Coding Standards for Java

Version 1.3

Why have coding standards?

It is a known fact that 80% of the lifetime cost of a piece of software goes to maintenance; therefore, it makes sense for all programs within an organization to be as consistent as possible. Code conventions also improve the readability of the software.

This document specifies the coding standards for all computer science courses using Java at Pacific University. It is important for you to adhere to these standards in order to receive full credit on your assignments.

This document is divided into four main sections:

- Naming Conventions
- Formatting
- Comments
- Printing

Naming Conventions

- Constants

A constant is to be mnemonically defined using all capital letters and underscore characters such as MAX_NAME_CHARS. Separate words using an underscore character. Further, your program is to contain no "magic constants." That is, all magic constants must be declared **final** to make the program easier to modify and easier to read. In the case below, 100 is a magic constant and if used in several places throughout a program, can create problems if 100 is to be modified for any reason. Also, the meaning of 100 is not obvious.

Poor Program Style

```
.....
for (indx = 0; indx < 100; ++indx)
{
.....
}
```

Correct Program Style

```
final byte MAX_GRADE_SCORES = 100;

.....
for (indx = 0; indx < MAX_GRADE_SCORES; ++indx)
{
.....
}
```

Note: Constants like 0 and 1 are usually acceptable unless they represent values such as true and false in which case they should be declared as constants.

- Variable Names

1) A variable name is defined in all lowercase letters unless the variable name contains multiple names such as readStudentRecord. After the first word, each subsequent word has the first letter capitalized with the remainder of the word made up of lowercase letters.

2) Variable names are to be mnemonic unless the variable is being used in a for loop in which case the names i, j, k, l, m, n are acceptable names to be used. If, however the nested loop is being used in conjunction with a two-dimensional array, then the names row and column should be used.

3) Global variables must begin with g so that a name such as gHashTable denotes a global variable.

4) To aid in identifying the type of a variable, we will use the following prefixes.

Type Indicator is a	**Text Prefix**	**Variable Name Example**
boolean	b	bIsCorrect
globals	g	char gNumFiles
object	c	cRational
object member	m	mNumerator, mDenominator

Note: Declare all variables at the top of the function, no in-code variable declarations EXCEPT for the initializer in a for loop.

Poor Program Style

```
public byte f (byte s)
{
  byte i;
  for (i = 0; n[i] != s; ++i);

  return i;
}
```

Good Program Style

```
public byte find (byte value)
{
  byte indx;
  byte foundAt = NOT_FOUND;

  for (indx = 0; indx  < mIdNumbers.length && value != mIdNumbers[indx]; ++indx)
  {
  }

  if (indx != mIdNumbers.length)
  {
    foundAt = indx;
  }

  return foundAt;
}
```

Class Implementation

Method and Constructor Names - methods and constructors are named using the standard naming conventions described for variables where the first word begins with a lowercase letter and each subsequent word has the beginning letter capitalized.

