Prepared by:
V Vijaya Raghava

Conventions for writing Robust Programming

using Java / J2EE Technologies

Agenda

- Purpose
- Getting Started with Conventions
- Conventions – Severity Level
- Different places where the Conventions can be applied
  - Naming Conventions
  - Comments Conventions
  - Coding Conventions
  - Exceptions Conventions
- Books to refer to for further reading
- Q&A

Purpose

- Coding, naming and documentation conventions are important to a company / organization for a number of reasons:
  - When everybody in your company follows these conventions it makes **maintenance** of your code (maybe by somebody else) a lot easier. It's like the same person has developed all of the code.
  - They improve the **readability** of software artifacts
  - They **reduce training management** and effort
  - They leverage organizational commitment towards **standardization**
  - Naming and Coding conventions can also highlight potential bugs or avoid junior **mistakes**.
  - Following documentation conventions will not only result in a correct entered **JavaDoc** output but will also improve readability of your Java code.

Getting Started with Conventions

- In order for to be able to determine if conventions are followed, rules must be defined.
- Each rule will be given a severity level indicating the importance of the rule. The following table gives an overview of the severity levels used in this presentation.

Severity	Impact
Enforced	This priority is the same like 'High' but the related rule can be enforced through the use of an auditing tool.
High	The convention should be followed at all times. No exceptions are allowed! The source code should be changed to conform to the convention. Rules with a 'High' priority cannot (yet) be enforced using an auditing tool.
Normal	The convention should be followed whenever it is possible. Exceptions are allowed but must be documented.
Low	The convention should be followed. Exceptions are allowed and should not be documented.

Different places where Conventions can be applied

- ❑ In this presentation we are going to cover the following conventions for both Java and J2EE Technologies:

Naming Conventions

Comments Conventions

Coding Conventions

Exceptions Conventions

1 Naming Conventions

Naming Rules for various artifacts in Java / J2EE Technologies

Naming Conventions

- The Java and J2EE naming conventions will ensure that every developer in your team will think the same when he or she needs to create a package, class, methods or variables and more.

Java Naming Conventions

J2EE Naming Conventions

Java Naming Conventions

- Match a class name with its file name [High]

WRONG

```
/**
 * Copyright Notice
 * Filename: Hello.java
 */
package learning.com.myprograms;
public class HelloWorld {

}
```

RIGHT

```
/**
 * Copyright Notice
 * Filename: HelloWorld.java
 */
package learning.com.myprograms;
public class HelloWorld {

}
```

Java Naming Conventions *contd…*

- ❑ Group operations with the same name together [Low]

WRONG

```
package learning.com.myprograms;
public class HelloWorld {
    void operation() {
    }
    void function() {
    }
    void operation(int param) {
    }
}
```

RIGHT

```
package learning.com.myprograms;
public class HelloWorld {
    void operation() {
    }
    void operation(int param) {
    }
    void function() {
    }
}
```

Java Naming Conventions *contd…*

- Use A correct name for a class or interface [Enforced]
  - Try to keep your class names simple and descriptive. Use whole words, avoid acronyms and abbreviations.
  - WRONG

    -
      ```
      Public class _HelloWorld{
       }
      ```
  - RIGHT

    -
      ```
      public class ChainOfResponsibility {
      }
      ```

    -
      ```
      public class HelloWorld {
      }
      ```
  - Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Also known as the CamelNotation.

Java Naming Conventions *contd…*

- Use a correct name for a non final field [Enforced]
  - Non-final field names that are not constants should contain only letters, starting with a lower-case letter
  - WRONG

```
int AMOUNT = 100;
```

  - RIGHT

```
int amount = 100;
```

Java Naming Conventions *contd…*

- Use A correct name for a constant [Enforced]
  - Names of constants (final fields that are assigned a constant value) should contain only upper-case letters and underscores. This makes Java code easier to read and maintain by enforcing the standard naming conventions.
  - WRONG

```
public static final int height= 100;
```

  - RIGHT

```
public static final int HEIGHT = 100;
```

Java Naming Conventions *contd…*

- Use A correct name for a method [Enforced]
  - Method names should contain only letters, starting with a lower-case letter
  - WRONG

```
void CALCULATE(){
//...
}
```

  - RIGHT

```
void calculate(){
//...
}
```

Java Naming Conventions *contd…*

- Use a correct name for a package [Enforced]
  - Package names should contain only lower-case letters. Makes it easier to view a directory listing and be able to differentiate packages from classes.
  - Package Name should not start with java or sun.
  - Package names beginning with javax should also be avoided.
  - WRONG

```
package learning.com.java.algorithms._functions;
```

  - RIGHT

```
package learning.com.programs.algorithms.functions;
```

Java Naming Conventions *contd…*

- Name an exception class ending with Exception [High]
  - Names of classes that directly or indirectly inherit from java.lang.Exception or java.lang.RuntimeException should end with Exception.
  - WRONG

```
Class InsufficientFundsEx extends Exception{
}
```

  - RIGHT

```
Class InsufficientFundsException extends Exception{
}
```

Java Naming Conventions *contd…*

- Use a conventional variable name when possible [Normal]
  - One-character local variable or parameter names should be avoided, except for temporary and looping variables, or where a variable holds an undistinguished value of a type.
    - b for a byte
    - c for a char
    - d for a double
    - e for an Exception
    - f for a float
    - i, j or k for an int
    - l for a long
    - o for an Object
    - s for a String
    - in for an InputStream
    - out for an OutputStream
  - WRONG

```
void foo(double d) {
char s;
Object f;
String k;
Object UK;
}
```

  - RIGHT

```
void foo(double d) {
char c;
Object o;
String s;
}
```

Java Naming Conventions *contd…*

- Do not use `$` in a name [Enforced]
  - Do not use the character `$` as part of any identifiers (even though the Java Language Specification allows this).
  - WRONG

```
char character$;
Object studentObj$;
```

  - RIGHT

```
char character;
Object studentObj;
```

Java Naming Conventions *contd…*

- Do not use names that only differ in case [High]
  - Names that only differ in case makes it difficult for the reader to distinguish between.
  - WRONG

```
String anSQLDatabaseConnection;
String anSqlDatabaseConnection;
```

  - RIGHT

```
String anSQLDatabaseConnection1;
String anSQLDatabaseConnection2;
```

Java Naming Conventions *contd…*

- Make A constant private field static final [Normal]
  - private attributes that never get their values changed should be declared final. When you explicitly declare them like this, the source code provides some information to the reader about how the attribute is supposed to be used.
  - WRONG

```
Class AmountTransfer{
   private int dailyLimit = 1000000;
   //...
}
```

  - RIGHT

```
Class AmountTransfer{
   private static final int DAILYLIMIT = 1000000;
   //...
}
```

Java Naming Conventions *contd…*

- Do not declare multiple variables in one statement [High]
  - You should not declare several variables (attributes and local variables) in the same statement. If you define multiple attributes on the same line you won't be able to provide javadoc information for each of the attributes.
  - WRONG

```
Class AmountTransfer{
   private int amount1,amount2,amount3;
   //...
}
```

  - RIGHT

```
Class AmountTransfer{
   private int amount1;
   private int amount2;
   private int amount3;
   //...
}
```

Java Naming Conventions *contd…*

- Use A correct order for class or interface member declarations [Normal]
  - According to Sun coding conventions, the parts of a class or interface declaration should appear in the following order:
    1. Class (static) variables. First the public class variables, then the protected, then package level (no access modifier), and then the private.
    2. Instance variables. First the public class variables, then the protected, then package level (no access modifier), and then the private.
    3. Constructors
    4. Methods

Java Naming Conventions *contd…*

- Use A correct order for class or interface member declarations [Normal]
  - WRONG

```
public class HelloWorld {
public void calculate() {
}
public HelloWorld() {
}
public static void main(String[] args) {
}
private static final double PI = Math.PI;
private static Logger logger = Logger.getLogger
(HelloWorld.class.getName());
}
```

  - RIGHT

```
public class HelloWorld {
private static final double PI = Math.PI;
private static Logger logger = Logger.getLogger
(HelloWorld.class.getName());
public HelloWorld() {
}
public void calculate() {
}
public static void main(String[] args) {
}
}
```

Java Naming Conventions *contd…*

- Use A correct order for modifiers [Enforced]
  - Checks for correct ordering of modifiers:
    - For classes: visibility (public, protected or private), abstract, static, final.
    - For attributes: visibility (public, protected or private), static, final, transient,
    - volatile.
    - For operations: visibility (public, protected or private), abstract, static, final, synchronized, native.

