

# **Technical Paper: Java - J2EE Conventions and Guidelines**

**Release 1.0.2**

**Stephan Janssen**

**The JCS Team**

---

# **Technical Paper: Java - J2EE Conventions and Guidelines: Release 1.0.2**

Stephan Janssen

The JCS Team

Copyright © 1998-2004 JCS Int. NV. ([www.jcs.be](http://www.jcs.be))

Copyright © 1995-1999 Adapted with permission from CODE CONVENTIONS FOR THE JAVA™ PROGRAMMING LANGUAGE. Copyright 1995-1999 Sun Microsystems, Inc. All rights reserved.

Copyright © 1995-1999 Sun, Sun Microsystems, the Sun logo, Java, J2EE, JavaServer Pages, Enterprise JavaBeans, Java Naming and Directory Interface, EJB, JSP, J2EE, J2SE and the Java Coffee Cup logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

---

---

---

---

# Table of Contents

Preface .....	13
Why Read This Document? .....	13
Who Should Read This? .....	13
Conventions Used In This Document .....	13
Typographical Conventions .....	14
Post Your Feedback .....	14
Example Application .....	14
More Information .....	14
1. Conventions .....	1
Why Use Conventions? .....	1
Convention Rules .....	1
Defining .....	1
Enforcing .....	1
Naming Conventions .....	2
Introduction .....	2
Java Naming .....	2
J2EE Naming .....	7
Comments Conventions .....	11
Java Comments .....	11
Coding Conventions .....	18
Introduction .....	18
Java Coding .....	18
J2EE Coding .....	26
Exceptions .....	41
Exceptions Defined .....	41
When To Use Exceptions .....	41
Exception Types .....	42
Throwing Exceptions .....	43
Catching Exceptions .....	43
Cleaning Up After An Exception Using A finally Clause .....	44
Exception Matching .....	45
Exceptions And Error Messages .....	46
Exception Chaining .....	47
Creating Custom Exceptions .....	47
Exceptions And EJBs .....	48
2. Guidelines .....	51
Introduction .....	51
Project Directory Structure .....	51
Introduction .....	51
Proposed Directory Structure .....	52
Build Tools .....	53
ANT .....	53
Maven .....	65
Logging .....	71
Introduction .....	71
Why Use Logging? .....	71
Logging Example .....	71
Logging Levels .....	73
Localized Messages .....	74
Configuration File .....	74
Database Independency .....	75
Introduction .....	75
JDBC Basics .....	75

---

Use Of SQL .....	81
Know Your Target Database(s) .....	81
Architecture .....	82
JMS .....	82
Introduction .....	82
Overview Of JMS .....	83
JMS Examples .....	91
Assertions .....	94
Introduction .....	94
Coding Assertions .....	95
Design By Contract .....	97
Testing .....	99
Automated Regression Testing .....	99
Unit Tests (White Box) .....	100
Scenario Tests (Black Box) .....	111
Integration of JUnit with development tools .....	113
Stress Tests (Volume Tests) .....	114
Stress testing with JMeter .....	115
JMX .....	116
Introduction .....	117
JMX Basics .....	117
Why And When To Use JMX .....	118
How To Use JMX .....	119
JMX Pitfalls .....	121
MBeans .....	122
JAAS .....	124
Introduction .....	124
Why Use JAAS .....	124
JAAS Basics .....	124
Complete JAAS Examples .....	130
A. Conventions Rules .....	137
Java Naming Conventions Rules .....	137
Overview .....	137
JAN_001: Match A Class Name With Its File Name (High) .....	137
JAN_002: Group Operations With The Same Name Together (Low) .....	138
JAN_003: Use A Correct Name For A Class Or Interface (Enforced) .....	139
JAN_004: Use A Correct Name For A Non Final Field (Enforced) .....	139
JAN_005: Use A Correct Name For A Constant (Enforced) .....	139
JAN_006: Use A Correct Name For A Method (Enforced) .....	139
JAN_007: Use A Correct Name For A Package (Enforced) .....	140
JAN_008: Name An Exception Class Ending With Exception (High) .....	140
JAN_009: Use A Conventional Variable Name When Possible (Normal) .....	140
JAN_010: Do Not Use \$ In A Name (Enforced) .....	141
JAN_011: Do Not Use Names That Only Differ In Case (High) .....	142
JAN_012: Make A Constant private Field static final (Normal) .....	142
JAN_013: Do Not Declare Multiple Variables In One Statement (High) .....	142
JAN_014: Use A Correct Order For Class Or Interface Member Declarations (Normal) .....	143
JAN_015: Use A Correct Order For Modifiers (enforced) .....	144
JAN_016: Put The main Method Last (High) .....	144
JAN_017: Put The public Class First (High) .....	145
J2EE Naming Conventions Rules .....	145
Overview .....	145
JEN_001: Name An EJB Bean Class Like [Name]EJB or [Name]Bean (High) .....	146
JEN_002: Name An EJB Remote Home Interface Like [Name]Home (High) .....	146
JEN_003: Name An EJB Remote Interface Like [Name] (High) .....	147
JEN_004: Name An EJB Local Home Interface Like [Name]LocalHome (High) .....	147
JEN_005: Name A Transfer Object Like [Name]TO (High) .....	148