Rational Class Implementation

```
package edu.pacificu.cs.Rational;

/**
 * Creates a Rational class where all operations work on actual
 * rational numbers of the form numerator / denominator. Assumes all
 * Rationals are positive and there is no error checking.
 *
 * @author  CS, Pacific University
 */



public class Rational
{
  private int mNumerator;
```

```java
private int mDenominator;

/**
* Initializes data members to default values representing 0
*/

public Rational ()
{
  mNumerator = 0;
  mDenominator = 1;
}

/**
* Initializes Rational number using parameter list values
*
* @param numerator  the numerator of the Rational number
*
* @param denominator  the denominator of the Rational number
*/

public Rational (int numerator, int denominator)
{
  this.mNumerator = numerator;
  this.mDenominator = denominator;
}

/**
* Changes the value of the numerator to the passed in value
*
* @param  numerator the numerator of the rational number
*/

public void setNumerator (int numerator)
{
  this.mNumerator = numerator;
}

/**
* Changes the value of the denominator to the value input
*
* @param  denominator denominator of the rational number
*/

public void setDenominator (int denominator)
{
  this.mDenominator = denominator;
}

/**
* Returns the value of the numerator
*
* @return the value of the numerator
*/

public int getNumerator ()
{
  return this.mNumerator;
}
```

```
/**
* Returns the value of the denominator
*
* @return the value of the denominator
*/

public int getDenominator ()
{
  return this.mDenominator;
}

/**
* Outputs a Rational number in the form numerator / denominator
* to the screen
*/

public void print ()
{
  // Just showing method calls instead of using member variables directly
  System.out.print (getNumerator () + "/" + getDenominator ());

}

/**
* Compares two objects of Rational returning a value of true if the
* numerators and denominators if both rational numbers are the same. If this
* method is not provided then == will be used on two Rational
* objects and == will check the reference values not the two Rational
* object values.
*
* @param fraction a rational number object
*
* @return true if objects are equal; else, false
*/

public boolean equals (Rational fraction)
{
  return (this.mNumerator == fraction.mNumerator &&
          this.mDenominator == fraction.mDenominator);
}

/**
* Multiplies the numerators and denominators of two objects.
*
* @param  fraction  a rational number object
*
* @return a Rational object that contains the result of the
* multiplication
*/

public Rational multiply (Rational fraction)
{
  Rational tempFraction = new Rational ();

  tempFraction.setNumerator (this.mNumerator * fraction.getNumerator());
  tempFraction.setDenominator (this.mDenominator * fraction.getDenominator());

  return tempFraction;
}
```

```
  /**
  * Reduces the rational number by dividing the numerator and
  * denominator by the greatest common divisor
  */

  public void reduce ()
  {
    int gcd;

    gcd = greatestCommonDivisor (this.mNumerator, this.mDenominator);

    if (gcd > 1)
    {
      this.mNumerator /= gcd;
      this.mDenominator /= gcd;
    }
  }

  /**
  * Returns a Rational object as numerator / denominator
  *
  * @return a representation of a Rational object
  */

  @Override
  public String toString ()
  {
    return Integer.toString (mNumerator) + "/" +
           Integer.toString (mDenominator);
  }

  /**
  * Finds the greatest common divisor of two integers using Euclid's method
  *
  * @param numOne first integer used in finding the gcd
  *
  * @param numTwo second integer used in finding the gcd
  *
  * @return the greatest common divisor of numOne and numTwo
  */

  private int greatestCommonDivisor (int numOne, int numTwo)
  {
    int tempInt;

    while (0 != numTwo)
    {
      numOne %= numTwo;
      tempInt = numOne;
      numOne = numTwo;
      numTwo = tempInt;
    }

    return numOne;
  }
}
```

