|  |
| --- |
|  |
|  |
|  |
| CA/CST  Java Coding Standards  Version 1.0 |
| January 31, 2014 |
|  |
|  |
|  |

Record of Changes

| No. | Date | | | | Reference: Page, Table, Figure, Paragraph | A = Add. M = Mod. D = Del. | | Change Description | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | | | 01/30/2014 | Entire Document | |  | | Initial Draft | | |
|  | |  | | |  | |  | |  |
|  |  | | | |  |  | |  | | |
|  |  | | | |  |  | |  | | |
|  |  | | | |  |  | |  | | |
|  |  | | | |  |  | |  | | |
|  |  | | | |  |  | |  | | |
|  |  | | | |  |  | |  | | |
|  |  | | | |  |  | |  | | |
|  |  | | | |  |  | |  | | |
|  |  | | | |  |  | |  | | |
|  |  | | | |  |  | |  | | |
|  |  | | | |  |  | |  | | |
|  |  | | | |  |  | |  | | |

Contents

[1 Introduction 1](#_Toc379531712)

[1.1 Purpose 1](#_Toc379531713)

[1.2 Scope 1](#_Toc379531714)

[1.3 Code Change Scope 1](#_Toc379531715)

[2 References 3](#_Toc379531716)

[3 Java Best Practices 5](#_Toc379531717)

[4 Naming Conventions 6](#_Toc379531718)

[4.1 Packages and Files 7](#_Toc379531719)

[4.2 Constants 8](#_Toc379531720)

[5 Formatting Conventions 10](#_Toc379531721)

[5.1 Class File Structure 10](#_Toc379531722)

[5.2 Comments 10](#_Toc379531723)

[5.2.1 Comments: Class Header 11](#_Toc379531724)

[5.2.2 Comments: Method Header 11](#_Toc379531725)

[5.2.3 Definition of Comments 12](#_Toc379531726)

[5.2.4 Non-Javadoc Comment 13](#_Toc379531727)

[5.3 Indentation 13](#_Toc379531728)

[5.4 White Space 13](#_Toc379531729)

[5.4.1 Blank Lines 13](#_Toc379531730)

[5.4.2 Blank Spaces 13](#_Toc379531731)

[6 Statements 15](#_Toc379531732)

[6.1 Compound Statements 15](#_Toc379531733)

[6.2 Return Statement 15](#_Toc379531734)

[6.3 if, if-else, if else-if else Statements 16](#_Toc379531735)

[6.4 for Statements 16](#_Toc379531736)

[6.5 for-each Statements 17](#_Toc379531737)

[6.6 while Statements 17](#_Toc379531738)

[6.7 do-while Statements 18](#_Toc379531739)

[6.8 switch Statements 18](#_Toc379531740)

[6.9 try-catch Statements 19](#_Toc379531741)

[6.10 Exceptions 19](#_Toc379531742)

[6.10.1 Throwing An Exception 19](#_Toc379531743)

[6.10.2 Catching An Exception 20](#_Toc379531744)

[6.10.3 Top-Level Exception 21](#_Toc379531745)

[6.10.4 Best Practices for Exception Handling 22](#_Toc379531746)

[6.10.5 The Nature of Exceptions 23](#_Toc379531747)

[6.10.6 Types of Exceptions in Java 23](#_Toc379531748)

[6.10.7 Best Practices for Designing the API 24](#_Toc379531749)

[6.10.8 Best Practices for Using Exceptions 26](#_Toc379531750)

[6.10.9 Information Exposure Through an Error Message 28](#_Toc379531751)

[7 Resources Management 30](#_Toc379531752)

[8 Code Level Metrics 31](#_Toc379531753)

[9 Security Standards 33](#_Toc379531754)

[9.1 HTML I/O Handling (XSS) 33](#_Toc379531755)

[9.2 Dynamic SQL 34](#_Toc379531756)

[9.3 Buffer Copy without Checking Size of Input ('Classic Buffer Overflow') 35](#_Toc379531757)

[9.4 Cross-Site Request Forgery (CSRF) 35](#_Toc379531758)

[9.5 Improper Access Control (Authorization) 35](#_Toc379531759)

[9.6 Reliance on Un-trusted Inputs in a Security Decision 36](#_Toc379531760)

[9.7 Path Traversal 36](#_Toc379531761)

[9.8 Unrestricted Upload of File with Dangerous Type 36](#_Toc379531762)

[9.9 Operating System Commands 37](#_Toc379531763)

[9.10 Missing Encryption of Sensitive Data 37](#_Toc379531764)

[9.11 Use of Hard-coded Credentials 37](#_Toc379531765)

[9.12 Buffer Access with Incorrect Length Value 38](#_Toc379531766)

[9.13 Improper Validation of Array Index 38](#_Toc379531767)

[9.14 Improper Check for Unusual or Exceptional Conditions 38](#_Toc379531768)

[9.15 Error Message Handling 39](#_Toc379531769)

[9.16 Integer Overflow or Wraparound 40](#_Toc379531770)

[9.17 Incorrect Calculation of Buffer Size 40](#_Toc379531771)

[9.18 Missing Authentication for Critical Function 41](#_Toc379531772)

[9.19 Download of Code Without Integrity Check 41](#_Toc379531773)

[9.20 Incorrect Permission Assignment for Critical Resource 42](#_Toc379531774)

[9.21 Allocation of Resources Without Limits or Throttling 42](#_Toc379531775)

[9.22 URL Redirection to Un-trusted Site ('Open Redirect') 43](#_Toc379531776)

[9.23 Use of a Broken or Risky Cryptographic Algorithm 43](#_Toc379531777)

[9.24 Race Conditions 44](#_Toc379531778)

[10 Section 508 Compliance 46](#_Toc379531779)

[11 Simple Coding Guidelines 47](#_Toc379531780)

[12 Declarations 49](#_Toc379531781)

[12.1 Number Per Line 49](#_Toc379531782)

[12.2 Initialization 49](#_Toc379531783)

[12.3 Placement 49](#_Toc379531784)

[12.4 Class, Enumeration, and Interface Declarations 50](#_Toc379531785)

[12.5 Visibility 51](#_Toc379531786)

