The full document is available at http://geosoft.no/development/javastyle.html

1 Introduction

This document lists Java coding recommendations common in the Java development community.

The recommendations are based on established standards collected from a number of sources, individual experience, local requirements/needs, as well as suggestions given in [1], [2], [3], [4] and [5].

Woodley says: I have removed quite a bit of Java-specific from this document. Rather than trying to completely re-write it I have just removed things that I thought were not universal enough to keep. Some things will not apply and those writing code using these guidelines will have to use their judgment. Corrections can be made during code reviews. Questions about application of rules can also be asked during code reviews.

Layout for the recommendations is as follows:

n. Guideline short description
`Example if applicable`
Motivation, background and additional information.

1.2 Recommendation Importance

. A *must* requirement must be followed,

a *should* is a strong recommendation, and

a *can* is a general guideline.

2 General Recommendations

1. Any violation to the guide is allowed if it enhances readability.
The main goal of the recommendation is to improve readability and thereby the understanding and the maintainability and general quality of the code. It is impossible to cover all the specific cases in a general guide and the programmer should be flexible.

3 Naming Conventions

3.2 Specific Naming Conventions

13. The terms *get/set* must be used where an attribute is accessed directly.
`employee.getName();`

```
employee.setName(name);

matrix.getElement(2, 4);
matrix.setElement(2, 4, value);
```

Common practice in the Java community and the convention used by Sun for the Java core packages.

14. *is* prefix should be used for boolean variables and methods.

```
isSet, isVisible, isFinished, isFound, isOpen
```

This is the naming convention for boolean methods and variables used by Sun for the Java core packages.

Using the *is* prefix solves a common problem of choosing bad boolean names like *status* or *flag*. *isStatus* or *isFlag* simply doesn't fit, and the programmer is forced to chose more meaningful names.

Setter methods for boolean variables must have *set* prefix as in:

```
  void setFound(boolean isFound);
```

There are a few alternatives to the *is* prefix that fits better in some situations. These are *has*, *can* and *should* prefixes:

```
  boolean hasLicense();
  boolean canEvaluate();
  boolean shouldAbort = false;
```

15. The term *compute* can be used in methods where something is computed.

```
valueSet.computeAverage();
matrix.computeInverse()
```

Give the reader the immediate clue that this is a potential time consuming operation, and if used repeatedly, he might consider caching the result. Consistent use of the term enhances readability.

16. The term *find* can be used in methods where something is looked up.

```
vertex.findNearestVertex();
matrix.findSmallestElement();
node.findShortestPath(Node destinationNode);
```

Give the reader the immediate clue that this is a simple look up method with a minimum of computations involved. Consistent use of the term enhances readability.

17. The term *initialize* can be used where an object or a concept is established.

```
printer.initializeFontSet();
```

The American *initialize*should be preferred over the English *initialise*. Abbreviation *init* must be avoided.

19. Plural form should be used on names representing a collection of objects.

```
Collection<Point>  points;
int[]              values;
```

Enhances readability since the name gives the user an immediate clue of the type of the variable and the operations that can be performed on its elements.

20. *n* prefix should be used for variables representing a number of objects.

```
nPoints, nLines
```

The notation is taken from mathematics where it is an established convention for indicating a number of objects.

Note that Sun use *num* prefix in the core Java packages for such variables. This is probably meant as an abbreviation of *number of*, but as it looks more like *number* it makes the variable name strange and misleading. If "number of" is the preferred phrase, *numberOf* prefix can be used instead of just *n*. *num* prefix must not be used.

21. *Num* suffix should be used for variables representing an entity number.

```
tableNum, employeeNum
```

The notation is taken from mathematics where it is an established convention for indicating an entity number.

An elegant alternative is to prefix such variables with an *i*: `iTable, iEmployee`. This effectively makes them *named* iterators.

