Java Programming Standards
&
Reference Guide

Version 3.2

Office of Information & Technology
Department of Veterans Affairs

VA | U.S. Department of Veterans Affairs
Office of Information and Technology
Product Development

REVISION HISTORY

DATE	VER.	DESCRIPTION	AUTHOR	CONTRIBUTORS
10-26-15	3.2	Added Logging Standards , updated checkstyle installation instructions and package name rules.	Sid Everhart Vic Pezzolla	JSC
11-14-14	3.1	Added ground rules for enforcement	Vic Pezzolla	JSC
9-26-14	3.0	Document is continually being edited for technical accuracy and compliance to JSC standards.	Raymond Steele OI&T / PD	JSC and several noteworthy Subject Matter Experts (SMEs)
12-1-09	2.0	Document Updated	Michael Huneycutt Sr	
4-7-05	1.2	Document Updated	Sachin Sharma	Mai L Vo Lyn D Teague Rajesh Somannair Katherine Stark Niharika Goyal Ron Ruzbacki
3-4-05	1.0	Document Created	Sachin Sharma	

ABSTRACT

The VA Java Development Community has been establishing standards, capturing industry best practices, and applying the insight of experienced (and seasoned) VA developers to develop this "Java Programming Standards & Reference Guide".

The Java Standards Committee (JSC) team is encouraging the use of CheckStyle (in the Eclipse IDE environment) to quickly scan Java code, to locate Java programming standard errors, find inconsistencies, and generally help build program conformance.

The benefits of writing quality Java code infused with consistent coding and documentation standards is critical to the efforts of the Department of Veterans Affairs (VA).

This document stands for the quality, readability, consistency and maintainability of code development and it applies to all VA Java programmers (including contractors). Adherence to these standards and rules will become a measurement of the quality of their work.

```
NOTE: Good Java programming practices empower new
personnel to quickly grasp the intention and purpose
of the application, understand the style and theme
being used, and construct enhancements that blend in
well.
```

Please read and follow the standards, conventions, suggestions, and general concerns outlined in this document. This document will continue to evolve and is meant to be a helpful programmer's tool.

- All aspects of mature software development are implemented when new code can be written and Quality-Inspected quickly and thoroughly.
- Most software is not maintained indefinitely by the original software developer. It is maintained by a multitude of contributing software engineers that in time will add patches, tighten the code, add functionality, and keep the applications current to the latest standards.
- Feedback in the form of corrections or suggestions for improvement of this document is encouraged in order to remove useless parts and add new parts as the Java language itself evolves.
- The VA JSC is comprised of volunteers that champion these standards to attain the industry proven benefits for VA.
- The developed software must comply with the appropriate standards and conventions established for the programming language found in the Technical Reference Model/Standards Profile (TRMSP).
- The VA requires a 508 compliance certificate along with other documentation to certify software compliance prior to acceptance.

- Comments may be sent to the VA OI&T Java Standards Committee <VAOITJavaStandardsCommittee@va.gov>

> "Any fool can write code that a computer can understand.
>
> Good programmers write code that humans can understand." By Mr. Martin Fowler, On Refactoring: Improving the Design of Existing Code

TABLE OF CONTENTS

Field Code Changed ... [32]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 25
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 25
Unknown
Field Code Changed ... [33]
Unknown
Field Code Changed ... [34]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 27
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 27
Unknown
Field Code Changed ... [35]
Unknown
Field Code Changed ... [36]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 27
Unknown
Field Code Changed ... [37]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 27
Unknown
Field Code Changed ... [38]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 28
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 28
Unknown
Field Code Changed ... [39]
Unknown
Field Code Changed ... [40]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 28
Unknown
Field Code Changed ... [41]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 29
Unknown
Field Code Changed ... [42]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 29
Unknown
Field Code Changed ... [43]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 29
Unknown
Field Code Changed ... [44]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 30
Unknown
Field Code Changed ... [45]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 31
Unknown
Field Code Changed ... [46]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 31
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 31
Unknown
Field Code Changed ... [47]
Unknown
Field Code Changed ... [48]
Department of Vete..., 3/29/2016 8:26 AM
Unknown
Field Code Changed ... [49]
Department of Vete..., 3/29/2016 8:26 AM

FIGURES & TABLES

Field Code Changed ... [65]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 39
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 40
Unknown
Field Code Changed ... [66]
Unknown
Field Code Changed ... [67]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 42
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 43
Unknown
Field Code Changed ... [68]
Unknown
Field Code Changed ... [69]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 44
Unknown
Field Code Changed ... [70]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 44
Unknown
Field Code Changed ... [71]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 46
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 46
Unknown
Field Code Changed ... [72]
Unknown
Field Code Changed ... [73]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 46
Unknown
Field Code Changed ... [74]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 47
Unknown
Field Code Changed ... [75]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 48
Unknown
Field Code Changed ... [76]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 50
Unknown
Field Code Changed ... [77]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 58
Unknown
Field Code Changed ... [78]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 60
Unknown
Field Code Changed ... [79]
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 61
Department of Vete..., 3/29/2016 8:26 AM
Deleted: 61
Unknown
Field Code Changed ... [80]
Unknown
Field Code Changed ... [81]
Department of Vete..., 3/29/2016 8:26 AM
Unknown
Field Code Changed ... [82]
Department of Vete..., 3/29/2016 8:26 AM

1 OVERVIEW

1.1 Introduction

This Java Programming Standards and Reference Guide has been compiled to provide current and future Java programmers with a set of coding conventions and standards to be followed when developing applications using the Java Programming Language.

This document reflects best practices for Java programming standards and coding conventions to follow, that by themselves do nothing. When combined with the following disciplined practices it will produce well-formed, functional, readable, and maintainable VA Java applications.

- Use proper design and analysis techniques.
- Participate in individual and group code reviews.
- Build upon test-driven development efforts.
- Use continuous integration and coordinated implementation practices.
- Test locally before deployment globally.
- Use known Good (Best) Business Practices and VA Lessons Learned.
- Look for Java programming standards to evolve and move with it.

Take a look at the big picture and see how your contributions and thoughtful application of these standards helps not only yourself, but the VA as a whole:

- Strongly improves development and inspection communications by offering a common reference point.
- Reduces the learning curve for new developers tasked with enhancements.
- Minimizes common coding issues / mistakes often made.
- Discourages improper coding artifacts that lead to defects and failures.
- Encourages code reuse thereby enhancing efficiency.
- Passes on quality Java programming applications and functionality onto the VA Enterprise Network and beyond.
- Bridges gaps between other departments and in commercial applications.
- Sets standards for present and future VA Java Programming development.

1.2 Standards and Conventions Derivation

The key to large-scale Programmer's adoption of these VA Java programming conventions and standards, is to make them easy to deploy and relevant to those who must learn and use them.

JSC draws from the best of the industry-standard Java programming practices published by Oracle (formerly Sun). Adherence to proper coding conventions / standards can lead to higher quality software artifacts that are easier to support and maintain.

This document is broken down into four broad categories of standards:

STANDARD	DESCRIPTION
NAMING (CONVENTION)	The very first aspect of development that requires convention is how development artifacts are named. Consistent and proper naming of packages, types, variables and other code artifacts will help to insure that any skilled Java Programmer can pick up a piece of code and understand its function.
DOCUMENTATION	All languages provide the ability to create (embed) inline comments that may be viewed while looking through the code. Java has the ability to formally document code (in line) allowing it to be extracted and presented in the form of a browsable reference guide.
STYLE	The programming style for most programmers is a matter of preference. However, there are certain coding / programming conventions that should be standardized to insure proper readability and to produce software artifacts that are easier to maintain.
STATEMENT DESIGN **CLASS DESIGN** **POTENTIAL CODING ISSUES** **IMPORTS**	Though detailed design is outside of the scope of this document, there are some programming standards and conventions related to your design that when outlined, will help avoid defects. An example is the use of Boolean. TRUE, Boolean. FALSE as opposed to new Boolean (true), new Boolean (false).

1.3 The Benefits of Using a Consistent Style

The importance and ultimate benefits of programming using a consistent style are: improved **readability** and greater **maintainability**.

A consistent programming style also facilitates the sharing of code among programmers, especially in dedicated teams of programmers all working on the same project.

A consistent programming style supports the use of (and development of) automated programming tools that can greatly speed up program development. These types of tools can automatically setup a structured format and frequently generate “pretty-print” source code for easier code analysis.

From a software group engineering process (and development perspective), a strong consistent style makes it easier to conduct code reviews. It is easier to see relationships, commands, loops and integrations. Furthermore, the group VA engineering practice of regular code reviews only reinforces the overwhelming need for all VA programmers (including contractors) to learn (and implement) consistent group styles.

