs`
	`int x, y, z, num, gizmo, somethingelse;`
	
	`// sep operands`
	`x = -b + sqrt(b*b - 4*a*c);`
	
	`// sep logical lines`
	`tf1 = new TextField();`
	`tf2 = new TextField();`
	
	`cbg = new CheckboxGroup();`
	`cbRed = new Checkbox("red", cbg, true);`
	`cbGreen = new Checkbox("green", cbg, false);`
	`cbBlue = new Checkbox("blue", cbg, false);`

7. Use meaningful variable and method names and adopt consistent naming conventions. Use standard Java capitalization: underscores are rarely used; rather, each important word in a name is capitalized, but while class names start with a capital letter, variables and method names do not:

- class RangeException
- double getPosNum(...)
- double totalCost

Wrong	*Right*

`double ss, scnt, sdev, sav;`
 	
`double yards, Price, tax_amt;`
`String priceAsString, taxIn;`
 	
`double get_price() {...}`
`double yardsIn() {...}`
`double getTax() {...}`
	`double studentScore,`
	`studentCount,`
	`studentStandDev,`
	`studentAvg;`
	
	`double yards, price, taxAmt;`
	`String priceStr, taxStr;`
	
	`double getPrice() {...}`
	`double getYards() {...}`
	`double getTax() {...}`

Here is a complete description of Java Naming Conventions:

Identifier Type	Rules for Naming	Examples

Classes	Class names should be nouns, in mixed case with the first letter of each word capitalized (including the first). Try to keep your class names simple and descriptive. Use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).	class Raster; class ImageSprite;

Methods	Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.	run();
		runFast();
		getBackground();

Variables	Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore _ or dollar sign $ characters, even though both are allowed.
	
	Variable names should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are `i`, `j`, `k`, `m`, and `n` for integers; `c`, `d`, and `e` for characters.
		`int numScores;`
		`char userChoice;`
		`float average;`

Constants	The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_"). (ANSI constants should be avoided, for ease of debugging.)	static final int MIN_WIDTH = 4;
		
		static final int MAX_WIDTH = 999;
		
		static final int GET_THE_CPU = 1;

8. Keep lines short enough to print without wrapping. Find a print program that will print things neatly (UltraEdit?). Notepad is not the best. If you need line comments and they are long, put them *above* the line of code they document. (Also see point 4 above)

When an expression will not fit on a single line, break it according to these general principles:

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line.
- If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

Wrong	*Right*

1)
`result = methodCall(parm1, parm2, parm3); // calculate the gizmo from blah blah`
 	
2)
`x = (-b + sqrt(b*b - 4*a*c)) / (2*a) + p*(p - s1) * (p - s2) * (p-s3);`
	1)
	`{`
	`// calculate the gizmo from blah blah`
	`result = methodCall(parm1, parm2, parm3);`
	`}`
	
	or even better
	
	`{`
	`// calculate the gizmo from blah blah`
	`result = methodCall(parm1,`
	`parm2,`
	`parm3);`
	`}`
	
	`2) x = (-b + sqrt(b*b - 4*a*c)) / (2*a) +`
	`p *`
	`(p - s1) *`
	`(p - s2) *`
	`(p-s3);`

9. Line up groups of assignment statements (and declarations) where possible. See points 3, 4, 6 for small examples. You can't always do this, and it is very hard to do it the same amount at all places in the code, but even doing it in small groups helps a lot.

Wrong	*Right*

`TextField priceIn = new TextField();`
`TextField itemQuantity = new TextField();`
`TextField total = new TextField();`
 	
`sum = 0;`
`count = 0;`
`minDeposit = 1000.00;`
 	
`final int MAX_SCORE = 100, MIN_SCORE = 0,`
`MIN_A = 90, MIN_B = 80, MIN_C = 70,`
`MIN_D = 60;`
	`TextField priceIn = new TextField();`
	`TextField itemQuantity = new TextField();`
	`TextField total = new TextField();`
	
	`sum = 0;`
	`count = 0;`
	`minDeposit = 1000.00;`
	
	`final int MAX_SCORE = 100,`
	`MIN_SCORE = 0,`
	`MIN_A = 90,`
	`MIN_B = 80,`
	`MIN_C = 70,`
	`MIN_D = 60;`

10. Use boolean variables to record boolean conditions. Don't *ever* name them *flag*. Name them so you know what they are about.

Wrong	*Right*
`String ok = "yes"; // ok is too vague`	`boolean gotInput = true;`

```
while (ok.equals("yes"))
  {
  .....
  if (somecond)
  ok = "no";
  }
```

```
while (gotInput)
  {
  ....
  if (somecond)
    gotInput = false;
  }
```

11. Use Parentheses

It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others-you shouldn't assume that other programmers know precedence as well as you do.

```
if (a == b && c == d)      // AVOID!
if ((a == b) && (c == d)) // RIGHT
```

12. Document any *special* error detection and handling that your code performs in some appropriate place, including special Exception classes or techniques to prevent invalid input.

13. Don't comment bad code. Rewrite it. If you find that you have to explain a lot then maybe the code itself is bad and could be clarified by re-thinking and re-writing it.

14. Don't use inheritance on documentation. That is, don't inherit the documentation from the program you just wrote into the program you are now writing (in other words, don't copy documentation from the old one and paste it into the new one) and figure you will fix it later. You might, but then again you might not. And then six months later when your read it, you will be reading lies. And you will be confused. And you won't know what to trust and what not to trust.

Conclusion

It may seem that there is a lot to remember here. This is probably true. You will not be able to remember all of these points as you are writing code because you are concentrating on getting the code right, not on how it looks or on how efficient or elegant it is.

Nevertheless, you should try. Start by concentrating on one or two of these guidelines as you write. Soon (a day, a week, ...) these two will become habit and you won't have to think about them all the time. Then move on to trying to remember a couple more. Over time, most of these things will become automatic and you will not have to spend too much time reviewing and re-writing your code after it is "done" but before you submit it for public viewing (or grading).

In the meantime, before you do all these thing automatically, you should plan on revising your code. You can do this as you write (one method at a time, or at the end of a coding session) or you can take a hour or two (or whatever you need) after the program is "done" to go back and apply these principles and guidelines to your program.

Re-formatting is fairly fast and easy, and you get a big improvement for little effort. Re-thinking variable and method names may take longer - but once you have made a decision to change, your editor's global search-and-replace can help. But be careful not to change something that shouldn't be changed by accident. Using the "confirm each replacement" option is probably wise. Other issues will require deeper thought - but you should make the effort. Always ask yourself: Is this the best way to do this? Is there a better way? Be prepared to make changes when you see they should be made.

After you have done all this, your program is (maybe) "done".

References and Further Reading

1. *The Practice of Programming*, Brian Kernighan and Rob Pike, Addison Wesley, 1999. ISBN 0-201-61586-X

2. *Code Complete*, Steve McConnell, Microsoft Press, 1993. ISBN 1-55615-484-4

3. *The Pragmatic Programmer*, Andrew Hunt and David Thomas, Addison Wesley Longman 2000. ISBN 0-201-61622-X