Java Naming Conventions *contd…*

- Use A correct order for modifiers [Enforced] *contd…*
  - WRONG

```
final public class HelloWorld {
        final static public double PI = Math.PI;
        static public int attr2;
        public HelloWorld() {
        }
        static void public main(String[] args) {
        }
}
```

  - RIGHT

```
Public final class HelloWorld {
        public static final double PI = Math.PI;
        public static int attr2;
        public HelloWorld() {
        }
        public static void main(String[] args) {
        }
}
```

Java Naming Conventions *contd…*

- Put the main method last [High]
  - WRONG

```
public final class HelloWorld {
        public static final double PI = Math.PI;
        public HelloWorld() { }
        public static void main(String[] args) { }
        public calculate() {}
}
```

  - RIGHT

```
public final class HelloWorld {
        public static final double PI = Math.PI;
        public HelloWorld() { }
        public calculate() {}
        public static void main(String[] args) {}
}
```

Java Naming Conventions *contd…*

- Put the public class first [High]
  - According to Sun coding conventions, the public class or interface should be the first class or interface in the file.
  - WRONG

```
class Helper{
}
class HelloWorld {
}
public class Animal {
}
```

  - RIGHT

```
public class Animal {
}
class Helper{
}
class HelloWorld {
}
```

J2EE Naming Conventions

- Name a Servlet like [Name]Servlet [High]
  - The name of a Servlet should always end with Servlet, for example when implementing a Front Controller Servlet the name should be FrontControllerServlet.

- Name A Filter Servlet Like [Name]Filter [High]
  - The name of a Filter Servlet should always end with Filter, for example DecodingFilter.

J2EE Naming Conventions

- Name A Data Access Object Like [Name]DAO [High]
  - Implementations of the J2EE pattern Data Access Object must follow the naming convention [Name]DAO.
  - WRONG

```
public class Account {
 //...
}
```

  - RIGHT

```
public class AccountDAO {
 //...
}
```

J2EE Naming Conventions

- Use A Correct Name For An Enterprise Application Display Name [High]
  - The enterprise application display name within the deployment descriptor is the application name, written in mixed cases, with a suffix EAR.
  - **WRONG**

    ```
    <display-name>MyProject</display-name>
    ```

  - **RIGHT**

    ```
    <display-name>MyProjectEAR</display-name>
    ```

J2EE Naming Conventions

- ❑ Use A Correct Name For A Web Module Display Name [High]
  - The Web module display name within the deployment descriptor is the web module name, written in mixed cases, with a suffix WAR.
  - WRONG

```
<display-name>MyProject</display-name>
```

  - RIGHT

```
<display-name>MyProjectWAR</display-name>
```

J2EE Naming Conventions

- JSP Naming conventions
  - All lowercase (myfile.jsp) – *This is recommended one.*
  - Camel case with the first letter lowercase (myFile.jsp)
  - Camel case with the first letter uppercase (MyFile.jsp)

2

Comments Conventions

Rules for Declaring, Defining and Using comments in Java / J2EE Technologies

Comments Conventions

- Comments should be used to provide additional information which 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.

Comments Conventions

- Comments should be used to provide additional information which 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.
- Java / J2EE programs can have two kinds of comments:
  - Implementation comments
  - Documentation comments.

Comments Conventions

- ❑ In this section, we are going to cover the Comments Conventions related to Java / J2EE Technologies.

Java Comments Conventions

J2EE Comments Conventions

Java Comments Conventions

- Do Not Use An Invalid Javadoc Comment Tag [High]
  - This rule verifies code against accidental use of improper Javadoc tags. Replace misspelled tags.

Java Comments Conventions

- ❑ Provide A Correct File Comment For A File [High]
  - According to Sun coding conventions, all source files should begin with a C-style comment that lists the copyright notice and optionally the file name.

```
/**
 * Copyright Notice
 */
```

  - This audit rule verifies whether the file begins with a C-style comment. It may optionally verify whether this comment contains the name of the top-level class the given file contains.

Java Comments Conventions

- Provide A JavaDoc Comment For A Class [Enforced]
  - WRONG

```
public class Account {
  //...
}
```

  - RIGHT

```
/**
 * More information about the class.
 * @author firstName lastName - companyName
 * @version 1.1 - 14/09/2013
 *
 */
public class Account {
  //...
}
```

Java Comments Conventions

- Provide A JavaDoc Comment For A Constructor [Enforced]
  - WRONG

```
public class HelloWorld {
public HelloWorld(String s) { }
}
```

  - RIGHT

```
public class HelloWorld {
/**
 * More information about the constructor here
 * @param s the string to be tested
 */
public HelloWorld(String s) { }
}
```

Java Comments Conventions

- Provide A JavaDoc Comment For A Method [Enforced]
  - WRONG

```
public class HelloWorld {
public boolean isValid() {
 //...
}
}
```

  - RIGHT

```
public class HelloWorld {
/**
 * More information about the method here
 * @return <code>true</code> if the character is valid.
 */
public boolean isValid() {
 //...
}
}
```

Java Comments Conventions

- Provide A JavaDoc comment For A Field [Enforced]
  - WRONG

```
public class HelloWorld {
        public static final double PI = Math.PI;
}
```

  - RIGHT

```
public class HelloWorld {
        /**PI constant value (set to Math.PI)*/
        public static final double PI = Math.PI;
}
```

Java Comments Conventions

- Provide a package.html per package [Low]
  - Each package directory in the source file tree should have a package.html file.
  - This file should provide a brief overview of the package's content.

J2EE Comments Conventions

- Maintain copyright comments.
  - Always it is a best practice to include/update copyright section, whenever a JSP page is getting created / updated respectively.
  - **Way – 1** (Comments will be displayed to client side.)

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

  - **Way – 2** (Comments will be excluded during translation and thus will not be displayed to client side.)

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

J2EE Comments Conventions

- Avoid client side style comments
  - Avoid creating comments using <!-- --!>. These comments will retain during translation and will propagate to the client side (visible to a browser).
  - Sometimes it is required, but most of the times it is a security concern. Hence it must be avoided.
  - WRONG

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

  - RIGHT

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

3

Coding Conventions

Coding Conventions

- ❑ Many of the Java coding conventions can be found from in Java Language Specification from James Gosling, Bill Joy, Guy L. Steele Jr and Gilad Bracha.

