Guidelines – Java Coding Guidelines

JSAG Suggests Series

Nagarro Software Pvt. Ltd.
Java Software Architecture Group

Revision History			
Version	**Date**	**Author/Contributor**	**Comments**
1.0	18-Mar-2010	Amit Sharma	First Version
1.2	21-Sep-2011	Kameswari SSV	Review and Revision as per latest guidelines from Oracle
1.3	24-Oct-2011	Amit Balwani	Added sections on Threads, Generics, Annotations.
1.4	03-Nov-2011	Amit Sharma	Final review. Changed JAMG to JSAG
1.5	07-Feb-2012	Amit Balwani	Changed the line length recommendation from 80 to 120
1.6	15-May-2012	Amit Balwani	Added sample code for some sections and some additional guidelines
1.7	08-Jun-2012	Kuldeep Singh	Added the Comparator and Comparable guidelines .
1.8	21-Jun-2012	Kuldeep Singh	Updated guidelines for using parseInt in javascript.
1.9	25-Oct-2012	Kuldeep Singh	Updated Interface Guidelines
2.0	19-Nov-2012	Amit Balwani	Added sections for static code analysis and frameworks
2.1	28-Dec-2012	Kuldeep Singh	Updated URLs for Sonar and Jenkins
2.2	14-Jun-2013	Rajneesh Aggarwal	Added section for Artifactory Server
2.3	28-Jun-2013	Rahul Kumar	JSAG Logging framework section added
2.4	28-Aug-2013	Rahul Kumar	HTML and CSS guidelines added, JS guidelines updated.
2.5	11-Dec-2013	Kuldeep Singh	Updated URLs for Sonar and Jenkins

Contents

1. Overview

This document is a working document - it is not designed to meet the requirement that we have "a" coding standard but instead it is an acknowledgment that we can make our lives much easier in the long term if we all agree to a common set of conventions when writing code. The guidelines here have been compiled from coding guidelines published by Sun**,** best practices followed across the industry, and our own experience.

Anybody with a few months of programming experience can write 'working code'. Writing 'good code' is an art and has the following characteristics

- Readable
- Reliable
- Maintainable
- Efficient

Code conventions are important to programmers for a number of reasons:

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create

There are several standards that exist in the programming industry. None of them are wrong or bad and you may follow any of them. What is more important is, selecting one standard approach and ensuring that everyone is following it.

This document is not fixed in stone; but it is not a suggestion, either. However, if you think that something could be improved, or even if you think that we've made a wrong call somewhere then please bring it to the notice of the architecture group.

2. When this document does and does not apply

It is the intention that all code written for or by Nagarro Software adheres to this standard. However, there are some cases where it is impractical or impossible to apply these conventions.

This document applies to all code **except the following**:

2.1. Code changes made to existing systems not written to this standard

In general, it is a good idea to make your changes conform to the surrounding code style wherever possible. You might choose to adopt this standard for major additions to existing systems.

2.2. Code written for customers that require that their standards should be adopted

Nagarro Software works with customers that have their own coding standards. Most coding standards applicable to Java derive at least some of their content from Sun coding guidelines. For this reason many coding standards are broadly compatible with each other. This document goes a little further than most in some areas; however it is likely that these extensions will not conflict with most other coding standards. We must be absolutely clear on this point: if there is a conflict, the customer's coding standards are to apply - always.

3. File Organization Guidelines

A file consists of sections that should be separated by blank lines and an optional comment identifying each section.

Files longer than 2000 lines are cumbersome and should be avoided.

Each Java source file contains a single public class or interface. When private classes and interfaces are associated with a public class, you can put them in the same source file as the public class. The public class should be the first class or interface in the file.

Java source files should have the following ordering:

- ***Beginning comments***- All source files should begin with a c-style comment that lists the programmer(s), the date, a copyright notice, and also a brief description of the purpose of the program

```
/*
* Class name
*
* Version info
*
* Copyright notice
*
* Author info
*
* Creation date
*
* Last updated By
*
* Last updated Date
*
* Description
*/
```

- ***Package and Import statements*** - The first non-comment line of most Java source files is a package statement. After that, import statements can follow. For example:

```
package java.awt;
import java.awt.peer.CanvasPeer;
```

- ***Class and interface declarations*** - The following table describes the parts of a class or interface declaration, in the order that they should appear.

	Part of Class/Interface Declaration	**Notes**
1	Class/interface documentation comment (/**...*/)	
2	Class, interface, enum or annotation statement	

3	Class/interface implementation comment (/*...*/), if necessary	This comment should contain any class-wide or interface-wide information that wasn't appropriate for the class/interface documentation comment.
4	Class (static) variables	First the public class variables, then the protected, then package level (no access modifier), and then the private.
5	Instance variables	First public, then protected, then package level (no access modifier), and then private.
6	Constructors	
7	Methods	These methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in between two public instance methods. The goal is to make reading and understanding the code easier.
8	Inner Classes	Are placed at the bottom of the file

4. Indentation

Four spaces should be used as the unit of indentation. The exact construction of the indentation (spaces vs. tabs) is unspecified.

4.1. Line Length

Avoid lines longer than 120 characters, since they're not handled well by many terminals and tools.

4.2. Wrapping Lines

When an expression will not fit on a single line, break it according to these general principles:

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line.
- If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

Here are some examples of breaking method calls:

```
someMethod(longExpression1, longExpression2, longExpression3,
        longExpression4, longExpression5);

var = someMethod1(longExpression1,
someMethod2(longExpression2,
                        longExpression3));
```

Following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

```
longName1 = longName2 * (longName3 + longName4 - longName5)
          + 4 * longname6; //Preferred
longName1 = longName2 * (longName3 + longName4
                       - longName5) + 4 * longname6; // Avoid
```

Following are two examples of indenting method declarations. The first is the conventional case. The second would shift the second and third lines to the far right if it used conventional indentation, so instead it indents only 8 spaces.

```
//Conventional indentation
someMethod(intanArg, Object anotherArg, String yetAnotherArg,
           Object andStillAnother) {
    ...
}

//Indent 8 spaces to avoid very deep indents
private static synchronized horkingLongMethodName(intanArg,
        Object anotherArg, String yetAnotherArg,
        Object andStillAnother) {
    ...
}
```

Line wrapping for *if* statements should generally use the 8-space rule, since conventional (4 space) indentation makes seeing the body difficult. For example:

```
// Don't use this indentation
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) { //Bad wraps
    doSomethingAboutIt();           //Make this line easy to miss
}

// Use this indentation instead
if ((condition1 && condition2)
        || (condition3 && condition4)//8 space indentation
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}

// Or use this
if ((condition1 && condition2) || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

Here are three acceptable ways to format ternary expressions:

```
alpha = (aLongBooleanExpression) ? beta: gamma;

alpha = (aLongBooleanExpression) ? beta
                                 : gamma;

alpha = (aLongBooleanExpression)
        ? beta
        : gamma;
```

5. Comments

Java programs can have two kinds of comments:

- Implementation comments – Implementation comments are those found in C++, which are delimited by /*...*/, and //. They are meant for commenting out code or for comments about the particular implementation

- Documentation comments - Documentation comments (known as "Doc comments") are Java-only, and are delimited by /**...*/. Doc comments can be extracted to HTML files using the javadoc tool. They are meant to describe the specification of the code, from an implementation-free perspective to be read by developers who might not necessarily have the source code at hand.

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. For example, information about how the corresponding package is built or in what directory it resides should not be included as a comment.

Discussion of non-trivial or non-obvious design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code. It is too easy for redundant comments to get out of date. In general, avoid any comments that are likely to get out of date as the code evolves.

Note: The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer.

Comments should not be enclosed in large boxes drawn with asterisks or other characters.
Comments should never include special characters such as form-feed and backspace.

5.1. Implementation Comment Formats

Programs can have four styles of implementation comments: block, single-line, trailing, and end-of-line.

5.1.1. Block Comments

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments may be used at the beginning of each file and before each method. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe.

A block comment should be preceded by a blank line to set it apart from the rest of the code.

```
/*
 * Here is a block comment.
 */
```

Block comments can start with `/*-`, which is recognized by **indent**(1) as the beginning of a block comment that should not be reformatted. Example:

```
/*-
 * Here is a block comment with some very special
 * formatting that I want indent(1) to ignore.
 *
 *    one
 *        two
 *            three
 */
```

*Note: If you don't use **indent**(1), you don't have to use /*- in your code or make any other concessions to the possibility that someone else might run **indent**(1) on your code.*

5.1.2. Single-Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format. A single-line comment should be preceded by a blank line. Here's an example of a single-line comment in Java code:

```
if (condition) {

    /* Handle the condition. */
    ...
}
```

5.1.3. Trailing Comments

Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same tab setting.

Here's an example of a trailing comment in Java code:

```
if (a == 2) {
        return true;            /* special case */
} else {
        Return isPrime(a);      /* works only for odd a */
}
```

5.1.4. End-Of-Line Comments

The // comment delimiter can comment out a complete line or only a partial line. It shouldn't be used on consecutive multiple lines for text comments; however, it can be used in consecutive multiple lines for commenting out sections of code. Examples of all three styles follow:

```
if (foo > 1) {

    // Do a double-flip.
    ...
}
else {
    return false;          // Explain why here.
}
//if (bar > 1) {
//
//    // Do a triple-flip.
//    ...
//}
//else {
//    return false;
//}
```

5.2. Documentation Comments

For further details, see "How to Write Doc Comments for Javadoc" which includes information on the doc comment tags (@return, @param, @see):

http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html. For further details about doc comments and Javadoc, see the Javadoc home page at:

http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html. Doc comments describe Java classes, interfaces, constructors, methods, and fields. Each doc comment is set inside the comment delimiters /**...*/, with one comment per class, interface, or member. This comment should appear just before the declaration:

```
/**
 * The Example class provides ...
 */

public class Example { ...
```

Notice that top-level classes and interfaces are not indented, while their members are. The first line of doc comment (/**) for classes and interfaces is not indented; subsequent doc comment lines each have 1 space of indentation (to vertically align the asterisks). Members, including constructors, have 4 spaces for the first doc comment line and 5 spaces thereafter.

If you need to give information about a class, interface, variable, or method that isn't appropriate for documentation, use an implementation block comment or single-line comment immediately *after* the declaration. For example, details about the implementation of a class should go in such an implementation block comment *following* the class statement, not in the class doc comment.

Doc comments should not be positioned inside a method or constructor definition block, because Java associates documentation comments with the first declaration *after* the comment.

6. Declarations

6.1. Number per Line

One declaration per line is recommended since it encourages commenting. In other words,

```
int level; // indentation level
int size;  // size of table
```

is preferred over

```
int level, size;
```

Do not put different types on the same line. Example:

```
int foo,  fooarray[]; //Wrong!
```

Note: The examples above use one space between the type and the identifier. Another acceptable alternative is to use tabs, e.g.:

```
int     level;          // indentation level
int     size;   // size of table
Object  currentEntry;   // currently selected table entry
```

6.2. Initialization

Try to initialize local variables where they're declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.

```
Public void persistEmployee(Employee employee) {
    int version = 0;       // Preferred approach
    ……

     int newVersion;           // Avoid
    newVersion = 2;
}
```

6.3. Placement

Put declarations only at the beginning of blocks. (A block is any code surrounded by curly braces "{" and "}".) Don't wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

```
voidmyMethod() {
int int1 = 0;          // beginning of method block

if (condition) {
    int int2 = 0;      // beginning of "if" block
        ...
```

```
        }
    }
```

The one exception to the rule is indexes of *for* loops, which in Java can be declared in *for* statement:

```
for (int i = 0; i<maxLoops; i++) { ... }
```

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

```
int count;
...
myMethod() {
if (condition) {
    int count = 0;      // Avoid!
        ...
    }
    ...
}
```

6.4. Class and Interface Declarations

When coding Java classes and interfaces, the following formatting rules should be followed:

- No space between a method name and the parenthesis "(" starting its parameter list
- Open brace "{" appears at the end of the same line as the declaration statement
- Closing brace "}" starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the "}" should appear immediately after the "{"

```
class Sample extends Object {
int ivar1;
int ivar2;

Sample(inti, int j) {
    ivar1 = i;
    ivar2 = j;
}

int emptyMethod() {}

    ...
}
```

- Methods are separated by a blank line

7. Statements

7.1. Simple Statements

Each line should contain at most one statement. Example:

```
argv++;       // Correct
argc--;       // Correct
```

```
argv++; argc--;        // Avoid!
```

7.2. Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces "{statements}". See the following sections for examples.

- The enclosed statements should be indented one more level than the compound statement.
- The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.
- Braces are used around all statements, even single statements, when they are part of a control structure, such as an if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.
- If (employeeActive) {
- updateEmployeeAddress(employee, address);
- updateEmployeeProfile(empployeeProfile);

}

7.3. return Statements

A return statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

```
return;

returnmyDisk.size();

return (size ? size : defaultSize);
```

7.4. if, if-else, if else-if else Statements

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;
}
```

Note: if statements always use braces {}. Avoid the following error-prone form:

```
if (condition) //Avoid! this omits the braces {}!
    statement;
```

7.5. for Statements

A for statement should have the following form:

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

An empty for statement (one in which all the work is done in the initialization, condition, and update clauses) should have the following form:

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

When using the comma operator in the initialization or update clause of a *for* statement, avoid the complexity of using more than three variables. If needed, use separate statements before the *for* loop (for the initialization clause) or at the end of the loop (for the update clause).

7.6. while Statements

A while statement should have the following form:

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

An empty while statement should have the following form:

```
while (condition);
```

7.7. do while Statements

A do-while statement should have the following form:

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

7.8. switch Statements

A switch statement should have the following form:

```
switch (condition) {
case ABC:
```

```
        statements;
        /* falls through */

    case DEF:
        statements;
        break;

    case XYZ:
        statements;
        break;

    default:
        statements;
        break;
    }
```

Every time a *case* falls through (doesn't include a break statement), add a comment where the *break* statement would normally be. This is shown in the preceding code example with the /* falls through */ comment.

Every *switch* statement should include a default case. The break in the default case is redundant, but it prevents a fall-through error if later another case is added.