---

JEN_006: Name An EJB In The Deployment Descriptor Like [Name]EJB (High) .....	148
JEN_007: Name An EJB Display Name In The Deployment Descriptor Like [Name]JAR (High) .....	148
JEN_008: Name A Servlet Like [Name]Servlet (High) .....	149
JEN_009: Name A Primary Key Class Like [Name]PK (High) .....	149
JEN_010: Name A Filter Servlet Like [Name]Filter (High) .....	149
JEN_011: Name A Local Interface Like [Name]Local (High) .....	149
JEN_012: Name A Data Access Object Like [Name]DAO (High) .....	150
JEN_013: Use A Correct Name For An Enterprise Application Display Name (High) .....	150
JEN_014: Use A Correct Name For A Web Module Display Name (High) .....	150
JEN_015: Use A Correct Name For An EJB Environment Reference Name (High) .....	151
JEN_016: Name A JMS Destination Like [Name]Queue Or [Name]Topic (High) .....	151
JEN_017: Use A Correct Name For A JMS Environment Reference Name (High) .....	152
JEN_018: Use A Correct Name For A JDBC Environment Reference Name (High) ....	152
JEN_019: Name A Database Like [Name]DB (High) .....	153
Java Comments Conventions Rules .....	153
Overview .....	153
JAD_001: Do Not Use An Invalid Javadoc Comment Tag (High) .....	154
JAD_002: Provide A Correct File Comment For A File (High) .....	154
JAD_003: Provide A Javadoc Comment For A Class (Enforced) .....	154
JAD_004: Provide A Javadoc Comment For A Constructor (Enforced) .....	155
JAD_005: Provide A Javadoc Comment For A Method (Enforced) .....	155
JAD_007: Provide A Javadoc comment For A Field (Enforced) .....	156
JAD_008: Provide a package.html per package (Low) .....	156
Java Coding Conventions Rules .....	156
Overview .....	157
JAC_001: Do Not Make An Attribute Non final And static (Low) .....	159
JAC_002: Do Not Reference A static Member Through An Object Instance (High) ....	159
JAC_003: Do Not Make A File Longer Than 2000 Lines (Enforced) .....	160
JAC_004: Do Not Make A Line Longer Than 120 Characters (Normal) .....	160
JAC_005: Do Not Use Complex Variable Assignment (High) .....	160
JAC_006: Do Not Code Numerical Constants Directly (Low) .....	161
JAC_007: Do Not Place Multiple Statements On The Same Line (High) .....	161
JAC_008: Parenthesize The Conditional Part Of A Ternary Conditional Expression (High) .....	161
JAC_009: Provide An Incremental In A for Statement (High) .....	162
JAC_010: Do Not Use A Demand Import (Enforced) .....	162
JAC_011: Provide A default case In A switch Statement (Enforced) .....	162
JAC_012: Use The Abbreviated Assignment Operator When Possible (Normal) .....	163
JAC_013: Do Not Make A Method Longer Than 60 Lines (Normal) .....	163
JAC_014: Do Not Make A switch Statement With More Than 256 Cases (Normal) ....	163
JAC_015: Do Not Chain Multiple Methods (Normal) .....	163
JAC_016: Use A Single return Statement (Normal) .....	164
JAC_017: Do Not Duplicate An import Declaration (Enforced) .....	164
JAC_018: Do Not Import A Class Of The Package To Which The Source File Belongs (Enforced) .....	165
JAC_019: Do Not Import A Class From The Package java.lang (Enforced) .....	165
JAC_020: Do Not Use An Equality Operation With A boolean Literal Argument (Enforced) .....	166
JAC_021: Do Not Import A Class Without Using It (Enforced) .....	166
JAC_022: Do Not Unnecessary Parenthesize A return Statement (Normal) .....	167
JAC_023: Do Not Declare A private Class Member Without Using It (Enforced) .....	167
JAC_024: Do Not Use Unnecessary Modifiers For An Interface Method (Low) .....	168
JAC_025: Do Not Use Unnecessary Modifiers For An Interface Field (Low) .....	168
JAC_026: Do Not Use Unnecessary Modifiers For An Interface (High) .....	169
JAC_027: Do Not Declare A Local Variable Without Using It (Enforced) .....	169
JAC_028: Do Not Do An Unnecessary Typecast (High) .....	169

