

Appendix

A1

Java Language Coding Guidelines

A1.1 Introduction

This coding style guide is a simplified version of one that has been used with good success both in industrial practice and for college courses.

A style guide is a set of mandatory requirements for layout and formatting. Uniform style makes it easier for you to read code from your instructor and classmates. You will really appreciate that if you do a team project. It is also easier for your instructor and your grader to grasp the essence of your programs quickly.

A style guide makes you a more productive programmer because it *reduces gratuitous choice*. If you don't have to make choices about trivial matters, you can spend your energy on the solution of real problems.

In these guidelines, several constructs are plainly outlawed. That doesn't mean that programmers using them are evil or incompetent. It does mean that the constructs are not essential and can be expressed just as well or even better with other language constructs.

If you already have programming experience, in Java or another language, you may be initially uncomfortable at giving up some fond habits. However, it is a sign of professionalism to set aside personal preferences in minor matters and to compromise for the benefit of your group.

These guidelines are necessarily somewhat dull. They also mention features that you may not yet have seen in class. Here are the most important highlights:

- ◆ Tabs are set every three spaces.
- ◆ Variable and method names are lowercase, with occasional upperCase characters in the middle.

- ◆ Class names start with an Uppercase letter
- ◆ Constant names are UPPERCASE, with an occasional UNDER_SCORE.
- ◆ There are spaces after keywords and surrounding binary operators.
- ◆ Braces must line up horizontally or vertically.
- ◆ No magic numbers may be used.
- ◆ Every method, except for `main` and overridden library methods, must have a comment.
- ◆ At most 30 lines of code may be used per method.
- ◆ No `continue` or `break` is allowed.
- ◆ All non-`final` variables must be private.

Note to the instructor: Of course, many programmers and organizations have strong feelings about coding style. If this style guide is incompatible with your own preferences or with local custom, please feel free to modify it. For that purpose, this coding style guide is available in electronic form from the author.

A1.2 Source Files

Each Java program is a collection of one or more source files. The executable program is obtained by compiling these files. Organize the material in each file as follows:

- ◆ `package` statement, if appropriate
- ◆ `import` statements
- ◆ A comment explaining the purpose of this file
- ◆ A `public` class
- ◆ Other classes, if appropriate

The comment explaining the purpose of this file should be in the format recognized by the `javadoc` utility. Start with a `/**`, and use the `@author` and `@version` tags:

```
/**
 * COPYRIGHT (C) 1997 Harry Hacker. All Rights Reserved.
 * Classes to manipulate widgets.
 * Solves CS101 homework assignment #3
 * @author Harry Hacker
 * @version 1.01 1997/2/15
 */
```

A1.3 Classes

Each class should be preceded by a class comment explaining the purpose of the class.

List features in the following order:

- ◆ `public` instance methods
- ◆ `public static` methods
- ◆ `public static final` constants
- ◆ `private` instance methods
- ◆ `private static` methods
- ◆ `private` instance variables
- ◆ `private static final` constants
- ◆ `private static` variables (but see the last paragraph in this section)
- ◆ Inner classes

Leave a blank line after every method.

All non-`final` variables must be `private`. (However, instance variables of a `private` inner class may be `public`.) Methods and final variables can be either `public` or `private`, as appropriate.

All features must be tagged `public` or `private`. Do not use the default visibility (that is, package visibility) or the `protected` attribute.

Avoid static variables (except `final` ones) whenever possible. In the rare instance that you need static variables, you are permitted one static variable per file.