22. Iterator variables should be called *i*, *j*, *k* etc.

```
for (Iterator i = points.iterator(); i.hasNext(); ) {
  :
}

for (int i = 0; i < nTables; i++) {
  :
}
```

The notation is taken from mathematics where it is an established convention for indicating iterators.

Variables named *j*, *k* etc. should be used for nested loops only.

Woodley's note: *Make sure that what you are using is actually just an iterator. Just because you iterate using this index doesn't mean that you can always use a generic iterator. My favorite example is when moving through a raster. That's not just an iteration, you are moving through rows and columns and should name your iterator variables accordingly*.

23. Complement names must be used for complement entities [1].

```
get/set, add/remove, create/destroy, start/stop, insert/delete,
increment/decrement, old/new, begin/end, first/last, up/down,
min/max,
next/previous, old/new, open/close, show/hide, suspend/resume,
```

```
etc.
```

Reduce complexity by symmetry.

30. Classes that create instances on behalf of others (*factories*) can do so through method *new[ClassName]*

```
class PointFactory
{
  public Point newPoint(...)
  {
    ...
  }
}
```

Indicates that the instance is created by *new* inside the factory method and that the construct is a controlled replacement of `new Point()`.

31. Functions (methods returning an object) should be named after what they return and procedures (*void* methods) after what they do.

Increase readability. Makes it clear what the unit should do and especially all the things it is *not* supposed to do. This again makes it easier to keep the code clean of side effects.

5 Statements

5.2 Classes and Interfaces

40. Class and Interface declarations should be organized in the following manner:

1. **Class/Interface documentation.**
2. **`class` or `interface` statement.**
3. **Class (`static`) variables in the order `public`, `protected`, package (no access modifier), `private`.**
4. **Instance variables in the order `public`, `protected`, package (no access modifier), `private`.**
5. **Constructors.**
6. **Methods (no specific order).**

Reduce complexity by making the location of each class element predictable.

5.3 Methods

41. Method modifiers should be given in the following order:
<access> static abstract synchronized <unusual> final native

The *<access>* modifier (if present) must be the first modifier.

```
public static double square(double a);  // NOT: static public
double square(double a);
```

<access> is one of *public*, *protected* or *private* while *<unusual>* includes *volatile* and *transient*. The most important lesson here is to keep the *access* modifier as the first modifier. Of the possible modifiers, this is by far the most important, and it must stand out in the method declaration. For the other modifiers, the order is less important, but it make sense to have a fixed convention.

5.4 Types

42. Type conversions must always be done explicitly. Never rely on implicit type conversion.

```
floatValue = (int) intValue; // NOT: floatValue = intValue;
```

By this, the programmer indicates that he is aware of the different types involved and that the mix is intentional.

5.5 Variables

44. Variables should be initialized where they are declared and they should be declared in the smallest scope possible.

This ensures that variables are valid at any time. Sometimes it is impossible to initialize a variable to a valid value where it is declared. In these cases it should be left uninitialized rather than initialized to some phony value.

45. Variables must never have dual meaning.

Enhances readability by ensuring all concepts are represented uniquely. Reduce chance of error by side effects.

46. Class variables should never be declared public.

The concept of information hiding and encapsulation is violated by public variables. Use private variables and access functions instead. One exception to this rule is when the class is essentially a data structure, with no behavior (equivalent to a C++ `struct`). In this case it is appropriate to make the class' instance variables public [2].

47. Arrays should be declared with their brackets next to the type.

```
double[] vertex;  // NOT: double vertex[];
int[]    count;   // NOT: int    count[];

public static void main(String[] arguments)
```

```
public double[] computeVertex()
```

The reason for is twofold. First, the *array-ness* is a feature of the class, not the variable. Second, when returning an array from a method, it is not possible to have the brackets with other than the type (as shown in the last example).

48. Variables should be kept alive for as short a time as possible.

Keeping the operations on a variable within a small scope, it is easier to control the effects and side effects of the variable.