7.9. Try-catch Statements

A try-catch statement should have the following format:

```
    try {
        statements;
    } catch (ExceptionClass e) {
        statements;
    }
```

A try-catch statement may also be followed by finally, which executes regardless of whether or not the try block has completed successfully.

```
    try {
        statements;
    } catch (ExceptionClass e) {
        statements;
    } finally {
        statements;
    }
```

8. White Space

8.1. Blank Lines

Blank lines improve readability by setting off sections of code that are logically related.

Two blank lines should always be used in the following circumstances:

- Between sections of a source file
- Between class and interface definitions

One blank line should always be used in the following circumstances:

- Between methods
- Between the local variables in a method and its first statement
- Before a block or single-line comment
- Between logical sections inside a method to improve readability

8.2. Blank Spaces

Blank spaces should be used in the following circumstances:

- A keyword followed by a parenthesis should be separated by a space. Example:

```
while (true) {
    ...
}
```

  Note that a blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

- A blank space should appear after commas in argument lists.

- All binary operators except "." should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands. Example:

```
a += c + d;
a = (a + b) / (c * d);

while (d++ = s++) {
    n++;
}
printSize("size is " + foo + "\n");
```

- The expressions in a "for" statement should be separated by blank spaces. Example:

```
for (expr1; expr2; expr3)
```

- Casts should be followed by a blank space. Examples:

```
myMethod((byte) aNum, (Object) x);
myMethod((int) (cp + 5), ((int) (i + 3))
                                + 1);
```

9. Naming Conventions

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier-for example, whether it's a constant, package, or class-which can be helpful in understanding the code.

Identifier Type	Rules for Naming	Examples

Packages	The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981.
	
	Subsequent components of the package name vary according to an organization's own internal naming conventions. Such conventions might specify that certain directory name components be division, department, machine, and project or login names.
		`com.sun.eng`
		`com.apple.quicktime.v2`
		`edu.cmu.cs.bovik.cheese`

Classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).	`class Raster;` `class ImageSprite;`
Interfaces	Interface names should be capitalized like class names.	`interface RasterDelegate;` `interface Storing;`

Methods	Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.	`run();`
		`runFast();`
		`getBackground();`

Variables	Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore _ or dollar sign $ characters, even though both	`Int i;`
		`char c;`
		`float myWidth;`

	are allowed. Variable names should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.	

Constants	The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_"). (ANSI constants should be avoided, for ease of debugging.)	`static final int MIN_WIDTH = 4;`
		`static final int MAX_WIDTH = 999;`
		`static final int GET_THE_CPU = 1;`

10. Programming Practices

10.1. Providing Access to Instance and Class Variables

Don't make any instance or class variable public without good reason. Often, instance variables don't need to be explicitly set or gotten—often that happens as a side effect of method calls.

One example of appropriate public instance variables is the case where the class is essentially a data structure, with no behaviour. In other words, if you would have used a *struct* instead of a class (if Java supported *struct*), then it's appropriate to make the class's instance variables public.

10.2. Referring to Class Variables and Methods

Avoid using an object to access a class (static) variable or method. Use a class name instead.

For example:

```
classMethod(); //Ok
AClass.classMethod(); //Ok
anObject.classMethod(); //Avoid!
```

10.3. Multiple exit points

Methods should ideally have a single exit point. This improves readability, extensibility and the debugging process. If there is a need to put an exit log statement or to log the output value, it is easy with the single exit point approach.

This approach also avoids arbitrary scenario handling code (sometimes duplicate) being added in the beginning of the methods with an exit point.

The following example shows the multiple exit points approach which should be avoided.

```
public static double max(double a, double b) {
    double result;    // Stores the maximum value until the return.
    if (a > b) {
        return a;
    } else {
        return b;     // Avoid
    }
}
```

Instead, the approach given below should be followed:

```
public static double max(double a, double b) {
    double result;    // Stores the maximum value until the return.
    if (a > b) {
        result = a;
    } else {
        result = b;
    }
    return result;   // Preferred approach
}
```

10.4. Constants

Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a *for* loop as counter values.

For example:

```
for(int i = 0; i < arraySize; i++) //Ok
int resolution = 1024 + defaultResolution ; //Avoid
```

10.5. Variable Assignments

Avoid assigning several variables to the same value in a single statement. It is hard to read.
Example:

```
fooBar.fChar = barFoo.lchar = 'c'; // Avoid!
```

Do not use the assignment operator in a place where it can be easily confused with the equality operator. Example:

```
if (c++ = d++) { // Avoid! Java disallows
...
}
```

It should be written as

```
if ((c++ = d++) != 0) {
...
}
```

Do not use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler, and besides, it rarely actually helps. Example:

```
d = (a = b + c) + r; // Avoid!
```

It should be written as

```
a = b + c;
```

```
d = a + r;
```

10.6. Use of Static Variables

Static member variables should be used very carefully as their inappropriate use may result in memory leaks in the application.

Static variables should be used to store data which does not change frequently but will be accessed frequently during the application lifetime. Static variables should be used to provide application level storage; they should not be used for instance level data storage. Using mutable static variables for instance data will keep on increasing memory usage even though the data is not required after the instance is garbage collected.

For example:

```
public class EmployeeServiceImpl{

    // static - will remain in memory for the application lifetime
    private static HashMap<String, String> errorMap
            = new HashMap<String, String>();

    public EmployeeServiceImpl (List<String> salaryGenErrors) {
        if(null != salaryGenErrors) {
            for(String errorMessage : salaryGenErrors) {
                // Avoid for instance data as Map is static
                errorMap.put (generateErrorId(), errorMessage);
            }
        }
    }
}
```

A scenario where the use of static member variable would be valid is given below.

```
public class CacheManagerServiceImpl{

    // static - cache gender code and description mapping
    private static HashMap<String, String> genderMap
            = new HashMap<String, String>();

    public void populateGenderList (List<Gender> genderList) {
        if(genderMap.isEmpty()) {
            for(Gender gender :genderList) {
                genderMap.put (gender.getCode(),
                    gender.getDescription());
            }
        }
    }
}
```

10.7. Code reuse

As much as possible, code must be compact and easy to read by using utility methods (preferably static) from utility classes categorized by the programming area they serve. For example, there could be classes for String manipulation, date manipulation, pagination, caching etc.

For example:

```
public class StringUtil {
    public static String concatenateStrings(String… stringValues)
            throws Exception {
        StringBuilder sbConcatenatedValue = null;
        if(null != stringValues && stringValues.length > 0) {
            sbConcatenatedValue = new StringBuilder();
            for (int i = 0; i < stringValues.length; i++) {
                sbConcatenatedValue.append(stringValues[i]);
            }
        }
        return sbConcatenatedValue;
    }
}
```

If there are common methods or member variables which are component or layer specific, they should be moved to the base class. If, if-else and if-else if-else blocks should not have duplicate code in if, else and else-if blocks. It should be moved either before or after the if-else block.

10.8. Method parameter validation

All non-private methods should validate the input parameters for null, empty and other required conditions before being used for any operations. Validation being done in the caller should not be relied upon as future modifications or new callers might easily oversee this error vulnerability.

Even for private methods it is recommended to perform basic input validation like null and empty to avoid run time exceptions.

For example,

```
public class FundsUtil {
    public static double calcFundsPercentage(double orgFund, double
            totalFunds) throws Exception {
        double orgFundPercentage = 0.0;

        // validate values before usage
        if(orgFund > 0.0 && totalFunds > 0.0) {
            orgFundPercentage = (orgFund / totalFunds) * 100.0;
        }
        return orgFundPercentage;
    }
}
```

10.9. Configurability

All business and technical parameters which have even a rarest possibility of getting changed in future should be kept configurable either through System properties or application level configuration files or through any other available mechanism for the system. If the information is sensitive (for eg. Server URLs, account info etc.) it should be encrypted and then stored in the configuration layer.

All log messages, screen labels and messages should come from locale based property files even if there is no requirement from the client to support i18n. It will keep the system flexible for future extensions.

10.10. Miscellaneous Practices

10.10.1. Parentheses

It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others—you shouldn't assume that other programmers know precedence as well as you do.

```
if (a == b && c == d) // Avoid!
if ((a == b) && (c == d)) // Right
```

10.10.2. Returning Values

Try to make the structure of your program match the intent. Example:

```
if (booleanExpression) {
    return TRUE;
} else {
    return FALSE;
}
```

It should instead be written as

```
return booleanExpression;
```

Similarly,

```
if (condition) {
    return x;
}
return y;
```

It should be written as

```
return (condition ? x : y);
```

10.10.3. Expressions before '?' in the Conditional Operator

If an expression containing a binary operator appears before the *?* in the ternary *?*: operator, it should be parenthesized. Example:

```
(x >= 0) ? x : -x
```

10.10.4. Special Comments

Use XXX in a comment to flag something that is bogus but works. Use FIXME to flag something that is bogus and broken

11. Threads Guidelines

- **start() Method**
  Avoid calling the *run()* method directly from a thread object.
  Calling *start()* on a thread object in turn calls run() on a new instance of the thread object. It is not advisable to call *run()* explicitly as this will not achieve multi-threading. A dead thread cannot be started again. You can call *start()* on a Thread object only once. Avoid calling start() more than once on a Thread object, it will throw a **IllegalThreadStateException -** if the thread was already started.

  Syntax:
  ```
  Thread thread = new MyThread();  //Thread class object.
  thread.start();
  ```

- It is legal to create many Thread objects using the same Runnable object as the target.
- **run() Method**
  In most cases, the Runnable interface should be used if you are only planning to override the run() method and no other Thread methods. This is important because classes should not be sub-classed unless the programmer intends on modifying or enhancing the fundamental behaviour of the class.
- There's no guarantee that threads will take turns in any fair way. It's up to the thread scheduler, as determined by the particular virtual machine implementation. If you want a guarantee that your threads will take turns regardless of the underlying JVM, you can use the sleep() method. This prevents one thread from hogging the running process while another thread starves. (In most cases, though, yield() works well enough to encourage your threads to play together nicely.)

Concurrent Access Problems and Synchronized Threads

- When sleep is called from a synchronized context, its locks will be unavailable to other threads. Try to avoid such circumstances. This can result in a deadlock.

- All three methods wait (), notify (), and notifyAll()—must be called from within a synchronized context! A thread invokes wait() or notify() on a particular object, and the thread must currently hold the lock on that object. If any of these methods is called from a non-synchronized context, it will result in IllegalMonitorStateException.
- **Executor Interface:** An object that executes submitted Runnable tasks. This interface provides a way of decoupling task submission from the mechanics of how each task will be run, including details of thread use, scheduling, etc. An Executor is normally used instead of explicitly creating threads. For example, rather than invoking new Thread(new(RunnableTask())).start() for each of a set of tasks, you might use:

```
Executor executor = anExecutor;
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
```

However, the Executor interface does not strictly require that execution be asynchronous. In the simplest case, an executor can run the submitted task immediately in the caller's thread:

```
class DirectExecutor implements Executor {
     public void execute(Runnable r) {
         r.run();
     }
 }
```

More typically, tasks are executed in some thread other than the caller's thread. The executor below spawns a new thread for each task.

```
class ThreadPerTaskExecutor implements Executor {
     public void execute(Runnable r) {
         new Thread(r).start();
     }
}
```

The Executor implementations provided in java.util.concurrent package implement ExecutorService, which is a more extensive interface. The ThreadPoolExecutor class provides an extensible thread pool implementation. The Executors class provides convenient factory methods for these Executors. When finished using an ExecutorService, you need to shut it down explicitly

- **ThreadLocal variables:** These variables differ from their normal counterparts in that each thread that accesses one (via its `get` or `set` method) has its own, independently initialized copy of the variable. *ThreadLocal* instances are typically private static fields in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID).
  When to use ThreadLocal:

  Basically, objects where:
  - The objects are non-trivial to construct;
  - An instance of the object is frequently needed by a given thread;
  - The application pools threads, such as in a typical server (if every time the thread-local is used it is from a new thread, then a new object will still be created on each call!);
  - It doesn't matter that Thread A will never share an instance with Thread B;
  - It's not convenient to subclass Thread. If you can subclass Thread, you could add extra instance variables to your subclass instead of using ThreadLocal. But for example, if you are writing a servlet running in an off-the-shelf servlet runner such as Tomcat, you generally have no control over the class of created threads. Of course, even if you can subclass Thread, you may simply prefer the cleaner syntax of ThreadLocal.

  That means that typical objects to use with ThreadLocal could be:
  - Random number generators (provided a per-thread sequence was acceptable);

- Collators;
- native ByteBuffers (which in some environments cannot be destroyed once they're created);
- XML parsers or other cases where creating an instance involves going through slightly non-trivial code to 'choose a registered service provider';
- Per-thread information such as profiling data which will be periodically collated.

Note that it is generally better not to re-use objects that are trivial to construct and finalize. (By "trivial to finalize", we mean objects that don't override finalize.) This is because recent garbage collector implementations are optimized for "temporary" objects that are constructed, trivially used and then fall out of scope without needing to be added to the finalizer queue. Pooling something trivial like a StringBuffer, Integer or small byte array can actually degrade performance on modern JVMs.