[13 Application Logging 52](#_Toc379531787)

[13.1 Logging Levels 53](#_Toc379531788)

[14 Database JDBC Best Practice 54](#_Toc379531789)

[15 Code Example 56](#_Toc379531790)

Tables

[Table 4‑1. Identifier Naming Rules 6](#_Toc273023665)

[Table 4‑2. Package and File Naming Rules 7](#_Toc273023666)

[Table 4‑3. File Extensions 7](#_Toc273023667)

[Table 6‑1. Exceptions 24](#_Toc273023668)

[Table 8‑1. Code level metrics 31](#_Toc273023669)

[Table 9‑1. HTML characters to replace 34](#_Toc273023670)

[Table 13‑1. Log levels 53](#_Toc273023671)

Figures

[Figure 6-1. NullPointerException hierarchy 23](#_Toc273023672)

# Introduction

This document is derived from the Office of Biometric Information Management (OBIM) Java coding standards developed and leveraged by the United States Visitor and Immigrant Status Indicator Technology (US-VISIT) program team during the course of all Java development activities.

## Purpose

Coding conventions help make sure that project code has a consistent structure and style. They are intended to make the code easier to read, understand, review, and maintain and further reduce the complexity of the code.

Additional coding guidelines in the form of code metrics (measurements) and coding rules are provided in this document for developers to use. The code metrics guidelines consist of “hand-calculable” measurements within each method and each class. The coding rules are based on information gathered from multiple sources to enhance the security, reliability, maintainability, testability and performance of the code.

This document is intended for the following uses:

* Desk-side reference for CA/CST Java Web Services developers during coding.
* Source for GTM code review checklists and criteria.
* Reference for developers who must develop extend and maintain the CA/CST Java Web Services.
* As a Governance Tool for CA/CST GTM’s.

## Scope

This document describes the following for the CA/CST:

* Naming conventions for projects, files, objects, variables, and other code constructs.
* Formatting conventions for code modules and their comments.
* Error handling conventions.
* Complexity conventions.
* Section 508 compliance.
* Security standards.
* Logging conventions.
* Coding practices and recommendations.

## Code Change Scope

The Java coding standards described in this document apply to new applications (new code) and existing code in the following ways:

* New code in the middle of an existing file should follow new coding standards with explanations regarding the change being made to be consistent with new guidelines, ensuring no disruption to the existing code structure. Exceptions can be made to this rule if following new guidelines creates significant and unnecessary and potentially dangerous re-work.
* New files within existing application must follow the Java coding standards documented in this document.
* New applications must follow the Java coding standards documented in this document.

# References

The following sources were used in creation of the original version of this standard.

* US-VISIT Java Coding Standards Version 1.1 November 19, 2010 CI: USVISIT-TO027-CSD01555-SE-Coding\_Standard-001-F
* Coding Conventions for the Java Programming Language, http://www.oracle.com/technetwork/java/codeconvtoc-136057.html
* Draft Coding Standard, [http://g.oswego.edu/dl/html/javaCodingStd.html](http://g.oswego.edu/dl/html/javaCodingStd.html%20)
* Exception Handling, <http://www.onjava.com/lpt/a/4345>
* Maven 1.x Best Practice, <http://maven.apache.org/maven-1.x/using/bestpractices.html>
* Javadoc Information, <http://java.sun.com/j2se/javadoc/writingdoccomments/>
* Javadoc Tool Information, <http://www.oracle.com/technetwork/java/index.html>
* JUnit Information, <ftp://ftp.sei.cmu.edu/public/documents/97.reports/ps/97hb001.ps> Handbook: CMU/SEI-97-HB-001. January 1997. C4 Software Technology Reference Guide —A Prototype, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213
* A Proposed Taxonomy for Software Development Risks for High-Performance Computing (HPC) Scientific/Engineering Applications, TECHNICAL NOTE, CMU/SEI-2006-TN-039 January 2007, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213
* Applying and Interpreting Object Oriented Metrics, Presenter Dr. Linda H. Rosenberg, <http://www.literateprogramming.com/ooapply.pdf>
* Minimizing code defects to improve software quality and lower development costs, Development Solutions White paper, October 2008, IBM/Rational
* Managing Software Risks in Software Intensive Systems with Metrics and Measures, Robert A. Martin, MITRE, presentation to SEI/CMU Conference on the Acquisition of Software-Intensive Systems 2003, 30 January 2003.
* NIST Special Publication 500-235, Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric, Arthur H. Watson, Thomas J. McCabe, Prepared under NIST Contract 43NANB517266 , Dolores R. Wallace, Editor Computer Systems Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899-0001, August 1996
* Shyam R. Chidamber, Chris F. Kemerer. A Metrics suite for Object Oriented design. M.I.T. Sloan School of Management E53-315. 1993. <http://maisqual.squoring.com/wiki/images/5/5c/Chid_kem_metrics.pdf>
* Victor Basili, Lionel Briand and Walcelio Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. IEEE Transactions on Software Engineering. Vol. 22, No. 10, October 1996. <http://www.cs.umd.edu/users/basili/publications/journals/J60.pdf>
* Laing, Victor & Coleman, Charles: Principal Components of Orthogonal Object-Oriented Metrics. White Paper Analyzing Results of NASA Object-Oriented Data. SATC, NASA, 2001. <http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=CB520F14993302E2669C6BD3F4C0AC90?doi=10.1.1.95.6760&rep=rep1&type=pdf>
* O’Rielly on Java.com; <http://www.onjava.com/>
* "Does Java need Checked Exceptions?" by Bruce Eckel
* "Exceptional Java," by Alan Griffiths
* "The trouble with checked exceptions: A conversation with Anders Hejlsberg, Part II" on Artima.com
* "Checked exceptions are of dubious value," on C2.com

# Java Best Practices

This section describes Java coding best practices and the resources used to support the best practices. For more information on the tools, refer to the links provided.