In its truest sense, coding in a consistent style allows programmers to focus on the semantics of the code rather than wasting time trying to conjure up new formats.

```
NOTE: Consistency of coding style is more important
than using a particular style. When a given situation
falls outside of the scope of this document,
experience and informed judgment should be used. The
VA OI&T Java Standards Committee wants to know your
concerns, suggestions, and coding problems.
```

1.4 Intended Audience

This document is primarily intended for Java programmers (developers and maintenance programmers) although it is also of interest to Program Managers, Systems Architects, Software Quality Assurance (SQA), and Software Technical Writers.

We fully realize that set, standardized programming guidelines will always be a compromise and may not equally apply to every situation you encounter. If a guideline rule exception is found and exercised (a special circumstance), it is highly recommended to annotate that exception in the actual source code or the project’s documentation and then seek a waiver from the JSC group. You may reach the VA OI&T Java Standards Committee here: VAOITJavaStandards Committee@va.gov.

```
NOTE: The most important consideration is that agreed
upon "best practices" for programming Java source code
is unilaterally applied in a pre-defined (not
arbitrary) manner (consistency is key).
```

1.5 Terminologies Used in This Guide

Terms and acronyms used throughout this document are defined below.

Table 1 - Acronyms and Definitions

TERM	DEFINITION
IDE	Integrated Development Environment
Integration Test	Exercising a software item, or collection of software items, as a whole. Integration testing is typically concerned with confirming behavior and outputs based on inputs or other stimuli, rather than leveraging knowledge of the implementation. Integration testing is frequently performed on the target hardware platform.
JSC	Java Standards Committee
lowerCamelCase	Naming starts with a lowercase letter and capitalizes the first letter of any subsequent word or acronym in the name
NHD	The National Help Desk. It is located at http://vaww.essremedy.va.gov The National Help Desk line is: 888-596-4357
OI&T	Office of Information & Technology
PD	Product Development
Programming	Covers all Java programming including developer and maintenance programming
Software Artifact	A collection of software Items that comprise a computer program.
Software Item	Any identifiable part of a computer program, comprised of one or more software units.
Software Unit	An indecomposable collection of code. A software item that is not subdivided into other items.
Tools	An application used to assist in the development of code such as an IDE (e.g. Eclipse) or code analysis tool (e.g. Fortify or CheckStyle).
Unit Test	Exercising a software unit in isolation. Unit tests may leverage knowledge of the specific implementation, and

	typically does not need to be performed on the target hardware platform.
UpperCamelCase	Naming starts with an uppercase letter and capitalizes the first letter of any subsequent word or acronym in the name
VA	Department of Veterans Affairs

1.6 Technology

There are a number of tools/technologies that can be used in conjunction with the Java Development Environment to guide Java Software Programmers.

In formulating this document the JSC considered various technologies and have chosen to leverage CheckStyle (an open source technology) for defining and enforcing programming standards within Java Integrated Development Environments and Continuous Integration/Build environments.

CheckStyle will help us all implement greater uniformity of code and style.

Using CheckStyle's code analysis functions (and possibly some other selected tools) programmers will discover inconsistencies and a myriad of "violations" that exist.

Not everything flagged is important or relevant to our intent. Some rules (and violations thereof) are critical to these standards and **must be enforced**.

The CheckStyle configuration file can be loaded into your IDE and used as an aid in standards compliance.

```
Note: Both the installation of, and use of CheckStyle
is covered (see TOC for applicable CheckStyle sections
and examples).
```

1.7 Enforcement

New Projects	New project must adhere to standards and any deviation would require a waiver.
Existing Projects	Adherence to pre-existing local standards takes precedence over the VA Java Programming Standards; however, it's highly recommended that non-conflicting standards be incorporated in the existing code base as time permits. Any new modules would need to be fully compliant with the existing team standard that includes all the non-conflicting VA Java Programming standards. All conflicting standards would need to be team documented and provided to team members (including contractors and SQA) to ensure proper compliance. Note: ○ No waiver is required for existing projects that have conflicting standards. ○ JSC will help projects create a custom checkstyle to reflect these standards

1.8 Acknowledgements

Portions of this document are based on Oracle (formerly Sun) programming styles, conventions, and formats. There are many such Java coding examples included that should already be familiar to most Java developers.

The guidelines presented here were not created in a vacuum. In the process of writing this document, the group has read numerous amounts of existing (and popular) Java code conventions, Java coding articles, visited many Java Internet Blogs, participated in several Java forums and spoken in depth with our most seasoned VA programmers.

We have considered popular coding styles being used in current governmental and commercial practice. All this was in an effort to find common denominators and best practices for the VA. This work yielded Standards and Conventions that can be easily learned and with practice, become second nature.

This document builds upon and borrows heavily from several sources listed in the “References” section at the end of this document. The most heavily used sources are The Java Language Specification [1] and C++ Style Guide [3] (see References).

The language and terminology used here, as well as several suggested naming conventions, are taken directly from The Java Language Specification [1].

1.9 Source File Organization

A Java Source File shall contain 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.

A public class should be the first class or interface declaration in the file.

A Java source file shall contain the following elements, in the following order:

1. Package declaration.
2. Import declarations.
3. Class comment including description, author and version.
4. One or more class/interface declarations. Starting with the Public class or interface declaration.

1.10 Source File Naming

A Java source file name shall use the prefix of the name of the class or interface defined in the source file.

Java Source file names shall use the suffix: .java.

For Example:

```
Java Source File Name LayoutManager.java contains:


public class LayoutManager{
….
}
```

1.11 Package Declaration

A Java source file shall contain a package declaration specifying the namespace to which the class belongs. Omitting the package declaration causes the types to be part of an unnamed package, with implementation-defined semantics.

For Example:

```
package gov.va.sample
```

2 NAMING CONVENTIONS

Naming conventions make programs more understandable by making them easier to read and ensuring consistency.

Naming conventions also provide information about the function of the identifier-

For Example, Whether or not it is a constant, package, or class-which can be helpful in understanding the code.

The following lists the Java naming standards that shall be adhered to when coding Java applications.

2.1 Package Names

A package name shall contain only lower-case letters and digits with no underscore characters.

For Example:

```
java.lang
java.awt.image
dinosaur.theropod.velociraptor
```

A unique package prefix is constructed by using the components of the VA Internet domain name of the host site in reverse order. The top two levels of the package prefix shall be: gov.va.**For Example:**

```
gov.va.security
gov.va.med.pharmacy
```

Currently COM, EDU, GOV, MIL, NET, ORG, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981 are considered valid for a top level package name.

For more information, refer to the documents stored at ftp://rs.internic.net/rfc (rfc920.txt and rfc1032.txt).

Rule:

```
Regex Check: ^[gov.va]+(\.[a-z_][a-z0-9_]*)*$
```

Violation:

```
Error
```

2.2 Type Names

Type names (classes and interfaces) shall use the UpperCamelCase style. UpperCamelCase naming starts with an uppercase letter and capitalizes the first letter of any subsequent word in the name. If an acronym is used then only the first character in the acronym should be capitalized. All other characters in the name are lowercase. Underscore characters are not to be used to separate words.

Rule:

```
Regex Check: ^[A-Z][a-zA-Z0-9]*$
```

Violation:

```
Error
```

Class names shall be nouns or noun phrases. Classes that implement interfaces should be suffixed with Impl to differentiate with interface names.

Interface names depend on the salient purpose of the interface.

If the purpose is primarily to endow an object with a particular capability, then the name shall be an adjective (ending in -able or -ible if possible), that describes the capability (e.g., Searchable, Sortable, Network Accessible**). Otherwise, use nouns or noun phrases.**

For Example:

```
// GOOD type names:
LayoutManager, AWTException,
ArrayIndexOutOfBoundsException
// BAD type names:
ManageLayout                    // verb phrase
awtException                    // first letter lower-case
array_out_of_bounds_exception   // underscores
```

2.3 Member Names

Member variables or non-static fields (reference types, or non-final primitive types) should follow the lowerCamelCase style. The lowerCamelCase style begins with a lowercase letter and capitalizes the first letter of any subsequent word or acronym in the name.

All other characters in member names should be lowercase.

Underscores or other special characters should not be used to separate words.