12. Exception handling guidelines

- Avoid empty catch block. This indicates that an exception which should either be acted on or reported is being swallowed.

```
public void doSomething() {
try {
FileInputStreamfis = new FileInputStream("/tmp/bugger");
  } catch (IOExceptionioe) {
    // not good
  }
}
```

When catching an exception, some options include:

1. inform the user (strongly recommended)
2. log the problem, using the JDK logging services, or similar tool
3. Log the user id, timestamp, class name and other relevant details as per the implementation which might help in issue resolution at a later stage (Do not include sensitive data in the logs)
4. send an email describing the problem to an administrator

- Avoid empty try blocks - what's the point?

```
public class Foo {
public void bar() {
try {
} catch (Exception e) {
    e.printStackTrace();
  }
 }
}
```

- Avoid empty finally blocks - these can be deleted.

```
public class Foo {
public void bar() {
try {
    int x=2;
} finally {
    // empty!
  }
 }
}
```

- Avoid returning from a finally block - this can discard exceptions.

```
public class Bar {
public String foo() {
try {
    throw new Exception( "My Exception" );
  } catch (Exception e) {
       throw e;
  } finally {
       return "A. O. K."; // Very bad.
  }
 }
}
```

- Avoid instance of checks in catch clause. Each caught exception type should be handled in its own catch clause.

```
try { // Avoid this
 // do something
} catch (Exception ee) {
      if (ee instanceof IOException) {
         cleanup();
      }
}

try {  // Prefer this:
 // do something
} catch (IOExceptionee) {
    cleanup();
}
```

- Preserve stack trace. Throwing a new exception from a catch block without passing the original exception into the new exception will cause the true stack trace to be lost, and can make it difficult to debug effectively.

```
public class Foo {
void good() {
try{
    Integer.parseInt("a");
} catch(Exception e){
     throw new Exception(e);
  }
}
```

```
void bad() {
try{
    Integer.parseInt("a");
} catch(Exception e){
      throw new Exception(e.getMessage());
  }
}
```

- Avoid catching Throwable. This is dangerous because it casts too wide a net; it can catch things like OutOfMemoryError.

```
public class Foo {
public void bar() {
try {
   // do something
} catch (Throwable th) {  //Should not catch Throwable
      th.printStackTrace();
 }
 }
}
```

- Be specific in throws clause i.e. a method/constructor shouldn't explicitly throw java.lang.Exception. Do not group together related exceptions in a generic exception class - that would represent a loss of possibly important information.

```
public void methodThrowingException() throws Exception {//Bad Practice
}
```

An alternative is the exception translation practice, in which a low level exception is first translated into a higher level exception before being thrown out of the method. The data layer, for example, can profit from this technique. Here, the data layer seeks to hide almost all of its implementation details from other parts of the program. It even seeks to hide the basic persistence mechanism - whether or not a database or an ad hoc file scheme is used, for example. If the rest of the program is to remain truly ignorant of the persistence mechanism, then these exceptions cannot be allowed to propagate outside the data layer, and must be translated into some higher level abstraction - DataAccessException, say.

- Never use exceptions as flow control as it leads to GOTOish code and obscures true exceptions when debugging.

```
public class Foo {
void bar() {
try {
try {
} catch (Exception e) {
        throw new WrapperException(e);
    // this is essentially a GOTO to the WrapperException catch block
}
```

```
} catch (WrapperException e) {
   // do some more stuff
  }
 }
}
```

- Avoid catching NullPointerException. Consider removing the cause of the NullPointerException. A catch block may hide the original error, causing other more subtle errors in its wake.

```
public class Foo {
void bar() {
try {
   // do something
}  catch (NullPointerException npe) {
       // Avoid
   }
 }
}
```

- Avoid throwing certain exception types. Rather than throwing a raw RuntimeException, Throwable, Exception, or Error, use a sub-classed exception or error instead.

```
public class Foo {
public void bar() throws Exception {
throw new Exception();  //Avoid
 }
}
```

- Avoid throwing a NullPointerException - it's confusing because most people will assume that the virtual machine threw it. Consider using an IllegalArgumentException instead; this will be clearly seen as a programmer-initiated exception.

```
public class Foo {
void bar() {
throw new NullPointerException(); // Avoid
 }
}
```

- A catch statement that catches an exception only to re-throw it or re-throw a caught exception wrapped inside a new instance of the same type should be avoided. This adds to code size and runtime complexity.

```
public class Foo {
void bar() {
try {
    // do something
}  catch (SomeException se) {
       throw se;     // Avoid
   }
  }
}
```

```
public class Foo {
void bar() {
    try {
       // do something
    }  catch (SomeException se) {
       throw new SomeException(se);     // Avoid
    }
  }
}
```

- Exceptions should not extend java.lang.Error as they are system exceptions.

```
public class Foo extends Error { } // Avoid
```

- Throwing exception in a finally block is confusing. It may mask exception or a defect of the code, it can also render code clean-up unstable.

13. JUnit guidelines

- The suite() method in a JUnit test needs to be both public and static.

```
Import junit.framework.*;
public class Foo extends TestCase {
public void suite() {} // oops, should be static
private static void suite() {} // oops, should be public
}
```

- Use setUp() method instead of parameterized constructors while developing the JUnits.

```
public class SomeTest extends TestCase
publicSomeTest (String testName) {
    super (testName);
      // Perform test set-up
}
}
```

- The order of test cases run cannot be assumed. For ordering testsuites can be used.

```
public static Test suite() {
    suite.addTest(new SomeTestCase ("testDoThisFirst";));
    suite.addTest(new SomeTestCase ("testDoThisSecond";));
    return suite;
}
```

- Put the common code for context initialization, cleanup code etc. in a super class's setUp and tearDown methods. Ensure that the setUp(), tearDown() methods are named correctly, have no arguments, return void and are either public or protected. The setup() method should call super.setUp() and tearDown method should call super.tearDown().

- JUnit assertions should include a message - i.e., use the three argument version of assertEquals(), not the two argument version.

```
public class Foo extends TestCase {
public void testSomething() {
assertEquals("foo", "bar");
  // Use the form:
  // assertEquals("Foo does not equals bar", "foo", "bar");
  // instead
 }
}
```

- JUnit tests should include at least one assertion. This makes the tests more robust, and using assert with messages provide the developer a clearer idea of what the test does.

```
public class Foo extends TestCase {
public void testSomething() {
      Bar b = findBar();
  // This is better than having a NullPointerException
  // assertNotNull("bar not found", b);
b.work();
  }
}
```

- Test classes should end with the suffix Test. Having a non-test class with that name is not a good practice, since most people will assume it is a test case. Test classes have test methods named testXXX.

```
//Consider changing the name of the class if it is not a test
//Consider adding test methods if it is a test
public class CarTest {
public static void main(String[] args) {
    // do something
}
// code
}
```

- A JUnit test assertion with a boolean literal is unnecessary since it always will evaluate to the same thing. Consider using flow control (in case of assertTrue(false) or similar) or simply removing statements like assertTrue(true) and assertFalse(false). If you just want a test to halt, use the fail method.

```
public class SimpleTest extends TestCase {
public void testX() {
  // Why on earth would you write this?
assertTrue(true);
 }
}
```

- Use assertEquals(x, y) instead of assertTrue(x.equals(y))
- Use assertNull(x) instead of assertTrue(x==null), or assertNotNull(x) vsassertFalse(x==null)

- Use assertSame(x, y) instead of assertTrue(x==y), or assertNotSame(x,y) vsassertFalse(x==y)
- Avoid negation in an assertTrue or assertFalse test. For example, rephrase assertTrue(!expr) as
- assertFalse(expr).
- JUnit 4 should indicate test suites via annotations, not the suite method.

```
public class BadExample extends TestCase{

public static Test suite(){
        return new Suite();
    }
}

@RunWith(Suite.class)
@SuiteClasses( { TestOne.class, TestTwo.class })
public class GoodTest {
}
```

- JUnit 4 should use @Test annotation to indicate tests that need to be executed.

```
public class MyTest {
public void testBad() {
    doSomething();
}

@Test
public void testGood() {
    doSomething();
}
}
```

- JUnit 4 tests should use @After annotation instead of tearDown() and @Before annotation instead of setup().

```
public class MyTest2 {

@Before
public void setUp() {
    good();
}

@After
public void tearDown() {
   good();
}
}
```

- It is not recommended to use an absolute location while using test data in writing test cases.

```
public void setup(){
     ......................
     File f = new File("D:\Dir\myTestData.dat");  //wrong
     ........................................
}
```

Test data along with test cases can be stored in common configuration location for accessing these files with relative paths.

```
public void setUp () {
    FileInputStreaminp ("myTestData.dat");
    ...
}
```

- Keep tests in the same location as the source code so that both of them are in sync but they should never be a part of the deployment.
- Use JUnit assertion and remove redundant catch clauses as any unhandled exception should fail the JUnit.
- When testing classes that process XML, it pays to write a routine that compares two XML DOMs for equality. You can then programmatically define the correct DOM in advance (or read from a file) and compare it with the actual output from your processing methods.
- While testing servlets you can write a dummy servlet framework and pre configure the framework during test case. The framework must contain derivations of classes found in the normal servlet environment. These derivations should allow you to pre configure their responses to method calls from the servlet. For example:
    1. HttpServletRequest can be sub-classed to allow the test class to specify the header, method, path info, and other data

    2. HttpServletResponse can be sub-classed to return an output stream that stores the servlet's responses in a string for later checking
- For testing JUnits for synchronization we should avoid using TestRunner from JUnit. Instead we can use MultiThreadTestRunner framework which can be used to run threads asynchronously inside a JUnit. Unlike a standard JUnitTestRunner, the MultiThreadedTestRunner will wait until all threads have terminated to exit. This forces JUnit to wait while the threads do their work, nicely solving our problem from earlier.

MultiThreadedTestRunnerProgram.txt

- For testing taglibs, Struts framework and other container dependant frameworks we can use the Cactus framework.

Cactus test code.txt

14. Code sizing guidelines

- The NPath complexity of a method i.e. the number of acyclic execution paths through a method should not be greater than 200.
- Avoid really long methods; ideally a method text should fit within a single screen view. Try to reduce the method size by creating helper methods and removing any copy/pasted or duplicated code.
- Avoid really long parameter lists. Long parameter lists can indicate that a new object should be created to wrap the numerous parameters. Basically, try to group the parameters together. The number of parameters of a method or constructor should not be greater than 7.
- Avoid really long classes. Long Class files are indications that the class may be trying to do too much. Try to break it down, and reduce the size to something manageable. A class with too many methods is probably a good suspect for refactoring, in order to reduce its complexity and find a way to have more fine grained objects.
- Cyclomatic complexity of a method should not be very high. Complexity is determined by the number of decision points in a method plus one for the method entry. The decision points are 'if', 'while', 'for', and 'case labels'. Generally, 1-4 is low complexity, 5-7 indicates moderate complexity, 8-10 is high complexity, and 11+ is very high complexity.

```
// Cyclomatic Complexity = 12

public class Foo {
1   public void example()  {
2       if (a == b)  {
3           if (a1 == b1) {
fiddle();
4           } else if a2 == b2) {
fiddle();
}  else {
fiddle();
            }
5       } else if (c == d) {
6           while (c == d) {
fiddle();
            }
7        } else if (e == f) {
8           for (int n = 0; n < h; n++) {
fiddle();
            }
        } else{
switch (z) {
9               case 1:
fiddle();
break;
10              case 2:
fiddle();
break;
11              case 3:
fiddle();
break;
12              default:
```

```
            fiddle();
            break;
                        }
                    }
                }
            }
```

- Avoid declaring large number of public methods and attributes in a class. A large number of public methods and attributes declared in a class can indicate the class may need to be broken up as increased effort will be required to thoroughly test it.
- Classes that have too many fields could be redesigned to have fewer fields, possibly through some nested object grouping of some of the information. For example, a class with city/state/zip fields could instead have one Address field.

15. Guidelines for Collection framework

- Choose the right collection. The best general purpose or 'primary' implementations are likely ArrayList, LinkedHashMap, and LinkedHashSet. Their overall performance is better, and you should use them unless you have special requirements for ordering or sorting.

Interface	HasDuplicates?	Implementations					Historical
Set	no	HashSet	...	LinkedHashSet*	...	TreeSet	...
List	yes	...	ArrayList*	...	LinkedList	...	Vector, Stack
Map	no duplicate keys	HashMap	...	LinkedHashMap*	...	TreeMap	Hashtable, Properties

Principal features of non-primary implementations:

1. HashMap has slightly better performance than LinkedHashMap, but its iteration order is undefined.
2. HashSet has slightly better performance than LinkedHashSet, but its iteration order is undefined.
3. TreeSet is ordered and sorted, but slow.
4. TreeMap is ordered and sorted, but slow.
5. LinkedList has fast adding to the start of the list, and fast deletion from the interior via iteration.

Iteration order for above implementations:

1. HashSet - undefined
2. HashMap - undefined
3. LinkedHashSet - insertion order
4. LinkedHashMap - insertion order of keys (by default), or 'access order'
5. ArrayList - insertion order
6. LinkedList - insertion order
7. TreeSet - ascending order, according to Comparable / Comparator
8. TreeMap - ascending order of keys, according to Comparable / Comparator

- The keys of a ***Map*** or items in a ***Set*** should be immutable objects i.e. they must not change their state.
- It is recommended to encapsulate collections as they are not immutable objects. As such, one must often exercise care that collection fields are not unintentionally exposed to the caller. One technique is to define a set of related methods which prevent the caller from directly using the underlying collection, such as :

```
1. addThing(Thing)
2. removeThing(Thing)
3. getThings() - return an unmodifiable Collection
```

- Be extra careful if you need to get an array of a class from your collection. If you need to get an array of a class from your Collection, you should pass an array of the desired class as the parameter of the toArray method. Otherwise you will get a ClassCastException.

```
Import java.util.ArrayList;
Import java.util.Collection;

public class Test {

public static void main(String[] args) {
    Collection c=new ArrayList();
    Integer obj=new Integer(1);
    c.add(obj);

    // this would throw a ClassCastException if executed
    Integer[] a=(Integer [])c.toArray();

    // this wouldn't
    Integer[] b=(Integer [])c.toArray(new Integer[c.size()]);
}
}
```

Additionally, a call to Collection.toArray() should always use the Collection's size rather than an empty array of the desired type. Doing this one sizes the destination array, avoiding a reflection call in some collection implementations.

```
class Foo {
void bar(Collection x) {

    // A bit inefficient
    x.toArray(new Foo[0]);

    // Much better
    x.toArray(new Foo[x.size()]);
}
}
```

- Always substitute calls to size() == 0 (or size() != 0) with calls to isEmpty().The isEmpty() method on java.util.Collection is provided to see if a collection has any elements. Comparing the value of size() to 0 merely duplicates existing behaviour.
- Always use ArrayList as compared to Vector to avoid synchronization overhead.

- Always use interface references to Collections. The user of such a reference will be protected from possible changes to the underlying implementation class. So habitually refer collection objects using List, Set, Deque, Queue, and Map interface references.
- Don't use removeAll() to clear a collection. If you want to remove all elements from a collection c, use c.clear(), not c.removeAll(c). Calling c.removeAll(c) to clear a collection is less clear, susceptible to errors from typos, less efficient and for some collections, might throw a ConcurrentModificationException.
- Never add a reference of collection to itself as it would generate a stack overflow in the Java virtual machine.

```
LinkedList list = new LinkedList();

// add list to itself
list.add(list);

// dies with infinite recursion
String contents = list.toString();
```

- Take care of concurrent modification exception while using collections. The container keeps track of the number of mutating operations (such as adding and removing elements). Each iterator keeps a separate count of the number of mutating operations that it was responsible for. At the beginning of each iterator method, the iterator simply checks whether its own mutation count equals that of the container. If not, it throws a ConcurrentModificationException.

```
LinkedList list = . . .;
ListIterator iter1 = list.listIterator();
ListIterator iter2 = list.listIterator();
iter1.next();
iter1.remove();

// throws ConcurrentModificationException
iter2.next();
```

16. File I/O Guidelines

- It's common to "wrap" a BufferedReader around a FileReader or a BufferedWriter around a FileWriter, to get access to higher-level (more convenient) methods.
- FileWriter and FileReader are low-level I/O classes. You can use them to write and read files, but they should usually be wrapped.
- A new File object doesn't mean there's a new file on your hard drive. File objects can represent either a file or a directory.
- A class must implement Serializable before its objects can be serialized.
  The ObjectOutputStream.writeObject() method serializes objects, and the ObjectInputStream.readObject() method de-serializes objects.