Java Coding Conventions

J2EE Coding Conventions

Java Coding Conventions

- Do not reference a static member through an object instance [High]
  - static members should be referenced through class names rather than through objects.
  - WRONG

```
package learning.com.myprograms.staticusage;
public class StaticUsage {
void calculate() {
Employee employee = new Employee();
String grade = employee.GRADE; // don't access through
an object reference
employee.calculateSalary();
}
}
class Employee {
static final String GRADE = "Manager";
static void calculateSalary() {
}
}
```

Java Coding Conventions

- Do not reference a static member through an object instance [High]
  - RIGHT

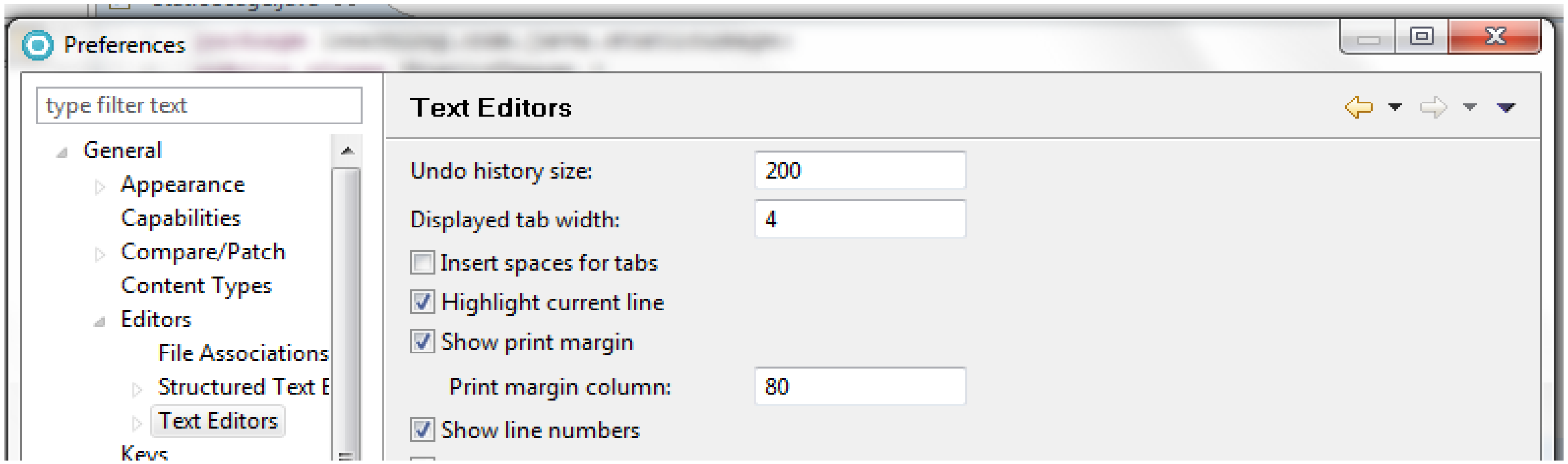
```
package learning.com.myprograms.staticusage;
public class StaticUsage {
void calculate() {
//Employee employee = new Employee();
String grade = Employee.GRADE;
//employee.calculateSalary();
Employee.calculateSalary();
}
}
class Employee {
static final String GRADE = "Manager";
static void calculateSalary() {
}
}
```

Java Coding Conventions

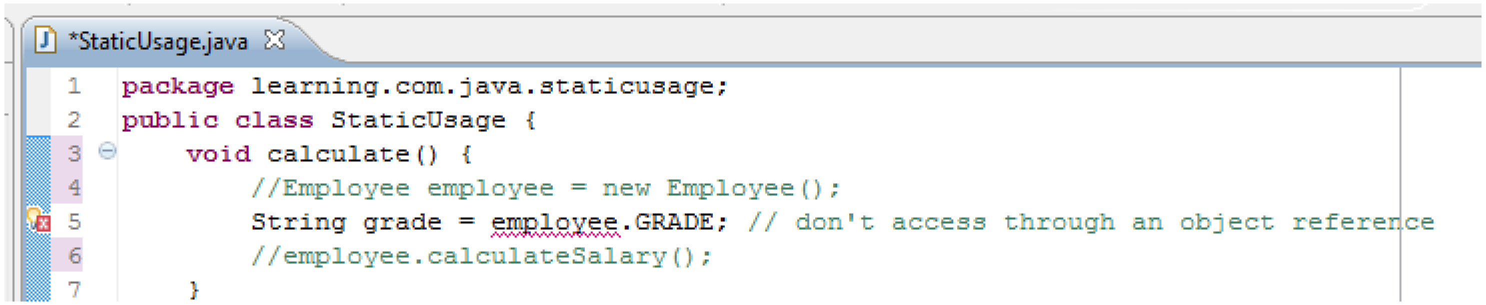
- ❑ Do Not Make A File Longer Than 2000 Lines [Enforced]
  - According to the Sun coding conventions, files longer than 2000 lines are cumbersome and should be avoided.

Java Coding Conventions

- ❑ Do Not Make A Line Longer Than 120 Characters [Normal]
  - According to Sun coding conventions, lines longer than 80 characters should be avoided, since they're not handled well by many terminals and tools.



Undo history size: 200
Displayed tab width: 4
[ ] Insert spaces for tabs
[✓] Highlight current line
[✓] Show print margin
Print margin column: 80
[✓] Show line numbers



```java
package learning.com.java.staticusage;
public class StaticUsage {
    void calculate() {
        //Employee employee = new Employee();
        String grade = employee.GRADE; // don't access through an object reference
        //employee.calculateSalary();
    }
```

Java Coding Conventions

- ❑ Do Not Use Complex Variable Assignment [High]
  - This rule checks for the occurrence of multiple assignments and assignments to variables within the same expression. You should avoid using assignments that are too complex, since they decrease program readability.
  - **WRONG**

```
// compound assignment
i *= j++;
k = j = 10;
l = j += 15;
// nested assignment
i = j++ + 20;
i = (j = 25) + 30;
```

  - **RIGHT**

```
// instead of i *= j++;
j++;
i *= j;
// instead of k = j = 10;
k = 10;
j = 10;
// instead of l = j += 15;
j += 15;
l = j;
// instead of i = j++ + 20;
j++;
i = j + 20;
// instead of i = (j = 25) + 30;
j = 25;
i = j + 30;
```

Java Coding Conventions

- Do not code numerical constants directly [Low]
  - According to Sun Code Conventions, numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a for loop as counter values.
  - Add static final attributes for numeric constants.
  - See also rule : Use a correct name for a constant [Enforced].

Java Coding Conventions

- ❑ Do Not Place Multiple Statements On The Same Line [High]
  - According to Sun coding conventions, each line should contain at most one statement.
  - WRONG

```
if (someCondition) someMethod();
i++; j++;
```

  - RIGHT

```
if (someCondition) {
someMethod();
}
i++;
j++;
```

Java Coding Conventions

- Parenthesize the conditional part of a ternary conditional expression [High]
  - According to Sun coding conventions, if an expression containing a binary operator appears before the ? in the ternary ?: operator, it should be parenthesized.
  - WRONG

    ```
    return x >= 0 ? x : -x;
    ```
  - RIGHT

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

Java Coding Conventions

- Provide an incremental In a for statement [High]
  - This rule checks whether the third argument of a for statement is missing. That is: either provide the incremental part of the for structure, or cast the for statement into a while statement.
  - WRONG

```
for (Enumeration enumeration = getEnum();
enumeration.hasMoreElements();) {
Object o = enumeration.nextElement();
doSomeProcessing(o);
}
```

  - RIGHT

```
Enumeration enumeration = getEnum();
while (enumeration.hasMoreElements()) {
Object o = enumeration.nextElement();
doSomeProcessing(o);
}
```

Java Coding Conventions

❑ Do Not Use A Demand Import [Enforced]

– Demand import-declarations must be replaced by a list of single import-declarations that are actually imported into the compilation unit. In other words, import statements should not end with an asterisk.

– WRONG

```
import java.util.*;
import java.awt.*;
import learning.com.algorithms.*;
```

– RIGHT

```
import java.util.Enumeration;
import java.awt.ActiveEvent;
import learning.com.algorithms.sorting.BubbleSort;
```

Java Coding Conventions

- Provide A default case In A switch Statement [Enforced]
  - According to Sun coding conventions, every switch statement should include a default case.
  - Put an assertion in the default case if you expect the default case never to be called.

