Java Coding Standards

1. Introduction

This document provides guidelines to follow to improve maintainability of Java programs. The guidelines include proper formatting of source code using spaces, alignments, ordering of methods, and statements which improves source code readability. This document also provides guidelines on naming conventions to be followed and how to comment properly. The Rules are to be strictly followed and it is the best practice to follow recommendations. The guidelines are further demonstrated with many examples.

2. Source Code Structure

Rule 1. Do not use tab characters in implementation files, use plain spaces instead.

- Use four spaces of indentation

Rule 2. Use the following order to declare members of a class:

- Class variables (declared static).
- Instance variables.
- Constructors (at least one).
- finalize method (destructor) if necessary.
- Class methods (declared static).
- Methods:
    - set/get
    - Other methods

Rule 3. *Avoid lines longer than 80 characters.*

Rule 4. *When breaking long lines, follow these guidelines:*

- Break after commas.
- Break before operators.
- Prefer breaking between large high-level subexpressions to breaking between smaller lower-level subexpressions.
- Align the text in a new line, with the beginning of the expression at the same syntactical level on the previous line.
- If the above guidelines lead to confusing or cumbersome formatting, indent the second line 8 spaces instead.

Example

- Break after commas. Prefer

```
system.out.printline(final_time,
                     (final_time – initial_time) * 4);
```

to

```
system.out.printline(final_time, (final_time –
                      initial_time) * 4);
```

- Break before operators. Prefer

```
final_price = basic_price * tax_rate
              + initial_price;
```

(break before "+") to

```
final_price = basic_price * tax_rate +

                    initial_price;
```

(break after "+".)

- Prefer breaking between large high-level subexpressions to breaking between smaller lower-level subexpressions:

```
pos = pos0 + speed * time +

        (acceleration * time * time) / 2;
```

should be preferred to

```
pos = pos0 + speed * time + (acceleration * time
                              * time) / 2;
```

- Align the text in a new line, with the beginning of the expression at the same syntactical level on the previous line. Prefer

```
self.getClientList().retrieve(baseClient – processCount
                                  + offset);
```

to

```
self.getClientList().retrieve(baseClient – processCount
                                                   + offset);
```

- If the above guidelines lead to confusing or cumbersome formatting, indent the second line 8 spaces instead. Prefer:

```
members.getMemberByName(memberName).store(
        self.database, self.getPriority() + priorityIncrease);
```

to

```
members.getMemberByName(memberName).store(self.database,
                                          self.getPriority() +
                                           priorityIncrease);
```

Rule 5: Put single variable definitions in separate lines.

Example

Instead of writing

```
int counter, total; // Wrong!
```

write

```
int counter;
int total;
```

Rule 6: Put single statements in separate lines.

Example

Instead of writing

```
counter = initial; counter++; // Wrong!
```

write

```
counter = initial;
counter++;
```

3. Naming

Rule 7: Restrict identifiers to the ASCII character set.

Recommendation 1: Pick identifiers that accurately describe the corresponding program entity.

Recommendation 2: Use terminology applicable to the domain.

Recommendation 3: Avoid long (e.g. more than 20 characters) identifiers.

Recommendation 4: Use abbreviations sparingly and consistently.

Rule 8: Avoid names that differ only in case.

Rule 9: Capitalize the first letter of standard acronyms.

Rule 10: Do not hide declarations.

Do not declare names in one scope that hide names declared in a wider scope. Pick different names as necessary.

Rule 11: Capitalize the first letter of each word that appears in a class or interface name.

Example

```
  public class PrintStream extends FilterOutputStream
 {
        ...
 } // end class
public interface ActionListener extends EventListener
{
    ...
} // end interface
```

Rule 12: Use nouns when naming classes.

Example

Some examples from the Java standard class library:

SocketFactory

KerberosTicket

MediaName

Modifier

StringContent

Time

Rule 13: Pluralize the names of classes that group related attributes, static services or constants.

Example

Some examples from the Java standard class library:

BasicAttributes

LifespanPolicyOperations

PageRanges

RenderingHints

Method Names

Rule 14: Use lower-case for the first word and capitalize only the first letter of each subsequent word that appears in a method name.

Example

insertElement

computeTime

save

extractData

Rule 15: Use verbs in imperative form to name methods that:

- Modify the corresponding object (modifier methods).
- Have border effects (i.e., displaying information, writing to or reading from a storage device, changing the state of a peripheral, etc.)

Example

Modifier methods:

insert

projectOrthogonal

deleteRepeated

Methods with border effects:

save

drawLine

sendMessage

Rule 16: Use nouns to name variables and attributes.

Example

shippingAddress

counter

currentPosition

maximalPower

Rule 40: Qualify instance variable references with this to distinguish them from local variables.

Example

```
public class AtomicAdder
{
   private int count;

   public AtomicAdder(int count)
   {
      this.count=count;
```

```
    } // end method
    public synchronized int fetchAndAdd(int value)
    {
       int temp = this.count;
       this.count += value;
       return temp;
    } // end method
    ...
} // end class
```

Constant Names

Rule 17: Use upper-case letters for each word and separate each pair of words with an underscore when naming Java constants.

Example

```
class Byte
{
    public static final byte MAX_VALUE = +127;
    public static final byte MIN_VALUE = 0;
     ...
} // end class
```

4. Documentation and Commenting Conventions

Comment Types

The Java programming language supports three comment types:

- A one-line or end-line comment that begins with "//" and continues through to the end of the line.
- A standard, or C-style, comment, which starts with "/*" and ends with "*/".
- A documentation comment that starts with "/**" and ends with "*/". The Javadoc tool processes only comments of this type.

Rule 18: Provide a summary description and overview for each package.

The Javadoc utility provides a mechanism for including package descriptions in the documentation it generates. Use this capability to provide a summary description and overview for each package you create.
The Javadoc documentation [SUNDoc] explains how to make use of this feature.

Rule 19: Document public, protected, package, and private members.

Rule 20: Fully describe the signature of each method.

The documentation for each method shall always include a description for each parameter, each checked exception, any relevant unchecked exceptions, and any return value.

Recommendation 5: Document local variables with an end-line comment.