- If you mark an instance variable transient, it will not be serialized even though the rest of the object's state will be.
- If a superclass implements Serializable, then its subclasses do automatically.
- All streams opened in a method should be closed in the finally block

17. Generics Guidelines

- A non-generic collection can hold any kind of object! A non-generic collection is quite happy to hold anything that is NOT a primitive. This meant it was entirely up to the programmer to be…careful. Having no way to guarantee collection type wasn't very programmer-friendly for such a strongly typed language.
- Generics let you enforce compile-time type safety on Collections (or other classes and methods declared using generic type parameters).
- When using generic collections, a cast is not needed to get (declared type) elements out of the collection.

```
List<String>gList = new ArrayList<String>();
List list = new ArrayList();
// more code
String s = gList.get(0);            // no cast needed
String s = (String)list.get(0);     // cast required
```

- You can pass a generic collection into a method that takes a non-generic collection, but the results may be disastrous. The compiler can't stop the method from inserting the wrong type into the previously type safe collection.
  If you pass a List<String> into a method declared as

```
void foo(List aList) { aList.add(anInteger); } // add would be
disastrous
```

- Polymorphic assignments applies only to the base type, not the generic type parameter. You can say

```
List<Animal>aList = new ArrayList<Animal>();   // yes
```

  You can't say, `List<Animal>aList = new ArrayList<Dog>();  // no`

- The polymorphic assignment rule applies everywhere an assignment can be made. Hence:

```
void foo(List<Animal>aList) { }  // cannot take a List<Dog>
List<Animal>bar() { }            // cannot return a List<Dog>
```

- However, note that the wildcard keyword extends is used to mean either "extends" or "implements." So in <? extends Dog>, Dog can be a class or an interface.

- When using a wildcard, List<? extends Dog>,the collection can be accessed but not modified.

- Generics are a little different from arrays, which give you BOTH compile-time and runtime protection. There no type information at runtime to support legacy code. What you gain from using generics is compile-time protection that guarantees that you won't put the wrong thing into a typed collection, and it also eliminates the need for a cast when you get something out. Hence, we have to pay close attention to compiler warnings. There is no need for runtime protection unless we mix up generic and non-generic code.

- When using a wildcard, List<?>, any generic type can be assigned to the reference, but for access only, no modifications.
  List<Object> refers only to a List<Object>, while List<?> or List<? extends Object> can hold any type of object, but for access only.

  For Modifications you can use the super keyword instead.

  Like List <? Super Animal > will take list<Dog> as parameter for modification operations.
- A method that takes, say, a collection of superclass-type for instance- ArrayList<Animal> will NOT be able to accept a collection of any Animal subtype! That means ArrayList<Dog> cannot be passed into a method with an argument of `ArrayList<Animal>`, even though this works just fine with plain old arrays.
  i.e. An error shows up at runtime.

- The generics type identifier can be used in class, method, and variable declarations:

```
class Foo<t> { }     // a class
T anInstance;        // an instance variable
Foo(T aRef) {}       // a constructor argument
void bar(T aRef) {}  // a method argument
T baz() {}           // a return type
```

  The compiler will substitute the actual type.
- You can declare a generic method using a type not defined in the class:

```
public<T> void makeList(T t) { }
```

  is NOT using T as the return type. This method has a void return type, but to use T within the method's argument you must declare the <T>, which happens before the return type.

18. Inner Classes Guidelines

- An inner class is a full-fledged member of the enclosing (outer) class, so it can be marked with an access modifier as well as the abstract or final modifiers.

The relationship between the inner and outer class gives the inner class access to all of the outer class's members, including those marked private.

```
classMyOuter {
privateint x = 7;

   // inner class definition
classMyInner {
public void seeOuter() {
logger.debug("Outer x is " + x);
      }
   } // close inner class definition

} // close outer class
```

- You must have instance of the outer class to instantiate the inner class.

```
public static void main(String[] args) {
MyOutermo = new MyOuter();     // gotta get an instance!
MyOuter.MyInner inner = mo.newMyInner();
inner.seeOuter();
 }
```

- From code outside the enclosing class's instance methods, you can instantiate the inner class only by using both the inner and outer class names, and a reference to the outer class as follows:

```
MyOutermo = new MyOuter();
MyOuter.MyInner inner = mo.newMyInner();
```

- From code within the inner class, the keyword ***this*** holds a reference to the inner class instance. To reference the outer this (in other words, the instance of the outer class that this inner instance is tied to) precede the keyword this with the outer class name as follows:

```
MyOuter.this;
```

- A method-local inner class can be instantiated only within the method where the inner class is defined. In other words, for a method local inner class to be used, you must instantiate it, and that instantiation must happen within the same method, but after the class definition code.

```
class MyOuter2 {
private String x = "Outer2";
voiddoStuff() {
classMyInner {
```

```
public void seeOuter() {
logger.debug ("Outer x is " + x);
            } // close inner class method
          } // close inner class definition

MyInner mi = new MyInner();  // This line must come
                                        // after the class
mi.seeOuter();
      } // close outer class method doStuff()
} // close outer class
```

- The inner class object cannot use the local variables of the method the inner class is in unless the local variables are marked final!.
  Because the local variables aren't guaranteed to be alive as long as the method-local inner class object, the inner class object can't use them.

  Marking the local variable z as final fixes the problem:

```
final String z = "local variable";  //Now inner object can use it
```

- The only modifiers you can apply to a method-local inner class are abstract or final. (Never both at the same time, though.)

- An anonymous inner class is always created as part of a statement; don't forget to close the statement after the class definition with a curly brace. This is a rare case in Java, a curly brace followed by a semicolon.

```
class Popcorn {
public void pop() {
logger.debug ("popcorn");
}
}
class Food {
   Popcorn p = new Popcorn() {
public void pop() {
logger.debug("anonymous popcorn");
      }
   };
}
```

- Because of polymorphism, the only methods you can call on an anonymous inner class reference are those defined in the reference variable class (or interface), even though the anonymous class is really a subclass or implementer of the reference variable type.

```
class Popcorn {
public void pop() {
logger.debug("popcorn");
}
}

class Food {
   Popcorn p = new Popcorn() {
public void sizzle() {
    logger.debug("anonymous sizzling popcorn");
}
public void pop() {
    logger.debug("anonymous popcorn");
}
};

public void popIt() {
    p.pop();     // OK, Popcorn has a pop() method
    p.sizzle();  // Not Legal! Popcorn does not have
sizzle()
 }
}
```

- An anonymous inner class can extend one subclass or implement one interface. Unlike non-anonymous classes (inner or otherwise), an anonymous inner class cannot do both. In other words, it cannot both extend a class and implement an interface, nor can it implement more than one interface.

```
class Popcorn {
public void pop() {
    logger.debug("popcorn");
}
}

class Food {
   Popcorn p = new Popcorn() {
```

```
public void sizzle() {
    logger.debug("anonymous sizzling popcorn");
}
public void pop() {
    logger.debug("anonymous popcorn");
}
};

public void popIt() {
    p.pop();     // OK, Popcorn has a pop() method
    p.sizzle();  // Not Legal! Popcorn does not have
sizzle()
}
}
```

(OR)

```
interfaceCookable {
public void cook();
}
class Food {
Cookable c = new Cookable() {
public void cook() {
    logger.debug("anonymous cookable implementer");
}
};
}
```

- If the nested class is static, it does not share any special relationship with an instance of the outer class. In fact, you don't need an instance of the outer class to instantiate a static nested class. Instantiating a static nested class requires using both the outer and nested class names as follows:

```
BigOuter.Nested n = new BigOuter.Nested();
```

19. Clone implementation & object copy guidelines

- Avoid implementing clone. The clone method relies on strange/hard to follow rules that do not work it all situations. Consequently, it is difficult to override correctly. Below are some of the rules/reasons why the clone method should be avoided:

    1. Classes supporting the clone method should implement the Cloneable interface but the *Cloneable* interface does not include the clone method. As a result, it doesn't enforce the method override.
    2. The *Cloneable* interface forces the Object's clone method to work correctly. Without implementing it, the Object's clone method will throw a CloneNotSupportedException.
    3. Non-final classes must return the object returned from a call to super.clone().
    4. Final classes can use a constructor to create a clone which is different from non-final classes.
    5. If a super class implements the clone method incorrectly all subclasses calling super.clone() are doomed to failure.
    6. If a class has references to mutable objects then those object references must be replaced with copies in the clone method after calling super.clone().
    7. The clone method does not work correctly with final mutable object references because final references cannot be reassigned.
    8. If a super class overrides the clone method then all subclasses must provide a correct clone implementation.

    Two alternatives to the clone method, in some cases, is a copy constructor or a static factory method to return copies of an object. Both of these approaches are simpler and do not conflict with final fields. They do not force the calling client to handle a CloneNotSuportException. They also are typed therefore no casting is necessary. Finally, they are more flexible since they can take interface types rather than concrete classes.

    Sometimes a copy constructor or static factory is not an acceptable alternative to the clone method. The example below highlights the limitation of a copy constructor (or static factory). Assume Square is a subclass for Shape.

```
Shape s1 = new Square();
System.out.println(s1 instanceof Square); //true
```

    Assume at this point the code knows nothing of s1 being a Square that's the beauty of polymorphism but the code wants to copy the Square which is declared as a Shape, its super type.

```
Shape s2 = new Shape(s1); //using the copy constructor
System.out.println(s2 instanceof Square); //false
```

The working solution (without knowing about all subclasses and doing many casts) is to do the following (assuming correct clone implementation).

```
Shape s2 = s1.clone();
System.out.println(s2 instanceof Square); //true
```

- Object clone() should be implemented with super.clone().

```
class Foo{
public Object clone(){
    return new Foo(); // This is bad
}
}
```

- The method clone() should throw a CloneNotSupportedException.

```
public class MyClass implements Cloneable{
public Object clone() { // will cause an error
MyClass clone = (MyClass)super.clone();
    return clone;

}
}
```

- The method clone() should only be implemented if the class implements the Cloneable interface with the exception of a final method that only throws CloneNotSupportedException.

```
public class MyClass {
public Object clone() throws CloneNotSupportedException {
    return foo;
}
}
```

- Constructors and methods receiving arrays should clone objects and store the copy. This prevents that future changes from the user affect the internal functionality.

```
public class Foo {
private String [] x;
public void foo (String [] param) {
// Don't do this, make a copy of the array at least
    this.x=param;
}
}
```

20. equals and hashcode implementation guidelines

- Override both public Boolean Object.equals(other Object), and public int Object.hashCode(), or override neither.
- Even if you are inheriting a hashCode() from a parent class, consider implementing hashCode and explicitly delegating to your super class.

- equals and hashCode must depend on the same set of "significant" fields. You must use the same set of fields in both of these methods. You are not required to use all fields. For example, a calculated field that depends on others should very likely be omitted from equals and hashCode.
- hashCode must generate equal values for equal objects.
- Avoid using equals() to compare against null

```
class Bar {
void foo() {
        String x = "foo";
if (x.equals(null)) { // bad!
doSomething();
        }
    }
}
```

- Use equals() to compare object references; avoid comparing them with ==.
- Equals method should not assume anything about the type of its argument. The equals(Object o) method shouldn't make any assumptions about the type of o. It should simply return false if o is not the same type as this.
- Never compare a class literal with the class of the argument in equals() method. The method will be broken if it is inherited by sub classes.

```
Foo.class == o.getClass())//Wrong

this.getClass() == o.getClass()//Correct
```

- equals() method should return false if a null value is passed as argument.
- Objects placed in a List, Set, or Map (as either a key or value) should have an appropriate definition of equals.
- When implementing equals, fields are compared differently, according to their type:
  1. ***object fields, including collections*** : use equals
  2. ***type-safe enumerations*** : use either equals or == (they amount to the same thing, in this case)
  3. ***possibly-null object fields*** : use both == and equals
  4. ***array fields*** : use Arrays.equals
  5. ***primitive fields other than float or double*** : use ==
  6. ***float*** : convert to int using Float.floatToIntBits, then use ==
  7. ***double*** : convert to long using Double.doubleToLongBits, then use ==

- In an equals() method, it is usually worthwhile to order field comparisons such that the most significant comparisons are performed first. That is, fields most likely to differ should be evaluated first. This allows the && "short-circuit" logical operator to minimize execution time.

21. Comparable and Comparator implementation guidelines

21.1. Comparable

- Implementing Comparable allows, calling Collections.sort and Collections.binarySearch, calling Arrays.sort and Arrays.binarySearch, using objects as keys in a TreeMap and using objects as elements in a TreeSet.
- x.compateTo(y) method Returns a negative integer, zero, or a positive integer as this x is less than, equal to, or greater than the specified object y.
- x.compareTo(y) is the opposite sign of y.compareTo(x) for all the values of x and y.
- x.compareTo(y) throws exactly the same exceptions as y.compareTo(x)
- if x.compareTo(y)>0 and y.compareTo(z)>0, then x.compareTo(z)>0 (and same for less than)
- if x.compareTo(y)==0, then x.compareTo(z) has the same sign as y.compareTo(z)
- It is strongly recommended, but not strictly required that (x.compareTo(y)==0) == (x.equals(y)). Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals.