Java Coding Conventions

- Use The Abbreviated Assignment Operator When Possible [Normal]
  - Use the abbreviated assignment operator in order to write programs more rapidly. Also some compilers run faster when you do so.
  - WRONG

```
void bar() {
int i = 0;
i = i + 20;
i = 30 * i;
}
```

  - RIGHT

```
void bar() {
int i = 0;
i += 20;
i *= 30;
}
```

Java Coding Conventions

- Do Not Make A Method Longer Then 60 Lines [Normal]
  - A method should not be longer than 1 page (60 lines). If a method is more than one page long, it can probably be split.
  - A method should do only one thing.

Java Coding Conventions

- Do not make a switch statement with more than 256 cases [Normal]
  - Switch statements should not have more than 256 cases.
  - This rule is redundant if you consider rule: Do not make a method longer then 60 lines [Normal]

Java Coding Conventions

- Do Not Chain Multiple Methods [Normal]
  - Do not chain method calls. Exceptionally, a chain of 2 method calls can be used. If the return value from one of the methods is null you would get a NullPointerException
  - WRONG

```
ret = object.method1().method2().method3();
```

  - RIGHT

```
ret1 = object.method1();
ret2 = ret1.mehthod2();
ret3 = ret2.method3();
```

Java Coding Conventions

- Use A Single return Statement [Normal]
  - Try to only use 1 return statement in a method (one point of entry and exit). This makes it easier to read and debug the code.
  - Using a single return also allows for post-conditions to be checked.
  - WRONG

```
public int indexOf(Object something) {
        for (int i = 0; i < arrayList.size(); i++) {
                if (arrayList.get(i).equals(something)) {
                return i;
                }
        }
        return -1;
}
```

Java Coding Conventions

- Use A Single return Statement [Normal]
  - RIGHT

```
public int indexOf(Object something) {
        int index = -1;
        for (int i = 0; (index == -1) && (i <
                arrayList.size()); i++) {
                if (arrayList.get(i).equals(something)) {
                index = i;
                }
        }
        return index;
}
```

Java Coding Conventions

- ❑ Do Not Duplicate An import Declaration [Enforced]
  - There should be at most one import declaration that imports a particular class/package.
  - WRONG

```
package learning.com.myprograms.mypackage.mysubpackage;
mport java.io.IOException;
import java.io.Reader;
import java.sql.Time;
import java.sql.Time;
public class StaticMemberVariable {...}
```

  - RIGHT

```
package learning.com.myprograms.mypackage.mysubpackage;
mport java.io.IOException;
import java.io.Reader;
import java.sql.Time;
public class StaticMemberVariable {...}
```

❑ Do Not Import A Class Of The Package To Which The Source File Belongs [Enforced]

- No classes or interfaces need to be imported from the package that the source code file belongs to. Everything in that package is available without explicit import statements.
- WRONG

```java
package learning.com.algorithms.functions;
import java.util.ArrayList;
import java.util.Calendar;
import learning.com.algorithms.functions.Palindrome;
public class CalculateDate {...}
```

- RIGHT

```java
package learning.com.algorithms.functions;
import java.util.ArrayList;
import java.util.Calendar;
public class CalculateDate {...}
```

Java Coding Conventions

- ❑ Do Not Import A Class From The Package java.lang [Enforced]
  - Explicit import of classes from the package java.lang should not be performed.
  - WRONG

```
package learning.com.algorithms.functions;
import java.lang.String;
public class StringManipulation {
    public void someStringOperation(String s){
    //...
    }
}
```

  - RIGHT

```
package learning.com.algorithms.functions;
public class StringManipulation {
    public void someStringOperation(String s){
    //...
    }
}
```

Java Coding Conventions

- ❑ Do Not Use An Equality Operation With A boolean Literal Argument [Enforced]
  - Avoid performing equality operations on boolean operands. You should not use true and false literals in conditional clauses.
  - By following this rule, you save some byte code instructions in the generated code and improve performance.
  - In most situations, you also improve readability of the program.
  - WRONG

    ```
    boolean isLoaded = true;
    if (isLoaded == true) {
    // ...
    }
    ```
  - RIGHT

    ```
    boolean isLoaded = true;
    if (isLoaded) {
    // ...
    }
    ```

Java Coding Conventions

- Do Not Import A Class Without Using It [Enforced]
  - This rule checks only those classes and interfaces that are explicitly imported by their names. It is not legal
  - to import a class or an interface and never use it. If unused class and interface imports are omitted,
  - the amount of meaningless source code is reduced - thus the amount of code to be understood by a reader is minimized.

Java Coding Conventions

- Do Not Import A Class Without Using It [Enforced]
  - WRONG

```
package learning.com.algorithms.functions;
import java.util.Date;
import java.awt.*;
public class CalculateDate {
    public void someDateOperation(Date s){
    //...
    }
}
```

  - RIGHT

```
package learning.com.algorithms.functions;
import java.util.Date;
public class CalculateDate {
    public void someDateOperation(Date s){
    //...
    }
}
```

Java Coding Conventions

- Do Not Unnecessary Parenthesize A return Statement [Normal]
  - According to Sun coding conventions, a return statement with a value should not use parentheses unless they make the return value more obvious in some way.
  - WRONG
    ```
    return (myDisk.size());
    ```
  - RIGHT
    ```
    return myDisk.size();
    ```

Java Coding Conventions

- Do Not Declare A private Class Member Without Using It [Enforced]
  - An unused class member might indicate a logical flaw in the program. The class declaration has to be reconsidered in order to determine the need of the unused member(s).
  - Examine the program. If the given member is really unnecessary, remove it (or at least comment it out).

Java Coding Conventions

- Do Not Use Unnecessary Modifiers For An Interface Method [Low]
  - All interface methods are implicitly public and abstract. However, the language permits the use of these modifiers with interface methods. Such use is not recommended. Some day, this may be disallowed by Java. It is far less painful to clean up your code before this happens.
  - WRONG

```
interface Employee {
public abstract void calculateSalary();
}
```

  - RIGHT

```
interface Employee {
void calculateSalary();
}
```

Java Coding Conventions

- ❑ Do Not Use Unnecessary Modifiers For An Interface Field [Low]
  - All interface fields are implicitly public, static and final.
  - If the below interface Employee is used to collect all the constant then access the constant variable ***DESIGNATION_M*** through the interface and not by implementing the interface.
  - To avoid this design misuse just place all your constants in a class.
  - WRONG

```
interface Employee {
public static final String DESIGNATION_M = "MANAGER";
public abstract void calculateSalary();
}
```

  - RIGHT

```
interface Employee {
String DESIGNATION_M = "MANAGER";
public abstract void calculateSalary();
}
```

Java Coding Conventions

- ❑ Do Not Use Unnecessary Modifiers For An Interface [High]
  - The modifier `abstract` is considered obsolete and should not be used.
  - WRONG
    ```
    abstract interface Employee {
    }
    ```
  - RIGHT
    ```
    interface Employee {
    }
    ```

Java Coding Conventions

- ❑ Do Not Declare A Local Variable Without Using It [Enforced]
  - Local variables should be used.
  - WRONG

```
public static int sumofDigits(int number) {
    int sum, i, d;
    int total;
    sum = 0;
    while (number > 0) {
    d = number % 10;
    number = number / 10;
    sum = sum + d;
    }
return sum;
}
```

Java Coding Conventions

- Do Not Declare A Local Variable Without Using It [Enforced]
  - RIGHT

```
public static int sumofDigits(int number) {
    int sum, i, d;
    sum = 0;
    while (number > 0) {
    d = number % 10;
    number = number / 10;
    sum = sum + d;
    }
return sum;
}
```

Java Coding Conventions

- Do Not Do An Unnecessary Typecast [High]
  - Checks for the use of type casts that are not necessary, including redundant upcasts.
  - Delete redundant upcasts to improve readability.
  - WRONG

```java
public class ClassA {
}
class ClassB extends ClassA {
    public void someFunction(ClassB b) {
    ClassA a = (ClassA) b; // VIOLATION
    }
}
```

