Java Programming Style Guide

Computer Science Program
Cedarville University

Goal: Our goal is to produce well-written code that can be easily understood and will facilitate life-cycle maintenance. These guidelines lay out principles that Java programmers have found useful for producing readable code that contains fewer bugs and is easier to maintain. It is important to remember these are just guidelines, and efficiency and understandability should not be sacrificed to blindly follow them. Remember: there are no style points for "slick code."

"*Any fool can write code that a computer can understand. Good programmers write code that humans can understand*." - Martin Fowler

Layout and Comments:

1. Each public class should be placed in a separate file (of the same name). If you use private classes, they can be located in the same file as the public class they support (the public class should be first in the file). Logically associated public classes should be placed in the same directory and included in the same *package*. Each file will contain a *package* statement, necessary import statements, and the code for the class being developed.

2. The first line in the file should be the package statement. This should be followed by any required import statements. The package and import statements should be located before the class header.

3. Each class will begin with a class header block. This block should contain, as a minimum, the name and brief description of the class, the author, the name of the file containing the class, the date created, a summary of modifications, and a copyright statement. The block should also explain the use/purpose of the class. For example, explain the hierarchy it belongs to, what classes derive from it, and how the class is intended to be used. JavaDoc will be used to the maximum extent possible. Mandatory JavaDoc fields include @author and @version.

```
package mathfunctions;

import java.util.*;

/**
*   This class implements the Rational ADT, providing support for manipulation
*   of fractions.
*
*   @author Joe Smith
*   @version 1.0
*    File: rational.java
*    Created: Jun 2000
*    ©Copyright Cedarville University, its Computer Science faculty, and the
*    authors.  All rights reserved.
*    Summary of Modifications:
*         14 Jul 2000 – JMS – corrected bug in Multiply routine
*         24 Jul 2000 – JMS – added Rational::Reduce method
*
* Description:  This class provides an ADT for fractions.  Users of this class are
* provided methods to manipulate fractions in an intuitive manner.  For example,
* arithmetic (+, -, *, /), comparison (<=, ==), and I/O (<<, >>) operators work for
* Rationals.  Rational does not inherit from other classes, and is not anticipated to
* be a base class for other classes.
*/
```

4. An appropriate amount of comments should be used throughout your code. Although it is most common for programmers to provide too few comments, over-commenting can also negatively impact the readability of the code. It is important to comment on any sections of code whose function is not obvious (to another person). For algorithmic-type code which follows a sequence of steps, it may be appropriate to summarize the algorithm at the beginning of the section (e.g., in the method header) and then highlight each of the major steps throughout the code. Either style (C or C++) comments may be used; however, the same style should be used throughout your program. Comments should be indented at the same level as the code they describe, and should appear before the code to which they refer.

<u>Naming Conventions</u>:

1. Use descriptive names for variables and methods in your code, such as xPosition, distanceToGo, or findLargest(). The only exception to this may be loop iteration variables, where no descriptive name makes sense. In this case, use the lower case letters beginning with *i* (i, j, k, etc).

2. For variables, the first word in the name should be lower case; the first letter of subsequent words in the name should be capitalized (sometimes referred to as "camelcase"). For example, upperLimit, averageValue, makeConnection(), addBody(). This distinguishes function and variable names from class names at quick

glance. Whenever possible, method names should be verbs: draw(), getX(), setPosition().

3. For constants, capitalize all letters in the name, and separate words in the name using an underscore, e.g., PI, MAX_ARRAY_SIZE. Any numeric constants needed in your code (other than very simple ones like –1, 0, and 1) should be replaced by a named constant.

4. For boolean variables and functions, the name should reflect the boolean type, e.g., isEmpty (), isLastElement, hasChanged. Names should always be positive; i.e., use hasChanged rather than hasNotChanged.

5. Do not use the name of a class instance variable as the name for method formal parameter or for local variables within a method. This causes ambiguity which can confuse the reader and promote bugs.

6. Package names should be all lowercase letters.

Statements:

1. Only one statement is allowed per line, and each line of code will be no more than 80 characters in length (to prevent line-wrap). If a statement requires more than one line, subsequent lines will be indented to make it obvious that the statement extends over multiple lines, as shown below. The general rules for line breaks are to break after a comma or before an arithmetic operator.

```
int myFunction (int variableA, int variableB, int variableC, int variableD,
                int variableE, int variableF ) {
}
```

2. A single declaration per line is preferred. Never mix multiple types on the same line, or initialized and uninitialized variables.

```
int id, grades[10];          // bad
int height, weight = 5;      // bad
```

3. Braces can be done in one of two styles: the opening brace can be put on the end of the line defining the block, or it can be on a separate line by itself, as shown below. Code inside the braces will always be indented.

```
for (i = 0; i <= maxSize; i++)   {
  …
  …
}
```

```
for (i = 0; i <= maxSize; i++)
{
    …
}
```

4. Make consistent use of horizontal white space. For example, include spaces after commas, and between operands and operators in expressions. Avoid using tabs to create white space. Note the spacing in the for-loop example in paragraph 3 above.

5. Use vertical white space to separate your code into "paragraphs" of logically connected statements.

6. Use parentheses as needed for clarity.

7. Do not compare a Boolean variable explicitly against true/false.
   OK: if (isEmpty) … if (counter == 0) …
   Avoid: if(!counter) … if (ifEmpty == true)

8. Write straightforward code and avoid unnecessarily complicated code. For example, *for* statements should typically only include initialization, test, and increment of iteration variable, not additional statements which belong in the loop body.
   Avoid: for (i = 0, sum=0; i <= maxCount; i++, sum+=i*4) {…}

9. For *if* statements, the nominal or more frequent case should typically be placed in the *then* block, and the exceptional or less frequent case in the *else* block.

10. Looping: Kava provides two primary statements for iteration: the *while* statement and the *for* statement. The *while* statement should be preferred when the number of loop iterations is unknown at the outset of the loop or when early termination from the loop is a possibility. In addition, *while* loops are preferred when the exit condition is a non-counting condition. The *for* statement should be preferred when the number of loop iterations is known, i.e., they usually can be counted. Early termination of *for* loop is discouraged. Using *do* … *while* loops is discouraged.

11. The *break* and *continue* statements should be used sparingly, and only if another more straightforward approach results in less readable/maintainable code.

12. Structured programming encourages a single exit point (i.e., a single return statement) from a function. Multiple return statements from a function should be used sparingly.

13. The *goto* statement is not to be used.

14. Enumerations should be used when you need to specify a set of possible integer values.

Classes:

1. Within the class, items will be listed in the following order:
   a. Static data members.

b. Instance data members.
c. Constructors.
d. Static methods (except *main*()).
e. Instance methods.
f. *main*(). Note: alternatively, you can put *main*() right after the constructors; i.e., make it the first method in the file.

2. Member data should usually be private, and public or protected accessor methods [e.g., getX(), setX()] provided.

3. Use of modifiers should be maximized.

4. Methods which are "helpers" for other methods and are not part of the "interface" of the class should be declared private.

5. Significant methods (public methods where are part of the "interface" and other major methods) should provide JavaDoc documentation. These comments should include a brief description, and specify parameters, return type, and any exceptions thrown. For example:

```
/**
 * Main entry point to begin simulation of the circuit.
 *
 *  @param FileName The name of the file containing the circuit description
 * @return true if the simulation runs to completion
 * @throws FileNotFoundException If the specified file could not be opened
 */
```