21.2. Comparator

- Similarly Implementing Comparator allows calling Collections.sort(List,Comparator), Arrays.sort(Object[],Comparator) to control the sort order.
- Comparators can be used to control the order of certain data structures (such as sorted sets or sorted maps), or to provide an ordering for collections of objects that don't have natural sorting order.
- compare (x, y) compares and returns a negative integer, zero, or a positive integer as the x is less than, equal to, or greater than the y.
- compare(x, y) is the opposite sign of compare(y, x) for all the values of x and y.
- compare(x, y) throws exactly the same exceptions as compare(y, x)
- ((compare(x, y)>0) && (compare(y, z)>0)) implies compare(x, z)>0
- compare(x, y)==0 then (compare(x, z)) must have same sign as (compare(y, z)) for all z.
- It is generally the case, but not strictly required that (compare(x, y)==0) == (x.equals(y)). Generally speaking, any comparator that violates this condition should clearly indicate this fact. The recommended language is "Note: this comparator imposes orderings that are inconsistent with equals."

22. Annotations Guidelines

- Annotation (introduced in Java 1.5) is a special form of syntactic metadata that can be added to Java source code. Classes, methods, variables, parameters and packages may be annotated. Unlike Javadoc tags, Java annotations can be reflective in that they can be embedded in class files generated by the compiler and may be retained by the Java VM to be made retrievable at run-time.

- Annotations can be used to
  - Simplify Configuration
  - To define requirements
  - To enforce local policies
  - Avoiding unneeded interfaces (with care)
  - Avoiding marker interfaces (no methods)
  - Replace Naming Convention Based Development
  - To avoid boilerplate code
  - To make meta-data about code available at runtime
- Annotations' use should be avoided for
  - Environment Specific Information
  - Application configuration
  - Database configuration
  - JNDI lookups
  - A substitute for macros

23. JDBC and Database handling guidelines

- Minimize the Use of Database Metadata Methods. Compared to other JDBC methods, database metadata methods that generate ResultSet objects are relatively slow. Applications should cache information returned from result sets that generate database metadata methods so that multiple executions are not needed.

- Avoid Search Patterns in database metadata methods. Using null arguments or search patterns in database metadata methods results in generating time-consuming queries. In addition, network traffic potentially increases due to unwanted results. Always supply as many non-null arguments as possible to result sets that generate database metadata methods. For example:

  ResultSetWSrs = WSdbmd.getTables (null, null, "WSTable", null);

  In this example, an application uses the getTables method to determine if the WSTable table exists. A JDBC driver interprets the request as: return all tables,views, system tables, synonyms, temporary tables, and aliases named "WSTable" that exist in any database schema inside the database catalog.

  In contrast, the following request provides non-null arguments as shown:

  String[] tableTypes = {"TABLE"}; WSdbmd.getTables ("cat1","johng", "WSTable", "tableTypes");

Clearly, a JDBC driver can process the second request more efficiently that it can process the first request.

- Use a Dummy Query to Determine Table Characteristics. Avoid using the getColumns method to determine characteristics about a table. Instead, use a dummy query with getMetadata. Consider an application that allows the user to choose the columns that will be selected. Should the application use getColumns to return information about the columns to the user or, instead, prepare a dummy query and call getMetadata?

```
Case 1: getColumns Method

ResultSetWSrc = WSc.getColumns (... "UnknownTable" ...);
// This call to getColumns will generate a query to
// the system catalogs... possibly a join
// which must be prepared, executed, and produce
// a result set
. . .
WSrc.next();
stringCname = getString(4);
. . .

// user must retrieve N rows from the server
// N = # result columns of UnknownTable

Case 2: getMetaData Method
// prepare dummy query
PreparedStatementWSps = WSc.prepareStatement
("SELECT * FROM UnknownTable WHERE 1 = 0");
// query is never executed on the server - only prepared
ResultSetMetaDataWSsmd=WSps.getMetaData();
intnumcols = WSrsmd.getColumnCount();
...
intctype = WSrsmd.getColumnType(n)
...
// result column information has now been obtained
```

In both cases, a query is sent to the server. However, in Case 1, the potentially complex query must be prepared and executed, result description information must be formulated, and a result set of rows must be sent to the client. In Case 2, we prepare a simple query where we only retrieve result set information. Clearly, Case 2 is the better performing model.

- Because retrieving long data across a network is slow and resource intensive, applications should not request long data unless it is necessary. Most users don't want to see long data. If the user does want to see these result items, the application can query the database again, specifying only the long columns in the Select list. This method allows the average user to retrieve the result set without having to pay a high performance penalty for network traffic.
- Reduce the size of data being retrieved. Sometimes long data must be retrieved. When this is the case, remember that most users do not want to see 100 KB, or more, of text on the screen. To reduce network traffic and improve performance, you can reduce the size of any data being retrieved to some manageable limit by calling ***setMaxRows***, ***setMaxFieldSize***, and the driver-

specific ***setFetchSize***. Another method of reducing the size of the data being retrieved is to decrease the column size. In addition, be careful to return only the rows and columns you need. If you return five columns when you only need two columns, performance is decreased especially if the unnecessary rows include long data.

- Choose the right data type. Retrieving and sending certain data types can be expensive. When you design a schema, select the data type that can be processed most efficiently. For example, integer data is processed faster than floating-point data. Floating-point data is defined according to internal database-specific formats, usually in a compressed format. The data must be decompressed and converted into a different format so that it can be processed by the database wire protocol.
- Do not write code that relies on the number of result rows from a query because drivers must fetch all rows in a result set to know how many rows the query will return. Instead, you can either count the rows by iterating through the result set or get the number of rows by submitting a query with a COUNT column in the Select clause.
- When calling stored procedures, always use parameter markers for argument markers instead of using literal arguments. JDBC drivers can call stored procedures on the database server either by executing the procedure as a SQLquery or by optimizing the execution by invoking a Remote Procedure Call (RPC)directly on the database server. When you execute a stored procedure as a SQLquery, the database server parses the statement, validates the argument types,and converts the arguments into the correct data types.

  Remember that SQL is always sent to the database server as a character string, for example, "{call getCustName (12345)}". In this case, even though the application programmer may have assumed that the only argument to getCustName was an integer, the argument is actually passed inside a character string to the server. The database server parses the SQL query, isolates the single argument value 12345, and then, converts the string '12345' into an integer value before executing the procedure as a SQL language event. By invoking a RPC on the database server, the overhead of using a SQL character string is avoided. Instead, the JDBC driver constructs a network packet that contains the parameters in their native data type formats and executes the procedure remotely.

Case 1: Not Using a Server-Side RPC

In this example, the stored procedure getCustName cannot be optimized to use a server-side RPC. The database server must treat the SQL request as a normal language event, which includes parsing the statement, validating the argument types, and converting the arguments into the correct data types before executing the procedure.

```
CallableStatementcstmt = conn.prepareCall (
"{callgetCustName (12345)}");
ResultSetrs = cstmt.executeQuery ();
```

Case 2: Using a Server-Side RPC

In this example, the stored procedure getCustName can be optimized to use a server-side RPC. Because the application avoids literal arguments and calls the procedure by specifying all arguments as parameters, the JDBC driver can optimize the execution by invoking the stored procedure directly on the database as an RPC. The SQL language processing on the database server is avoided and execution time is greatly improved.

```
CallableStatementcstmt – conn.prepareCall (
"{callgetCustName (?)}");
cstmt.setLong (1,12345);
ResultSetrs = cstmt.executeQuery();
```

- Understand the usage of Statement and PreparedStatement Object. JDBC drivers are optimized based on the perceived use of the functions that are being executed. Choose between the PreparedStatement object and the Statement object depending on how you plan to use the object. The Statement object is optimized for a single execution of a SQL statement. In contrast, the PreparedStatement object is optimized for SQL statements that will be executed two or more times.

  The overhead for the initial execution of a PreparedStatement object is high. The benefit comes with subsequent executions of the SQL statement. For example, suppose we are preparing and executing a query that returns employee information based on an ID. Using a PreparedStatement object, a JDBC driver would process the prepare request by making a network request to the database server to parse and optimize the query. The execute results in another network request. If the application will only make this request once during its lifespan using a Statement object instead of a PreparedStatement object results in only a single network roundtrip to the database server. Reducing network communication typically provides the most performance gains.

  This guideline is complicated by the use of prepared statement pooling because the scope of execution is longer. When using prepared statement pooling, if a query only will be executed once, use the Statement object. If a query will be executed infrequently, but may be executed again during the life of a statement pool inside a connection pool, use a Prepared Statement object. Under similar circumstances without statement pooling, use the Statement object.
- Using Batches Instead of Prepared Statements. Updating large amounts of data typically is done by preparing an Insert statement and executing that statement multiple times, resulting in numerous network roundtrips. To reduce the number of JDBC calls and improve performance, you can send multiple queries to the database at a time using the add Batch method of the PreparedStatement object.

  Case 1: Executing Prepared Statement Multiple Times

```
PreparedStatement ps = conn.prepareStatement(
"INSERT INTO employees VALUES (?, ?, ?)");
for (n = 0; n < 100; n++) {
    ps.setString(name[n]);
    ps.setLong(id[n]);
    ps.setInt(salary[n]);
```

```
    ps.executeUpdate();
}
```

Case 2: Using a Batch

```
PreparedStatement ps = conn.prepareStatement(
"INSERT INTO employees VALUES (?, ?, ?)");
for (n = 0; n < 100; n++) {
    ps.setString(name[n]);
    ps.setLong(id[n]);
    ps.setInt(salary[n]);
    ps.addBatch();
}
ps.executeBatch();
```

In Case 1, a prepared statement is used to execute an Insert statement multiple times. In this case, 101 network roundtrips are required to perform 100 Insert operations: one roundtrip to prepare the statement and 100 additional round trips to execute its iterations. When the addBatch method is used to consolidate 100 Insert operations, as demonstrated in Case 2, only two network roundtrips are required — one to prepare the statement and another to execute the batch. Although more database CPU cycles are involved by using batches, performance is gained through the reduction of network roundtrips. Remember that the biggest gain in performance is realized by reducing network communication between the JDBC driver and the database server.

- Use the get Methods Effectively. JDBC provides a variety of methods to retrieve data from a result set, such as getInt, getString, and getObject. The getObject method is the most generic and provides the worst performance when the non-default mappings are specified. This is because the JDBC driver must perform extra processing to determine the type of the value being retrieved and generate the appropriate mapping. Always use the specific method for the data type.

  To further improve performance, provide the column number of the column being retrieved, for example, getString(1), getLong(2), and getInt(3), instead of the column name. If column numbers are not specified, network traffic is unaffected, but costly conversions and lookups increase. For example, suppose you use getString("foo") ... A driver might have to convert foo to uppercase (if necessary), and then compare foo with all the columns in the column list. If, instead, the driver went directly to result column 23, a significant amount of processing would be saved.

  For example, suppose you have a result set that has 15 columns and 100 rows, and the column names are not included in the result set. You are interested in three columns, EMPLOYEENAME (a string), EMPLOYEENUMBER (a long integer), and SALARY (an integer). If you specify getString(“EmployeeName”), getLong(“EmployeeNumber”), and getInt(“Salary”), each column name must be converted to the appropriate case of the columns in the database metadata and lookups would increase considerably. Performance would improve significantly if you specify getString(1), getLong(2), and getInt(15).
- Manage connections effectively for application performance. Optimize your application by connecting once and using multiple statement objects, instead of performing multiple connections. Avoid connecting to a data source after establishing an initial connection.

Although gathering driver information at connect time is a good practice, it is often more efficient to gather it in one step rather than two steps. For example, some applications establish a connection and then call a method in a separate component that reattaches and gathers information about the driver. Applications that are designed as separate entities should pass the established connection object to the data collection routine instead of establishing a second connection.

Another bad practice is to connect and disconnect several times throughout your application to perform SQL statements. Connection objects can have multiple statement objects associated with them. Statement objects, which are defined to be memory storage for information about SQL statements, can manage multiple SQL statements.

You can improve performance significantly with connection pooling, especially for applications that connect over a network or through the World Wide Web Connection pooling lets you reuse connections. Closing connections does not close the physical connection to the database. When an application requests a connection, an active connection is reused, thus avoiding the network input/output needed to create a new connection.

- Manage Commits in Transactions. Committing transactions is slow because of the amount of disk input/output, and potentially network input/output, that is required. Always turn auto-commit off by using the WSConnection.setAutoCommit(false) setting.

  What does a commit actually involve? The database server must flush back to disk every data page that contains updated or new data. This is usually a sequential write to a journal file, but nevertheless, it involves disk input/output. By default, auto-commit is on when connecting to a data source and auto-commit mode usually impairs performance because of the significant amount of disk input/output needed to commit every operation.

  Furthermore, most database servers do not provide a native auto-commit mode.
  For this type of server, the JDBC driver must explicitly issue a COMMIT statement and a BEGIN TRANSACTION for every operation sent to the server. In addition to the large amount of disk input/output required to support auto-commit, a performance penalty is paid for up to three network requests for every statement issued by an application.

  Although using transactions can help application performance, do not take this tip too far. Leaving transactions active can reduce throughput by holding locks on rows for longer than necessary, preventing other users from accessing the rows. Commit transactions in intervals that allow maximum concurrency.
- Choose the Right Transaction Model. Many systems support distributed transactions; that is, transactions that span multiple connections. Distributed transactions are at least four times slower than normal transactions due to the logging and network input/output necessary to communicate between all the components involved in the distributed transaction (the JDBC driver, the transaction monitor, and the DBMS). Unless distributed transactions are required, avoid using them. Instead, use local transactions when possible. Many Java application servers provide a default transaction behavior that uses distributed transactions.

  For the best system performance, design the application to run using a single Connection object.

24. String, StringBuilder and StringBuffer guidelines

- Avoid instantiating String objects; this is usually unnecessary.

```
public class Foo {
// just do String bar = "bar";
private String bar = new String("bar");
}
```

- Use StringBuffer or StringBuilder for string concatenation rather than using String objects which are immutable

```
String myString = "Hello";
myString = myString + "Guest";  // Avoid

StringBuilder sb = new StringBuilder("Hello");
sb.append("Guest"); // instead do this
```

- StringBuffer and StringBuilder (introduced in Java 1.5) have the same methods with one difference and that's of synchronization. StringBuffer is synchronized (which means it is thread safe and hence you can use it when you implement threads for your methods) whereas StringBuilder is not synchronized(which implies it isn't thread safe).

  So, if you aren't going to use threading then use the **StringBuilder** class as it'll be more **efficient** than **StringBuffer** due to the **absence** of **synchronization**.