A1.4 Methods

Every method (except for `main`) starts with a comment in `javadoc` format.

```
/**
 * Convert calendar date into Julian day.
 * Note: This algorithm is from Press et al., Numerical Recipes
 * in C, 2nd ed., Cambridge University Press, 1992
 * @param day day of the date to be converted
 * @param month month of the date to be converted
 * @param year year of the date to be converted
 * @return the Julian day number that begins at noon of the
 * given calendar date.
 */
```

```
public static int dat2jul(int day, int month, int year)
{ . . .
}
```

Parameter names must be explicit, especially if they are integers or Boolean:

```
public Employee remove(int d, double s)
    // Huh?
public Employee remove(int department, double severancePay)
    // OK
```

Of course, for very generic methods, short parameter names may be very appropriate:

```
public static void sort(int[] a)
    // OK
```

Methods must have at most 30 lines of code. The method header, comments, blank lines, and lines containing only braces are not included in this count. Methods that consist of one long `if/else` or `switch` may be longer, provided each branch is 10 lines or fewer. This rule forces you to break up complex computations into separate methods.

A1.5 Variables and Constants

Do not define all variables at the beginning of a block:

```
public static double squareRoot(double a)
{ double xold; // Don't
  double xnew;
  boolean more;
  . . .
}
```

Define each variable just before it is used for the first time:

```
public static double squareRoot(double a)
{ . . .
  while (more)
  { double xnew = (xold + a / xold) / 2;
    . . .
  }
  . . .
}
```

Do not define two variables on the same line:

```
int dimes = 0, nickels = 0; // Don't
```

In Java, constants must be defined with the keyword `final`. If the constant is used by multiple methods, declare it as `static final`. It is a good idea to define static final variables as `private` if no other class has an interest in them.

Do not use *magic numbers*! A magic number is a numeric constant embedded in code, without a constant definition. Any number except `-1`, `0`, `1`, and `2` is considered magic:

```
if (p.getX() < 300) // Don't
Use final variables instead:
final double WINDOW_WIDTH = 300;
. . .
if (p.getX() < WINDOW_WIDTH) // OK
```

Even the most reasonable cosmic constant is going to change one day. You think there are 365 days per year? Your customers on Mars are going to be pretty unhappy about your silly prejudice. Make a constant

```
public static final int DAYS_PER_YEAR = 365;
```

so that you can easily produce a Martian version without trying to find all the 365s, 364s, 366s, 367s, and so on, in your code.

A1.6 Control Flow

A1.6.1 The `if` Statement

Avoid the “`if...if...else`” trap. The code

```
if ( ... )
    if ( ... ) ...;
else ...;
```

will not do what the indentation level suggests, and it can take hours to find such a bug. Always use an extra pair of `{...}` when dealing with “`if...if...else`”:

```
if ( ... )
{ if ( ... ) ...;
} // {...} are necessary
else ...;
```

```
if ( ... )
{ if ( ... ) ...;
  else ...;
} // {...} not necessary, but they keep you out of trouble
```

A1.6.2 The for Statement

Use `for` loops only when a variable runs from somewhere to somewhere with some constant increment/decrement:

```
for (int i = 0; i < a.length; i++)
    System.out.println(a[i]);
```

Do not use the `for` loop for weird constructs such as

```
for (a = a / 2; count < ITERATIONS; System.out.println(xnew))
    // Don't
```

Make such a loop into a `while` loop. That way, the sequence of instructions is much clearer.

```
a = a / 2;
while (count < ITERATIONS) // OK
{
    . . .
    System.out.println(xnew);
}
```

A1.6.3 Nonlinear Control Flow

We recommend that you not use the `switch` statement, because it is easy to fall through accidentally to an unwanted case. Use `if/else` instead.

We recommend that you not use the `break` or `continue` statements. Use another `boolean` variable to control the execution flow.

A1.6.4 Exceptions

Do not tag a method with an overly general exception specification:

```
Widget readWidget(Reader in)
    throws Exception // Bad
```

Instead, specifically declare any checked exceptions that your method may throw:

```
Widget readWidget(Reader in)
    throws IOException, MalformedWidgetException // Good
```

Do not “squelch” exceptions:

```
try
{
    double price = in.readDouble();
}
catch (Exception e)
{} // Bad
```

Beginners often make this mistake “to keep the compiler happy”. If the current method is not appropriate for handling the exception, simply use a `throws` specification and let one of its callers handle it.

A1.7 Lexical Issues

A1.7.1 Naming Convention

The following rules specify when to use upper- and lowercase letters in identifier names.