For Example:

```
boolean isResizable;
char recordDelimiter;
```

Rule:

```
Regex Check: ^[a-z][a-zA-Z0-9]*$
```

Violation:

```
Warning
```

Member names shall be nouns or noun phrases. One-character field names should be avoided except for temporary and looping variables. In these cases, use the following:

- b for a byte
- c for a char
- d for a double
- e for an Exception object
- f for a float
- g for a Graphics object
- i, j, k, m, n for integers
- p, q, r, s for String, StringBuffer, or char[] objects

There may be exceptions to this list in cases where a strong convention exists for the use of a single character variable such as the use of x and y to denote screen coordinates.

The single character variable l ("el") should not be used because it is hard to distinguish it from 1 ("one") on some printers and displays.

2.4 Method Names

Method names should use the lowerCamelCase style.

Start with a lowercase letter and capitalize the first letter of any subsequent word in the name. If an acronym is used then only the first character in the acronym should be capitalized. All other characters in the name are lowercase.

Underscores or other special characters should not be used to separate words.

In Java, constructors are not considered methods; constructors always have the same name as the class.

Rule:

```
Regex Check: ^[a-z][a-zA-Z0-9]*$
```

Violation:

```
Warning
```

Method names shall be imperative verbs or verb phrases.

```
NOTE:   This is identical to the naming convention for
non-constant fields. However, it will be easy to
distinguish the two by their context (verbs or nouns).
```

For Example:

```
// GOOD method names:
showStatus(), drawCircle(), addLayoutComponent()


// BAD method names:
mouseButton() // noun phrase; doesn't describe function
DrawCircle() // starts with upper-case letter
add_layout_component() // underscores


// The function of this method is unclear. Does it start
the
// server running (better: startServer()), or test
whether or not
// it is running (better: isServerRunning())?
serverRunning() // verb phrase, but not imperative
```

2.5 Constant Names

It is important to differentiate class or instance constant variables from regular instance variables. The names of fields being used as constants should be all uppercase characters and individual words should be separated using an underscore.

For Example:

```
static final int HTTP_OK_RESPONSE = 200;

static final String GNUTELLA_CONNECT = "GNUTELLA
CONNECT\n\n";
```

The following are considered constants:

- All static final primitive types. (Remember that all interface fields are inherently static final.)
- All static final object reference types that are never followed by "." (dot).
- All static final arrays that are never followed by "[" (square bracket).

For Example:

```
MIN_VALUE, MAX_BUFFER_SIZE, OPTIONS_FILE_NAME
```

Rule:

```
Regex Check: ^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$
```

Violation:

```
Warning
```

2.6 Parameter Names

Parameter names should be short yet meaningful. The choice of a parameter name should indicate what is being passed.

Parameter names for public and protected methods often become part of the class/interface contract and are published in the Javadoc, therefore meaningful names are extremely important.

Names such as arg0 or single character parameter names should be avoided.

Parameter names should follow the lowerCamelCase style.

For Example:

```
void updatePatientData(String patientKey, String
patientData) {

   ….

}
```

Rule:

```
Regex Check: ^[a-z][a-zA-Z0-9]*$
```

Violation:

```
Warning
```

2.7 Static Variable Names

Static 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.

Static variable name should follow the lowerCamelCase style.

For Example:

```
static String logFilePath = null;
```

Rule:

```
Regex Check: ^[a-z][a-zA-Z0-9]*$
```

Violation:

```
Warning
```

2.8 Specific Naming Conventions

1. The terms *get* or *set* shall be used where an attribute is accessed directly.

   This is the naming convention for accessor methods used by Oracle for the Java core packages (mandatory for Java) and is part of the Bean Pattern.

   A method to get or set a property of the class should be called getProperty() or setProperty() respectively, where Property is the name of the property.

For Example:

```
getHeight(), setHeight()
```

2. The *is* prefix shall be used for Boolean variables and methods.

   This is the naming convention for Boolean methods and variables used by Oracle for the Java core packages.

   When writing Java beans this convention is actually enforced for methods.

   Using the is prefix solves a common problem of choosing bad Boolean names like status or flag. isStatus or isFlag simply does not fit, and the Programmer is forced to choose names that are more meaningful.

For Example:

```
isSet, isVisible, isFinished, isFound, isOpen
```

   There are a few alternatives to the *is* prefix that works better in some situations. These are *has*, *can,* and *should* prefixes.

For Example:

```
Boolean hasLicense();
Boolean canEvaluate();
Boolean shouldAbort = false;
```

3. JFC (Java Swing) variables should be suffixed by the type of the JFC element.

   This convention enhances readability since the name gives the user an immediate clue of the type of the variable and thereby the available resources of the object.

```
widthScale, nameTextField, leftScrollbar, mainPanel,
fileToggle, minLabel, printerDialog
```

4. Negated Boolean variable names should not be used.

   A problem arises when the logical not operator is used and a double negative arises. It is not immediately apparent what !isNotError means.

```
boolean isError; // NOT: isNotError
```

```
boolean isFound; // NOT: isNotFound
```

5. Exception classes should be suffixed with *Exception*.

```
class AccessException
```

Exception classes are really not part of the main design of the program, and adding this suffix allows them to stand out relative to the other classes.

```
NOTE: This standard is followed by Oracle (formerly
Sun) in the basic Java library.
```

3 DOCUMENTATION

Java supports two kinds of comments: documentation and general.

- **General Comments** are comments which are delimited by /*...*/, and //. General comments are meant to aid developers in further understanding code and implementation decisions.
- **Documentation Comments** (known as "doc comments") are comments, which adhere to the requirements of the Javadoc technology to allow documentation to be extracted to HTML files. Documentation comments are meant to describe the specification of the code and its intended use from an implementation-free perspective.

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 nontrivial or unobvious 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.

```
NOTE: In general, avoid any comments that are likely
to get out of date as the code evolves.
```

Comments should not be enclosed in large boxes drawn with asterisks or other characters and should not include special characters such as form-feed and backspace.

General guidelines for comment usage are listed below. These are described separately in the subsequent sections:

- Comments should help a reader understand the purpose of the code. They should guide the reader through the flow of the program, focusing especially on areas that might be confusing or obscure.
- Avoid comments that are obvious from the code, as in this famously bad comment example:

```
i = i + 1; // Add one to i
```

- Remember that misleading comments are worse than no comments at all.
- Avoid putting any information into comments that is likely to become out-of-date.
- Temporary comments that are expected to be changed, or removed later, shall be marked with the special tag “XXX:” so that they can easily be found afterwards.

- Ideally, all temporary comments shall have been removed by the time a program is ready to be shipped.

For Example:

```
// XXX: Change this to call viewOrder() when the bugs
// are fixed
```

```
NOTE: Please refer to listed References [11] and [13]
for further guidance in proper comment placement and
usage.
```

3.1 Beginning Comments

Source files should begin with a comment that describes the class/interface and provides the name(s) of the author(s).

Some source code control systems also provide the ability to leverage tags that are replaced with the last update date and version number.

Goncer, Asli 2/5/2016 3:58 PM
Comment [1]: Because of this last sentence prior to the example, it leads you to believe the example shows this date/version usage. So the example may be a little confusing.

For Example:

```
/*
 * Description
 *  Author
 *  Usage Restrictions
 */
```

Violation:

```
Warning
```

3.2 General Comment Formats

Programs can have four styles of implementation comments:

- BLOCK COMMENTS
- SINGLE-LINE COMMENTS
- TRAILING COMMENTS
- END-OF-LINE COMMENTS

3.2.1 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 or they can 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. This sets it apart from the rest of the code.

For Example:

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

indent(1) is a program that makes code easier to read by inserting or deleting whitespaces. Block comments can start with /*-, which is recognized by *indent(1)* as the beginning of a block comment that should not be reformatted.

For Example:

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

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

3.2.2 Single-Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment cannot be written in a single line, it should follow the block comment format (see Section 3.1.1). A single-line comment should be preceded by a blank line, unless it is the first line following a “{“.

Here is an example of single-line comments in Java code (also see "Documentation Comments" in section 3.4.2):

```
if (condition) {
    /* Initial Comment */
    doSomething();

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

3.2.3 Trailing Comments

Trailing comments are very short comments that appear on the same line as the code they describe.

Trailing comments should be shifted far enough to the right in order to separate them from the statements.

Multiple trailing comments contained in a section of code should be indented to the same tab setting.

Here is a sample of a trailing comment used in good Java code:

For Example

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

3.2.4 End-Of-Line Comments

The // comment delimiter can comment out a complete line or only a partial line.

The // comment delimiter should not 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;
//}
```

3.3 Comments with TODO or FIXME

In addition to general comments, some IDEs allow developers to place TODO and FIXME comments in code to indicate areas where there is additional work to be completed or a known issue needs to be corrected.

