Fourth AP Edition

Java Methods

Object-Oriented Programming and Data Structures

Maria Litvin
Phillips Academy, Andover, Massachusetts

Gary Litvin
Skylight Software, Inc.

Skylight Publishing
Andover, Massachusetts

Skylight Publishing
9 Bartlet Street, Suite 70
Andover, MA 01810

web: http://www.skylit.com
e-mail: sales@skylit.com
support@skylit.com

Library of Congress Control Number: 2021944689

ISBN 978-0-9972528-2-8

Appendix A: The 17 Bits of Style

> The language is perpetually in flux: it is a living stream, shifting, changing, receiving new strength from a thousand tributaries, losing old forms in the backwaters of time. To suggest that a young writer not swim in the main stream of this turbulence would be foolish indeed, and such is not the intent of these cautionary remarks. The intent is to suggest that in choosing between formal and the informal, the regular and the offbeat, the general and the special, the orthodox and the heretical, the beginner err on the side of conservatism, on the side of established usage.
>
> William Strunk Jr. and E.B. White, *The Elements of Style*

Style is a crucial component of professionalism in software development. Clean code that follows stylistic conventions is easier to read, maintain, and share with colleagues. Programmers who code in good style are less likely to have silly bugs and will actually spend less time developing and debugging their code. Finally, good style in programs is a concern because for us, humans, style and aesthetics are a concern in everything we do.

Following stylistic conventions is easy, and, after a little practice, becomes second nature. Occasionally, an especially independent-minded student resists all conventions, arguing, "But it works!" To this person we can point out two things. First, a programmer's product is not an executable program but its source code. In the current environment the life expectancy of a "working" program is just a few months, sometimes weeks. On the other hand, source code, updated periodically, can live for years; to be of any use, it must be readable to others. Second, bad or unconventional style is uncool. It immediately gives away an amateur, a kind of social misfit in the software developers' culture. As nerdy as this culture might be, it has its own sense of aesthetic pride.

In *The Elements of Style*, William Strunk wrote:

> There is no satisfactory explanation of style, no infallible guide to good writing, no assurance that a person who thinks clearly will be able to write clearly, no key that unlocks the door, no inflexible rule by which the young writer may shape his course.

These words, which come in the book's closing chapter, describe (and illustrate) the deeper meaning of style — after the straightforward rules of correct usage have been discussed and illustrated in the previous chapters. In programming, too, style has a deeper meaning: it is that elusive quality which makes one person's code elegant and easy to follow and another person's convoluted and obscure although in line with all the superficial stylistic conventions. The mystery is not as wide open in writing code as in creative writing: there are many firm design principles, and "a person who thinks clearly" usually is able to code clearly. Still, some mystery remains. Here we discuss only the superficial stylistic conventions, leaving the deeper meaning alone.

Bit 0. At the top of each source module, put in a comment that states the purpose of the class, its author, date of completion and other pertinent information, such as a copyright message, special instructions on how to run the program, data files that your program reads or creates, and a history of revisions.

Bit 1. Place all `import` directives at the top of your source module. Start with standard Java packages followed by your own or your organization's packages.

Bit 2. Separate different methods in a class with blank lines and separator comment lines (or more formal documentation comments for *javadoc*). Split your code into "paragraphs" that represent meaningful steps or actions in your program by inserting blank lines and, if necessary, comment lines.

Bit 3. Place each statement on a separate line. Indent.

Java code is usually indented by two character positions in the body of a method, and within braces under `if`, `else`, `switch`, and `for`, `while`, and `do-while` loops.

Bit 4. Keep braces visible.

There are different styles of placing braces (naturally, this is a subject of heated debates). Many programmers place the opening brace at the end of the previous line, and the closing brace on a separate line. In our Java book we place both the opening and the closing braces on separate lines.

Bit 5. Use spaces liberally. Do not cram things together.

There are no clear-cut rules on where to add spaces — different people have different tastes. We use spaces on both sides of the assignment operator and other binary operators: arithmetic, logical, relational. For example, we find

```
for (i=0;i<n;i++)
    sum+=scores[i];
```

less readable than

```
for (i = 0; i < n; i++)
    sum += scores[i];
```

With longer names, spaces become more important. For example:

```
taxAmount=saleAmount*taxRate;
```

appears too crammed. We prefer

```
taxAmount = saleAmount * taxRate;
```

Unary operators, such as - (negation) , ++, --, !, are usually attached to their operands. For example:

```
if (! match)
  count --;
```

is too fancy. People normally write:

```
if (!match)
  count--;
```

Most people do not use spaces around brackets, or the "dot" class member access operators.

```
Color c = Color . white;
```

compiles correctly, but you won't see it in programs.