5.6 Loops

49. Only loop control statements must be included in the *for()* construction.

```
sum = 0;                              // NOT: for (i = 0, sum = 0; i < 100; i++)
for (i = 0; i < 100; i++)                        sum += value[i];
  sum += value[i];
```

Increase maintainability and readability. Make a clear distinction of what *controls* and what is *contained* in the loop.

50. Loop variables should be initialized immediately before the loop.

```
isDone = false;              // NOT: bool isDone = false;
while (!isDone) {            //       :
  :                          //       while (!isDone) {
}                            //         :
                             //       }
```

51. The use of *do-while* loops can be avoided.

do-while loops are less readable than ordinary *while* loops and *for* loops since the conditional is at the bottom of the loop. The reader must scan the entire loop in order to understand the scope of the loop.

In addition, *do-while* loops are not needed. Any *do-while* loop can easily be rewritten into a *while* loop or a *for* loop. Reducing the number of constructs used enhance readbility.

52. The use of *break* and *continue* in loops should be avoided.

These statements should only be used if they prove to give higher readability than their structured counterparts.

5.7 Conditionals

53. Complex conditional expressions must be avoided. Introduce temporary boolean variables instead [1].

```
bool isFinished = (elementNo < 0) || (elementNo > maxElement);
bool isRepeatedEntry = elementNo == lastElement;
if (isFinished || isRepeatedEntry) {
  :
}

// NOT:
if ((elementNo < 0) || (elementNo > maxElement)||
     elementNo == lastElement) {
  :
}
```

By assigning boolean variables to expressions, the program gets automatic documentation. The construction will be easier to read, debug and maintain.

54. The nominal case should be put in the *if*-part and the exception in the *else*-part of an if statement [1].

```
boolean isOk = readFile(fileName);
if (isOk) {
  :
}
else {
  :
}
```

Makes sure that the exception does not obscure the normal path of execution. This is important for both the readability and performance.

55. The conditional should be put on a separate line.

```
if (isDone) {        // NOT: if (isDone) doCleanup();
  doCleanup();
}
```

This is for debugging purposes. When writing on a single line, it is not apparent whether the test is really true or not.

56. Executable statements in conditionals must be avoided.

```
InputStream stream = File.open(fileName, "w");
if (stream != null) {
  :
```

```
}

// NOT:
if (File.open(fileName, "w") != null)) {
  :
}
```

Conditionals with executable statements are simply very difficult to read. This is especially true for programmers new to Java.

5.8 Miscellaneous

57. The use of magic numbers in the code should be avoided

```
private static final int  TEAM_SIZE = 11;
:
Player[] players = new Player[TEAM_SIZE]; // NOT: Player[]
players = new Player[11];
```

If the number does not have an obvious meaning by itself, the readability is enhanced by introducing a named constant instead.

Even obvious values should be named. While the context of “7” may indicate that it it obviously the number of days in a week you should still name that DAYS_PER_WEEK.

58. Floating point constants should always be written with decimal point and at least one decimal.

```
double total = 0.0;     // NOT:  double total = 0;
double speed = 3.0e8;   // NOT:  double speed = 3e8;

double sum;
:
sum = (a + b) * 10.0;
```

This emphasize the different nature of integer and floating point numbers. Mathematically the two model completely different and non-compatible concepts.

Also, as in the last example above, it emphasize the type of the assigned variable (`sum`) at a point in the code where this might not be evident.

59. Floating point constants should always be written with a digit before the decimal point.

```
double total = 0.5;  // NOT:  double total = .5;
```

The number and expression system in Java is borrowed from mathematics and one should adhere to mathematical conventions for syntax wherever possible. Also, 0.5 is a lot more readable than .5; There is no way it can be mixed with the integer 5.

60. Static variables or methods must always be refered to through the class name and never through an instance variable.

```
Thread.sleep(1000);     // NOT: thread.sleep(1000);
```

This emphasize that the element references is static and independent of any particular instance. For the same reason the class name should also be included when a variable or method is accessed from within the same class.