* Use Logback for logging when developing code. For more information on Logback logging, please check the Logback library at URL: http://Logback.qos.ch/index.html. Logback implements the SLF4J API and is very flexible. It can, via a configuration file, select log events and filtered content may be sent to different destinations. Proper error logging quickly helps find application errors and business logic issues in development, test, and production environments. Logging should have several levels of detail and each level should be configurable at run-time.
* Use Javadoc tool or if not possible, Javadoc comment style for documenting comments within Java programs. Properly commenting the Java Application Programming Interface (API) allows for easier maintenance of the code.

1. Comment any complex and hard-to-understand code constructions, workarounds or assumptions made by the code.
2. Remove dead code rather than comment it out.
3. Use @TODO task tag to comment unfinished tasks or that code requires additional work.
4. Use FIXME task tag to document issues.
5. Use XXX task tag to comment any content that may not be appropriate.
6. All task tags should be handled and removed prior to IV&V Testing.
7. To add to the traceability of the system you are working on, use class-level comments to point to the system requirements and design that the code implements or the individual enhancement or defect report request that requires the change or update you are working on
8. Suppress or handle all compiler warnings prior to Test Readiness Review (TRR).

* If your project compiles with the latest Java release, and you have no need to support previous versions of Java, you are encouraged to use the new release features. New projects are encouraged to use the latest IT CCB approved release to take advantage of new features, as well as implement improved performance.
* Follow proper Exception Handling best practices. All exception handling should be logged to the application log. All log files should contain identifiable exceptions for operational debugging purposes. Errors should never be silently ignored. At a minimum, the condition should be logged.
* Please refer to our policy on exception handling. Specific exceptions should be used whenever useful, as they allow other code to catch and handle specific problems without falsely handling unrelated issues. If a specific exception does not exist, creating one is encouraged, even if it does not carry data.

# Naming Conventions

The section below contains naming conventions for packages, class names, directory paths and public method names that apply to new applications. Do not change existing applications since there will be large impact to the existing software. This means new code in the middle of an existing file must follow new guidelines as much as possible without disrupting the existing code. However, new files within existing application must follow these new naming conventions.

Meaningful names are important for maintaining understandable and high-quality code. Developers should take the time to choose names that are descriptive and unambiguous.

Except for user-defined constants, all identifier names should be written as a mix of upper and lower case letters. This coding standard uses the following rules for formatting identifier names:

Table 4‑1. Identifier Naming Rules

| Identifier Type | Naming Rules | Example |
| --- | --- | --- |
| Variables | Variables 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 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.  No instance variables should be declared as public; getters and setters should be preferred. | int I = 0;  String xmlDoc = 0;  float price = 0; |
| 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, project, machine, or login names.  The prefix for an CA/CST package will be (for example) gov.state.ca1.portal. | gov.state.ca1. Interface.daos  gov.state.ca1.generic.captureapplet.app  gov.state.ca1.generic.interfaces |
| Classes and Enumerations | Class names and enumeration 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). Acronym should be capitalized. | class ReqReqDAO class ResReqHome |
| Constants (user defined) | Capitalize all letters in each word.  Use underscores in place of spaces. | static final int MIN\_WIDTH = 4; |
| Methods | Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized. Variables in methods should be public only at last resort; getters and setters should be preferred. | getCustomerData(); setCustomerData(int data); |
| Interfaces | Interface names should be capitalized like class names. | interface Storing |

## Packages and Files

This section describes the naming conventions for development files created in the Java project. These files will be named according to the following rules:

Table 4‑2. Package and File Naming Rules

|  |  |  |
| --- | --- | --- |
| File type | Naming Rules | Example |
| Source files | The same as the class name in the file. | Transaction.java |
| Compressed package files | Name should not be capitalized like class names. | ca1-dal-jpa-2.0.jar |

The following file extensions will be used for all CA/CST files:

Table 4‑3. File Extensions

|  |  |
| --- | --- |
| File Type | Extension |
| Java source file | .java |
| Java class file | .class |
| Java compressed package file | .jar |
| Java Web Archive file | .war |
| Java Enterprise Application Archive file | .ear |
| Property files in XML format | .xml |
| Resource file | .rar |
| Property file | .properties |

## Constants

* Use keywords enum and static final to declare constants.
* Use constants to avoid hard-coded values in the code.
* Use uppercase letters and replace spaces with underscores to format the name of the constant for user-defined constants.
* Use common sense and provide meaningful constant names (not e, var, io, etc).
* If there are several related values, use a type-safe enumeration or a class implementing some interface to group the constants rather than statics.
* Constants used over multiple files (classes) should be placed in a separate constants file with a meaningful name.
* Constants which may need to be modified over the lifetime of the project should be placed in properties files or configuration files where they can be modified without recompiling the code.

The following is an example of specifying a constant:

static final String APP\_CODE = "RES";

…

application\_code = APP\_CODE;

The following is an example of using type-safe enumeration to specify a group of related constants: (change example)

public enum Direction {

EAST(0),

SOUTH(1),

WEST(2),

NORTH(3);

private int code;

private Direction(int code) {

this.code = code;

}

private int getCode() {

return code;

}

The following is an example of using interface to specify a group of related constants:

public interface Direction {

final int EAST = 0;

final int SOUTH = 1;

final int WEST = 2;

final int NORTH = 3;

}

# Formatting Conventions

The length of Java source lines, including comments, should not exceed 79 characters. This ensures that any monitor or printer is useable.

Use liberal amounts of white space to improve code readability.

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

1. Break after a comma.

2. Break before an operator.

3. Prefer higher-level breaks to lower-level breaks.

4. Align the new line with the beginning of the expression at the same level on the previous line.

5. If the above rules lead to confusing code or to code that is too flushed against the right margin, outdent one additional level.

## Class File Structure

The elements of a class or interface file should appear in the following order:

1. Class (static) variables

2. Instance variables

3. Constructors

4. Methods grouped by functionality

## Comments

A good metric for comments is that comments should account for minimum 20 percent (ideally more) of the total Lines of Code (LOC).

The formatting for class, method, and member comments follow the requirements of the JavaDoc Tool. A documentation comment is made up of two parts -- a description followed by zero or more tags, with a blank line (containing a single asterisk "\*") between these two sections:

/\*\*

\* This is the description part of a doc comment

\*

\* @tag Comment for the tag

\*/

The first line is indented to line up with the code below the comment, and starts with the begin-comment symbol (/\*\*) followed by a return.