JAC_029: Do Not Do An Unnecessary instance of Evaluation (High)	170
JAC_030: Do Not Hide An Inherited Attribute (High)	171
JAC_031: Do Not Hide An Inherited static Method (High)	171
JAC_032: Do Not Declare Overloaded Constructors Or Methods With Different Visibility Modifiers (High)	172
JAC_033: Do Not Override A Non abstract Method With An abstract Method (High)	172
JAC_034: Do Not Override A private Method (High)	173
JAC_035: Do Not Overload A Super Class Method Without Overriding It (High)	174
JAC_036: Do Not Use A Non final static Attribute For Initialization (High)	175
JAC_037: Do Not Use Constants With Unnecessary Equal Values (High)	175
JAC_038: Provide At Least One Statement In A catch Block (Normal)	176
JAC_039: Do Not Catch java.lang.Exception Or java.lang.Throwable (Normal)	176
JAC_040: Do Not Give An Attribute A public Or Package Local Modifier (Enforced)	177
JAC_041: Provide At Least One Statement In A Statement Body (Enforced)	178
JAC_042: Do Not Compare Floating Point Types (Low)	178
JAC_043: Enclose A Statement Body In A Loop Or Condition Block (Enforced)	179
JAC_044: Explicitly Initialize A Local Variable (High)	179
JAC_045: Do Not Unnecessary Override The finalize Method (High)	180
JAC_046: Parenthesize Mixed Logical Operators (High)	180
JAC_047: Do Not Assign Values In A Conditional Expression (High)	181
JAC_048: Provide A break Statement Or Comment For A case Statement (Normal)	181
JAC_049: Use equals To Compare Strings (Enforced)	182
JAC_050: Use L Instead Of l At The End Of A long Constant (Enforced)	183
JAC_051: Do Not Use The synchronized Modifier For A Method (Normal)	183
JAC_052: Declare Variables Outside A Loop When Possible (Normal)	184
JAC_053: Do Not Append To A String Within A Loop (High)	185
JAC_054: Do Not Make Complex Calculations Inside A Loop When Possible (Low)	185
JAC_055: Provide At Least One Statement In A try Block (Enforced)	186
JAC_056: Provide At Least One Statement In A finally Block (Enforced)	186
JAC_057: Do Not Unnecessary Jumble Loop Incrementors (High)	187
JAC_058: Do Not Unnecessary Convert A String (High)	187
JAC_059: Override The equals And hashCode Methods Together (Enforced)	188
JAC_060: Do Not Use Double Checked Locking With Lazy Initialization (Enforced)	188
JAC_061: Do Not Return From Inside A try Block (High)	189
JAC_062: Do Not Return From Inside A finally Block (High)	189
JAC_063: Do Not Use A try Block Inside A Loop When Possible (Normal)	189
JAC_064: Do Not Use An Exception For Control Flow (High)	190
JAC_065: Do Not Unnecessary Use The System.out.print or System.err.print Methods (High)	191
JAC_066: Do Not Return In A Method With A Return Type of void (High)	191
JAC_067: Do Not Reassign A Parameter (Enforced)	192
JAC_068: Close A Connection Inside A finally Block (Enforced)	192
JAC_069: Do Not Declare A Method That Throws java.lang.Exception or java.lang.Throwable (Normal)	193
JAC_070: Do Not Use An instance of Statement To Check An Exception (High)	193
JAC_071: Do Not Catch A RuntimeException (High)	194
JAC_072: Do Not Use printStackTrace (High)	194
JAC_073: Package Declaration Is Required (Enforced)	194
J2EE Coding Conventions Rules	195
Overview	195
JEC_001: Do Not Create A Class Loader (High)	196
JEC_002: Do Not Read Or Write A File Descriptor (High)	197
JEC_003: Do Not Use A AWT, Swing Or Other UI API In An EJB (High)	197

JEC_004: Do Not Attempt To Load A Native Library In An EJB (High) .....	197
JEC_005: Do Not Attempt To Obtain The Security Policy Information In An EJB (High) .....	197
JEC_006: Do Not Attempt To Set The Socket Factory In An EJB (High) .....	197
JEC_007: Do Not Attempt To Listen On A Socket In An EJB (High) .....	197
JEC_008: Do Not Attempt To Use The Subclass Or Object Substitution Features Of The Java Serialization Protocol (High) .....	198
JEC_009: Do Not Attempt To Manage A Thread (High) .....	198
JEC_010: Do Not Use The java.io Package To Access The File System (Normal) .....	198
JEC_011: Do Not Pass An EJB's this Reference As An Argument Or Method Result (Enforced) .....	198
JEC_012: Do Not Throw A RemoteException From An EJB Implementation Method (High) .....	198
JEC_013: Make All EJB interfaces and classes public (High) .....	199
JEC_014: Do Not Make A EJB implementation class final (High) .....	199
JEC_015: Provide An ejbCreate Method For A Session Bean (High) .....	199
JEC_016: Provide A public Default Constructor For A EJB implementation class (High) .....	200
JEC_017: Do Not Override The finalize Method For A EJB implementation class (High) .....	200
JEC_018: Return void For An ejbCreate Method Of A Session Bean (High) .....	201
JEC_019: Return The EJB Remote Interface For A create Method Of An EJB Remote Home Interface (Enforced) .....	202
JEC_020: Make A create Method Of An EJB Home Interface Throw An javax.ejb.CreateException (Enforced) .....	202
JEC_022: Do Not Declare A Finder Method For A CMP EJB (High) .....	203
JEC_024: Match An ejbPostCreate Method To An ejbCreate Method For An Entity Bean (High) .....	203
JEC_025: Declare An ejbCreate Method public For An Entity Bean (High) .....	204
JEC_026: Return The Primary Key For An ejbCreate[Name] Method Of An Entity Bean (High) .....	204
JEC_027: Do Not Declare An ejbCreate Or ejbPostCreate Method As final Or static For An Entity Bean (High) .....	205
JEC_028: Return void For An ejbPostCreate Method Of An Entity Bean (High) .....	205
JEC_029: Declare An ejbFindByPrimaryKey Method For A BMP EJB (High) .....	205
JEC_030: Declare A ejbFind Method public For A BMP EJB (High) .....	206
JEC_031: Do Not Declare A ejbFind Method final Or static (High) .....	206
JEC_032: Make A find Method Of An EJB Home Interface throw javax.ejb.FinderException (High) .....	207
JEC_033: Declare An ejbSelect Method Of An Entity Bean public (High) .....	207
JEC_034: Make An ejbSelect Method Of An Entity Bean throw javax.ejb.FinderException (High) .....	208
JEC_035: Return The EJB Local Interface For A create Method In An EJB Local Home Interface (Enforced) .....	208
JEC_036: Declae An ejbPostCreate Method Of An Entity Bean public (High) .....	208
JEC_037: Make A Method Of An EJB Remote Interface Throw java.rmi.RemoteException (Enforced) .....	209
JEC_038: Make A Method Of An EJB Remote Home Interface Throw java.rmi.RemoteException (Enforced) .....	209
JEC_039: Do Not Make A Method Of An EJB Local Interface Throw java.rmi.RemoteException (Enforced) .....	210
JEC_040: Do Not Make A Method Of An EJB Local Home Interface Throw java.rmi.RemoteException (Enforced) .....	210
JEC_042: Declare An ejbSelect Method Of A CMP EJB abstract (High) .....	211
JEC_043: Make An ejbCreate Method Of An Entity Bean Throw javax.ejb.CreateException (High) .....	211
JEC_044: Declare An ejbHome Method Of An Entity Bean public (High) .....	211
JEC_045: Do Not Declare An ejbHome Method Of An Entity Bean final Or static	