These types of comments indicate that the code is not complete.

Released source code shall not contain TODO and FIXME comments.

Rule:

```
Code check – looks for comments in the form:
//TODO: something needs to be done
//FIXME: something needs to be fixed
```

Violation:

```
Error
```

3.4 Javadoc Comments

Javadoc is not just another way of commenting your code. It provides a powerful mechanism for documenting code through Javadoc style comments that can then be extracted as documentation in the form of HTML pages using the Javadoc tool.

Classes, public methods, and important fields shall be commented using Javadoc style comments.

This does not mean that you should forget about normal commenting - normal comments and Javadoc comments can, and should, exist side by side in your program!

For Example:

Classes are commented as:

```
/**
* Car represents cars ... A description of the class
* should be place here. Note that the description begins
* on the second line and that there is a space between
* the asterisk and the text. Next we will add some fields
* indicating who the authors of the class are and
* other useful information. Notice the newline!
*
* @author Sachin Sharma
* @usage restrictions
*/
```

```
public class Car{
```

```
NOTE:  There is no extra newline between the end of
the Javadoc comment and the beginning of the class.
```

Methods may be commented as shown in this example:

```
/**
* A description of what the method does...
*
* @param n  a description of the parameter
* @return   a description of the return value
*/
public int factorial(int n){
```

Some, but not all, fields are commented using Javadoc:

```
/**
* Short description of the variable (one line)
*/
type variable;
```

What should be commented using Javadoc and what should be commented normally?

> *Well, think of it this way. Everything you comment using Javadoc will be seen on the document pages of your classes.*

A person reading this documentation would be most interested in what the class represents, what methods it contains, how to use these methods (what type of arguments are to be given), and what they will return.

Some fields, such as public variables or constants, might also be of general interest.

You should assume that a person only wants to use your class without knowing anything about what it really looks like inside.

This is the information that you should provide, and this can be done using Javadoc comments.

```
NOTE: The things you should not comment using Javadoc
are the things that are of interest to the Programmer
who wants to modify the contents of your class.
```

Normal comments should help the reader of the code understand all its inner details and *secrets*.

3.5 Type Javadoc

It is important that every Type (Class or Interface) be documented outlining the role of the Type and its intended usage.

Types that have a package, protected or public scope, should include a Javadoc comment that describes the Type.

For Example

Refer to the example used in paragraph 3.3 as a good code example.

Rule:

```
Code check – looks for missing Javadoc comments Classes
and

Interfaces which have a scope of package, protected or
public.
```

Violation:

```
Warning
```

3.6 Method Javadoc

A method that is scoped at a package, protected or public level should include a Javadoc comment.

This Javadoc comment should describe the method, outline the parameters, return types, and document the exceptions thrown from the method.

Methods that leverage the *@Override* tag and are not polymorphic in nature do not require a Javadoc comment.

Methods that implement an interface should use the *@see* tag to refer to the documentation in the interface.

For Example:

```
//use of @override tag:
public class Animal {
    /**
    * Returns the animal text sound.
    *
    * @return Animal Sound
    */
    public String show animalSoundText(){
      return "animalSoundText";
    }
}
```

```
public class Cat extends Animal{

    @Override
    public final String animalSoundText(){
        return "myyyaaww";
    }
}
```

Here is the interface that contains the method Javadoc.

For Example: The Description:

```
//use of @see tag:
/**
* A description of what the method does...
*
* @param purchaseAmount – Amount of Purchase
* @return   Fees associated with purchase
*/
public int calculateFees(int purchaseAmount);

Here is a class method that implements the interface
above:
/** {@inheritDoc}
* @see src.PurchaseChargesInterface#calculateFees (int purchaseAmount)
*/
public int calculateFees (int purchaseAmount) {
…
}
```

NOTE: This convention works with Javadoc but has a warning with VACheckStyle.xml that can be removed by adding the {@inheritDoc} annotation (see example above).

Rule:

```
Code check which looks for methods with a scope of
package, protected or public that do not have a Javadoc
```

```
comment or have a Javadoc comment that is missing the
required annotations and does not leverage the @Override
tag or @see tag to refer to an interface or super class
that has the appropriate documentation.
```

Violation:

```
Warning
```

3.7 Variable Javadoc

Variables with a package, protected or public scope should be documented to insure proper understanding and usage.

Though most variables are scoped private (to insure that variables are scoped properly and documented properly) it is important to provide a Javadoc comment for all visible variables.

For Example:

```
/**
* Description of the variable here.
*/
protected boolean isResizable;
```

Rule:

```
Code check which looks for class or interface level
variables
which are of package, protected or public scope, but do
not have
a Javadoc comment associated.
```

Violation:

```
Warning
```

3.8 Style Javadoc

Javadoc comments should be well written and conform to the proper style as outlined in the referenced Javadoc documentation.

The following are general guidelines for producing well-formed and proper Javadoc comments in your code:

- The first sentence of a Javadoc comment should end with proper punctuation (That is a period, question mark, or exclamation mark, by default). Javadoc automatically places the first sentence in the method summary table and index. Without proper punctuation, the Javadoc may be malformed. All items eligible for the ***{@inheritDoc}*** tag are exempt from this requirement.

- Javadoc statements should have a description. This includes both completely empty Javadoc, and Javadoc with only tags such as *@param* and *@return*.
- HTML tags should be completed and well-formed.
- HTML tags should have corresponding end tags.
- The use of package level HTML documentation is not strictly enforced. However, if used, it should follow HTML rules and be well-formed.
- Only tags approved for use in Javadoc should be used when using HTML tags. The list of valid HTML tags can be found in the Javadoc reference documentation.

```
NOTE: These checks were patterned after the checks
made by the DocCheck doclet available from Oracle.

Change DocCheck link to:
http://www.oracle.com/technetwork/java/javase/document
ation/index-137868.html#doccommentcheckingtool
```

Rule:

```
This is a code check that looks for Type, Method, and
Variable Javadoc comments and insures that they are well
formed and follow proper style.
```

Violation:

```
Warning
```

4 STYLE

Coding style deals with issues not related to design or documentation, but related to the style of coding. These standards address readability and maintainability of code based on the developers style of coding.

4.1 Coding Size Limits

It has been proven that code readability and maintenance can be impacted by the style of coding which results in excessively long source files or coding artifacts.

Some areas can result in defects downstream or maintenance problems, others have less of an impact, but need to be addressed.

This section deals with size limits on various aspects of Java programming.

4.2 Maximum Line Length

Long lines may be hard to read in printouts or on the screen. If developers have limited screen space long lines may wrap or be cut-off making the code difficult to understand.

The maximum length of a line shall be 128 characters.

Hint:

Eclipse will always provide a line gutter.

```
Open Preferences, go to General -> Editors -> Text
Editors .  There's a "Show print margin" setting.  That
will set the right gutter.
```

Rule:

```
Limit Character Count 128
```

Violation:

```
Error
```

4.3 Maximum File Length

Source files that become excessively long can indicate a design issue and may be hard to understand or maintain.

Java source files shall contain 2000 or less lines of code.

Long classes require re-factoring into smaller classes.

Rule:

```
Line Count 2000
```

Violation:

```
Error
```

4.4 Maximum Anonymous Inner Class Length

Anonymous inner classes should be used for highly specialized cases. If an anonymous inner class is used, the maximum number of lines of code in the class should be limited to 40.

Because an inner class is enclosed in a public class, very long inner classes can become hard to read and understand.

It may be difficult for the developer to follow the flow of the method where the class is defined.

Rule:

```
Line Count 40
```

Violation:

```
Warning
```

4.5 Maximum Method Length

The body of a method shall be 150 lines or less.

Where a method involves a large number of statements it must be broken up into additional methods called by the primary method.

Long method bodies are hard to understand and maintain.

Method bodies that exceed 150 lines will be flagged as an error requiring refactoring into smaller methods.

Rule:

```
Line Count 150
```

Violation:

```
Error
```

4.6 Maximum Number of Parameters

As an object oriented programming language (OOPL), Java developers have a number of approaches for passing parameters into a constructor or method.

Methods and constructors that involve passing an excessive number of parameters can be hard to read and understand.

The maximum number of constructor or method parameters should be seven.

Methods and constructors exceeding this limit will be flagged with a warning and the developer should make every attempt to re-factor the code to reduce the number of parameters passed.

This may involve introduction of a class that can carry the desired parameters.

Rule:

```
Parameter Count 7
```

Violation:

```
Warning
```

4.7 Whitespace

The use of whitespace within Java code is a matter of personal preference and style. However, some areas can result in code that is difficult to read and maintain.