- Avoid concatenating non literals in a StringBuilder/StringBuffer constructor or append().

```
public class Foo {
void bar() {
    // Avoid this
    StringBuffer sb=new StringBuffer("tmp =
"+System.getProperty("java.io.tmpdir"));

    // Use instead something like this
    StringBuffer sb = new StringBuffer("tmp = ");
    sb.append(System.getProperty("java.io.tmpdir"));
 }
}
```

- Using equalsIgnoreCase() is faster than using toUpperCase/toLowerCase().equals()

```
public class Foo {
public boolean bar(String buz) {
    // should be buz.equalsIgnoreCase("baz")
    returnbuz.toUpperCase().equals("baz");
    // another unnecessary toUpperCase()
    returnbuz.toUpperCase().equalsIgnoreCase("baz");
  }
 }
```

- Use StringBuffer.length() to determine StringBuffer length rather than using StringBuffer.toString().equals("")or StringBuffer.toString().length() ==.

```
public class Foo {
void bar() {
    StringBuffersb = new StringBuffer();
    // this is bad
    if(sb.toString().equals("")) {}
    // this is good
    if(sb.length() == 0) {}
 }
}
```

- Use String.indexOf(char) when checking for the index of a single character; it executes faster.

```
public class Foo {
void bar() {
    String s = "hello world";
    // avoid this
    if (s.indexOf("d") {}
    // instead do this
    if (s.indexOf('d') {}
 }
}
```

25. General guidelines

- Avoid modifying an outer loop increment in an inner loop for update expression. It's usually a mistake, and it's confusing even if it's what's intended.

```
public class JumbledIncrementerRule1 {
public void foo() {
    for (int i = 0; i< 10; i++) {
    for (int k = 0; k < 20; i++) {
        System.out.println("Hello");
    }
   }
  }
 }
```

- Avoid unnecessary temporaries when converting primitives to Strings.

```
public String convert(int x) {
    // this wastes an object
    String foo = new Integer(x).toString();
    // this is better
    returnInteger.toString(x);
}
```

- Avoid instantiating Boolean objects; you can reference Boolean.TRUE, Boolean.FALSE, or call Boolean.valueOf() instead.

```
public class Foo {
```

```
        // just do a Boolean bar = Boolean.TRUE;
        Boolean bar = new Boolean("true");

        // just do a Boolean buz = Boolean.FALSE;
        Boolean buz = Boolean.valueOf(false);
    }
```

- Avoid creating BigDecimal with a decimal (float/double) literal. One might assume that "new BigDecimal(.1)" is exactly equal to .1, but it is actually equal to .1000000000000000055511151231257827021181583404541015625. This is so because .1 cannot be represented exactly as a double (or, for that matter, as a binary fraction of any finite length). Thus, the long value that is being passed in to the constructor is not exactly equal to .1, appearances notwithstanding. The (String) constructor, on the other hand, is perfectly predictable. 'newBigDecimal(".1")' is exactly equal to .1, as one would expect. Therefore, it is generally recommended that the (String) constructor be used in preference to this one.

- An operation on an Immutable object (String, BigDecimal or BigInteger) won't change the object itself. The result of the operation is a new object. Therefore, ignoring the operation result is an error.

```
importjava.math.*;

class Test {

void method1() {
    BigDecimalbd=new BigDecimal(10);
    bd.add(new BigDecimal(5)); // wrong
}

void method2() {
    BigDecimalbd=new BigDecimal(10);
    bd = bd.add(new BigDecimal(5)); // correct
 }
}
```

- Avoid using interface as a container of constants; interface should be used to model the behaviour of a class.
- Avoid static imports; they may lead to poor code readability as it may no longer be clear what class a member resides in (without looking at the import statement), also it can lead to naming conflicts between class' members.

26. JavaScript guidelines

26.1. JavaScript Files

- JavaScript programs should be stored in and delivered as .js files. Prototype functions must be defined in utility.js file.

- JavaScript code should not be embedded in HTML files unless the code is specific to a single session. Code in HTML adds significantly to page weight with no opportunity for mitigation by caching and compression.

- <script src=filename.js> tags should be placed as late in the body as possible. This reduces the effects of delays imposed by script loading on other page components.

26.2. Line Length

Avoid lines longer than 120 characters. When a statement will not fit on a single line, it may be necessary to break it. Place the break after an operator, ideally after a comma. A break after an operator decreases the likelihood that a copy-paste error will be masked by semicolon insertion. The next line should be indented 8 spaces.

26.3. Variable Declarations

All variables should be declared before used. JavaScript does not require this, but doing so makes the program easier to read and makes it easier to detect undeclared variables that may become implied global. The var statements should be the first statements in the function body. It is preferred that each variable be given its own line and comment. They should be listed in alphabetical order.

```
var currentEntry; // currently selected table entry
var level;        // indentation level
var size;         // size of table
```

26.4. Function Declarations

All functions should be declared before they are used. Inner functions should follow the var statement. This helps make it clear what variables are included in its scope. There should be no space between the name of a function and the (*(left parenthesis)* of its parameter list. There should be one space between the) (right parenthesis) and the { (left curly brace) that begins the statement body. The body itself is indented four spaces. The } (right curly brace) is aligned with the line containing the beginning of the declaration of the function.

```
function outer(c, d) {
    var e = c * d;

    function inner(a, b) {
        return (e * a) + b;
    }

    return inner(0, 1);
}
```

26.5. Names

- Names should be formed from the 26 upper and lower case letters (A .. Z, a .. z), the 10 digits (0 ... 9), and _ (underscore). Avoid use of international characters because they may not read well or be understood everywhere.
- Do not use $ (dollar sign) or \ (backslash) in names. Do not use _ (underbar) as the first character of a name.
- Function names should begin with a lowercase letter, and the first letter of each subsequent new word should be uppercase with all other letters lowercase.
- Global variables should be in all caps.
- Class names should begin with a capital letter, and the first letter of each subsequent new word should be capitalized with all other letters lowercase.
- Variable names should begin with a lowercase letter, and the first letter of each subsequent word should be uppercase with all other letters lowercase.
- Variable names should indicate their data type with a consistent prefix (bln – boolean; flt – floating point; int – integer; obj – object; str – string).

26.6. Whitespace

- The basic indentation is two spaces. Tabs are not to be used at all.
- Try to keep lines to 120 characters or less. When wrapping lines, try to indent to line up with a related item on the previous line.
- Lines should not contain trailing spaces, even after binary operators, commas or semicolons.
- Separate binary operators with spaces.
- Spaces after commas and semicolons, but not before.
- Spaces after keywords, e.g. if (x > 0).
- One (or two) blank lines between block definitions. Also consider breaking up large code blocks with blank lines.
- A keyword followed by (*(left parenthesis)* should be separated by a space.

26.7. Statements

26.7.1. Simple Statements

Each line should contain at most one statement. Put a ; (semicolon) at the end of every simple statement. Note that an assignment statement which is assigning a function literal or object literal is still an assignment statement and must end with a semicolon.

JavaScript allows any expression to be used as a statement. This can mask some errors, particularly in the presence of semicolon insertion. The only expressions that should be used as statements are assignments and invocations.

26.7.2. Compound Statements

Compound statements are statements that contain lists of statements enclosed in { } (curly braces).

- The enclosed statements should be indented four more spaces.
- The { (left curly brace) should be at the end of the line that begins the compound statement.
- The } (right curly brace) should begin a line and be indented to align with the beginning of the line containing the matching { (left curly brace).
- Braces should be used around all statements, even single statements, when they are part of a control structure, such as if or for statement. This makes it easier to add statements without accidentally introducing bugs.

26.7.3. Labels

Statement labels are optional. Only these statements should be labeled: while, do, for, switch.

26.7.4. return Statement

A return statement with a value should not use ( ) (parentheses) around the value. The return value expression must start on the same line as the return keyword in order to avoid semicolon insertion.

26.7.5. if Statement

The "if" 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
}
```

26.7.6. for Statement

A for class of statements should have the following form:

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

for (variable in object) {
    if (filter) {
        statements
    }
}
```

The first form should be used with arrays and with loops of a predetermined number of iterations.

The second form should be used with objects. Be aware that members that are added to the prototype of the *object* will be included in the enumeration. It is wise to program defensively by using the hasOwnProperty method to distinguish the true members of the *object*:

```
for (variable in object) {
      if (object.hasOwnProperty(variable)) {
          statements
      }
  }
```

26.7.7. while Statement

A while statement should have the following form:

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

26.7.8. switch Statement

A switch statement should have the following form:

```
switch (expression) {
case expression:
    statements
default:
    statements
}
```

Each case is aligned with the switch. This avoids over-indentation.

Each group of *statements* (except the default) should end with break, return, or throw. Do not fall through.

26.7.9. try Statement

The try class of statements should have the following form:

```
try {
    statements
} catch (variable) {
    statements
}

try {
    statements
} catch (variable) {
    statements
} finally {
    statements
}
```

26.7.10. continue & with statements

Avoid use of continue and with statements. Continue tends to obscure the control flow of the function

26.8. General

- All the functions should be declared such that they are targeting just one concern. If a single function is having the code for multiple functionalities, than we would want to consider breaking the function into two individual functions.
- Use {} instead of new Object().
- Use [] instead of new Array().
- In JavaScript blocks do not have scope. Only functions have scope. Do not use blocks except as required by the compound statements.
- Avoid using the *eval* function.
- Write a JavaScript code that is compatible across all the browsers.
- Always pass *radix* in the *parseInt(string, radix)* method to avoid inconsistent result.
- Sanitize the data to make sure that all the data that goes into systems is clean and exactly what is needed. If data is a number, than cast it to a number. If we know that it represents a Date, then convert it to a Date object.

```
//age is a String.
var age = document.getElementById('age').value;
//sanitize your data
age = (+age);//the unary operator will convert age to a number.
//or
age = parseInt(age, 10);
```

- Global variables should not be used, use closures instead. To keep variable declarations closer to where they're actually used in the code, use anonymous functions to give yourself an inline local scope.

26.9. MVC Guidelines

Below are the guidelines for MVC based JavaScript development using frameworks like Backbone.js, Angular.js etc. These frameworks are generally used for development of script-heavy single page application.

26.9.1. Single page application

- Model should be single source of data. Instead of storing data in the DOM or in random objects, there is a set of in-memory models which represent all of the state/data in the application.
- Instead of making things global, we should try to create small subsystems that are not interdependent. Dependencies make code hard to set up for testing.
- Browsers incompatibilities are in the DOM implementations, not in the JavaScript implementations, so it makes sense to minimize and isolate DOM -dependent code.
- By coding the page state into the URL, single-page web applications can support deep bookmarks and the browser's back button. Approach is to utilize the location hash, i.e. the local part of the URL.
- The URL must be changeable without reloading the page.
- Each and every action that modifies the current page must trigger a URL change.

26.9.2. Asynchronous Module Definition

- Use module pattern based on AMD (Asynchronous Module Definition). Separate code into modules, which each handles a single responsibility. Asynchronous module definition (AMD) is a JavaScript API for defining modules such that the module and its dependencies can be asynchronously loaded. It is useful in improving the performance of websites by bypassing synchronous loading of modules along with the rest of the site content.
- Use library like require.js for asynchronous module loading and callback management.
- The benefits of AMD includes:
    - Works cross domain
    - Prevents the need for globals
    - Load only what you need, expose only what you should
    - Loads modules only once and caches them

26.9.3. JS Templating

- Use JS templating to keep your JavaScript and HTML sufficiently decoupled; which will allow you to manage your HTML and JS files reliably and easily.
- Use frameworks like Handlebars.js, Mustache.js for client side JS templating.
- The features of JS templating includes:
    - It is lightweight
    - Easy to update the HTML structure (even for HTML-only coders)
    - Compatible with jQuery
    - Works with all modern browsers including Chrome, Firefox, Safari and IE7+

26.9.4. Performance

- Avoid or defer time consuming date, number formatting operations until absolutely necessary.
- Proper clean-up of resources after use and unbinding all events from events on lifecycle events like destroy to avoid memory leaks.
- Avoiding DOM reflow for large collections by using DocumentFragment.

26.9.5. Declaration and formatting tips

- Object declarations can be made on a single line if they are short (remember the line length limits). When an object declaration is too long to fit on one line, there must be one property per line. Property names only need to be quoted if they are reserved words or contain special characters:

```
var map = {
    ready: 9,
    when: 4,
    "you are": 15
```

- When a chain of method calls is too long to fit on one line, there must be one call per line, with the first call on a separate line from the object the methods are called on. If the method changes the context, an extra level of indentation must be used.

```
  Elements
    .addClass( "foo" )
    .children()
     .html( "hello" )
    .end()
    .appendTo( "body" );
};
```

27. HTML guidelines

Consistency is absolutely a prerequisite for maximizing maintainability and reusability. These general guidelines for coding style can form the basis of a set of standards that will help ensure that all developers in a project—or, better, in *all* projects across an organization—write code consistently.

- Use Well-formed HTML. Well-formedness guarantees a single unique tree structure for the document that can be operated on by the DOM.
- Pick Good Names and ID Values.
- Consistent indentation.
- Limit Line Length.
- Standardize Character Case.
- Use Comments Judiciously.
- Use strict HTML 5 and CSS 3.
- HTML and CSS will adhere to Google's HTML and CSS styling rules.
- Avoid in lining JavaScript unless there's a strong reason.
- Every DIV, INPUT, TABLE, SPAN, UL and anchor tag will have a unique ID.
- Every BODY, DIV, TABLE, SPAN, UL and anchor tag will reference a CSS class.
- HTML Tables will not be used for layout purposes.
- All file names will be in lower case.

Note: Detailed guidelines on HTML will be published in next version of this document.

28. CSS guidelines

- Write valid CSS: All CSS code should comply with CSS3 standards.
- Selectors should
  - be on a single line.
  - have a space after the previous selector.
  - end in an opening brace.
  - be closed with a closing brace on a separate line without indentation.

```
.book-navigation .page-next {
}
.book-navigation .page-previous {
}
.book-admin-form {
}
```

A blank line should be placed between each group, section, or block of multiple selectors of logically related styles.

- **Multiple selectors**

Multiple selectors should each be on a single line, with no space after each comma:

```
#forum td.posts,
#forum td.topics,
#forum td.replies,
#forum td.pager {
}
```

- **Properties**

All properties should be on the following line after the opening brace. Each property should:

- be on its own line
- be indented with two spaces, i.e., no tabs
- have a single space after the property name and before the property value
- end in a semi-colon

```
#forum .description {
        color: #EFEFEF;
        font-size: 0.9em;
        margin: 0.5em;
}
```

- **Alphabetizing properties**

Multiple properties should be listed in alphabetical order.