Subsequent lines start with an asterisk \*. They are indented an additional space so the asterisks line up. A space separates the asterisk from the descriptive text or tag that follows it.

Insert a blank comment line between the description and the list of tags.

Insert additional blank lines to create "blocks" of related tags.

The last line begins with the end-comment symbol (\*/) indented so the asterisks line up and followed by a return. Note that the end-comment symbol contains only a single asterisk (\*).

A blank line should precede and follow the comment block.

### Comments: Class Header

The following comments will appear before the definition of every class.

/\*\*

\* A description of what the class does.

\*

\* @author Author name

\* @author Contributor 1

\* @author Contributor 2

.

.

\* @author Contributor n

\*/

### Comments: Method Header

The following comments will appear before every method (including main).

/\*\*

\* A description of what the method does

\*

\* @param name description

\* @param name description

.

\* @param name description

\*

\* @return description

\*

\* @exception name description

\* @exception name description

.

.

\* @exception name description

\*

\* @pre precondition before execution

\* @post postcondition after execution

\*/

### Definition of Comments

**@param**

The @param tag is followed by the name (not type) of the parameter, followed by a description of the parameter. Additional spaces should be inserted between the name and description so that comments line up in a block. Dashes or other punctuation should not be inserted before the description. The name always starts with a lowercase letter. The description is most usually a phrase, starting with a lowercase letter and ending without a period, unless it contains a complete sentence.

**@return**

The @return tag is followed by a description of the return value. Whenever possible, detailed information (such as returns -1 when an out-of-bounds argument is supplied) should be provided. Spaces should be used to line the description up with the rest of the comments in the method.

**@exception**

An @exception tag should be included for at least any declared (checked) exceptions. It can also document any non-declared exceptions that can be thrown by the method, (normally those that appear directly in the implementation, rather than those that are indirectly thrown).

**@pre**

JavaDoc does not have a tag for preconditions, so the above tag is used. Preconditions should be expressed in prose or in a formal language if it is more appropriate for the situation.

**@post**

JavaDoc does not have a tag for postconditions, so the above tag is used. Postconditions should be expressed in prose or in a formal language if it is more appropriate.

### Non-Javadoc Comment

Use // for non-Javadoc comments, including multi-line comments.

## Indentation

Four spaces should be used as the unit of indentation. Do not use Tab for indentation.

## White Space

### 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.

### 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 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);

# Statements

Each line should contain at most one statement. Example:

argv++; // Correct

argc--; // Correct

argv++; argc--; // INCORRECT!

## Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces "{ statements }". For example:

while (!done) {

doSomething();

doMoreThing();

}

* 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.

## Return Statement

A return statement with a value should not use parentheses unless it is an expression that requires parentheses or the parentheses provide clarity. Example:

return;

return myDisk.size();

return (size ? size : defaultSize);

## if, if-else, if else-if else conditional operator ? 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) //INCORRECT !!

statement;

Note: the conditional operator “?” may be used to replace an “if” statement if there are only two conditions “?” statements use the Boolean value of one expression to decide which of two other expressions should be evaluated.:

ConditionalExpression:

ConditionalOrExpression

ConditionalOrExpression ? Expression : ConditionalExpression

## 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).

## for-each Statements

A “for-each” statement should have the following form:

for (type var : array or collection) {

statements;

}

## while Statements

A “while” statement should have the following form:

while (condition) {

statements;

}

An empty “while” statement should have the following form:

while (condition);

## do-while Statements

A “do-while” statement should have the following form:

do {

statements;

} while (condition);

## 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 (does not 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.

## 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;

}

## Exceptions

### Throwing An Exception

When throwing an exception, the statement should have the following sample format:

public void getCustomerData(String xmlDoc)

throws RemoteException {

CustomerData customerData = new CustomerData();

try {

customerData.checkXMLDocument(xmlDoc);

} catch (RemoteException ex) {

throw new

RemoteException("XML is not found.”);

}

}

A description of the exception should be included while throwing an exception. The text should be as simple and direct as possible, and end with a period. Uppercase is mandatory in the first word of the sentence, and should be used inside the sentence only when required.

Examples of correct exception text are:

“Capability for checking XML document not implemented yet.”

“Incorrect XML document is provided.”

“XML file not found in given path.”

Examples of incorrect exception text:

“We’re sorry, but we have not implemented the capability to delete entries at the moment.”

“Incorrect password.”

“Certificate file not there.

### Catching An Exception

When catching an exception, the statement should have the following sample format:

try {

OpenXMLDocument(“c:\\CA1\\”);

} catch (java.io.IOException ioe) {

log.error(ioe.getMessage());

}

The catch statement should only print the message supplied by the method throwing the exception.

### Top-Level Exception

Top-level java.lang.Exception can be caught in MDB onMessage() method to log all checked and runtime exceptions.

For example:

@Resource

private MessageDriveContext mdc;

public void onMessage(final Message msg) {

if (logger.isLoggable(Level.FINER)) {

logger.entering(CLASSNAME, "onMessage", msg.toString());

}

try {

// Log the receipt of the message.

logger.fine("Message id = " + msg.getJMSMessageID());

BytesMessage bytesMessage = (BytesMessage) msg;

byte[] body = new byte[(int)bytesMessage.getBodyLength()];

bytesMessage.readBytes(body);

String BiographicHistPayloadMg = new String(body);

// call submit BiographicHistRequest.submit method

BiographicHistRequest myReq = new BiographicHistRequest();

logger.fine("search request = " + BiographicHistPayloadMg);

myReq.submit(BiographicHistPayloadMg);

} catch (InvalidRequestException e) {

logger.log(Level.SEVERE, "Not submitting to CCD "

+ e.toString());

mdc.setRollbackOnly();

throw new IllegalStateException(e);

} catch (Exception unexpectedFailure) {

logger.log(Level.SEVERE, "Unexpected Failure, rolling back the transaction", unexpectedFailure);

mdc.setRollbackOnly();

throw new ESBException(unexpectedFailure);

}

}

### Best Practices for Exception Handling

One of the problems with exception handling is knowing when and how to use it. This section covers some of the best practices for exception handling. Programmers want to write quality code that solves problems. Unfortunately, exceptions come as side effects of code. Examine the following code:

public void consumeAndForgetAllExceptions(){

try {

...some code that throws exceptions

} catch (Exception ex){

ex.printStacktrace();

}

}

What is wrong with the code above?