The following two standards are such areas.

4.8 Operator Wrap

The use of operators can lead to a need to wrap them on a different line.

Defining a consistent mechanism for wrapping operators will help to insure that developers become accustomed to standard style for handling this area and improve overall readability and maintenance.

Operators that are used in a statement that is continued on more than one line should be the first item on the continuing line.

For Example:

```
//INCORRECT
Modifier.isPublic(member.getModifiers()) &&
Modifier.isPublic(clazz.getModifiers());
//CORRECT
Modifier.isPublic(member.getModifiers())
&& Modifier.isPublic(clazz.getModifiers());
```

Rule:

```
Code Check – Check for lines ending with &&, ||, & and
similar
characters
```

Violation:

```
Warning
```

4.9 Tab Character

Tab characters shall not be used in Java code.

The use of the tab character ('\t') in Java programming can have a hidden impact on the readability of the code. The use of the tab character may require developers to configure tab widths in their editor to properly view and edit code.

Additionally tabs can affect the source code control system and the ability to email code.

```
Hint: In Eclipse you can replace tabs with spaces by
selecting:

Preferences->General->Editors->Text Editors Insert
spaces for tabs
```

Rule:

```
Code Check – Existence of Character '\t'
```

Violation:

```
Error
```

4.10 Modifier Order

The order in which code appears based on the modifiers is a matter of style; however, consistently ordering code based on visibility helps improve readability and maintainability. Code should be structured such that items with public visibility (most relevant to the consumer) appear first in the source code.

The following order should be followed for best results:

1. public
2. protected
3. private
4. abstract
5. static
6. final
7. transient
8. volatile
9. synchronized
10. native
11. strictfp

Rule:

```
Code Check – the sequence of code based on scope is
evaluated to ensure that it follows the outlined order
with public scoped items appearing first.
```

Violation:

```
Warning
```

5 DESIGN

5.1 Simple Statements

A line of code shall contain at most one statement.

For Example:

```
a = b + c; count++;// WRONG
a = b + c; // RIGHT
count++; // RIGHT
```

Exceptions to this rule are compound looping statements.

5.2 Compound Statements

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

Study the following sections for appropriate compound statement 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 should be 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.

For Example

```
for (int i=0; i<aVariable; i++) {
    doSomething(i);
}
```

5.3 Return Statements

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

For Example:

```
//Correct
return myDisk.size();
return (size ? size : defaultSize);
//Incorrect
```

```
return (myDisk.size());
```

5.4 if, if-else, if else-if else Statements

The if-else class of statements should be structured in 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;
```

5.5 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, please 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 Example:

```
//Incorrect
for (int i=1, j=1, k=1, m=1; condition; i++, j++, k++,
m++) {
    …
}

//Correct
j=1;
k=1;
m=1;
for (i=1; condition; i++) {
    …
    j++;
    k++;
    m++;
}
```

5.6 While Statements

A **while** statement should have the following form:

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

An empty while statement should have the following form:

```
while (condition);
```

5.7 Do-while Statements

A do-while statement should have the following form:

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

5.8 Switch Statements

A switch statement should always include a default statement.

Switch statements without a default cause maintenance issues.

A switch statement without a default does not convey the original intent and may result in a defect in the code.A missed default may result in a code path not intended by the developer.

Rule:

```
Code Check – Look for switch statements where no default
has been

provided.
```

Violation:

```
Warning
```

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;
}
```

For case statements that are intended to fall 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.

A 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.

5.9 Try-catch Statements

A **try-catch** statement should have the following form:

```
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;
}
```

6 CLASS DESIGN

6.1 Design for Extension

Java is an object oriented language. Java classes should be designed with extension usage in mind.

Java classes that contain only non-private and non-static methods shall use an *abstract*, *final*, or *have an implementation* modifier.

A class that is not designed for an extension will be flagged with a warning to allow the developer to reconsider the programming approach.

Rule:

```
Code Check – Class has non-private or non-static methods
and does

not have an implementation or is not declared abstract or
final
```

Violation:

```
Warning
```

6.2 Final classes

A class that does not have a public constructor shall be marked with a *final* modifier.

This indicates that it is not intended to be extended and helps insure a proper understanding of the extensibility of the class.

Classes like this that do not declare the class final, will be marked with an error to help the developer re-evaluate whether the constructor needs to be marked protected and therefore extensible or final indicating that no specialization is expected.

For Example:

```
Public final class xyz{}
```

Rule:

```
Code Check – Class has only private constructors and is
not marked final
```

Violation:

```
Error
```

6.3 Utility classes

If a class has only public methods, it may be considered a utility class.

The constructors of a utility class shall be marked private or protected to ensure that it is not improperly used.

An error will be flagged where a utility class has exposed the constructor as public and yet all methods are marked static.

Rule:

```
Code Check – Class with only static methods and does not
have constructors marked as private or protected.
```

Violation:

```
Error
```

6.4 Coding Metrics - Number of Conditions

The maximum number of Boolean conditions in a given expression should be 5.

Expressions which have a large number of Boolean conditions chain together by &&, ||, &, | or ^ can result in code that is hard to read and maintain.

If it is necessary to have more than 5 Boolean conditions, then it is recommended that the code should be broken up to make it more readable and easier to maintain.

Coding Metrics - A Sample Violation:

```
if ((a == b || b == c) &&c==d && (d == e || f == e) &&
x==y) {
```

Remediation:

```
if ((a == b || b == c) {
  If (c==d && (d != e && f != e) && x==y) {
```

Rule:

```
Limit on Conditions in a single Expression 5
```

Violation:

```
Warning
```

6.5 Class Fan Out Complexity

A class should be dependent on 20 or fewer classes.

A factor of code complexity and overall maintenance complexity is the number of classes that a given class depends on. A class that has a dependency on more than 20 other classes indicates a level of effort in maintenance four or more times that of a class that has a dependency on 20 or fewer classes.

As the number of dependencies increases, the required level of effort to maintain this code will increase by the number of dependencies - squared.

Rule:

```
Limit Class Dependency 20 Classes
```

Violation:

```
Warning
```

6.6 Cyclomatic Complexity

Cyclomatic or conditional complexity provides a metric that helps identify the number of linearly independent paths that can be taken through a given class.

Higher numbers indicate greater code complexity and can indicate code that is difficult to read and maintain.

Conditional complexity is difficult to avoid, but the overall cyclomatic complexity of a class should be 10 or less.

Rule:

```
Limit Cyclomatic Complexity 10
```

Violation:

```
Warning
```

6.7 Duplicate Code

Sequences of Java code should not be duplicated.

Duplication of code, unless through code generation, can lead to code that is difficult to maintain.

Even in the case of code generators, maintenance can become burdensome if the code needs to be maintained outside of the context of the code generator.

This is largely due to the independence of code segments. A required change may need to be applied to all copies of the code with the possibility of missing one. This would cause a deviation and over time could lead to significant differences.

Some duplication of code cannot be avoided, but it should not be the norm.

Avoid duplication of code by properly designing classes to allow for re-use or by leveraging utility classes.

Rule:

```
Code Check - Search for sequences of Java code which
differ only in indentation.
```

Violation:

```
Warning
```

7 POTENTIAL CODING ISSUES

7.1 Empty Statements/Empty Blocks

Java code should not contain an empty statement. Though empty statements and empty blocks may have their place, they generally make the code difficult to read and understand.

The following example of an empty statement is legal and very concise:

```
for (; in.available() != 0; sbuf.append(in.readline());
```

However the following is much easier to read

```
while (in.available() != 0) {
    sbuf.append(in.readline());
}
```

Empty statements generally indicate a misplaced semicolon and can point to a potential defect.

For Example:

```
for (int i=0; i<; i++); <- Problem
Var[i]=”Some value”;
```

The above example points to two statements that should be linked, but are not because of the misplaced semicolon.

Empty blocks like empty statements may be valid, but generally point to potential problems.

One example of an empty block most commonly seen is the empty catch block.

Though there are valid cases when this is used such as ignoring exceptions in special cases, it does not indicate to other developers if this was intentional or a coding mistake.

Java code should not contain an empty block.

Empty blocks do not promote code readability and may create maintenance problems as other developers try to understand the intention of the empty block.

Empty Block Example:

```
for (int i=0; i<x; i++) {} <- Empty
try {
….do something…
} catch (Exception e) {}  <- Empty
```

Rule:

```
Code Check – looks for cases where there are statements
with no action or blocks that have no action.
```

Violation:

```
Warning
```

7.2 Equals and HashCode

A common coding problem is overriding the implementation of equals() but failing to also override the used hashCode().