Rational Class Driver

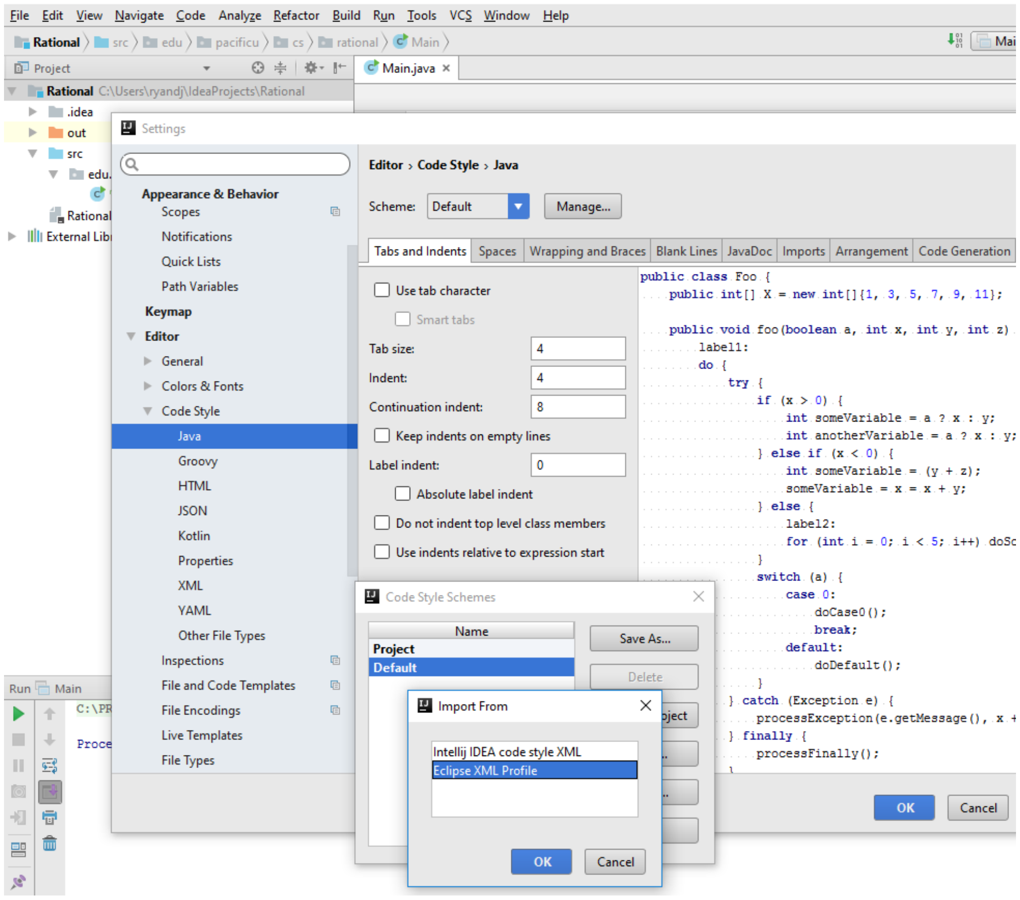
```
package edu.pacificu.cs.Rational;

import java.util.Scanner;

/**
 * Creates a Driver class to test all constructors and methods of
 * a Rational object
 *
 * @author   CS, Pacific University
 *
 * @see      Rational
 */

public class RationalDriver
{
  /**
  * Main testing method
  *
  * @param   args   command-line arguments
  *
  */
  static Scanner scanner = new Scanner (System.in);

  /**
   * A simple driver that creates a few Rational objects and calls their
   * methods doing some primitive operations. Also shows how to use the
   * scanner for input from the keyboard.
   *
   * @param args command line arguments if any
   */
  public static void main (String[] args)
  {
    Rational cRational1 = new Rational ();
    Rational cRational2 = new Rational (1, 2);
    Rational cRational3 = new Rational (2, 3);
    int numerator, denominator;

    cRational1.print();
    System.out.println ();
    cRational2.print();
    System.out.println ();
    cRational3.print();
    System.out.println ();
    cRational1 = cRational2.multiply(cRational3);
    cRational1.print();
    System.out.println ();
    cRational1.reduce();
    cRational1.print();
    System.out.println ();

    System.out.print ("Enter any Rational object [Form p / q]: ");
    numerator = scanner.nextInt ();
    scanner.next ();
    denominator = scanner.nextInt ();
    cRational1.setNumerator (numerator);
    cRational1.setDenominator (denominator);
    cRational1.print ();

    scanner.close ();
  }
}
```

Results if **12 / 13** is entered from the keyboard

```
0/1
1/2
2/3
2/6
1/3
Enter any Rational object [Form p / q]: 12 / 13
12/13
```

Formatting

- Indentation



Editor › Code Style › Java

Scheme: Default

- [ ] Use tab character
- [ ] Smart tabs

Tab size: 4
Indent: 4
Continuation indent: 8

- [ ] Keep indents on empty lines

Label indent: 0

- [ ] Absolute label indent
- [ ] Do not indent top level class members
- [ ] Use indents relative to expression start

Code Style Schemes: Project, Default (selected)

Import From: Intellij IDEA code style XML, Eclipse XML Profile (selected)

```
public class Foo {
    public int[] X = new int[]{1, 3, 5, 7, 9, 11};

    public void foo(boolean a, int x, int y, int z)
        label1:
        do {
            try {
                if (x > 0) {
                    int someVariable = a ? x : y;
                    int anotherVariable = a ? x : y;
                } else if (x < 0) {
                    int someVariable = (y + z);
                    someVariable = x = x + y;
                } else {
                    label2:
                    for (int i = 0; i < 5; i++) doSc
                }
                switch (a) {
                    case 0:
                        doCase0();
                        break;
                    default:
                        doDefault();
                }
            } catch (Exception e) {
                processException(e.getMessage(), x +
            } finally {
                processFinally();
```

An xml file with coding standard preferences will be provided.

Two spaces must be used as the unit of indentation per tab. Every IDE (Integrated Development Environment) such as Eclipse & IntelliJ IDEA includes an option for changing the number of spaces in a tab. These can usually be found in the preferences section. In Eclipse it is Window->Preferences and then you need to create a New Active profile. Then you can edit your active profile to set the Tabs, braces, ...
Select these options BEFORE typing in any of your code.