6 Layout and Comments

6.1 Layout

61. Basic indentation should be fixed and small (2-4). Spaces must be used. Tabs must never be used. Be consistent.

```
for (i = 0; i < nElements; i++)
  a[i] = 0;
```

Indentation is used to emphasize the logical structure of the code. Indentation of 1 is to small to acheive this. Indentation larger than 4 makes deeply nested code difficult to read and increase the chance that the lines must be split. Choosing between indentation of 2, 3 and 4; 2 and 4 are the more common, and 2 chosen to reduce the chance of splitting code lines. Note that the Sun recommendation on this point is 4.

62. Block layout should be as illustrated in example 1 below and must not be as shown in example 3. Class, Interface and method blocks should use the block layout of example 2.

```
while (!done) {
  doSomething();
  done = moreToDo();
}
```

```
while (!done)
{
  doSomething();
  done = moreToDo();
}
```

```
while (!done)
  {
    doSomething();
    done = moreToDo();
  }
```

Example 3 introduces an extra indentation level which doesn't emphasize the logical structure of the code as clearly as example 1 and 2.

63. The *class* and *interface* declarations should have the following form:

```
class Rectangle extends Shape
  implements Cloneable, Serializable
{
  ...
}
```

This follows from the general block rule above. Note that it is common in the Java developer community to have the opening bracket at the end of the line of the class keyword. This is not recommended.

64. Method definitions should have the following form:

```
public void someMethod()
```

```
  throws SomeException
{
  ...
}
```

See comment on `class` statements above.

65. The *if-else* class of statements should have the following form:

```
if (condition) {
  statements;
}

if (condition) {
  statements;
}
else {
   statements;
}

if (condition) {
  statements;
}
else if (condition) {
  statements;
}
else {
  statements;
}
```

This follows partly from the general block rule above. However, it might be discussed if an `else` clause should be on the same line as the closing bracket of the previous `if`or `else` clause:

```
  if (condition) {
    statements;
  } else {
    statements;
  }
```

This is equivalent to the Sun recommendation. The chosen approach is considered better in the way that each part of the `if-else` statement is written on separate lines of the file. This should make it easier to manipulate the statement, for instance when moving `else` clauses around.

66. The *for* statement should have the following form:

```
for (initialization; condition; update) {
  statements;
}
```

This follows from the general block rule above.

67. An empty *for* statement should have the following form:

```
for (initialization; condition; update)
```

```
  ;
```

This emphasize the fact that the for statement is empty and it makes it obvious for the reader that this is intentional.

68. The *while* statement should have the following form:

```
while (condition) {
  statements;
}
```

This follows from the general block rule above.

69. The *do-while* statement should have the following form:

```
do {
  statements;
} while (condition);
```

This follows from the general block rule above.

70. The *switch* statement should have the following form:

```
switch (condition) {
  case ABC :
    statements;
    // Fallthrough

  case DEF :
    statements;
    break;

  case XYZ :
    statements;
    break;

  default :
    statements;
    break;
}
```

This differs slightly from the Sun recommendation both in indentation and spacing. In particular, each `case` keyword is indented relative to the switch statement as a whole. This makes the entire switch statement stand out. Note also the extra space before the *:* character. The explicit *Fallthrough* comment should be included whenever there is a case statement without a `break` statement. Leaving the `break` out is a common error, and it must be made clear that it is intentional when it is not there.

71. A *try-catch* statement should have the following form:

```
try {
  statements;
}
```

```
catch (Exception exception) {
  statements;
}

try {
  statements;
}
catch (Exception exception) {
  statements;
}
finally {
  statements;
}
```

This follows partly from the general block rule above. This form differs from the Sun recommendation in the same way as the `if-else` statement described above.