NOT OK:

```
body {
```

```
        font-weight: normal;
background: #000;
        font-family: Helvetica, sans-serif;
        color: #FFF;
}
```

OK:

```
body {
        background: #000;
color: #FFF;
        font-family: Helvetica, sans-serif;
font-weight: normal;
}
```

Colors in RGB notation (e.g., #FFF) are preferred in uppercase, whereas non-color values should be in lowercase.

- **Properties with multiple values**

  Where properties can have multiple values, each value should be separated with a space. PIER Pro style guide will be utilized for the style to be followed for the user interface.

Note: Detailed guidelines on CSS will be published in next version of this document.

29. JSP guidelines

29.1. File Organization

A JSP file consists of the following sections in the order they are listed:

1. Opening comments
2. JSP page directive(s)
3. Optional tag library directive(s)
4. Optional JSP declaration(s)
5. HTML and JSP code

29.1.1. Opening Comments

A JSP file or fragment file begins with a server side style comment:

```
<%--
  - Author(s):
  - Date:
  - Copyright Notice:
  - Description:
--%>
```

This comment is visible only on the server side because it is removed during JSP page translation. Within this comment are the author(s), the date, and the copyright notice of the revision and a description about the JSP page for web developers. In some situations, the opening comments need to be retained during translation and propagated to the client side (visible to a browser) for authenticity and legal purposes. This can be achieved by splitting the comment block into two parts; first, the client-side style comment:

```
<!--
  - Author(s):
  - Date:
  - Copyright Notice:
  -->
```

And then a shorter server side style comment:

```
<%--
  - Description:
  --%>
```

29.1.2. JSP page directive(s)

A JSP page directive defines attributes associated with the JSP page at translation time. The JSP specification does not impose a constraint on how many JSP page directives can be defined in the same page. So the following two Code Samples are equivalent (except that the first example introduces two extra blank lines in the output):

Code Sample 1:

```
<%@ page session="false" %>
<%@ page import="java.util.*" %>
<%@ page errorPage="/common/errorPage.jsp" %>
```

If the length of any directive, such as a page directive, exceeds the normal width of a JSP page (120 characters), the directive is broken into multiple lines:

Code Sample 2:

```
<%@ page    session="false"
import="java.util.*"
errorPage="/common/errorPage.jsp"
%>
```

In general, Code Sample 2 is the preferred choice for defining the page directive over Code Sample 1. An exception occurs when multiple Java packages need to be imported into the JSP pages, leading to a very long import attribute:

```
<%@ page    session="false"
import="java.util.*,java.text.*,
com.mycorp.myapp.taglib.*,
com.mycorp.myapp.sql.*, ..."
...
%>
```

In this scenario, breaking up this page directive like the following is preferred:

```
<%-- all attributes except import ones --%>
<%@ page
...
%>
<%-- import attributes start here --%>
<%@ page import="java.util.*" %>
<%@ page import="java.text.*" %>
```

Note that in general the import statements follow the local code conventions for Java technology. For instance, it may generally be accepted that when up to three classes from the same package are used, import should declare specific individual classes, rather than their package. Beyond three classes, it is up to a web developer to decide whether to list those classes individually or to use the ".*" notation. In the former case, it makes life easier to identify an external class, especially when you try to locate a buggy class or understand how the JSP page interacts with Java code. For instance, without the knowledge of the imported Java packages as shown below, a web developer will have to search through all these packages in order to locate a Customer class:

```
<%@ page import="com.mycorp.bank.savings.*" %>
<%@ page import="com.thirdpartycorp.cashmanagement.*" %>
<%@ page import="com.mycorp.bank.foreignexchange.*" %>
```

In the latter case, the written JSP page is neater but it is harder to locate classes. In general, if a JSP page has too many import directives, it is likely to contain too much Java code. A better choice would be to use more JSP tags.

29.1.3. Optional Tag Library Directive(s)

A tag library directive declares custom tag libraries used by the JSP page. A short directive is declared in a single line. Multiple tag library directives are stacked together in the same location within the JSP page's body:

```
<%@ tagliburi="URI1" prefix="tagPrefix1" %>
<%@ tagliburi="URI2" prefix="tagPrefix2" %>
...
```

Just as with the page directive, if the length of a tag library directive exceeds the normal width of a JSP page (120 characters), the directive is broken into multiple lines:

```
<%@ taglib
uri="URI2"
prefix="tagPrefix2"
%>
```

Only tag libraries that are being used in a page should be imported.

From JSP 1.2 Specification, it is highly recommended that the JSP Standard Tag Library (JSTL) be used in your web application to help reduce the need for JSP scriptlets in your pages. Pages that use JSTL are, in general, easier to read and maintain.

29.1.4. Optional JSP Declaration(s)

JSP declarations declare methods and variables owned by a JSP page. These methods and variables are no different from declarations in the Java programming language, and therefore the relevant code conventions should be followed. Declarations are preferred to be contained in a single <%! ... %>JSP declaration block, to centralize declarations within one area of the JSP page's body. Here is an example:

Disparate declaration blocks	***Preferred declaration block***

<%! privateinthitCount; %>
<%! private Date today; %>
...
<%! public intgetHitCount() {
return hitCount;
}
%>
	<%!
	private inthitCount;
	private Date today;
	
	public intgetHitCount() {
	return hitCount;
	}
	%>

29.1.5. HTML and JSP Code

This section of a JSP page holds the HTML body of the JSP page and the JSP code, such as JSP expressions, scriptlets, and JavaBeans instructions.

29.2. Tag Library Descriptor

A tag library descriptor (TLD) file must begin with a proper XML declaration and the correct DTD statement. For example, a JSP 1.2 TLD file must begin with:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPEtaglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
    "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
```

This is immediately followed by a server-side style comment that lists the author(s), the date, the copyright, the identification information, and a short description about the library:

```
<!--
  - Author(s):
  - Date:
  - Copyright Notice:
  - @(#)
  - Description:
-->
```

The rest of tag descriptor file consists of the following, in the order they appear below:

1. Optional declaration of one tag library validator
2. Optional declaration of event listeners
3. Declaration of one or more available tags

It is recommended that you always add the following optional sub-elements for the elements in a TLD file. These sub-elements provide placeholders for tag designers to document the behaviour and additional information of a TLD file, and disclose them to web component developers.

TLD Element	**JSP 1.2 Recommended Sub-element**	**JSP 1.1 Recommended Sub-element**
attribute (JSP 1.2)	Description	
init-param (JSP 1.2)	Description	
Tag	display-name, description, example	name, info
Taglib	uri, display-name, description	uri, info
validator (JSP 1.2)	Description	
variable (JSP 1.2)	Description	

29.3. Indentation

29.3.1. General

Indentations should be filled with space characters. Tab characters cause different interpretation in the spacing of characters in different editors and should not be used for indentation inside a JSP page. Unless restricted by particular integrated development environment (IDE) tools, a unit of indentation corresponds to 4 space characters. Here is an example:

```
<myTagLib:forEachvar="client" items="${clients}">
<myTagLib:mail value="${client}" />
</myTagLib:forEach>
```

A continuation indentation aligns subsequent lines of a block with an appropriate point in the previous line. The continuation indentation is in multiple units of the normal indentation (multiple lots of 4 space characters):

```
<%@ page    attribute1="value1"
            attribute2="value2"
            ...
attributeN="valueN"
%>
```

29.3.2. Indentation of Scripting Elements

When a JSP scripting element (such as declaration, scriptlet, expression) does not fit on a single line, the adopted indentation conventions of the scripting language apply to the body of the element. The body begins from the same line for the opening symbol of the element <%=, and from a new line for the opening symbol <%=. The body is then terminated by an enclosing symbol of the element (%>) on a separate line. For example:

```
<%= (Calendar.getInstance().get(Calendar.DAY_OF_WEEK)
       =Calendar.SUNDAY) ?
       "Sleep in" :
       "Go to work"
%>
```

The lines within the body not containing the opening and the enclosing symbols are preceded with one unit of normal indentation (shown as in the previous example) to make the body distinctively identifiable from the rest of the JSP page.

29.3.3. Compound Indentation with JSP, HTML and Java

Compound indentation, for JSP elements intermingled with Java scripting code and template text (HTML), is necessary to reduce the effort of comprehending a JSP source file. This is because the conventional normal indentation might make seeing the JSP source file difficult. As a general rule, apply an extra unit of normal indentation to every element introduced within another one. Note that this alters the indentations of the final output produced for the client side to render for display. The

additional indentations, however, are usually ignored (by the browser) and have no effect on the rendered output on the browser. For instance, adding more space characters before a <TABLE> tag does not change the position of a table.

29.4. Comments

29.4.1. JSP Comments

JSP comments (also called server-side comments) are visible only on the server side (that is, not propagated to the client side). Pure JSP comments are preferred over JSP comments with scripting language comments, as the former is less dependent on the underlying scripting language, and will be easier to evolve into JSP 2.0-style pages. The following table illustrates this:

Line	JSP scriptlet with scripting language comment	Pure JSP comment
single	<% /** ... */ %> <% /* ... */ %> <% // ... %>	<%-- ... --%>
multiple	<% /* * ... * */ %>	<%-- - ... - -- %>
	<% // // ... // %>	

29.4.2. Client Side Comments

Client-side comments (<!-- ... -->) can be used to annotate the responses sent to the client with additional information about the responses. They should not contain information about the behaviour and internal structure of the server application or the code to generate the responses.

The use of client-side comments is generally discouraged, as a client / user does not need or read these kinds of comments directly in order to interpret the received responses. An exception is for authenticity and legality purposes such as the identification and copyright information as described above. Another exception is for HTML authors to use a small amount of HTML comments to embody the guidelines of the HTML document structures. For example:

```
<!-- toolbar section -->
    ...
<!-- left-hand side navigation bar -->
    ...
<!-- main body -->
```

```
    ...
<!-- footer -->
```

29.5. JSP Declarations

As per the Java code convention, declarations of variables of the same types should be on separate lines:

Not recommended	Recommended
`<%! Private int x, y; %>`	`<%! Private int x; %>` `<%! Private int y; %>`

JavaBeans components should not be declared and instantiated using JSP declarations but instead should use the <jsp:useBean> action tag.

In general, JSP declarations for variables are discouraged as they lead to the use of the scripting language to weave business logic and Java code into a JSP page which is designed for presentation purposes, and because of the overhead of managing the scope of the variables.

29.6. JSP Scriptlets

Where possible, avoid JSP scriptlets whenever tag libraries provide equivalent functionality. This makes pages easier to read and maintain, helps to separate business logic from presentation logic, and will make your pages easier to evolve into JSP 2.0-style pages (JSP 2.0 Specification supports but deemphasizes the use of scriptlets). In the following examples, for each data type representation of the customers, a different scriptlet must be written:

customers as an array of Customers

```
<table>
<%  for ( inti=0; i<customers.length; i++ ) { %>
<tr>
<td><%= customers[i].getLastName() %></td>
<td><%= customers[i].getFirstName() %></td>
</tr>
<%  } %>
</table>
```

customers as an Enumeration

```
<table>
<%  for ( Enumeration e = customers.elements();
e.hasMoreElements(); ) {
                Customer customer = (Customer)e.nextElement();
        %>
<tr>
<td><%= customer.getLastName() %></td>
<td><%= customer.getFirstName() %></td>
</tr>
```

```
<%  } %>
</table>
```

However, if a common tag library is used, there is a higher flexibility in using different types of customers. For instance, in the JSP Standard Tag Library, the following segment of JSP code will support both array and Enumeration representations of customers:

```
<table>
<c:forEachvar="customer" items="${customers}">
<tr>
<td><c:out value="${customer.lastName}"/></td>
<td><c:out value="${customer.firstName}"/></td>
</tr>
</c:forEach>
</table>
```

In the spirit of adopting the model-view-controller (MVC) design pattern to reduce coupling between the presentation-tier from the business logic, JSP scriptlets should not be used for writing business logic. Rather, JSP scriptlets are used if necessary to transform data (also called "value objects") returned from processing the client's requests into a proper client-ready format. Even then, this would be better done with a front controller servlet or a custom tag. For example, the following code fetches the names of customers from the database directly and displays them to a client:

```
<%
    // NOT RECOMMENDED TO BE DONE AS A SCRIPTLET!