  - RIGHT

```java
public class ClassA {
}
class ClassB extends ClassA {
    public void someFunction(ClassB b) {
    ClassA a = b;
    }
}
```

Java Coding Conventions

- Do Not Do An Unnecessary instanceof Evaluation [High]
  - The **instanceof** expression checks that the runtime type of the left-hand side expression is the same as the one specified on the right-hand side. However, if the static type of the left-hand side is already the same as the right-hand side, then the use of the **instanceof** expression is questionable.
  - WRONG

```
public class ClassA {
    public ClassA() {
    }
    public void someFunction() {
    }
}
class ClassB extends ClassA {
    public void someOtherFunction(ClassB b) {
        if (b instanceof ClassA) { // VIOLATION
        b.someFunction();
        }
    }
}
```

Java Coding Conventions

- Do Not Do An Unnecessary instanceof Evaluation [High]
  - RIGHT

```
public class ClassA {
    public ClassA() {
    }
    public void someFunction() {
    }
}
class ClassB extends ClassA {
    public void someOtherFunction(ClassB b) {
        b.someFunction();
    }
}
```

Java Coding Conventions

- Do Not Hide An Inherited Attribute [High]
  - This rule detects when attributes declared in child classes hide inherited attributes.
  - WRONG

```
class Foo {
int attrib = 10;
}
class Foo1 extends Foo {
int attrib = 0;
// ...
}
```

  - RIGHT

```
class Foo {
int attrib = 10;
}
class Foo1 extends Foo {
int foo1Attrib = 0;
// ...
}
```

❑ Do Not Hide An Inherited static Method [High]

– This rule detects when inherited static operations are hidden by child classes.

– WRONG

```
class Foo {
    static void oper1() {
    }
    static void oper2() {
    }
}
class Foo1 extends Foo {
    void oper1() {
    }
    static void oper2() {
    }
}
```

Java Coding Conventions

- Do Not Hide An Inherited static Method [High]
  - RIGHT

```
class Foo {
    static void oper1() {
    }

    static void oper2() {
    }
}
class Foo1 extends Foo {
    static void anOper2() {
    }

    void anOper1() {
    }
}
```

Java Coding Conventions

- Do Not Declare Overloaded Constructors Or Methods With Different Visibility Modifiers [High]
  - Overload resolution only considers constructors and methods visible at the point of the call. But when all the constructors and methods are considered, there may be more matches.
  - Imagine that FooB is in a different package than FooA. Then the allocation of FooB violates this rule, because the second and third constructor (FooA(char param) and FooA(short param)) are not visible at the point of the allocation because the modifier for both constructors is not public but package local.

Java Coding Conventions

- Do Not Declare Overloaded Constructors Or Methods With Different Visibility Modifiers [High]
  - WRONG

```
public class Animal{
    public FooA(int param) {
    }
    FooA(char param) {
    }
    FooA(short param) {
    }
}
```

  - RIGHT

```
public class Animal{
    public FooA(int param) {
    }
    public FooA(char param) {
    }
    public FooA(short param) {
    }
}
```

Java Coding Conventions

- Do Not Override A Non abstract Method With An abstract Method [High]
  - Checks for the overriding of non-abstract methods by abstract methods in a subclass.
  - If this is just a coincidence of names, then just rename your method. If not, either make the given method abstract in the ancestor or non abstract in the descendant.
  - WRONG

```
public class Animal {
    void func() {
    }
}
abstract class Elephant extends Animal {
    abstract void func();
}
```

Java Coding Conventions

- Do Not Override A Non abstract Method With An abstract Method [High]
  - RIGHT

```
public class Animal {
    void func() {
    }
}
abstract class Elephant extends Animal {
    abstract void extfunc();
}
```

Java Coding Conventions

- Do Not Override A private Method [High]
  - A subclass should not contain a method with the same name and signature as in a super class if these methods are declared to be private.
  - WRONG

```
public class Animal {
    private void func() {
    }
}
class Elephant extends Animal {
    private void func() {
    }
}
```

Java Coding Conventions

- Do Not Override A private Method [High]
  - RIGHT

```
public class Animal {
    private void func() {
    }
}
class Elephant extends Animal {
    private void someOtherfunc() {
    }
}
```

Java Coding Conventions

- Do Not Overload A Super Class Method Without Overriding It [High]
  - A super class method may not be overloaded within a subclass unless all overloaded methods in the super class are also overridden in the subclass.
  - It is very unusual for a subclass to be overloading methods in its super class without also overriding the methods it is overloading. More frequently this happens due to inconsistent changes between the super class and subclass, i.e. the intention of the user is to override the method in the super class, but due to an error, the subclass method ends up overloading the super class method.

Java Coding Conventions

- Do Not Overload A Super Class Method Without Overriding It [High]
  - WRONG

```
class FooB {
    public void bar(int i) {
    }
    public void bar(Object o) {
    }
}
class FooA extends FooB {
    // additional overloaded method
    public void bar(char c) {
    }
    public void bar(Object o) {
    }
}
```

Java Coding Conventions

- Do Not Overload A Super Class Method Without Overriding It [High]
  - RIGHT

```
class FooB {
    public void bar(int i) {
    }
    public void bar(Object o) {
    }
}
class FooA extends FooB {
    // additional overloaded method
    public void bar(char c) {
    }
    public void bar(int i) {
    }
    public void bar(Object o) {
    }
}
```

Java Coding Conventions

- Do Not Use A Non final static Attribute For Initialization [High]
  - Non final static attributes should not be used in initializations of attributes.
  - WRONG

```
public class FooA {
    static int state = 15;
    static int attr1 = state;
    static int attr2 = FooA.state;
    static int attr3 = FooB.state;
}
public class FooB {
        static int state = 25;
}
```

Java Coding Conventions

- Do Not Use A Non final static Attribute For Initialization [High]
  - RIGHT

```
public class FooA {
    static final int INITIAL_STATE = 15;
    static int state = 15;
    static int attr1 = INITIAL_STATE;
    static int attr2 = FooA.state;
    static int attr3 = FooB.state;
}
public class FooB {
    static int state = 25;
}
```

Java Coding Conventions

- Do Not Use Constants With Unnecessary Equal Values [High]
  - This rule catches constants with equal values. The presence of different constants with equal values can cause bugs if these constants have equal meaning.
  - WRONG

```
final static int SUNDAY = 0;
final static int MONDAY = 1;
final static int TUESDAY = 2;
final static int WEDNESDAY = 3;
final static int THURSDAY = 4;
final static int FRIDAY = 5;
final static int SATURDAY = 0;
// This method would never return "Saturday"
String getDayName(int day) {
    if (day == SUNDAY) {
        return "Sunday";
    // Other else if statements
    } else if (day == SATURDAY) {
        return "Saturday";
    }
}
```

Java Coding Conventions

- Provide At Least One Statement In A catch Block [Normal]
  - catch blocks should not be empty. Programmers frequently forget to process negative outcomes of a program and tend to focus more on the positive outcomes. There are situations where the programmer can prove that the exception will never be thrown, but still has to include a catch clause to satisfy the Java language rules.
  - When an empty catch block is specified it usually means that the exception being catched is not expected. In this case you should use an assertion to make sure that the exception is not thrown or throw a java.lang.Error or another runtime exception.