(High) .....	212
JEC_046: Do Not Make An ejbHome Method Of An Entity Bean Throw	
java.rmi.RemoteException (High) .....	212
JEC_047: Make A Message Bean Implementation Implement	
javax.jms.MessageListener (High) .....	212
JEC_052: Provide An ejbCreate Method For A Message Bean (High) .....	212
JEC_053: Return void For An ejbCreate Method Of A Message Bean (High) .....	213
JEC_054: Do Not Declare Arguments For An ejbCreate Method Of A Message Bean (High) .....	214
JEC_055: Provide A Valid RMI Method Signature For An EJB Remote Home Interface (High) .....	214
JEC_056: Do No Print Unnecessary Static Content From A Servlet (High) .....	214
JEC_057: Use Custom JSP Tags Instead Of Scriptlets (High) .....	216
JEC_058: Do Not Forward A Request From A JSP Page (High) .....	217
Exceptions Conventions Rules .....	217
Overview .....	217
B. Guidelines Rules .....	219
Introduction .....	219
Ant Rules .....	219
Overview .....	219
ANT_001: Use the default build.xml build filename .....	219
ANT_002: Do Not Hard Code An Absolute Directory Or File Path .....	219
ANT_003: Suffix The Name Of A Property That Represents A Directory With .dir ....	220
ANT_004: Use Package Like Names To Namespace A Property Or Target .....	221
ANT_005: Depend Only On Direct Dependencies .....	221
ANT_006: Make Each Target Directly Runnable .....	222
ANT_007: Use The location Attribute For A File Or Directory Based Property .....	223
ANT_008: Use The zipfileset Task To Create Archives .....	224
ANT_009: Keep Your Build File Platform Independent When Possible .....	224
ANT_010: Set A Needed Environment Variable In The Build Script .....	225
Logging Rules .....	225
Overview .....	225
LOG_001: Retrieve A Logger Based On The Fully Qualified Package And Class Name .....	226
LOG_002: Declare A Logger Instance private final static .....	226
LOG_003: Log A Caught Exception .....	227
LOG_004: Use A Correct FileHandler Log Name .....	228
LOG_005: Use The Log Level SEVERE Only For Non Recoverable Problems .....	229
LOG_006: Use The Log Level WARNING Only For Recoverable Problems .....	229
LOG_007: Use The Log Level INFO Only For Information Logs .....	229
LOG_008: Use The Log Level CONFIG Only For Configuration Problems .....	230
Database Independency Rules .....	230
Overview .....	230
JDBC_001: Do Not Hard Code A JDBC Driver Class Name .....	231
JDBC_002: Do Not Hard Code A JDBC Connection URL .....	231
JDBC_003: Do Not Use A Driver Management Method Of java.sql.DriverManager ....	231
JDBC_004: Do Not Import A JDBC Vendor Specific Class .....	231
JDBC_005: Close Connection, Statement And ResultSet .....	232
JDBC_006: Close JDBC Resources In The Correct Order .....	233
JDBC_007: Use PreparedStatement Instead Of Statement When Possible .....	233
JDBC_008: Check The Return value Of A Navigation Method Of ResultSet .....	234
JDBC_009: Use Standard SQL Only .....	235
JDBC_010: Check For A Nested SQLException .....	235
JDBC_011: Check For A SQLWarning .....	235
JMS Rules .....	235
Overview .....	235
JMS_001: Do Not Use A Vendor Specific Class .....	236
JMS_002: Do Not Hard Code An Initial Context Factory .....	236