Once an exception is thrown, normal program execution is suspended and control is transferred to the catch block. The catch block catches the exception and just suppresses it. Execution of the program continues after the catch block, as if nothing had happened.

It took the C++ community several years to decide on how to use exceptions. This is ongoing in the Java community. If not used correctly, exceptions can slow down programs, as it takes memory and CPU power to create, throw, and catch exceptions. If overused, exceptions make the code difficult to read and frustrating for the programmers using the API.

### The Nature of Exceptions

There are three different situations that cause exceptions to be thrown:

* Exceptions due to programming errors: In this category, exceptions are generated due to programming errors (e.g., NullPointerException and IllegalArgumentException). The client code usually cannot do anything about programming errors.
* Exceptions due to client code errors: Client code attempts something not allowed by the API, and thereby violates its contract. The client can take some alternative course of action, if there is useful information provided in the exception. For example: an exception is thrown while parsing an XML document that is not well-formed. The exception contains useful information about the location in the XML document that causes the problem. The client can use this information to take recovery steps.
* Exceptions due to resource failures: Exceptions that get generated when resources fail. For example: the system runs out of memory or a network connection fails. The client's response to resource failures is context-driven. The client can retry the operation after some time or just log the resource failure and bring the application to a halt.

### Types of Exceptions in Java

Java defines two kinds of exceptions:

* Checked exceptions: Exceptions that inherit from the Exception class are checked exceptions. Client code has to handle the checked exceptions thrown by the API, either in a catch clause or by forwarding it outward with the throws clause.
* Unchecked exceptions: RuntimeException also extends from Exception. However, all of the exceptions that inherit from RuntimeException get special treatment. There is no requirement for the client code to deal with them, and hence they are called unchecked exceptions.

By way of example, the following shows the hierarchy for NullPointerException:



Figure 6-1. NullPointerException hierarchy

In this diagram, NullPointerException extends from RuntimeException and hence is an unchecked exception.

A checked exception thrown by a lower layer is a forced contract on the invoking layer to catch or throw it. The checked exception contract between the API and its client soon changes into an unwanted burden if the client code is unable to deal with the exception effectively. Programmers of the client code may start taking shortcuts by suppressing the exception in an empty catch block or just throwing it and, in effect, placing the burden on the client's invoker.

Checked exceptions are also accused of breaking encapsulation. Consider the following:

public List getAllAccounts() throws

FileNotFoundException, SQLException{

...

}

The method getAllAccounts() throws two checked exceptions. The client of this method has to explicitly deal with the implementation-specific exceptions, even if it has no idea what file or database call has failed within getAllAccounts(), or has no business providing filesystem or database logic. Thus, the exception handling forces an inappropriately tight coupling between the method and its callers.

### Best Practices for Designing the API

When deciding on checked exceptions vs. unchecked exceptions, ask yourself, "What action can the client code take when the exception occurs?"

If the client can take some alternate action to recover from the exception, make it a checked exception. If the client cannot do anything useful, then make the exception unchecked. Take steps to recover from the exception and not just logging the exception. To summarize:

Table 6‑1. Exceptions

|  |  |
| --- | --- |
| Client's reaction when exception happens | Exception type |
| Client code cannot do anything | Make it an unchecked exception |
| Client code will take some useful recovery action based on information in exception | Make it a checked exception |

Moreover, prefer unchecked exceptions for all programming errors: unchecked exceptions have the benefit of not forcing the client API to explicitly deal with them. They propagate to where you want to catch them, or they go all the way out and get reported. The Java API has many unchecked exceptions, such as NullPointerException, IllegalArgumentException, and IllegalStateException. Work with standard exceptions provided in Java rather than creating my own. They make my code easy to understand and avoid increasing the memory footprint of code.

### Preserve encapsulation

Never let implementation-specific checked exceptions escalate to the higher layers. For example, do not propagate SQLException from data access code to the business objects layer. Business objects layer do not need to know about SQLException. You have two options:

* Convert SQLException into another checked exception, if the client code is expected to recuperate from the exception.
* Convert SQLException into an unchecked exception, if the client code cannot do anything about it.

Most of the time, client code cannot do anything about SQLExceptions. Do not hesitate to convert them into unchecked exceptions. Consider the following piece of code:

public void dataAccessCode(){

try{

..some code that throws SQLException

}catch(SQLException ex){

ex.printStacktrace();

}

}

This catch block just suppresses the exception and does nothing. The justification is that there is nothing the client could do about an SQLException. It can be handled as follows:

public void dataAccessCode(){

try{

..some code that throws SQLException

}catch(SQLException ex){

throw new RuntimeException(ex);

}

}

This converts SQLException to RuntimeException. If SQLException occurs, the catch clause throws a new RuntimeException. The execution thread is suspended and the exception gets reported. However, the business object layer is not corrupted layer with unnecessary exception handling, especially since it cannot do anything about an SQLException. If the catch needs the root exception cause, make use of the getCause() method available in all exception.

If you are confident that the business layer can take some recovery action when SQLException occurs, you can convert it into a more meaningful checked exception.

### Document exceptions

Use Javadoc's @throws tag to document both checked and unchecked exceptions that the API throws. Have some way by which the client code can learn of the exceptions that your API throws. Here is a sample unit test that tests for IndexOutOfBoundsException:

public void testIndexOutOfBoundsException() {

ArrayList blankList = new ArrayList();

try {

blankList.get(10);

fail("Should raise an IndexOutOfBoundsException");

} catch (IndexOutOfBoundsException success) {}

}

The code above should throw an IndexOutOfBoundsException when blankList.get(10) is invoked. If it does not, the fail ("Should raise an IndexOutOfBoundsException") statement explicitly fails the test. By writing unit tests for exceptions, it not only documents how the exceptions work, but also make the code robust by testing for exceptional scenarios.