❑ Provide At Least One Statement In A catch Block [Normal]

– WRONG

```
try {
        monitor.wait();
} catch (InterruptedException e) {
        // can never happen
}
```

– RIGHT

```
try {
        monitor.wait();
} catch (InterruptedException e) {
    logger.log(Level.WARNING, "Program Interrupted", e);
    throw new Error("Unexpected exception");
}
```

Java Coding Conventions

- Do Not Give An Attribute A public Or Package Local Modifier [Enforced]
  - Declare the attributes either private or protected and provide operations to access or change them. Also referred to as full encapsulation.
  - WRONG

```
public class ClassA {
int attr1;
public int attr2;
protected int attr3
}
```

Java Coding Conventions

- Do Not Give An Attribute A public Or Package Local Modifier [Enforced]
  - RIGHT

```
public class Foo {
    private int attr1;
    private int attr2;

    int getAttr1() {
        return attr1;
    }

    public int getAttr2() {
        return attr2;
    }

    protected int getAttr3() {
        return attr3;
    }
}
```

Java Coding Conventions

- Provide At Least One Statement In A Statement Body [Enforced]
  - As far as possible, avoid using statements with empty bodies.
  - Provide a statement body or change the logic of the program (for example, use a while statement instead of a for statement).
  - WRONG

```
StringTokenizer st = new
StringTokenizer(class1.getName(), ".", true);
String s;
for (s = ""; st.countTokens() > 2; s = s +
st.nextToken());
```

  - RIGHT

```
StringTokenizer st = new
StringTokenizer(class1.getName(), ".", true);
String s;
while (st.countTokens() > 2) {
s += st.nextToken();
}
```

Java Coding Conventions

- Do Not Compare Floating Point Types [Low]
  - Avoid testing floating point numbers for equality. Floating-point numbers that should be equal are not always equal due to rounding problems.
  - Replace direct comparison with estimation of absolute value of difference.
  - WRONG

```
void bar(double d) {
    if (d != 15.0) {
        for (double f = 0.0; f < d; f += 1.0) {
        // ...
        }
    }
}
```

Java Coding Conventions

- Do Not Compare Floating Point Types [Low]
  - RIGHT

```
void bar(double d) {
        if (Math.abs(d - 15.0) < Double.MIN_VALUE * 2) {
        for (double f = 0.0; d - f > DIFF; f += 1.0) {
                // ...
        }
        }
}
```

Java Coding Conventions

- Enclose A Statement Body In A Loop Or Condition Block [Enforced]
  - The statement of a loop should always be a block. The then and else parts of if-statements should always be blocks.
  - This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.
  - WRONG

```
if (st == null)
        st = "";
while (st.countTokens() > 2)
        s += st.nextToken();
```

  - RIGHT

```
if (st != null) {
st = "";
}
while (st.countTokens() > 2) {
s += st.nextToken();
}
```

Java Coding Conventions

- ❑ Explicitly Initialize A Local Variable [High]
  - Explicitly initialize all method variables. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.
  - WRONG

```
void bar() {
    int var;
    // ...
}
```

  - RIGHT

```
void bar() {
    int var = 0;
    // ...
}
```

Java Coding Conventions

- ❑ Do Not Unnecessary Override The finalize Method [High]
  - As mentioned in Java Language Specification, calling super.finalize() from finalize() is a good programming practice, even if the base class doesn't define the finalize method. This makes class implementations less dependent on each other.
  - However as also mentioned in Effective Java book, finalizers are unpredictable, often dangerous and generally unnecessary and as a rule of thumb finalizers should be avoided.
  - WRONG

```
public class Foo {
    public void finalize() {
        super.finalize();
    }
}
```

  - RIGHT

```
public class Foo {
    /* Remove the finalize method
    public void finalize() {
    super.finalize();
    }
    */
}
```

Java Coding Conventions

- ❑ Parenthesize Mixed Logical Operators [High]
  - An expression containing multiple logical operators together should be parenthesized properly.
  - Use parentheses to clarify complex logical expressions for the reader. Also when using boolean variables use the naming conventions is or has as a prefix.
  - For example: isLoading, hasFinished
  - WRONG

```
void bar() {
    boolean a = false
    boolean b = true;
    boolean c = false;
        // ...
    if (a || b && c) {
        // ...
    }
}
```

Java Coding Conventions

- Parenthesize Mixed Logical Operators [High]
  - RIGHT

```
void bar() {
    boolean a = false;
    boolean b = true;
    boolean c = false;
    // ...
    if (a || (b && c)) {
        // ...
    }
}
```

Java Coding Conventions

- Do Not Assign Values In A Conditional Expression [High]
  - Use of assignments within conditions makes the source code hard to understand.
  - WRONG

```
if ((dir = new File(targetDir)).exists()) {
        // ...
}
```

  - RIGHT

```
File dir = new File(targetDir);
if (dir.exists()) {
        // ...
}
```

Java Coding Conventions

- Provide A break Statement Or Comment For A case Statement [Normal]
  - According to Sun coding conventions, every time a case falls through (doesn't include a break statement), a comment should be added where the break statement would normally be.
  - The break in the default case is redundant, but it prevents a fall-through error if later another case is added.

Java Coding Conventions

- Provide A break Statement Or Comment For A case Statement [Normal]
  - WRONG

```
switch (c) {
case Constants.NEWLINE:
      result += Constants.NEWLINE;
      break;
case Constants.RETURN:
      result += Constants.RETURN;
      break;
case Constants.ESCAPE:
      someFlag = true;
      case Constants.QUOTE:
      result += c;
      break;
// ...
}
```

Java Coding Conventions

- Provide A break Statement Or Comment For A case Statement [Normal]
  - RIGHT

```
switch (c) {
case Constants.NEWLINE:
    result += Constants.NEWLINE;
    break;
case Constants.RETURN:
    result += Constants.RETURN;
    break;
case Constants.ESCAPE:
    someFlag = true;
    // falls through
case Constants.QUOTE:
    result += c;
    break;
// ...
}
```

Java Coding Conventions

- Use equals To Compare Strings [Enforced]
  - The == operator used on strings checks whether two string objects are two identical objects. However, in most situations, you simply want to check for two strings that have the same value. In this case, the method equals should be used.
  - Replace the '==' operator with the equals method.
  - WRONG

```
void function(String str1, String str2) {
    if (str1 == str2) {
        // ...
    }
}
```

  - RIGHT

```
void function(String str1, String str2) {
    if (str1.equals(str2)) {
        // ...
    }
}
```

Java Coding Conventions

- Use L Instead Of l At The End Of A long Constant [Enforced]
  - It is difficult to distinguish the lower case letter l from the digit 1. When the letter l is used as the long modifier at the end of an integer constant, it can be confused with digits. In this case, it is better to use an uppercase L.
  - WRONG

```
void bar() {
        long var = 0x0001111l;
}
```

  - RIGHT

```
void bar() {
        long var = 0x0001111L;
}
```

Java Coding Conventions

- Do Not Use The synchronized Modifier For A Method [Normal]
  - The synchronized modifier on methods can sometimes cause confusion during maintenance as well as during debugging. This rule therefore recommends using synchronized statements as replacements instead.
  - Use synchronized statements instead of synchronized methods.
  - WRONG

```
public class Foo {
    public synchronized void bar() {
        // ...
    }
}
```

  - RIGHT

```
public class Foo {
    public void bar() {
        synchronized(this) {
                // ...
        }
    }
}
```

Java Coding Conventions

- Declare Variables Outside A Loop When Possible [Normal]
  - This rule recommends declaring local variables outside the loops. The reason: as a rule, declaring variables inside the loop is less efficient.
  - Move variable declarations out of the loop
  - WRONG

```
for (int i = 0; i < 100; i++) {
    int var1 = 0;
    // ...
}
while (true) {
    int var2 = 0;
    // ...
}
do {
    int var3 = 0;
    // ...
} while (true);
```

Java Coding Conventions

- Declare Variables Outside A Loop When Possible [Normal]
  - RIGHT

```
int var1;
for (int i = 0; i < 100; i++) {
    var1 = 0;
    // ...
}
int var2;
while (true) {
    var2 = 0;
    // ...
}

int var3;
do {
    var3 = 0;
    // ...
} while (true);
```