The contract of the equals() method depends on the hashCode() and therefore when overriding one the other should be overridden to provide clarity of intent in the code and to enforce the contract.

Failure to properly implement equals() and hashCode() can lead to problems in other areas of the code which depend on these two methods.

For Example:

```
//Incorrect
public class CustomerID {
  private long crmID;
  private int nameSpace;

  public CustomerID(long crmID, int nameSpace) {
    super();
    this.crmID = crmID;
    this.nameSpace = nameSpace;
  }

  public boolean equals(Object obj) {
    //null instanceof Object will always return false
    if (!(obj instanceof CustomerID)) {
      return false;
    }
    if (obj == this) {
      return true;
    }
    return  this.crmID == ((CustomerID) obj).crmID &&
            this.nameSpace == ((CustomerID)
obj).nameSpace;
  }

  public static void main(String[] args) {
    Map m = new HashMap();
    m.put(new CustomerID(2345891234L,0),"Jeff Smith");
    System.out.println(m.get(new
CustomerID(2345891234L,0)));
```

```
  }

}
```

If you compiled and ran the above code, the output result is

```
null
```

What is wrong with this?

- The two instances of CustomerID are logically equal according to the class's *equals* method.
- Because the *hashCode()* method is not overridden, the identities for these two instances are not in common to the default *hashCode* implementation.
- Therefore, the *Object.hashCode* returns two seemingly random numbers instead of two equal numbers. Such behavior violates "Equal objects must have equal hash codes" rule defined in the hashCode contract.

```
//Correct:
```

Let us provide a simple *hashCode()* method to fix this problem:

```
public class CustomerID {
  private long crmID;
  private int nameSpace;

  public CustomerID(long crmID, int nameSpace) {
    super();
    this.crmID = crmID;
    this.nameSpace = nameSpace;
  }

  public boolean equals(Object obj) {
    //null instanceof Object will always return false
    if (!(obj instanceof CustomerID)) {
      return false;

   }
    if (obj == this) {
      return true;

   }
    return  this.crmID == ((CustomerID) obj).crmID &&
            this.nameSpace == ((CustomerID)
obj).nameSpace;
  }

  public int hashCode() {
    int result = 0;
    result = (int)(crmID/12) + nameSpace;
    return result;
  }
```

```
  public static void main(String[] args) {
    Map m = new HashMap();
    m.put(new CustomerID(2345891234L,0),"Jeff Smith");
    System.out.println(m.get(new
CustomerID(2345891234L,0)));
  }

}
```

When you compile and run the above code, the output result is

```
Jeff Smith
```

Rule:

```
Code Check – Looks for override of the equals() method
then checks that the hashCode() method has also be
overridden.
```

Violation:

```
Warning
```

7.3 Inner Assignment

Inner assignments should not be used.

The use of inner assignments though elegant and concise create both code readability problems and may present problems in the debugger.

For Example:

```
//Incorrect
string sPos = Integer.toString(pos = recordLocation + 1);
//Correct
pos = recordLocation + 1;
string sPos = Integer.toString(pos);
```

Though the incorrect example is perfectly legal, it may be difficult to read and within the debugger the value of what is passed in to toString() may not be visible.

Avoid such inner assignments with the exception of “for” loops where they are legal.

Rule:

```
Code Check – Looks for assignments that have inner
assignments
embedded.
```

Violation:

```
Warning
```

Magic Number

A numeric literal should only be used in an assignment to a constant.

The use of numeric literals such as 0, 1, -1, 9999, etc. that are not defined as constants is considered a bad coding practice.

The use of magic numbers can lead to code that is harder to maintain as the developers have to search for occurrences of a magic number when trying to debug or modify code.

For Example:

```
//Incorrect
while (i<1000){
    …
}
//Correct
static final int BUILD_TEST_SET_SIZE = 1000;
while (i<BUILD_TEST_SET_SIZE){
    …
}
```

Rule:

```
Code Check – Looks for instances of Integer, Float,
Double and Long where a number is used rather than a
constant.
```

Violation:

```
Warning
```

7.4 Boolean expressions and returns

Complex Boolean expressions should not be used.

The use of complex Boolean expressions and/or return statements that involve complex Boolean expressions can lead to code that is hard to read and maintain.

An example of complex Boolean expressions and return statements include the following:

```
Complex Boolean Expression:
if ((a == true || !b) && (c|| rt.isValid() ||
    !rt.isValidMessage())){
       …
}
Complex Boolean Return:
if (rt.isValid()) {
       return false;
} else {
       return true;
}
```

This could be written like this:

```
    return rt.isValid();
```

Rule:

```
Code Check – Checks code for instance of Boolean
expressions or Boolean return statements with too many
terms.
```

Violation:

```
Warning
```

7.5 Nested Blocks

Nested blocks should not be used.

Nested blocks often confuse the reader.

Most often nested blocks are the result of improper commenting or removal of debugging statements.

An example of a nested block:

```
public void someMethod() {
   string message=”Test Message”;
   {
```

```
      message=”My Message”;
    }
}
//if (logger.isDebugEnabled())
{
      Logger.debug(“This message”);
}
```

Rule:

```
Code Check – Code is searched for freely used blocks.
```

Violation:

```
Warning
```

8 IMPORTS

8.1 Wildcard imports

Import statements should contain fully qualified type names. Wildcard type-import-on-demand declarations (e.g. import java.util.*;) should not be used, unless Java reflection is used. Reasons for this include:

- Someone can later add new unexpected class files to the same package that you are importing. This new class can conflict with a type you are using from another package, thereby turning a previously correct program into an incorrect one without touching the program itself.
- Explicit class imports clearly convey to a reader the exact classes that are being used (and which classes are not being used).
- Explicit class imports provide better compile performance. While type-import-on-demand declarations are convenient for the Programmer and save a little bit of time initially, this time is paid for in increased compile time every time the file is compiled.

Most IDEs have a shortcut to format and organize the import statements (In Eclipse: Ctrl+Shift+O).

```
NOTE: Use this function regularly and certainly before
releasing the code for testing.
```

Rule:

```
Code Check – Looks for imports that leverage the *
notation to

import dependent Types.
```

Violation:

```
Warning
```

8.2 Illegal imports

Import statements shall not contain a name of an Oracle.* package.

Though Java is portable between Java Runtime Environments (JRE), it is possible to implement code that is dependent on a specific JRE, using illegal imports.

For Example the direct import of Oracle.* packages will result in code that is no longer 100% pure Java, therefore limiting the portability between JREs.

Rule:

```
Code Check – Looks for imports of the Oracle.* packages
within the

code.
```

Violation:

```
Error
```

8.3 Unused imports

Java code should not contain unused import statements.

Most IDEs provide a mechanism for automatic addition of imports and removal of unused imports.

It is important to remove unused imports to eliminate artificial binding to Types (classes and interfaces) that are not used within the enclosing class.

Failure to remove unused imports can result in code that does not compile or drives developers to include class libraries that are not actually required.

Rule:

```
Code Check – Looks for instance of imports with the
following characteristics: Evaluates only direct imports,
not wildcard imports such as java.io.*; An import
duplicates another import; The class imported is from the
java.lang package; A class imported is from the same
package; the class imported is not used within the
enclosing class.
```

Violation:

```
Warning
```

9 CHECKSTYLE INSTALLATION

The steps for installing CheckStyle may vary depending on the version of Eclipse or IBM Rational tool you are using.

Prior to installation, please review the CheckStyle home page at the following link: *http://checkstyle.sourceforge.net/index.html* for specific information about your IDE.

For Rational and Eclipse developers the CheckStyle plug-in can be downloaded from the following link: http://eclipse-cs.sf.net/update/.