### Best Practices for Using Exceptions

The next set of best practices show how the client code should deal with an API that throws checked exceptions.

* Always clean up after yourself

Make sure resources like database connections or network connections are cleaned-up. If the API you are invoking uses only unchecked exceptions, clean up resources after use, with try - finally blocks.

public void dataAccessCode(){

Connection conn = null;

try{

conn = getConnection();

..some code that throws SQLException

}catch(SQLException ex){

ex.printStacktrace();

} finally{

DBUtil.closeConnection(conn);

}

}

class DBUtil{

public static void closeConnection

(Connection conn){

try{

conn.close();

} catch(SQLException ex){

logger.error("Cannot close connection");

throw new RuntimeException(ex);

}

}

}

DBUtil is a utility class that closes the Connection. The important point is the use of finally block, which executes whether or not an exception is caught. In this example, the finally closes the connection and throws a RuntimeException if there is problem with closing the connection.

* Never use exceptions for flow control

Generating stack traces is expensive and the value of a stack trace is in debugging. In a flow-control situation, the stack trace would be ignored, since the client just wants to know how to proceed.

In the code below, a custom exception, MaximumCountReachedException, is used to control the flow.

public void useExceptionsForFlowControl() {

try {

while (true) {

increaseCount();

}

} catch (MaximumCountReachedException ex) {

}

//Continue execution

}

public void increaseCount()

throws MaximumCountReachedException {

if (count >= 5000)

throw new MaximumCountReachedException();

}

The useExceptionsForFlowControl() uses an infinite loop to increase the count until the exception is thrown. This not only makes the code difficult to read, but also makes it slower. Use exception handling only in exceptional situations.

* Do not suppress or ignore exceptions

When a method from an API throws a checked exception, take some counter action. If the checked exception does not make sense, do not hesitate to convert it into an unchecked exception and throw it again, but do not ignore it by catching it with {} and then continue as if nothing had happened.

* Do not catch top-level exceptions

Unchecked exceptions inherit from the RuntimeException class, which in turn inherits from Exception. By catching the Exception class, catch RuntimeException as in the following code:

try{

..

}catch(Exception ex){

}

The code above ignores unchecked exceptions, as well.

* Log exceptions just once

Logging the same exception stack trace more than once can confuse the programmer examining the stack trace about the original source of exception. So just log it once.

### Information Exposure Through an Error Message

Providing information in your error messages, could disclose secrets. The secrets could cover a wide range of valuable data, including personally identifiable information (PII), authentication credentials, and server configuration. Sometimes, the secrets might seem harmless and are that are convenient for admins, such as the full installation path of your software. Ensure that error messages only contain minimal details that are useful to the intended audience, and no one else. The messages need to strike the balance between too cryptic and not cryptic enough. The messages should not necessarily reveal the methods that were used to determine the error. Such detailed information can help an attacker craft another attack that now will pass through the validation filters.

If errors must be tracked in some detail, capture them in log messages - but consider what could occur if the log messages can be viewed by attackers. Avoid recording highly sensitive information such as passwords in any form. Avoid inconsistent messaging that might accidentally tip off an attacker about internal state, such as whether a username is valid or not

# Resources Management

Any resource obtained in the try block should be released in the finally block. It ensures the source code that releases or cleans up the resource will not be bypassed accidentally by a return, continue or break.

For example:

Connection conn = null;

PreparedStatement stmt = null;

ResultSet rset = null;

try {

conn = new LoggingConnection(getConnection());

stmt = conn.createPreparedStatement("select ...");

rset = stmt.execute();

.......

} catch (SQLException sqle) {

......

......

} finally {

//SQLUtil can be found in cst-lang-[version].jar

SQLUtil.closeQuietly(rset);

SQLUtil.closeQuietly(stmt);

SQLUtil.closeQuietly(conn);

}

# Code Level Metrics

Code level metrics (both at method and class levels) serve to keep the characteristics of the code within certain limits in order to enhance the readability/understandability, maintainability, testability, and in certain cases, security and performance of the code. The most commonly known metric is the Cyclomatic complexity or very roughly, the number of decision-making and/or branches in the code. High levels of complexity lead to difficulty understanding and testing code and therefore reduce the maintainability and testability of the code. The less complexity in the Java code, the less room for error, the easier it is to test and to understand and therefore maintain the code. Ensure your Java code meets the following metrics:

Table 8‑1. Code level metrics

| Class/Method Level | Description and How to Calculate | Recommended Threshold |
| --- | --- | --- |
| Class | | |
| Classes with High # of Methods: Such classes are difficult to understand, maintain and test. | Count the number of methods per class and keep them below the recommended threshold. | Less than or equal to 20 |
| Classes Dependent on their Children: This is a violation of OO programming guidelines as no parent class shall be dependent on its children. | Do not create class structures that have circular dependencies or parents that make calls to child methods | ZERO |
| Highly Coupled Classes: CBO: Such classes violate encapsulation and modularity guidelines and reduce potential re-use and component development capabilities since they are coupled with multiple other structures. | Count of number of references to classes outside the current class hierarchy and stay within the recommended threshold. | Less than or equal to 2 |
| Public/Protected Data (PUB\_DATA)  Ratio of Public/Private Data (Threshold =<20%)  Large percentages of Public Data violate encapsulation and modularity guidelines and reduce potential re-use and component development capabilities since they provide “global” data to other classes and methods, making all such units of code dependent on one another; i.e. coupled with multiple other structures. | The ratio of public data (attributes) in a class to the private data in same class | Less than or equal to 20 % |
|  |  |  |
| Method Level | | |
| Cyclomatic Complexity: V(g): can make a piece of code; i.e. a method harder to understand, maintain and test. In addition high levels of Cyclomatic complexity can introduce security risks as hard-to-understand code may perform undesirable actions. | The number of linearly-independent paths through a program module – roughly the Number of all decisions and loops in a method or # of branches – stay within the recommended threshold. | Less than or equal to 12 |
| DESIGN Complexity: iV(g) this complexity measure evaluates the dependence of a method on other methods and as such is a measure of how encapsulation and modularity (independence) guidelines have been followed. | The number of all decisions and loops that contain calls to subordinate modules | Less than or equal to 7 |
| High Depth of Code: creates complexity that may be hard to understand, follow and maintain. | The level of nesting of loops and decisions making statements | Limit to 5 |
| Global Data Complexity : gdV(g) :This measurement shows the dependence of a module of code on the global data present in a system and as such can violate encapsulation and modularity guidelines and reduce potential re-use and component development capabilities since they provide “global” data to other classes and methods, making all such units of code dependent on one another; i.e. coupled with multiple other structures. | The count of number of paths through global data | Less than or equal to 5 |
| Avoid high fan-out: high values of this metric indicate excessive interaction between modules and a coupling between disparate structures. This violates encapsulation and modularity guidelines and reduces potential re-use and component development capabilities since they are coupled with multiple other structures. | The count of the subordinate modules called from a single super-ordinate module | Less than or equal to 7 |
| # of Logical Branches in a method: very similar to Cyclomatic complexity. Useful for understanding high paths are in need of testing in any piece of code. A module with a v(G) of 12 for example, has 22 branches which means the minimum number of paths to cover during testing. | Logical branches: Correlates to threshold for v(G) of 10 - which would equal 19 branches, for a V(g) of 12 it is approximately 22 | Less that or equal to 22 |