JMS_003: Do Not Hard Code A Provider URL .....	237
JMS_004: Do Not Hard Code A Connection Factory Name For JNDI Lookup .....	237
JMS_005: Do Not Hard Code A Destination Name For JNDI Lookup .....	237
JMS_006: Use JNDI Lookups To Get A Connection Factory Or Destination .....	237
JMS_007: Close A Resource When It's No Longer Needed .....	238
JMS_008: Always Close Resources In The Correct Order .....	238
JMS_009: Do Not Produce A Message In A Transaction And Rely On It To Be Consumed Before The End Of The Transaction .....	239
JMS_010: Do Not Rely On JMS To Deliver Message In The Same Order As They Were Sent .....	239
JMS_011: Start A Producer Connection After Start A Consumer .....	239
JMS_012: Use A separate Transactional Session For transactional Messages And A Non-transactional Session For Non-transactional Messages .....	239
JMS_013: Set An Optimal Message Time To Live .....	239
JMS_014: Choose A Correct Message Type .....	240
JMS_015: Do Not Receive Messages Asynchronously In A Web Component, A Session Bean Or An Entity Bean .....	240
JMS_016: Do Not Use JMS Sessions In A Multi-threaded Context .....	240
Assertions Rules .....	240
Overview .....	240
ASSERT_001: Check A Method's Arguments .....	240
ASSERT_002: Do Not Use Assertions For Argument Checking In A public Method .....	241
ASSERT_003: Do Not Do Any Processing In An Assertion's Condition .....	241
ASSERT_004: Do Not Catch Assertion Related Exceptions .....	242
ASSERT_005: Use Assertions In A switch Statement's default Case Correct .....	242
ASSERT_006: Do Not Evaluate More Than One Condition In An Assertion .....	243
ASSERT_007: Make An Assertion Descriptive .....	244
Testing Rules .....	245
Overview .....	245
TEST_001: Provide A Unit Test For Each Utility Class, Library Or Base Layer .....	245
TEST_002: Name Test Methods Properly .....	246
TEST_003: Provide Mock Objects For A Unit Which Collaborates With Other Objects .....	246
TEST_004: Only Test What Can Possibly Break .....	246
TEST_005: Automate Running Unit Tests In The Build Process .....	246
TEST_006: Do Not Write Business Logic In Mock Objects .....	246
TEST_007: Run Scenario Tests As Part Of The Delivery Procedure Of A Product .....	247
TEST_008: Keep Mock Objects Independent From Each Other .....	247
TEST_009: Do Not Rely On The Order Of Tests Within A Testcase .....	247
TEST_010: Avoid Visual Inspection In Unit Tests .....	248
TEST_011: Avoid Code Duplication In Unit Tests .....	248
TEST_012: Call setUp() and tearDown() Methods Of A Testcase's Superclass .....	249
TEST_013: Isolate Test Data From Test Code .....	250
TEST_014: Do Not Load Data From Hardcoded Locations .....	250
TEST_015: Make Tests Locale Independent .....	250
TEST_016: Keep Unit Tests Small And Fast .....	250
JMX Rules .....	250
Overview .....	250
JMX_001: Name A MBean Interface Like [ImplementingClassName]MBean .....	251
JMX_002: Declare A MBean Interface public .....	252
JMX_003: Provide A public No Arg Constructor For A MBean Implementing Class .....	252
JMX_004: Make A Getter Or Setter Follow The Naming Conventions .....	253
JMX_005: Return The Property Type For A Getter .....	253
JMX_006: Declare A Getter Without Parameters .....	253
JMX_007: Declare A Setter With At Least 1 Parameter .....	254
JMX_008: Declare A Setter With At Most 1 Parameter .....	254

JMX_009: Do Not Define The Same Getter Twice .....	254
JMX_010: Match Getter And Setter .....	255
JMX_011: Do Not Construct An MBean .....	255
JAAS Rules .....	255
Overview .....	255
JAAS_001: Do Not Add A Principal Or Credential To A Subject At Login Time In A Login Module .....	256
JAAS_002: Only Clean A Principal Or Credential Previously Added To A Non Read-only Subject At Logout Time In A Login Module .....	257
JAAS_003: Return false From A login Method Only To Indicate To Ignore The Login Module .....	257
JAAS_004: Do Not Interact With A User Directly In A Login Module .....	258
JAAS_005: Ask For Credentials Only Once When Combining Multiple Login Modules .....	258
JAAS_006: Add A Debug Option To A Login Module .....	259
JAAS_007: Make A Principal Serializable .....	260
JAAS_008: Use Principals To Name Groups And Roles .....	260
JAAS_009: Use A PrincipalComparator To Structure Principals Hierarchically .....	261
JAAS_010: Use A PrivilegedExceptionAction To Raise Checked Exceptions From Secured Code Fragments .....	261
JAAS_011: Use The doAsPrivileged Method Of A Subject Instead Of The doAs Method .....	262
C. CheckStyle .....	263
Introduction .....	263
Integration With ANT .....	263
Installation .....	263
Usage .....	264
Integration With IntelliJ IDEA 4.x .....	264
Installation .....	264
Usage .....	268
Integration With Eclipse/WSAD .....	269
Installation .....	270
Usage .....	276
Integration With JBuilder X .....	276
Installation .....	276
Usage .....	277
Integration With JDeveloper .....	279
Installation And Usage .....	279
Glossary .....	280
Bibliography .....	287