Installation of CheckStyle involves downloading the Eclipse plug-in and extracting the contents. The process is as follows:

1. Within the Eclipse workbench (Luna), select Help -> Install New Software...

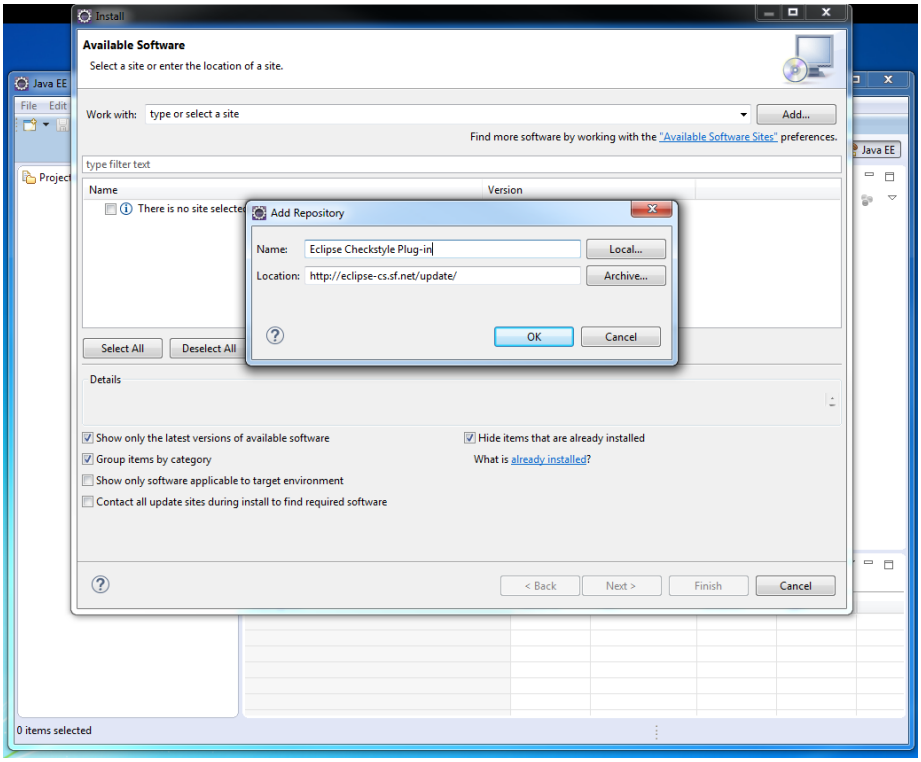

Figure 1 - Eclipse Screen Shot

2. Select Add… and complete the presented dialog (see Figure 1). Note: Unchecked 'Contact all update sites during install to find required software' to remove installation failure.
3. Confirm the selection of features to install (only Checkstyle is needed).
4. Select it and click next. Note: Eclipse will ask about installing unsigned content. Ok past security warning.
5. Agree to the license agreement. This will add CheckStyle to your IDE.

CheckStyle comes with a CheckStyle configuration based on the Oracle (formerly Sun) Java Coding Standards.

To load the VA CheckStyle configuration download a copy of the VACheckStyle.xml file:http://trm.oit.va.gov/files/VACheckStyle.xml.

1. In the IDE, proceed to Preferences -> CheckStyle -> New.
2. Select External Configuration and provide the Name VACheckStyle.
3. Browse to where VACheckStyle.xml was saved and then select it. This will configure the VA CheckStyle.
4. Once it is configured, select it as the default by clicking the Set As Default button.

CheckStyle is now fully configured with the VA CheckStyle configuration based on the standards in this document.

CheckStyle operates in two modes:

- CONTINUAL
- AS NEEDED.

To activate continual checking right click on your project and select CheckStyle -> Activate.

1. To check code periodically as needed right click on your project and select CheckStyle -> Check code with CheckStyle.
2. In both cases, CheckStyle violations are displayed in the Problems View of Eclipse and Rational Tools.

```
NOTE: CheckStyle doesn't allow tabs within source
code.

Please use Eclipse replace tabs with spaces by:

Preferences->General->Editors->Text Editors Insert
spaces for tabs
```

10JAVA PROGRAMMING RULES

The following table is a summary of the coding rules contained in this document. The “Required” column denotes which rules must be adhered to (error) and which rules are guidelines (warning). The “Auto” column refers to automated checking of this rule using CheckStyle.

- lowerCamelCase refers to a naming style that begins with a lower case letter and continues with capitalization of each subsequent word in the name.
- If an acronym is used then only the first character in the acronym should be capitalized. All other characters in the name should be lower case. Underscores or other special characters should not be used to separate words.
- UpperCamelCase is the same as described above except that the name begins with an upper case letter.

CATEGORY	RULE	REQUIRED	AUTO
SOURCE FILE	A Java source file shall contain a single public class or interface	Yes	No
	A public class or interface declaration should be the first class or declaration in the file	No	No
	A Java source file shall contain the following elements, in the following order: 1. Package declaration, 2. Import declarations, 3. Class/interface declarations.	Yes	No
	Java source file names shall use the prefix of the name of the class or interface	Yes	No
	Java source file names shall use the suffix: .java	Yes	No
	A Java source file shall contain a package declaration specifying the namespace to which the class belongs	Yes	No

	A package name shall contain only lower-case letters and digits with no underscore characters	Yes	Yes
CATEGORY	RULE	REQUIRED	AUTO
NAMING CONVENTIONS	A package prefix shall be constructed by using the components of the VA Internet domain name of the host site in reverse order	Yes	No
	The top two levels of the package prefix shall be: gov.va.	Yes	No
	Type names (classes and interfaces) shall use the UpperCamelCase style	Yes	Yes
	Class names shall be nouns or noun phrases	Yes	No
	Member variables shall use the lowerCamelCase style	Yes	Yes
	Member non-static fields (reference types, or non-final primitive types) should use the lowerCamelCase style	No	Yes
	Underscores or other special characters should not be used to separate words in member names	No	Yes
	Member variable names shall be nouns or noun phrases	Yes	No
	Member non-static field names shall be nouns or noun phrases	Yes	No
	The single character variable l (“el”) should not be used	No	No
	Method names should use the lowerCamelCase style	Yes	No
	Method names shall be imperative	Yes	No

	verbs or verb phrases		
	The accessor method to get a property of the class should be called getProperty() where Property is the name of the property	No	No
	The accessor method to set a property of the class should be called setProperty() where Property is the name of the property	No	No
	The accessor method to test a Boolean property of the class should be called isProperty(), where Property is the name of the property	No	No
	Constant variables should use all uppercase characters	No	Yes
	Individual words in a constant variable should be separated using an underscore character	No	Yes
	Parameter names should use the lowerCamelCase style	No	Yes
	Static variable names should use the lowerCamelCase style	No	Yes
	JFC (Java Swing) variables shall be suffixed by the type of the JFC element	No	No
	Negated Boolean variable names should not be used	No	No
	Exception classes should be suffixed with *Exception*	No	No

CATEGORY	RULE	REQUIRED	AUTO
COMMENTS	Comments should not be enclosed in large boxes drawn with asterisks or other characters	No	No
	Comments should not include special characters	No	No
	Source files should begin with a comment that describes the class and provides the name(s) of the author(s)	No	No
	Block comments inside a function or method should be indented to the same level as the code they describe	No	No
	A block comment should be preceded by a blank line	No	No
	A single-line comment should be preceded by a blank line	No	No
	Multiple trailing comments contained in a section of code should be indented to the same tab setting	No	No
	The // comment delimiter should not be used on consecutive multiple lines for text comments	No	No
	Released source code shall not contain TODO and FIXME comments	Yes	Yes
	Classes, public methods, and important fields shall be commented using Javadoc style comments	Yes	Yes
	Types with a scope of package, protected or public should include Javadoc comments	No	Yes

	Methods that are scoped at a package, protected or public level should include a Javadoc comment that describes the method, outlines the parameters and return type and documents the exceptions thrown from the method	No	Yes
	Methods which implement an interface should use the @see tag to refer to the documentation in the interface	No	No
	Variables with a package, protected or public scope should be documented	No	Yes
	The first sentence of a Javadoc comment should end with proper punctuation	No	Yes
	Javadoc statements should have a description	No	Yes
	HTML tags should be completed and well-formed	No	Yes
	HTML tags should have corresponding end tags	No	Yes
	HTML tags used in comments should be valid Javadoc HTML tags	No	Yes

CATEGORY	RULE	REQUIRED	AUTO
CODING	Java source files shall contain less than 2001 lines of code and the maximum length of a line shall be 128 characters	Yes	Yes
	An anonymous inner class should contain less than 41 lines of code	No	Yes
	A body of a method shall contain less than 151 lines of code	Yes	Yes
	The maximum number of parameters that can be passed into a constructor should be 7	No	Yes
	The maximum number of parameters that can be passed into a method should be 7	No	Yes
	Operators that are used in a statement that is continued on more than one line should be the first item on the continuing line	No	Yes
	The tab character ('\t') shall not be used in Java code	Yes	Yes
	Java modifiers should be arranged in source code in the following order: public, protected, private, abstract, static, final, transient, volatile, synchronized, native, strictfp	No	Yes
	A line of code shall contain at most one statement	Yes	No