# Security Standards

The following Security Standards are derived from the Mitre Common Weakness Enumeration (CWE) dictionary. Not all top 25 errors listed under CWE apply to Java language but we have included them in here as a general reference. The information has been used from this site and edited for brevity and how they apply to CA/CST application development. For more details, please visit http://cwe.mitre.org/top25/#CWE-79.

# Section 508 Compliance

In order to comply with section 508 guidelines, please visit the following websites and ensure adherence to the guidelines presented therein.

<http://www.section508.gov/>

Understanding accessibility:

<http://www-03.ibm.com/able/guidelines/software/accesssoftware.html>

General checklist from IBM:

<http://www-03.ibm.com/able/access_ibm/disability.html>

# Simple Coding Guidelines

The following coding guidelines assist with understandability, reliability, maintainability and testability of the code.

1. Avoid artifacts with more than 10 parameters

2. Avoid catching an exception of type Exception, RuntimeException, or Throwable

3. Avoid Classes implementing more than 5 Interfaces

4. Avoid Classes with a High Depth of Inheritance Tree; keep it under 10

5. Avoid Classes with a High Number Of Children; keep it under 10

6. Avoid Classes with a High Public Data Ratio, keep it under 20%

7. Avoid classes with a very low comment/code ratio, keep it over 20%

8. Avoid Classes with High Coupling Between Objects; keep it less than 3

9. Avoid Classes with High Weighted Methods per Class; keep it less than 10

10. Avoid declaring Final Instance Variables that are not dynamically initialized

11. Avoid declaring Instance Variables without defined access type

12. Avoid declaring Non Final Class Variables with Public or Package access type

13. Avoid declaring Public Instance Variables

14. Avoid declaring throwing an exception and not throwing it

15. Avoid direct Class inheritance from java.lang.Throwable

16. Avoid Dynamic SQL Objects With Queries on more than 4 Tables

17. Avoid empty catch blocks

18. Avoid empty finally blocks

19. Avoid hiding attributes

20. Avoid hiding static methods

21. Avoid High Response for a Class

22. Avoid inheritance down the Package path

23. Avoid instantiating Boolean

24. Avoid instantiations inside loops

25. Avoid large Classes - too many Constructors, keep it less than 10

26. Avoid large Classes - too many Fields, keep it less than 10

27. Avoid large Classes - too many Methods, keep it less than 10

28. Avoid large Methods - too many Lines of Code, keep it less than 50

29. Avoid large number of String concatenation; keep it less than 10

30. Avoid Methods with a very low comment/code ratio; keep it over 20%

31. Avoid missing default in switch statements

32. Avoid return statements in finally blocks

33. Avoid SQL Artifacts with High RAW SQL Complexity; keep it below 10

34. Avoid SQL Artifacts with queries on too many Tables; keep it below 10

35. Avoid SQL Artifacts with SQL statement using Group By clause

36. Avoid String concatenation in loops

37. Avoid Superclass knowing Subclass

38. Avoid the use of InstanceOf inside loops

39. Avoid throwing an exception of type Exception

40. Avoid unreferenced Classes

41. Avoid unreferenced Fields

42. Avoid unreferenced Interfaces

43. Avoid unreferenced Methods

44. Avoid Unused Imports

45. Avoid using anonymous Classes

46. Avoid using deprecated objects

47. Avoid using Dynamic instantiation

48. Avoid using exception handling inside loops

49. Avoid using fields (non Static final) from other Classes

# Declarations

## Number Per Line

Maximum one declaration per line is mandatory. The comment describing each variable should be placed above the declaration.

// Indentation level

int level = 0;

// Size of table

int size = 0;

The following is not allowed:

int level, size;

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

int foo, fooArray[]; // INCORRECT!

## Initialization

Local variables should be initialized where they are declared. The only reason not to initialize a variable where it is declared is if the initial value depends on some computation occurring first.

## Placement

Declarations should be placed only at the beginning of blocks. Do not wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

// beginning of method block

void myMethod() {

int int1 = 0;

// beginning of "if" block

if (condition) {

int int2 = 0;

...

}

}

The one exception to the rule is indexes of “for” loops, which in Java can be declared in the “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; // INCORRECT!

...

}

...

}

## Class, Enumeration, and Interface Declarations

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

* No space between a method name and the parenthesis "(" starting its parameter list, followed by the parenthesis “)”.
* 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 "{".

public class Sample extends Object {

int ivar1 = 0;

int ivar2 = 0;

Sample(int i, int j) {

ivar1 = i;

ivar2 = j;

}

int emptyMethod() {

}

...

}

* Methods are separated by a blank line

## Visibility

Java instance variables and methods have four levels of visibility: public, protected, package, and private. As visibility can have a big impact on maintainability of the software system, it is important to consider the following guidelines:

* Variables should be declared as locally as possible
* Variables and methods should have the least possible visibility
* Public instance variables should be avoided at all costs
* Protected instance variable should be avoided unless the instance variable is declared as final
* The only public methods that a class exposes will be the methods of the interface it implements and methods exposing JavaBean properties.