    Connection conn = null;
try {
    // Get connection
    InitialContextctx = new InitialContext();
    DataSource ds = (DataSource)ctx.lookup("customerDS");
    conn = ds.getConnection();

    // Get customer names
    Statement stmt = conn.createStatement();
    ResultSetrs = stmt.executeQuery("SELECT name FROM customer");

    // Display names
    while ( rs.next() ) {
        out.println(rs.getString("name") + "<br>");
    }
} catch (SQLException e) {
    out.println("Could not retrieve customer names:" + e);
} finally {
    if ( conn != null )
        conn.close();
    }
%>
```

The following segment of JSP code is better as it delegates the interaction with the database to the custom tag myTags:dataSource which encapsulates and hides the dependency of the database code in its implementation:

```
<myTags:dataSource
name="customerDS"
table="customer"
columns="name"
var="result" />
<c:forEachvar="row" items="${result.rows}">
<c:out value="${row.name}" />
<br />
</c:forEach>
```

“result” is a scripting variable introduced by the custom tag myTags:dataSource to hold the result of retrieving the names of the customers from the customer database. The JSP code can also be enhanced to generate different kinds of outputs (HTML, XML, WML) based on client needs dynamically, without impacting the backend code (for the dataSource tag). A better option is to delegate this to a front controller servlet which performs the data retrieval and provide the results to the JSP page through a request-scoped attribute.

In summary:

- JSP scriptlets should ideally be non-existent in the JSP page so that the JSP page is independent of the scripting language, and business logic implementation within the JSP page is avoided.

- If not possible, use value objects (JavaBeans components) for carrying information to and from the server side, and use JSP scriptlets for transforming value objects to client outputs.

- Use custom tags (tag handlers) whenever available for processing information on the server side.

29.7. JSP Expressions

JSP Expressions should be used just as sparingly as JSP Scriptlets. To illustrate this, let's look as the following three examples which accomplish equivalent tasks:

Example 1 (with explicit Java code):

```
<%= myBean.getName() %>
```

Example 2 (with JSP tag):

```
<jsp:getProperty name="myBean" property="name" />
```

Example 3 (with JSTL tag):

```
<c:out value="${myBean.name}" />
```

Example 1 assumes that a scripting variable called myBean is declared. The other two examples assume that myBean is a scoped attribute that can be found using PageContext.findAttribute(). The second example also assumes that myBean was introduced to the page using <jsp:useBean>.

Of the three examples, the JSTL tag example is preferred. It is almost as short as the JSP expression, it is just as easy to read and easier to maintain, and it does not rely on Java scriptlets (which would require the web developer to be familiar with the language and the API calls). Furthermore, it makes the page easier to evolve into JSP 2.0-style programming, where the equivalent can be accomplished by simply typing ${myBean.name} in template text. It should be noted that the JSTL example is actually slightly different in that it gets the value of myBean from the page context instead of from a local Java scripting variable.

Finally, JSP expressions have preference over equivalent JSP scriptlets which rely on the syntax of the underlying scripting language. For instance,

```
<%= x %>
```

is preferred over

```
<% out.print( x ); %>
```

29.8. White Space

White space further enhances indentation by beautifying the JSP code to reduce comprehension and maintenance effort. In particular, blank lines and spaces should be inserted at various locations in a JSP file where necessary.

29.9. Blank Lines

Blank lines are used sparingly to improve the legibility of <strike>the</strike>a JSP page, provided that they do not produce unwanted effects on the outputs. For the example below, a blank line inserted between two JSP expressions inside an HTML <PRE> block call causes an extra line inserted in the HTML output to be visible in the client's browser. However, if the blank line is not inside a <PRE> block, the effect is not visible in the browser's output.

JSP statements	HTML output to client

`<pre>`
`<%= customer.getFirstName() %>`
`<%= customer.getLastName() %>`
`</pre>`
	Joe
	Block

`<pre>`
`<%= customer.getFirstName() %>`
	Joe
	
	Block

<%= customer.getLastName() %> </pre>	
<%= customer.getFirstName() %> <%= customer.getLastName() %>	Joe Block

29.9.1. Blank Spaces

A white space character (shown as ␣) should be inserted between a JSP tag and its body. For instance, the following

```
<%=␣customer.getName()␣%>
```

is preferred over

```
<%=customer.getName()%>
```

There should also be space characters separating JSP comment tags and comments:

```
<%--
  -␣a multi-line comment broken into pieces, each of which
  -␣occupying a single line.
  --%>
<%-- a short comment --%>
```

29.10. Naming Conventions

29.10.1. JSP Names

A JSP (file) name should always begin with a lower-case letter. The name may consist of multiple words, in which case the words are placed immediately adjacent and each word commences with an upper-case letter. A JSP name can be just a simple noun or a short sentence. A verb-only JSP name should be avoided, as it does not convey sufficient information to developers. For example ***perform.jsp*** is not as clear as ***performLogin.jsp***

In the case of a verb being part of a JSP name, the present tense form should be used, since an action by way of backend processing is implied. For example ***showAccountDetails.jsp*** is preferred over ***showingAccountDetails.jsp.***

29.10.2. Tag Names

The naming conventions for tag handlers and associated classes are shown below:

Description	Class Name
XXX tag extra info (extending from javax.servlet.jsp.tagext.TagExtraInfo)	XXXTEI
XXX tag library validator (extending from javax.servlet.jsp.tagext.TagLibraryValidator)	XXXTLV

XXX tag handler interface (extending from javax.servlet.jsp.tagext.Tag/IterationTag/BodyTag)	XXXTag
XXX tag handler implementation	XXXTag

In addition, tag names must not violate the naming conventions of class and interface as specified in the relevant code convention for Java technology.

To further distinguish a tag-relevant class from other classes, a package suffix, tags, or taglib can be applied to the package name of the class. For example: `com.mycorp.myapp.tags.XXXTag`

29.10.3. Tag Prefix Names

A tag prefix should be a short yet meaningful noun in title case, and the first character in lower-case. A tag prefix should not contain non-alphabetic characters. Here are some examples:

Example	OK?
Mytaglib	no
myTagLib	yes
MyTagLib	no
MyTagLib1	no
My_Tag_Lib	no
MyTagLib	no

29.11. JSP Programming Practices

In general, avoid writing Java code (declarations, scriptlets and expressions) in your JSP pages, for the following reasons:

- Syntax errors in Java code in a JSP page are not detected until the page is deployed. Syntax errors in tag libraries and servlets, on the other hand, are detected prior to deployment.
- Java code in JSP pages is harder to debug.
- Java code in JSP pages is harder to maintain, especially for page authors who may not be Java experts.
- It is generally accepted practice not to mix complex business logic with presentation logic. JSP pages are primarily intended for presentation logic.
- Code containing Java code, HTML and other scripting instructions can be hard to read.

- The JSP 2.0 Specification is deemphasizing scriptlets in favour of a much simpler expression language. It will be easier to evolve your JSP pages to JSP 2.0-style programming if Java code is not used in your pages.

29.12. JavaBeans Component Initialization

JSP technology provides a convenient element to initialize all PropertyDescriptor-identified properties of a JavaBeans component. For instance:

```
<jsp:setProperty name="bankClient" property="*"/>
```

However, this should be used with caution. First, if the bean has a property, say, amount, and there is no such parameter (amount) in the current ServletRequest object or the parameter value is "", nothing is done: the JSP page does not even use null to set that particular property of the bean. So, whatever value is already assigned to amount in the bankClient bean is unaffected. Second, non-elementary properties that do not have PropertyEditors defined may not be implicitly initialized from a String value of the ServletRequest object and explicit conversion may be needed. Third, malicious users can add additional request parameters and set unintended properties of the bean, if the application is not carefully designed.

If you still prefer to use property="*" in the jsp:setProperty tag for the purpose of producing neater code, then we recommend that you add a comment preceding the jsp:setProperty tag about parameters expected to be present in the ServletRequest object to initialize the bean. So, in the following example, if we know that properties firstName and lastName are required to initialize the bankClient bean, then we should add a comment.

```
<%--
  -  requires firstName and lastName from the ServletRequest
  --%>
<jsp:setProperty name="bankClient" property="*" />
```

29.13. JSP implicit Objects

Direct use of JSP implicit objects to gain references to these objects rather than API calls is preferred. So, instead of using

```
getServletConfig().getServletContext().getInitParameter("param")
```

to access the initialization parameter as provided by the ServletContext instance, one can make use of the readily available implicit object:

```
application.getInitParameter("param")
```

In the case that only the value of an initialization parameter is outputted, it would be even better to use a JSTL tag to access the initialization parameter:

```
<c:out value="${initParam['param']}" />
```

29.14. Using Custom Tags

If a custom tag does not have a body content, the content should be declared explicitly with empty (rather than defaulting to the word "JSP") like this in the tag library descriptor:

```
<tag>
<name>hello</name>
<tag-class>com.mycorp.util.taglib.HelloTagSupport</tag-class>
<body-content>empty</body-content>
    ...
</tag>
```

This tells the JSP container that the body content must be empty rather than containing any JSP syntax to be parsed. The effect is to eliminate unnecessarily allocation of resources for parsing of empty body contents.

Empty tags should be in short XML elements, rather than using opening-closing XML element pairs to improve readability. So, `<myTag:hello />`is preferred over `<myTag:hello></myTag:hello>`

29.15. Using Custom Tags

Sometimes, the valid ways to use a tag library cannot be expressed using the TLD alone. Then, a TagExtraInfo class or a TagLibraryValidator class should be written and registered in the TLD so that errors in tag library can be caught at translation time.

29.16. Using JavaScript in JSP Files

JavaScript should be independent of particular features of browser types in order for the scripts to run properly.

Where it makes sense, it is a good idea to keep JavaScript code in individual files separate from JSP bodies, and use a statement like the following to import the JavaScript code into the JSP bodies:

```
<script language=javascriptsrc="/js/main.js">
```

This improves the chance for the JavaScript code to be reused, maintains the consistent behaviour of the JavaScript code across multiple JSP pages, and reduces the complexity of JSP pages.

29.17. Use of Composite View Patterns

When a JSP page requires a certain and complex structure which may also repeat in other JSP pages, one way to handle this is to break it up into pieces, using the Composite View pattern. For instance, a JSP page sometimes has the following logical layout in its presentation structure:

<table>
<tr><td colspan="2">Header</td></tr>
<tr><td>menu bar</td><td>main body</td></tr>
</table>

	Footnote
Footer	

In this manner, this composite JSP page can be divided into different modules, each realized as a separate JSP fragment. Constituent JSP fragments can then be placed in appropriate locations in the composite JSP page, using translation-time or request-time include JSP tags. In general, when static include directives are used to include a page that would not be requested by itself, remember to use the .jspf extension and place the file in the /WEB-INF/jspf/ directory of the Web application archive (war). For example:

```
<%@ include file="/WEB-INF/jspf/header.jspf" %>
...
<%@ include file="/WEB-INF/jspf/menuBar.jspf" %>
...
<jsp:include page="<%= currentBody %>" />
...
<%@ include file="/WEB-INF/jspf/footnote.jspf" %>
...
<%@ include file="/WEB-INF/jspf/footer.jspf" %>
```

30. Tools

30.1. Static code analysis

JSAG recommends using the static code analyser tools PMD, CheckStyle and FindBugs on each developer's machine before the code is checked-in to the project repository. Project build should be setup on the JSAG Jenkins Continuous integration server along with JSAG Sonar which should be configured to generate regular code analysis reports.

Please refer to the documents available at "JSAG Home Page" or the latest revision for Jenkins and Sonar build setup and configuration guidelines. Developers should configure IDE plugins for PMD, CheckStyle and FindBugs on their machines and load the JSAG selected rule configurations from the links given below:

- **PMD**
  http://sonar.nagarro.com/sonar/profiles/export?format=pmd&language=java&name=Nagarro%2520Way
- **CheckStyle**
  http://sonar.nagarro.com/sonar/profiles/export?format=checkstyle&language=java&name=Nagarro%2520Way
- **FindBugs**
  http://sonar.nagarro.com/sonar/profiles/export?format=findbugs&language=java&name=Nagarro%2520Way

30.2. Artifactory Server

Artifactory provides out-of-the-box, all that is needed to set up and run a robust secured repository within Nagarro projects. It is built with high concurrency in mind and it offers fast and reliable downloads and deployments on top of a consistent storage mechanism so that you'll never lose your precious artifacts.

Find the details of Artifactory Server and developer guide at JSAG Home Page

Some of the salient features of Artifactory server are:

- Binary Repository Manager for Maven, Ivy, Gradle modules, etc.
- Supports hosting and remote proxying of artifacts - browsable, secure, annotated & searchable.
- Integrates with CI servers, like Jenkins, TeamCity and Bamboo, for fully traceable builds.

31. JSAG Frameworks

JSAG recommends the frameworks listed below to be used in Nagarro Java projects for consistency, best practices and development acceleration. All new Java development projects must utilize the JSAG frameworks listed below. In case any project does not want to use the frameworks listed below, an email for approval should be sent to JSAG (mailto:jsag@nagarro.com) along with the reasons for not being able to use the particular framework(s).

Any queries, suggestions and issues regarding the framework(s) should be sent to the JSAG group email id.

31.1. Exception handling framework

JSAG Exception handling framework provides the basic set of exception and utility classes to capture and propagate exception related information between different layers and components in an application. It provides the following set of features:

- **BusinessException** – An abstract class which takes care of exceptions, which occur due to business rules violation. For example: Exception occurring due to invalid account number.
- **TechnicalException** – An abstract class which takes care of all recoverable technical exceptions. For example, if a connection timeout needs to be intercepted and retry mechanism needs to be implemented.
- **TechnicalRuntimeException -** An abstract class which takes care of all irrecoverable technical exceptions like Database listener being down.
- **i18n** (internationalization support) – This is for exception messages, which would be shown as per the locale. Application would need to provide the resource bundle with language based values.
- **XML Conversion -** Provides exception details in the XML format for easy debugging and troubleshooting.
- **Problematic object and method state -** Provides problematic object (the object on which the exception occurred) state snapshot, which will include the instance and local variable values.
- **Original exception -** Embeds the original exception

Please refer to the "JSAG Exception Handling Framework [v1.0].zip" or the latest revision for the binary code, sample application code, JavaDocs and the programming guide available on JSAG Home page . To get the latest available revision of the framework, the Project manager/Technical Lead/Management mentor should send an email request with project details to JSAG group email id.

31.2. JSAG Logging framework

JSAG logging framework provides a simplified abstraction layer on top of third party logging framework API's and with a pool of utility classes and methods while hiding the complexity of the underlying logging framework.

Features available in the current version:

- **Ease of configuration:**
  - Simple XML based configuration for JSAG Logging framework. The format of configuration file obviates the need of the application defining logging API specific configuration file.
  - Provides easy and flexible configuration mechanism to handle logging configuration.
- **Mandatory parameter contracts checking**
  - Minimal String manipulation and lazy parameter expansion.
  - Support mandatory parameters contract for the applications to check at compile time.
- **Internationalization of log messages**
  - Support for internationalizing log messages using the Logger interface.
  - Supports logging of a locale based log message specified by the bundle key, as opposed to the actual message itself.
- **Flexibility while logging exception stack traces**
  - The ability to tell Logger not to dump stack traces of exceptions even if the exception was logged using the exception log methods. This is useful if you wish to run your app in a slightly quieter mode - you might not care to see all the exception stack traces during a particular run.
  - Supports the logging of a message that is associated with a particular exception by passing the exception argument before the message key string. This is due to the nature of how varargs are parsed - putting the exception last in the argument list, as is the case with most other logging frameworks, would cause only the exception message to be logged and not its full stack trace.
- **Asynchronous logging**
  - Provides asynchronous log processor (handle the logging in a separate thread). Async Handler collects the events sent to it and then dispatches them to all the real handlers that are referred. It uses a separate thread to serve the events in its buffer
- **Performance Logging**
  - Supports Performance logging as it provided wrapper classes of Perf4J framework. Performance logging uses underlying JSAG Logging handler.

Please refer to the "JSAG Logging Framework Usage guide", sample application code, JavaDocs available on JSAG Home page. To get the latest available revision of the framework, the Project manager/Technical Lead/Management mentor should send an email request with project details to JSAG group email id.