---

## List of Tables

1.1. Severity .....	1
1.2. Naming Conventions For Enterprise Java Beans .....	8
1.3. Parts Of A Class Or Interface Declaration Notes .....	20
1.4. Exception Types .....	42
2.1. Project Directory Structure .....	52
2.2. Standard Ant Properties .....	58
2.3. Common Ant Targets .....	61
2.4. Comparison Of Ant And Maven .....	70
2.5. POM Naming .....	70
2.6. Logging Levels .....	73
2.7. JDBC Driver Types .....	76
2.8. DataSource Types .....	78
2.9. Statements Types .....	79
2.10. Header Types .....	84
2.11. Local Transactions .....	91
2.12. JVM Assertion Command Line Options .....	97
A.1. Java Naming Conventions Overview .....	137
A.2. J2EE Naming Conventions Overview .....	145
A.3. Java Comments Conventions Overview .....	153
A.4. Java Coding Conventions Overview .....	157
A.5. J2EE Coding Conventions Overview .....	195
A.6. Exception Overview .....	217
B.1. Ant Overview .....	219
B.2. Logging Overview .....	226
B.3. Database Independency Overview .....	230
B.4. JMS Overview .....	236
B.5. Assertions Overview .....	240
B.6. What Kind Of Assertion To Use .....	243
B.7. Testing Overview .....	245
B.8. JMX Overview .....	250
B.9. JAAS Overview .....	255

---

# Preface

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]preface](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]preface)]

## Why Read This Document?

When a J2EE project is not performing fast enough or some bugs can not be found a code audit is often performed. The first question asked is "Does the company use any Java coding conventions?" Many companies have a coding conventions document but when you then look at the code nobody within the team has followed the conventions.

So why bother writing another convention document that will only collect dust on a shelf of an IT manager or developer?

- First of all this document does not only cover Java and J2EE coding conventions, but documentation, naming conventions, guidelines and best practices.
- Secondly these conventions can be enforced by using existing Java tools. By enforcing these conventions we'll make a better developer out of you and hopefully we'll all produce better Java and J2EE code!
- Thirdly, and most importantly, this document was only possible through the support of the Flemish government and the Federal ICT (FEDict) department. The result is that future Java and J2EE projects executed for these departments have to follow and enforce the conventions written in this document!

## Who Should Read This?

People who want to introduce Java or J2EE standards within their company should read and use this document.

If you're a senior Java or J2EE developer then skip all chapters and focus on the appendix which lists the different rules.

If you're replying to a Request For Proposal (RFP) from either the Flemish government or FEDict then focus on the conventions and guidelines chapters, including the rules (with priority enforced and high) described in the appendix.

## Conventions Used In This Document

Throughout this document you will find tips and notes.



### Tip

A tip is a helpful hint from the author.



### Note

A note needs to be observed carefully by the reader.

Each rule in this document has a reference number, for example [JAN\_001] refers to the Java Naming

Conventions number 001. During manual (or even automatic) auditing reviews you can use this reference within your own audit report.

## Typographical Conventions

Class names are displayed like `ClassName`, method names like `methodName`, variables like `variableName` and literals like `literal`.

The Java code examples included in the rules are kept small, often not including the `JavaDoc` tags, to keep the examples readable and short.

Directory separators are displayed as `{ / }`

## Post Your Feedback

Every main section in this document has a feedback link. By clicking on the *Feedback* link you can email your comments, suggestions or ideas on the selected section or rule. If no feedback link is available you can always mail your input to `<feedback@jjguidelines.dev.java.net>`.

All the feedback emails are archived on the JJGuidelines mailing list at <https://jjguidelines.dev.java.net/servlets/SummarizeList?listName=Feedback>.

This document is hosted on the collaborative *java.net* community web site under the project name *JJGuidelines* (<https://jjguidelines.dev.java.net/>). On this web site you can post your comments in the discussion forums, submit issues or subscribe to the mailing list to receive notification emails when a new version of the document is available.

## Example Application

The code examples used in this document can be downloaded from the JJGuidelines web site at <https://jjguidelines.dev.java.net/>.

In a second phase we'll include a complete J2EE application that will demonstrate all the mentioned conventions, guidelines and best practices. This application can be a guide for developers just starting with J2EE or can be used by the more experienced developers as a source of inspiration.

## More Information

For the people reading this document in HTML, a PDF version is also available and can be downloaded from the following link: <https://jjguidelines.dev.java.net/servlets/ProjectDocumentList?folderID=267>.

In Bibliography you'll find more information on very interesting books and articles that were consulted.

---

# Chapter 1. Conventions

## Feedback

***[mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]conventions]***

## Why Use Conventions?

Coding, naming and documentation conventions are important to a company 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.
- These conventions will hopefully also be a helping hand for developers new to the Java language and the J2EE platform.
- 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.

So looking at the above advantages it's like a no-brainer to do this, right ?

## Convention Rules

### Defining

In order for to be able to determine if conventions are followed rules must be defined. All convention rules can be found in Appendix A, Conventions Rules . 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 document:

**Table 1.1. Severity**

Severity	Impact
Enforced	This priority is the same like ' <i>High</i> ' but the related rule can be enforced through the use of an auditing tool (see the section called "Enforcing").
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 ' <i>High</i> ' priority can not (yet) be inforced 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.