Java Coding Conventions

- Do Not Append To A String Within A Loop [High]
  - Operations on the class String (in package java.lang) are not very efficient when compared to similar operations on StringBuffer (in package java.lang). Therefore, whenever possible, you should replace uses of String operations by StringBuffer operations.
  - This rule checks for obvious misuse of String operations namely if a String object is appended to within a loop. Significant performance enhancements can be obtained by replacing String operations with StringBuffer operations.
  - Use StringBuffer class instead of String

Java Coding Conventions

- Do Not Append To A String Within A Loop [High]
  - WRONG

```
public class Foo {
    public String bar() {
        String var = "var";
        for (int i = 0; i < 10; i++) {
            var += (" " + i);
        }
        return var;
    }
}
```

  - RIGHT

```
public class Foo {
    public String bar() {
        StringBuffer var = new StringBuffer(23);
        var.append("var");
        for (int i = 0; i < 10; i++) {
            var.append(" " + i);
        }
        return var.toString();
    }
}
```

Java Coding Conventions

- ❑ Do Not Make Complex Calculations Inside A Loop When Possible [Low]
  - Avoid using complex expressions as repeat conditions within loops. By following this rule, you move repeated calculations within loops ahead of the loop where only a single calculation needs to be made.
  - Assign the expression to a variable before the loop and use that variable instead.
  - WRONG

```
void bar() {
    for (int i = 0; i < vector.size(); i++) {
        // ...
    }
}
```

  - RIGHT

```
void bar() {
    int size = vector.size();
    for (int i = 0; i < size; i++) {
        // ...
    }
}
```

Java Coding Conventions

- Do Not Unnecessary Jumble Loop Incrementors [High]
  - Avoid and correct jumbled loop incrementors because this is either a mistake or it makes the code unclear
  - to read.
  - WRONG

```
public void bar() {
for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; i++) {
                logger.info("i = " + i + ", j =" +j);
        }
}
}
```

  - RIGHT

```
public void bar() {
for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++) {
                logger.info("i = " + i + ", j =" +j);
        }
}
}
```

Java Coding Conventions

- ❑ Do Not Unnecessary Convert A String [High]
  - Avoid unnecessary String variables when converting a primitive to a String.
  - WRONG

```
public String bar(int x) {
    // ...
    String foo = new Integer(x).toString();
    return foo;
}
```

  - RIGHT

```
public String bar(int x) {
    // ...
    return Integer.toString(x);
}
```

Java Coding Conventions

- Override The equals And hashCode Methods Together [Enforced]
  - Override both public boolean Object.equals(Object other), and public int Object. hashCode(), or override neither.
  - Even if you are inheriting a hashCode() from a parent class, consider implementing hashCode and explicitly delegating to your super class.
  - WRONG

```
public class Foo {
    public boolean equals(Object o) {
        // do some comparison
    }
}
```

Java Coding Conventions

- Override The equals And hashCode Methods Together [Enforced]
  - RIGHT

```
public class Foo {
    public boolean equals(Object o) {
        // do some comparison
    }
    public int hashCode() {
        // return some hash value
    }
}
```

Java Coding Conventions

- Do Not Unnecessary Use The System.out.print or System.err.print Methods [High]
  - The System.out.print and System.err.print methods are often mis-used as a simple way to debug or log your application. Either use Logging API or centralize these print methods in a debug class.
  - WRONG

```
public void bar() {
    try {
        System.out.println("I got here");
        // ...
    } catch (JMSException e) {
        System.out.println("exception occurred");
        e.printStackTrace();
    } finally {
        System.out.println("cleaning up");
    }
}
```

Java Coding Conventions

- Do Not Unnecessary Use The System.out.print or System.err.print Methods [High]
  - RIGHT

```
public void bar() {
    logger.entering("Foo", "foo");
    try {
        // ...
    } catch (JMSException e) {
        logger.log(Level.WARNING, "JMS Problem", e);
    } finally {
        logger.info("cleaning-up");
    }
        logger.existing("Foo", "foo");
}
```

Java Coding Conventions

- Do Not Return In A Method With A Return Type of void [High]
  - Avoid unnecessary return statements, just remove this statement.
  - WRONG

```
public void bar() {
    // ...
    return;
}
```

  - RIGHT

```
public void bar() {
    // ...
}
```

Java Coding Conventions

- Do Not Reassign A Parameter [Enforced]
  - Avoid reassigning values to parameters, use a temporary local variable instead.
  - WRONG

```
public void foo(String bar) {
        bar = "Hello World";
}
```

  - RIGHT

```
public void foo(String bar) {
        String tmp = "Hello World";
}
```

Java Coding Conventions

- ❑ Package Declaration Is Required [Enforced]
  - All classes must have a package declaration. Classes that live in the null package cannot be imported.
  - Many novice developers are not aware of this.
  - WRONG

```
public class MyClass {
}
```

  - RIGHT

```
package some.package;
public class MyClass {
}
```

J2EE Coding Conventions

- The sources for these J2EE coding conventions are the different API specifications. Many of the J2EE coding rules can be verified with the J2EE Verifier tool included in the J2EE reference implementation from Sun Microsystems.

JSP Page Directive

- A JSP page directive defines attributes associated with the JSP page at translation time.
- The JSP specification does not impose a constraint on how many JSP page directives can be defined in the same page. So the following two Code Samples are equivalent (except that the first example introduces two extra blank lines in the output):
- Code Sample 1

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

- Code Sample 2 (Preferred)

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

J2EE Coding Conventions

- JSP Page Directive *contd…*
  - An exception occurs when multiple Java packages need to be imported into the JSP pages, leading to a very long import attribute:
  - Code Sample 1

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

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

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

J2EE Coding Conventions

❑ JSP Declarations

Disparate declaration blocks	Disparate declaration blocks

`<%!private int hitCount;%>`
`<%!private Date today;%>`
`...`
`<%!public int getHitCount()`
`    {`
`     return hitCount;`
`    }`
`%>`
	`<%!private int hitCount;`
	`private Date today;`
	`...`
	`public int getHitCount() {`
	`return hitCount;`
	`}%>`

`<%! private int x, y; %>`	`<%! private int x; %>`
	`<%! private int y; %>`

J2EE Coding Conventions

- JSP Programming Practices
  - In general, avoid writing Java code (declarations, scriptlets and expressions) in your JSP pages, for the following reasons:
  - Syntax errors in Java code in a JSP page are not detected until the page is deployed. Syntax errors in tag libraries and servlets, on the other hand, are detected prior to deployment.
  - Java code in JSP pages is harder to debug.
  - Java code in JSP pages is harder to maintain, especially for page authors who may not be Java experts.
  - It is generally accepted practice not to mix complex business logic with presentation logic. JSP pages are primarily intended for presentation logic.
  - Code containing Java code, HTML and other scripting instructions can be hard to read.

❑ Uniform use of Quoting

Non-uniform quoting	Preferred quoting
`<%@ page import=‘javabeans.*’ %>`	`<%@ page import="javabeans.*" %>`
`<%@ page import="java.util.*" %>`	`<%@ page import="java.util.*" %>`

4

Exceptions Conventions

Conventions for Handling Exceptions in Java / J2EE Programming

Exceptions Conventions

- Using and handling exceptions is strongly linked to the java coding process.
- Therefore the Exception conventions were added to both the Java and J2EE coding conventions.

Exceptions Conventions

- Do Not Catch java.lang.Exception Or java.lang.Throwable [Normal]
  - catch clauses must use exception types that are as specific as possible for the exception that may be thrown from the corresponding try block. By catching all exceptions a developer might also catch
  - For example a java.lang.OutOfMemoryException and interpret it in the wrong way (see example below). Following this rule catches programmer tardiness in fully enumerating the exception situations and properly handling them. As a rule developers should only catch what is thrown.
  - Exceptions to this rule are allowed if a method in a third party library throws a java.lang.Exception or java.lang.Throwable exception. In this case catching the general exception is allowed.