# Application Logging

Use Logback for logging when developing code. For more information on Logback logging, please check the Logback library at URL: http://Logback.qos.ch/index.html. Logback implements the SLF4J API and is very flexible. It can, via a configuration file, select log events and filtered content may be sent to different destinations. Follow proper error handling and error logging best practices. All exception handling should be logged to the application log and database log as appropriate. All log files should contain identifiable exceptions for operational debugging purposes. Errors should never be silently ignored. Below is a simple example of using Java util logging.

package logging.example2;

import java.util.logging.Logger;

public class HelloWorld {

private static CLAS\_NAME=’ HelloWorld.class.getName()”;

private static Logger logger = Logger.getLogger(CLAS\_NAME);

public static void main(String[] args) {

HelloWorld hello = new HelloWorld("Hello world!");

hello.sayHello();

}

private String theMessage = null;

public HelloWorld(String message) {

theMessage = message;

}

public void sayHello() {

// use the 'least important' type of message, one at

// the 'finest' level

logger.log(Level.FINEST, "Hello logging!");

System.err.println(theMessage);

}

}

Here is another example of using java.util.logging:

protected String[] getUserInfo(String firstname, String lastname)

throws GlobalConstantsServiceException, IdentSecurityException {

if (logger.isLoggable(Level.FINER)) {

logger.entering(CLASS\_NAME, "getUserInfo",

new String[]{firstname, lastname});

}

....

if (logger.isLoggable(Level.FINER)) {

logger.exiting(CLASS\_NAME, "getUserInfo", result);

}

}

## Logging Levels

Table 13‑1. Log levels

|  |  |
| --- | --- |
| Log Level | Usage Description |
| SEVERE | The highest value, intended for extremely important message (e.g. fatal program errors). A problem requiring attention from the system administrator has occurred. |
| WARNING | Intended for warning messages. An action occurred or a condition was  discovered that should be reviewed and may require action before an error occurs. |
| INFO | Informational runtime messages. A report of a normal action or event. This  could be a user operation, such as "login completed" or an automatic  operation such as a log file rotation. |
| CONFIG | Informational messages about configuration settings/setup. A configuration related message or problem. |
| FINE | Used for greater detail, when debugging/diagnosing problems. A trace or debug message used for debugging or performance monitoring. Typically contains detailed event data. |
| FINER | Even greater detail. A fairly detailed trace or debug message. |
| FINEST | The lowest value, greatest detail. A highly detailed trace or debug message. |

# Database JDBC Best Practice

* Use “PreparedStatement” - This is a very popular JDBC practice widely adopted by industry using the JDBC API in Java. “PreparedStatement” provides useful services such as prevention from SQL injection, Precompiled SQL queries and use of bind variables.
* Use “ConnectionPool” – “ConnectionPool” has become standard now days. Several frameworks provide built in “connectionPool” facility like Database Connection Pool in Spring, DBCP and if running in managed environments like J2EE Application Server (e.g. WAS or JBOSS Server) will provide Connection Pool facility. The rationale behind this JDBC best practice is that creating JDBC connections take relatively longer time which can increase overall response time, by caching JDBC connection in pool application can immediately access database.
* Disable auto commit mode - This best practice provides substantial performance gain in JDBC batch updates. It is recommended to run SQL query with auto commit mode disable. Rational behind this is that with auto commit mode disabled you can group SQL Statement in one transaction while in case of auto commit mode every SQL statement runs in its own transaction and committed as soon as it finishes. So always run queries with auto commit mode disabled.
* Use JDBC Batch Update - This JDBC best practice is very popular in industry. JDBC API provides “addBatch()” method to add SQL queries into batch and “executeBatch()” to send batch queries for execution. Rational behind this is that, JDBC batch update potentially reduce number of database roundtrip which result in significant performance gain. So always Use JDBC batch update for insertion and update queries.
* Access “ResultSet” using column name to avoid “invalidColumIndexError” - JDBC API allows to access data returned by SELECT query using “ResultSet”, which can further be accessed using either column name or column index. This JDBC best practice suggest using column name over column index in order to avoid “InvalidColumnIndexException” which comes if index of column is incorrect, most common of them is 0, since “ResultSet” column Index starts from 1, zero is invalid. Also you don't need to change your JDBC access code if order of column changed in SELECT SQL query, which is a major maintenance gain and a robust way to write JDBC code. Some Java programmer may argue that accessing column using index is faster than name, which is true but if you look in terms of maintenance, robustness and readability, It is preferable to access column using name in “ResultSet” iterator.
* Use Bind variables instead of String concatenation – Best practice suggest to use PreparedStatement in Java because of better performance. However performance can only be improved if you use bind variables denoted by “?” or place holders, which allows database to run same query with different parameter. This JDBC best practice will also result in better performance and provide protection against SQL injection.
* Always close Statement, PreparedStatement and Connection - Its common Java coding practice to close any resource in finally block as soon as you are done with that. JDBC Connection and other JDBC classes are costly resource and should be closed in finally block to ensure release of connection even in case of any SQLException. From Java 7 onwards Automatic Resource Management (ARM) Block can be used to close resources automatically.
* Choose suitable JDBC driver for application development - There are 4 types of JDBC driver in Java and it can directly affect the performance of DAO layer. Always use latest JDBC Driver if available and prefer type 4 native JDBC Drivers.
* Use standard SQL statement and avoid using db specific query until necessary - This JDBC best practice ensures writing portable code. Since most of JDBC code is filled up with SQL query, it’s easy to start using Database specific feature which may present in MySQL but not in Oracle etc. By using ANSI SQL or by not using DB specific SQL you ensure minimal change in your DAO layer in case you switch to another database.
* Use correct getXXX() method - This practice suggest using correct getter while getting data from ResultSet to avoid data conversion even though JDBC allows to get any data type using getString()or getObject().

# Code Example

The following example shows how to format a Java source file containing a single public class.

This sample code uses log4J API for logging. Please remember to use Java util API logging when developing your code. For more information on Java util logging, please check the Java 5.0 API specification.