- ◆ All variable and method names and all data fields of classes are in lowercase (maybe with an occasional upperCase in the middle); for example, `firstPlayer`.
- ◆ All constants are in uppercase (maybe with an occasional UNDER_SCORE); for example, `CLOCK_RADIUS`.
- ◆ All class and interface names start with uppercase and are followed by lowercase letters (maybe with an occasional UpperCase letter); for example, `BankTeller`.

Names must be reasonably long and descriptive. Use `firstPlayer` instead of `fp`. No dropping of vowels. Local variables that are fairly routine can be short (`ch`, `i`) as long as they are really just boring holders for an input character, a loop counter, and so on. Also, do not use `ctr`, `c`, `cntr`, `cnt`, `c2` for variables in your method. Surely these variables all have specific purposes and can be named to remind the reader of them (for example, `current`, `next`, `previous`, `result`, ...).

A1.7.2 Indentation and White Space

Use tab stops every three columns. That means you will need to change the tab stop setting in your editor!

Use blank lines freely to separate parts of a method that are logically distinct.

Use a blank space around every binary operator:

```
x1 = (-b - Math.sqrt(b * b - 4 * a * c)) / (2 * a); // Good
x1=(-b-Math.sqrt(b*b-4*a*c))/(2*a);//Bad
```

Leave a blank space after (and not before) each comma, semicolon, and keyword, but not after a method name:

```
if (x == 0) y = 0;
f(a, b[i]);
```

Every line must fit on 80 columns. If you must break a statement, add an indentation level for the continuation:

```
a[n] = .....
      + .....
```

Start the indented line with an operator (if possible).

If the condition in an `if` or `while` statement must be broken, be sure to brace the body in, *even if it consists of only one statement*:

```
if ( .....
    && .....
    || ..... )
{ . . .
}
```

If it weren't for the braces, it would be hard to separate the continuation of the condition visually from the statement to be executed.

A1.7.3 Braces

Opening and closing braces must line up, either horizontally or vertically:

```
while (i < n) { System.out.println(a[i]); i++; }

while (i < n)
{ System.out.println(a[i]);           // OK
  i++;
}
```

Some programmers place the `{` in a line all by itself:

```
while (i < n)
{                                     // OK
  System.out.println(a[i]);
  i++;
}
```

That is fine, but it does take away a line of precious screen space. If you like the first approach (which is the one used in this book), be sure to type a tab after the opening brace.

Some programmers don't line up vertical braces but place the `{` behind the key word:

```
while (i < n) {                       // Don't
  System.out.println(a[i]);
  i++;
}
```

Doing so makes it hard to check that the braces match.

A1.7.4 Unstable Layout

Some programmers take great pride in lining up certain columns in their code:

```
firstRecord = in.readInt();
lastRecord  = in.readInt();
cutoff      = in.readDouble();
```

This is undeniably neat, but the layout is not *stable* under change. A new variable name that is longer than the preallotted number of columns requires that you move *all* entries around:

```
firstRecord      = in.readInt();
lastRecord       = in.readInt();
cutoff           = in.readDouble();
marginalFudgeFactor = in.readDouble();
```

This is just the kind of trap that makes you decide to use a short variable name like `mff` instead.

Do not use `//` comments for comments that extend for more than two lines. You don't want to have to move the `//` around when you edit the comment.

```
// comment — don't do this
// more comment
// more comment
```

Use `/* ... */` comments instead. When using `/* ... */` comments, don't “beautify” them with additional asterisks:

```
/* comment—don't do this
 * more comment
 * more comment
 */
```

It looks neat, but it is a major disincentive to update the comment. Some people have text editors that lay out comments. But even if you do, you don't know whether the next person who maintains your code has such an editor.

Instead, format long comments like this:

```
/*
  comment
  more comment
  more comment
*/
```

or this:

```
/*
comment
more comment
more comment
*/
```

These comments are easier to maintain as your program changes. If you have to choose between pretty but unmaintained comments and ugly comments that are up to date, truth wins over beauty.