Exceptions Conventions

- Do Not Catch java.lang.Exception Or java.lang.Throwable [Normal]
  - WRONG

```
try {
file = new FileInputStream("myfile");
} catch (Throwable e) {
logger.log(Level.WARNING, "File myfile not found");
}
```

  - RIGHT

```
try {
file = new FileInputStream("myfile");
} catch (IOException e) {
logger.log(Level.WARNING, "File myfile not found");
} finally {
// If the file is open, close it here
}
```

Exceptions Conventions

- Provide At Least One Statement In A try Block [Enforced]
  - Empty try blocks should be removed because they bring no added value to your code.
  - WRONG

```
try {
} catch (SQLException e) {
      logger.log(Level.WARNING, "SQL Problem", e);
}
```

  - RIGHT

```
// Please remove this try-catch block !
```

Exceptions Conventions

❑ Provide At Least One Statement In A finally Block [Enforced]
- An empty finally block can be removed from the try/catch block.
- WRONG

```
try {
        name = rs.getString(1);
} catch (SQLException e) {
        logger.log(Level.WARNING, "SQL Problem", e);
} finally {
}
```

- RIGHT

```
try {
        name = rs.getString(1);
} catch (SQLException e) {
        logger.log(Level.WARNING, "SQL Problem", e);
}
```

Exceptions Conventions

- Do Not Return From Inside A try Block [High]
  - Code leaves the try block when it finishes normally, when an exception is thrown or a return, break or continue statement is executed. When a finally block exists, it will always be executed.
  - This means that a return in a try block is not guaranteed to be the normal flow of the code!
  - Putting a return statement in a try block might lead a developer to think that the finally block will not be executed.

Exceptions Conventions

- Do Not Return From Inside A finally Block [High]
  - Avoid returning from a finally block - this can discard exceptions.
  - WRONG

```
public void bar() {
try {
        throw new JMSException("My Exception");
} catch (JMSException e) {
        logger.log(Level.WARNING, "JMS Problem", e);
        throw e;
} finally {
        return;
}
}
```

Exceptions Conventions

- ❑ Do Not Return From Inside A finally Block [High]
  - RIGHT

```
public void bar() {
try {
    throw new JMSException("My JMS Exception");
} catch (JMSException e) {
    flag = "NOK";
    logger.log(Level.WARNING, "JMS Problem", e);
    throw e;
} finally {
        // cleanup here
}
}
```

Exceptions Conventions

- Do Not Use A try Block Inside A Loop When Possible [Normal]
  - An application might slow down when placing a try/catch block in a loop.
  - WRONG

```
public void bar() {
int balance = 0;
for (int i = 0; i < account.length; i++) {
        try {
                balance = account[i].getValue();
                account[i].setValue(balance * 1.2);
        } catch (AccountValueNotValidException e) {
                logger.log(Level.WARNING, "Account value
                not valid", e);
        }
}}
```

Exceptions Conventions

- Do Not Use A try Block Inside A Loop When Possible [Normal]
  - RIGHT

```
public void bar() {
int balance = 0;
try {
        for (int i = 0; i < account.length; i++) {
                balance = account[i].getValue();
                account[i].setValue(balance * 1.2);
        }
} catch (AccountValueNotValidException e) {
        logger.log(Level.WARNING, "Account value not
                valid", e);
}
}
```

Exceptions Conventions

- Do Not Use An Exception For Control Flow [High]
  - Using exceptions for flow control is just not a good idea. The example you can see below is a classic for people starting with the Java language.
  - WRONG

```
public void bar(int numbers[]) {
    int i = 0;
    try {
        while (true) {
            logger.info("number =" + numbers[i++]);
        }
    } catch (ArrayIndexOutOfBoundsException e) {
    // do nothing
    }
}
```

Exceptions Conventions

- Do Not Use An Exception For Control Flow [High]
  - RIGHT

```
public void bar(int numbers[]) {
    for (int i = 0; i < numbers.length; i++) {
        logger.info("number =" + numbers[i]);
    }
}
```

Exceptions Conventions

- Close A Connection Inside A finally Block [Enforced]
  - Ensure that a connection object is always closed within the finally block.
  - Use a helper method to close the connection, this makes the finally block small and readable.
  - WRONG

```
public void bar() {
        ServerSocket serverSocket = new ServerSocket(0);
    try {
        Socket mySocket = serverSocket.accept();
        // Do something with the mySocket
        mySocket.close();
    } catch (IOException e) {
        logger.log(Level.WARNING, "IO Problem", e);
    }
}
```

Exceptions Conventions

- Close A Connection Inside A finally Block [Enforced]
  - RIGHT

```
public void bar() {
    ServerSocket serverSocket = new ServerSocket(0);
    Socket mySocket = null;
    try {
        mySocket = serverSocket.accept();
        // Do something with the mySocket
    } catch (IOException e) {
        logger.log(Level.WARNING, "IO Problem", e);
    } finally {
        closeConnection(mySocket);
    }
}
```

Exceptions Conventions

- Do Not Declare A Method That Throws java.lang.Exception or java.lang.Throwable [Normal]
  - A method must not throw java.lang.Exception or java.lang.Throwable because of the unclarity. Use either a class derived from java.lang.RuntimeException or a checked exception with a descriptive name.
  - This rule is linked to rule: Do not catch java.lang.Exception or java.lang.Throwable [Normal].
  - WRONG

```
public void bar() throws Exception {
}
```

  - RIGHT

```
public void bar() throws MyException {
}
```

Exceptions Conventions

- ❑ Do Not Use An instanceof Statement To Check An Exception [High]
  - Do not check exceptions with instanceof, instead catch the specific exceptions.
  - WRONG

```
SimpleDateFormat sdf = new SimpleDateFormat("dd.MM.yyyy");
try {
       returnString = sdf.format(value);
} catch (Exception e) {
       if (e instanceof NumberFormatException) {
              NumberFormatException nfe =
              (NumberFormatException) e;
              logger.log(Level.WARNING, "NumberFormat
              exception", nfe);
       }
       if (e instanceof IllegalArgumentException) {
              IllegalArgumentException iae =
              (IllegalArgumentException) e;
              logger.log(Level.WARNING, "Illegal argument",
              iae);
       }
}
```

Exceptions Conventions

- Do not use an instanceof statement to check an exception [High]
  - RIGHT

```
try {
        returnString = sdf.format(value);
} catch (NumberFormatException e) {
        logger.log(Level.WARNING, "NumberFormat
        exception", e);
} catch (IllegalArgumentException e) {
        logger.log(Level.WARNING, "Illegal argument", e);
}
```

Exceptions Conventions

- Do Not Catch A RuntimeException [High]
  - Don't catch runtime exceptions. Runtime exceptions are generally used to report bugs in code and should therefore never be caught.
  - WRONG

```
public void doSomething() {
    try {
        doSomethingElse(null);
    }catch (NullPointerException e) {
        // exception handling
    }
}
private void doSomethingElse(String arg) {
        arg.trim();
}
```

  - RIGHT

```
Don't catch the NullPointerException runtime exception
in doSomething(), instead fix the
bug.
```

Exceptions Conventions

- Do Not Use printStackTrace [High]
  - Don't use printStackTrace to report exceptions.
  - The printStackTrace method prints the stack trace to standard error (usually the console).
  - Unless the console output is redirected to a file the stack trace will be lost once the console's screen buffer becomes full. A better alternative is to use a logger and log the exception with the appropriate logging level.

Books / Links to refer to for further reading…

Code Conventions for the Java Programming Language

Servlets and JSP Pages Best Practices

Code Conventions for the JavaServer Pages Technology Version 1.x Language

Guidelines, Patterns, and code for end-to-end Java applications

Questions