72. Single statement `if-else`, `for` or `while` statements must not be written without brackets.

```
if (condition) {
  statement;
}

while (condition) {
  statement;
}

for (initialization; condition; update) {
  statement;
}
```

It is a common recommendation (Sun Java recommendation included) that brackets should always be used in all these cases. However, brackets are in general a language construct that groups several statements. Brackets are per definition superfluous on a single statement. A common argument against this syntax is that the code will break*if* an additional statement is added without also adding the brackets. In general however, code should never be written to accommodate for changes that *might* arise.

75. Logical units within a block should be separated by one blank line.

```
// Create a new identity matrix
Matrix4x4 matrix = new Matrix4x4();

// Precompute angles for efficiency
double cosAngle = Math.cos(angle);
double sinAngle = Math.sin(angle);

// Specify matrix as a rotation transformation
matrix.setElement(1, 1,  cosAngle);
matrix.setElement(1, 2,  sinAngle);
matrix.setElement(2, 1, -sinAngle);
```

```
matrix.setElement(2, 2,  cosAngle);

// Apply rotation
transformation.multiply(matrix);
```

Enhances readability by introducing white space between logical units. Each block is often introduced by a comment as indicated in the example above.

77. Variables in declarations can be left aligned.

```
TextFile  file;
int       nPoints;
double    x;
```

77. Variables should be declared one line per variable..

```
TextFile  file;
int       nPoints;
double    x;
double    y;
```

Enhances readability. The variables are easier to spot from the types by alignment. Allows for simpler modification.

78. Statements should be aligned wherever this enhances readability.

```
if      (a == lowValue)    compueSomething();
else if (a == mediumValue) computeSomethingElse();
else if (a == highValue)   computeSomethingElseYet();

value = (potential        * oilDensity)   / constant1 +
        (depth            * waterDensity) / constant2 +
        (zCoordinateValue * gasDensity)   / constant3;

minPosition     = computeDistance(min,     x, y, z);
averagePosition = computeDistance(average, x, y, z);

switch (phase) {
  case PHASE_OIL   : text = "Oil";   break;
  case PHASE_WATER : text = "Water"; break;
  case PHASE_GAS   : text = "Gas";   break;
}
```

There are a number of places in the code where white space can be included to enhance readability even if this violates common guidelines. Many of these cases have to do with code alignment. General guidelines on code alignment are difficult to give, but the examples above should give some general hints. In short, any construction that enhances readability should be allowed.

6.3 Comments

79. Tricky code should not be commented but rewritten [1].

In general, the use of comments should be minimized by making the code self-documenting by appropriate name choices and an explicit logical structure.

80. All comments should be written in English.

In an international environment English is the preferred language.

82. There should be a space after the comment identifier.

```
// This is a comment

/**
 * This is a javadoc
 * comment
 */
```

```
NOT: //This is a comment

NOT: /**
 *This is a javadoc
 *comment
 */
```

Improves readability by making the text stand out.

84. Comments should be indented relative to their position in the code [1].

```
// CORRECT:
while (true) {
  // Do something
  something();
}
```

```
// INCORRECT:
while (true) {
// Do something
  something();
}
```

This is to avoid that the comments break the logical structure of the program.

85. The declaration of anonymous collection variables should be followed by a comment stating the common type of the elements of the collection.

```
private Vector  points_;    // of Point
private Set     shapes_;    // of Shape
```

Without the extra comment it can be hard to figure out what the collection consist of, and thereby how to treat the elements of the collection. In methods taking collection variables as input, the common type of the elements should be given in the associated JavaDoc comment.

Whenever possible one should of course qualify the collection with the type to make the comment superflous:

```
  private Vector<Point>  points_;
  private Set<Shape>     shapes_;
```

7 References

[1] Code Complete, Steve McConnel - Microsoft Press

[2] Java Code Conventions
http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

[3] Netscape's Software Coding Standards for Java
http://developer.netscape.com/docs/technote/java/codestyle.html

[4] C / C++ / Java Coding Standards from NASA
http://v2ma09.gsfc.nasa.gov/coding_standards.html

[5] Coding Standards for Java from AmbySoft
http://www.ambysoft.com/javaCodingStandards.html