CATEGORY	RULE	REQUIRED	AUTO
CLASS DESIGN	Braces should be used around all statements, even single statements, when they are part of a control structure	No	No
	A switch statement should always include a default case	No	Yes
	Java classes that contain only non-private and non-static methods shall use an *abstract*, *final*, or *have an implementation* modifier	No	Yes
	A Java class that does not have a public constructor shall use a *final* modifier	Yes	Yes
	The constructors of a utility class shall be marked private or protected	Yes	Yes
	The maximum number of Boolean conditions in a given expression should be 5	No	Yes
	The maximum number of class dependencies for a class should be 20	No	Yes
	The cyclomatic complexity of a class should be less than 11	No	Yes
	Overriding the implementation of the equal() method must be done in conjunction with overriding the implementation of the hashcode() method	No	Yes
	Java code should not contain an empty statement	No	No
	Inner assignments should not be used	No	Yes
	A numeric literal should only be	No	Yes

	used in an assignment to a constant		
	Complex Boolean expressions should not be used	No	Yes
	Import statements should contain fully qualified type names	No	Yes
	Import statements shall not contain a name of an Oracle.* package	Yes	Yes
	Classes and interfaces should only ‘include’ required packages.	No	Yes

11 NAMING CONVENTION REFERENCE

IDENTIFIER TYPE	RULES FOR NAMING	EXAMPLES
Package	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. Subsequent components of the package name vary according to an organization's own internal naming conventions. Compound statements Such conventions might specify that certain directory name components be division, department, project, machine, or login names.	gov gov.va gov.va.vha
Type (Class)	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 Patient; class PatientAllergy;
Interface (Class)	Interface names should be capitalized like class names.	interface PatientDelegate; interface Storing;

IDENTIFIER TYPE	RULES FOR NAMING	EXAMPLES

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

Variable	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 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.	int i; char c; float myWidth;

Constant	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;

12JAVA SECTION 508 COMPLIANCE

The Department of Veterans Affairs is committed to providing accessible electronic data and information technology to disabled Federal employees as well as disabled members of the public seeking information and services from VA. Software developers must design applications in a manner that will support Section 508 compliance.

Designing compliance into the application is far easier than correcting an application after the fact to meet Section 508 Guidelines.

The VA Section 508 site (http://vaww.section508.va.gov/SECTION508/) contains general information about Section 508 and specific documents required for submission to obtain conformance to Section 508.

The site also includes checklists (http://vaww.section508.va.gov/Standards_Checklist_Artifacts.asp) to assist with providing section 508 compliant products to include:

- Software
- Electronic Documents
- Websites
- Web Pages
- Multimedia
- Elearning Courses

These checklists include best practices for providing operating systems and software applications including web apps, applets, plug-ins, and applications required to use them (e.g. Flash, Java apps, media players) that conform to Section 508.

Please visit the VA Section 508 site for the latest information.

13 BEST PRACTICES

13.1 Logging Standards

Application logging should be well thought out and structured so that teams maintaining the software have the information needed to troubleshoot issues. Logging toolsets need to be addressed as well, preferring slf4j (www.slf4j.org) as this allows the concrete logging implementation to be changed at deployment to best suit the target environment. The recommended concrete logging implementation is log4j version 2 (www.logging.apache.org/log4j/2.x/). Developers should consider what data is required for proper troubleshooting and only log PHI or PII data when necessary.

- Dynamically Adjustable Logging Levels
  Logging shall be enabled for applications using Java Platform, Enterprise Edition (Java EE) framework with a built-in support for JMX, such that activation, deactivation, and level, can be controlled without restarting the JVM
- Session Data Capture
  Application logging shall be used to record session information useful for troubleshooting, at `info` level, for applications using Java EE framework. Meaningful attributes, to be recorded in log file, shall be extracted from events such as login, useful VistA RPC calls, message transactions, session termination for any cause, user privileges, CDATA from xml packets, transport layer common services connection, and aggregate data.

  Logged data will reside on application server or database server depending on architecture. Local storage devices and client workstations shall not be used. The developer will work with data center administrators to setup logs placement for secure access control.

14 REFERENCES

[1] Gosling, J., Joy, B., Steele, G., "The Java Language Specification", Addison-Wesley, 1996

[2] "Inner Classes Specification".

[3] Reddy, A., "C++ Style Guide", Oracle Internal Paper

[4] Skinner, G., Shah, S., Shannon, B., "*C Style and Coding Standards*", Oracle Internal Paper,

[5] "Java Beans 1.0 Specification", JavaSoft.

[6] Pike, R., "*Notes on Programming in C*", Bell Labs technical paper.

[7] Cannon, L., Spencer, H., Keppel, D., *et al*, "*Recommend C Style and Coding Standards"*,

[8] Goldsmith, D., Palevich, J., "*Unofficial C++ Style Guide"*, develop, April 1990.

[9] Plocher, J., Byrne, S., Vinoski, S., C++ Programming Style With Rationale", Oracle Internal

[10] ISO Standard 3166, 1981

[11] Baecker, R., Marcus, A., Human Factors and Typography for More Readable Programs,

[12] Kernighan, B., Ritchie, D., *The C Programming Language,* Prentice-Hall, 1978

[13] McConnell, Steven, Code Complete, Chapter 19: Self-Documenting Code

[14] Flanagan, David, Java in a Nutshell, O'Reilly & Associates, 1997, Chapter 5 – Inner Classes and Other New Language Features

14.1 Web Resources

Java Coding Conventions - Oracle Microsystems
Writing Javadoc Comments - Oracle Micrososystems
Java Programming Style Guide - David Wallace Croft
Java Style Guide - Catharina Candolin
Java Programming Style Guide - Java Ranch
Design by Contract – JavaWorld
StringBuffer Example Take Three – WikiWeb
Metrics for netBeans – netBeans
http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-135099.html#367 – Oracle Code Conventions
www.javaprogrammingworld.com/java-coding-conventions.doc - Java Programming World
http://google-styleguide.googlecode.com/svn/trunk/javaguide.html - Google Style Guide

15SO WHAT AM I?

The following job descriptions are **humorous and insightful**.

And yes we do have a sense of humor too!

JOB TITLE	THE LOW DOWN
COMPUTER SCIENTIST	They write code. It may not be the prettiest or most well-factored code, but it gets the job done. It is not about the design of the code or "good" practices; it is about proving what they set out to prove. A computer scientist is as much a mathematician as they are a technologist (they have 31337 math skills), they don't just need to know that stuff works, they have to prove it. Communication and people skills are desirable traits, but not emphasized. Software process and team dynamics skills are desirable traits, but not emphasized. They have a good breadth of general knowledge for their whole field, but they deeply specialize in one or several narrow areas. In these areas they are considered world-class experts. They work on stuff related to their research in their personal time.
PROGRAMMER	Programmers write awesome code and make it clean, well-factored and error free are very important concerns, but not at the expense of getting the job done. It is all about knowing the meaning of "good code" within their domain. They need to have some math skills, but this is not a paramount concern. They need to know of good (best) solutions to problems, but they don't need to prove it is the best solution. A good breadth of algorithmic knowledge is imperative. They have a depth of skill in a wide area of expertise and have reasonably good knowledge of related areas as well. Communication and people skills are desirable traits, but not emphasized. Software process and team dynamics skills are desirable traits, but not emphasized.

	They work on personal software projects they find of interest in their off time.
DEVELOPER	They write code and make it well-factored and clean (super important) but other factors often take priority. Math skills are very much optional, but it does help to be aware of common problems and solutions related to the domain they are in. Communication and people skills are paramount. Process and team dynamics are bread and butter skills. They are consummate generalists without any truly deep specializations. They are expert at finding ways around problems and plugging components together to fulfill a set of requirements. In their personal time they are either trying to build the next Facebook, or engage in activities that have nothing to do with programming, developing, or computer science. • Developers are programmers to a greater or lesser extent. • Computer scientists are programmers to a greater or lesser extent. • Enterprise software is the domain of the developer. • The Googles and Microsofts of the world are after programmers (and to lesser extent computer scientists). The developers who end up there become product managers. • RnD and academia are the domain of the computer scientist (and to a lesser extent the Programmer)
THE THING TO REMEMBER HERE	Any title they call you is derogatory or "bad" in any way. One is not more or less desirable than any of the others. They are simply different dimensions (with some crossover) of the field we are all involved in. Particular personalities will identify more with one but that does not mean that all three can't "bleed" into each other and combine favorably. It is entirely possible to be both an awesome developer and a great Programmer (although it is difficult with so many important things to focus on).

Here are the real definitions as publically seen.
http://en.wikipedia.org/wiki/Computer_scientist
http://en.wikipedia.org/wiki/Computer_programmer
http://en.wikipedia.org/wiki/Software_developer