### Enforcing

This document will not only define the different Java and J2EE conventions. We'll use open-source auditing tools to automatically enforce important rules. So when you deliver your J2EE project you can include the auditing reports showing that your implementation is at least compliant to the described *enforced* conventions. In the current release not all high rules can be enforced yet by means of one tool. In time, hopefully soon, and with the help of the java community, more *high* rules will make it to the status *enforced*.



### Note

The current version of JJGuidelines only uses checkstyle [<http://checkstyle.sourceforge.net>] as its main auditing tool. In future release other tools (like PMD [<http://pmd.sourceforge.net/>]) will be included. All rules marked as *enforced* are covered by checkstyle

## Checkstyle

Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. It automates the process of checking Java code to spare humans of this boring (but important) task. This makes it ideal for projects that want to enforce a coding standard.

Checkstyle is highly configurable and can be made to support almost any coding standard. The use of checkstyle in combination with the JJGuidelines conventions and guidelines is described in the Appendix C, CheckStyle

## Naming Conventions

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]naming](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]naming)]

## Introduction

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.

Of course we need to make sure that this really happens we'll need an automated way to verify that the naming conventions are followed. In this section (and others) you'll find references to convention rules that can be automatically enforced by tools.



### Tip

Sun Microsystems has recently started a naming conventions document for enterprise applications. This document covers all of them plus many others. More information can be found at <http://java.sun.com/blueprints/code/namingconventions.html>.

## Java Naming

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]namingJava](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]namingJava)]

## Package Name

Construct your base package name based on the customer's domain name but reversed. The domain name is the main identifying part of a web address (without the 'www' prefix). So for example if the web address of the customer is *www.vlaanderen.be* then the domain name is *vlaanderen.be*, so all packages for this customer should start with *be.vlaanderen*. The project name and sub packages are appended to the reversed domain name.



```
package be.vlaanderen.myproject.subpackage;
```

The project name and sub packages follow the reversed domain name.



### Note

A package name should be all lowercase characters. Also the package name can not start with *java* or *sun*. See rule JAN\_007: Use A Correct Name For A Package (Enforced).

## Classes

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).

For example:

```
public class HelloWorld {  
}  
  
public class SocketHandlerThread {  
}
```



### Note

Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Also known as the CamelNotation. See rule JAN\_003: Use A Correct Name For A Class Or Interface (Enforced).



### Note

Many of the code examples in this document will use the class name `Foo` and the method name `bar`.

## Interfaces

Interface names should be capitalized like class names. Don't use special prefixes or suffixes to specify interfaces (e.g.: don't use `IMyInterface`). Treat interface names the same as you do with classes.

For example:

```
public interface RasterDelegate {  
}
```



### Note

Interface names should be nouns, in mixed case with the first letter of each internal word capitalized. See rule JAN\_003: Use A Correct Name For A Class Or Interface (Enforced).

## Methods

Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each inter-

nal word capitalized. See rule JAN\_006: Use A Correct Name For A Method (Enforced) and also the *Getter* and *Setter* methods.

For example:

```
public void run() {
}

public boolean isRunning() {
}

public Color getBackground() {
}

public void setBackground(Color color) {
}
```

## Getters And Setters Methods

Feedback

[mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]namingJava.gettersAndSetters]

The JavaBeans specification defines a standard way in which the properties for a JavaBean instance should be accessed. (The JavaBeans specification can be found at <http://java.sun.com/products/javabeans/docs/spec.html>)

This same technique can also be applied to regular classes and interfaces to access their attributes.

The methods are divided into *accessor* and *mutator* methods:

Getter (Accessor)	Get an attribute/property value
Setter (Mutator)	Set an attribute/property value

The default name for a getter method is `getXXX()` ;

and for a setter method `setXXX()` ;

Where *XXX* represents the name of the attribute/property.

The name of a getter/setter method can be different depending on the type of attribute/property it gets or sets.

## Boolean properties

Getter: `getXXX()` OR `isXXX()` OR `hasXXX()` OR `canXXX()`

Setter: `setXXX(boolean value)`

For example:

```
boolean enabled;
isEnabled();
setEnabled(true);
```

## Indexed properties

Getter: `getXXX(int index)`

Setter: `setXXX(int index, value)`

For example:

```
ArrayList name = new ArrayList();  
name.get(0);  
name.set(0, "value");
```

## Variables

Variable names should be short yet meaningful. Non-final variables must start with lower-case characters and the internal words start with capital letters. See rule JAN\_004: Use A Correct Name For A Non Final Field (Enforced).

Variable names should not start with underscore `_` or dollar sign `$` characters, even though both are allowed. See rule JAN\_010: Do Not Use \$ In A Name (Enforced).

One-character variable names should be avoided except for temporary *throwaway* variables. See rule JAN\_009: Use A Conventional Variable Name When Possible (Normal).

Common names for temporary variables are:

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

Avoid variables names that only differ in case which makes it difficult for the reader to distinguish the variables. See rule JAN\_011: Do Not Use Names That Only Differ In Case (High).

Don't use the *Hungarian Notation* for any of your variable names.



### Note

In the early days of DOS, Microsoft's Chief Architect Dr. Charles Simonyi introduced an identifier naming convention that adds a prefix to the identifier name to indicate the functional type of the identifier. It came to be known as "Hungarian notation" because the prefixes make the variable names look a bit as though they're written in some non-English language and because Simonyi is originally from Hungary.