Some people like to leave spaces on both sides of a parenthesis in expressions and in method calls. We normally leave a space only before an opening parenthesis and after a closing parenthesis.

Bit 6. Omit needless parentheses.

Java uses a well-defined order of precedence among operators. All unary operators have higher rank than (are applied before) all binary operators. Among the binary operators, arithmetic operators have higher rank than relational operators, which in turn apply before logical operators. There is no need to clutter your code with too many parentheses. For example, instead of

```
if ((!match) && (i < (length - 1)))...
```

you can write

```
if (!match && (i < length - 1))...
```

or even

```
if (!match && i < length - 1)...
```

You have to be a little more careful with `&&` and `||`: the `&&` operator has a higher rank than `||`. For example, you need to keep the parentheses in

```
if (year % 4 == 0 && (year % 100 != 0 || year % 400 == 0))...
```

Occasionally, parentheses facilitate reading long logical expressions:

```
if ((x > -3 && x < - 1) || (x > 1 && x < 3))...
```

Use parentheses whenever you are unsure about the order of operations.

Bit 7. Comment each method: state its purpose, arguments, assumptions ("preconditions"), results ("postconditions"), and return value.

Bit 8. Avoid redundant comments.

```
if (a[i] >= 0)
  count++;   // Increment the count
```

Bit 9. Self-explanatory code is usually better than heavily commented code.

For example, instead of

```
if (s.equals("MA") && a < 16)  // If state is Massachusetts and
                               //   age is less than Massachusetts
                               //   legal driving age...
```

it is better to write:

```
final int massDrivingAge = 16;
if (stateCode.equals("MA") && age < massDrivingAge)
```

Bit 10. Explain your intentions ahead of time in difficult algorithms.

```
// Going backwards from a[n-1], shift elements to the
//   right until you find an element a[i] <= aTemp:
i = n;
while (i > 0 && aTemp < a[i-1])
{
  a[i] = a[i-1];
  i--;
}
```

Bit 11. Comment obscure code or unusual usage.

```
int mask = KeyEvent.SHIFT_MASK | KeyEvent.CTRL_MASK;

// If both control and shift are pressed:
if ((e.getModifiers() & mask) == mask)
  ...
```

Bit 12. Use meaningful names.

```
totalAmt = saleAmt * (1 + taxRate);
```

is better than

```
tot = a * (1 + r);
```

But overly long COBOL-style names will clutter your code:

```
totalAmountDue = totalSaleAmount * (1 + salesTaxRate);
```

In Java, method names should usually sound like verbs. Make names of classes, objects, and variables sound like nouns.

Method names do not have to be too long. The same name may be used for methods that take different numbers or types of arguments (a feature known as *overloading*). Methods in different classes may have the same name, too. In fact, the object-oriented programming approach encourages the use of the same name for methods that perform semantically similar tasks. The object's name often combines with the method's name and the argument's name to create a readable phrase:

```
display.setText(message);
```

Bit 13. It is acceptable to use simple names for loop-control variables and other temporary "throw-away" variables. Declare these variables locally in each method and use the same name where appropriate. Note that longer names, such as `index`, `subscript`, `counter`, `lcv`, or `loopControlVariable` are no more meaningful than `i` or `k`. Likewise, `xCoordinate` is no more expressive than `x`.

If possible, use the same name for similar purposes. For example, the same names for local variables `row` and `col` may be used in different methods to indicate row and column in a 2-D array.

Use names that make sense in context. For example:

```
for (student = 0; student < numStudents; student++)
  sum += gpa[student];
```

Bit 14. Use the upper and lower case consistently.

Java is case-sensitive, and all reserved words are lowercase. Other than that, there are no syntax rules for using the upper or lower case. However, you should follow the Java style:

- Start all class names with a capital letter.
- Start all names of variables and methods with a lowercase letter and capitalize subsequent words.
- Use all caps occasionally for prominent constants.

Bit 15. Code defensively.

```
while (n-- > 0)
{
  ...
}
```

may save one line of code, but it may also lead to a bug if `n` is used inside the loop.

```
while (n > 0)
{
    ...
    n--;
}
```

is safer.

Bit 16. *Non Sibi* ("Not for Self") is the motto of Phillips Academy in Andover. Keep these words in mind when writing code.