- Line Length

Lines must be no longer than 80 characters. Anything longer than 80 characters is normally not handled well in many terminals and tools. Further, a line longer than 80 will not wrap nicely when outputted.

- Wrapping Lines

If an expression cannot fit on a single line, then break it:
•After a comma
•Before an operator

Make sure that the new line is aligned with the beginning of the expression at the same level on the previous line. As an example, consider a long output statement as follows:

```
   System.out.println( " " + getNumerator() + "/" + getDenominator() + "
");
```

would be written as:

```
   System.out.println( " " + getNumerator() +
                       "/" + getDenominator() + " ");
```

- Spaces

All arithmetic and logical operators must have one space before and after the operator. The only exceptions are:

- •Unary operators
- •The period
- •No spaces before the comma and only one space after the comma

- Blank Lines

Use blank lines to separate distinct pieces of code. For example, one blank line before and after a while loop helps the code reader. The important thing to remember is that blank lines must be used consistently.

- Braces

Any curly braces that you use in your program (e.g. surrounding classes, methods) must appear on their own lines. Any code within the braces must be indented relative to the braces.

```java
private int greatestCommonDivisor (int numOne, int numTwo)
{
  int tempInt;

  while (0 != numTwo)
  {
    numOne %= numTwo;
    tempInt = numOne;
    numOne = numTwo;
    numTwo = tempInt;
  }

  return numOne;
}
```

Comments

Comments are used to explain the purpose of the code fragment they are grouped with. Comments state what the code is doing, while the code itself shows how you are doing it.

Use comments as follows:

- Class Header

The main purpose of a class header is to explain the purpose of the class as briefly as possible. You must include the following Javadoc tags in your class header:

```java
/**
 * Creates a Rational class where all operations work on actual
 * rational numbers of the form numerator / denominator
 *
 * @author  Joe Bloggs
 *
 */
```

- Declaration Comments

Variables must be declared one per line. Each variable can have a sidebar comment to the right indicating the variable's purpose if the purpose of the variable is not totally obvious. Do not put any blank lines between the variables being declared. You must also group together variables that are related.

```java
int seconds;
int minutes;
int hours ;
```

```
string firstName;
string lastName;
```

None of the above variables need to be documented because it is obvious what they represent. When your programs become more complicated, some variable names might need a little documentation.

- Sidebar Comments

A sidebar comment appears on the same line as the single statement it is describing. The comment must be brief and not exceed that line. Try and line up your sidebar comments as much as possible for easy reading. Even though the following line of code is obvious to an experienced programmer and need not be commented, here is an example of as sidebar comment.

```
value <<= 1; // multiply value by 2
```

- In-line Comments

In-line comments appear on their own lines and precede the segment of code they describe. You must use in-line comments to describe complex code that is not limited to a single statement. You must use one blank line to separate the comment from the segment of code being described. Every block of code (if statement, for loop, while loop, do-while loop, switch statement, …) must have an in-line comment preceding the actual code briefly letting the reader know what the code is doing. Here is an example.

```
    int tempInt;

    // Euclid's Algorithm for finding the greatest common divisor
    while (0 != numTwo)
    {
      numOne %= numTwo;
      tempInt = numOne;
      numOne = numTwo;
      numTwo = tempInt;
    }
```

- Method Header

In the same way that a class header is used to describe the purpose of the class, the method header must be used to describe the purpose of the function. All your method headers must include the following if they exist:

•Method Description
•Parameters

•Returned
•Since

```
/**
* Finds the greatest common divisor of two integers
*
* @param numOne first integer used in finding the gcd
*
* @param numTwo second integer used in finding the gcd
*
* @return the greatest common divisor of numOne and numTwo
*
*/

private int greatestCommonDivisor (int numOne, int numTwo)
{
  int tempInt;

  while (0 != numTwo)
  {
    numOne %= numTwo;
    tempInt = numOne;
    numOne = numTwo;
    numTwo = tempInt;
  }

  return numOne;
}
```

Printing

When printing your code you must use a fixed width font. Courier and Courier New are examples of fixed width fonts. You must also make sure that your lines do not wrap nor do they get cut off when printing. All printing is to be done in Portrait.
The final output you will turn in is to be printed in color since comments, keywords, strings, etc. are highlighted for easy reading. Multiple pages are to be printed double-sided (i.e. the ENTIRE job is printed by the printer double-sided without feeding in additional paper with a different orientation during the print job) and stapled in the upper-left corner.