## Constants

The names of variables declared as class constants (final and static) should be all uppercase with words separated by under-scores `_`. See rule JAN\_005: Use A Correct Name For A Constant (Enforced).

You can define constants on the class or interface that *owns* them or use one or more separate `xxxConstants` classes to bundle them.



### Caution

Before defining something as a constant make sure that it really is a constant and not a configurable property or setting.

For example declaring constants on a class that owns them:

```
package be.vlaanderen.examples;

public class ServiceLocator {

    /** The JNDI name of the session EJB */
    public static final String JNDI_NAME = "sessionBean";

    /** The default J2EE Reference Implementation URL
     * If possible move this information to an LDAP storage
     * or (if applicable) place it in the J2EE deployment descriptor
     * as an environment variable.
     */
    public static final String JNDI_URL = "iiop://localhost:1050";

    // ...
}
```

For example a `xxxConstants` class:

```
package be.vlaanderen.examples;

public final class JNDIConstants {

    /** The JNDI name of the session EJB */
    public static final String JNDI_NAME = "sessionBean";

    /** The default J2EE Reference Implementation URL
     * If possible move this information to an LDAP storage
     * or (if applicable) place it in the J2EE deployment descriptor
     * as an environment variable.
     */
    public static final String JNDI_URL = "iiop://localhost:1050";

    /**
     * Hide constructor, constant classes should never be instantiated
     */
    private JNDIConstants() {
    }
}
```



### Note

From JDK 1.5 onwards you'll be able to import your Constant variables through the use of 'static imports'. For example:

```
import static be.vlaanderen.examples.JNDIConstants.JNDI_NAME;

// ...

public class ServiceLocator {

    // ...

    public Object lookup() throws NamingException {
        return context.lookup(JNDI_NAME);
    }
}
```

## Exceptions

The names of Exception classes (classes that extend from `java.lang.Exception` or `java.lang.Throwable`) should always end with `Exception`. See rule JAN\_008: Name An Exception Class Ending With Exception (High).

For example:

```
public class PersonNotFoundException extends Exception {
    // ...
}
```



### Note

Try to be as descriptive as possible when choosing the name of your exception class because often only the name is used to determine the way the exceptional condition should be handled.

## J2EE Naming

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]namingJ2ee](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]namingJ2ee)]

### Application Name

A J2EE application is delivered in an Enterprise Archive (EAR) file. An EAR file is a standard Java Archive (JAR) file. The Application name must have a `.ear` extension and written in all-lowercase letters.

For example:

```
myproject.ear
```

The display name within the deployment descriptor is the application name, written in mixed cases, with a suffix `EAR`. See rule JEN\_013: Use A Correct Name For An Enterprise Application Display Name (High).

For example:

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

**Tip**

More information on how to develop J2EE applications can be found in the book *The J2EE 1.4 Tutorial* by Sun Microsystems.

## Web Module Name

Feedback

[mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]namingJ2ee.webModule]

Web modules contain JSP files, class files for servlets, GIF and HTML files, and a web deployment descriptor. Web modules are packaged as JAR files and must have a `.war` (Web ARchive) extension and written in all lower-case letters.

For example:

```
mywebproject.war
```

The display name within the deployment descriptor is the web module name, written in mixed cases, with a suffix `WAR`. See rule JEN\_014: Use A Correct Name For A Web Module Display Name (High).

For example:

```
<display-name>MyWebProjectWAR</display-name>
```

## Servlet Name

To clearly identify a servlet we'll suffix the servlet class name with `Servlet`, for example `BookStoreServlet`. See rule JEN\_008: Name A Servlet Like [Name]Servlet (High).

The servlet display name within the deployment descriptor is the servlet class name.

## Filter Name

As of version 2.3 of the Servlet API, a standard way to define filters has been specified. The filtering API is defined by the `Filter`, `FilterChain`, and `FilterConfig` interfaces in the `javax.servlet` package. You define a filter by implementing the `Filter` interface. The name of any filter must have the suffix 'Filter', for example `BasicDecodeFilter`. See rule JEN\_010: Name A Filter Servlet Like [Name]Filter (High).

The filter display name within the deployment descriptor is the filter class name.

## Enterprise Java Beans

Because Enterprise Java Beans are composed of multiple parts, it's useful to follow a naming convention for your J2EE applications. The table below summarizes the conventions for the Enterprise Java Beans.

(DD) means that the item is an element in the Enterprise Java Bean's deployment descriptor.

### Table 1.2. Naming Conventions For Enterprise Java Beans

Item	Syntax	Example	Rule
EJBJAR display name (DD)	[Name]JAR	ProductJAR	JEN_007: Name An EJB Display Name In The Deployment Descriptor Like [Name]JAR (High)
Enterprise Java Bean display name (DD)	[Name]EJB	ProductEJB	JEN_006: Name An EJB In The Deployment Descriptor Like [Name]EJB (High)
Enterprise bean class	[Name]EJB or [Name]Bean	ProductEJB or ProductBean	JEN_001: Name An EJB Bean Class Like [Name]EJB or [Name]Bean (High)
Home interface	[Name]Home	ProductHome	JEN_002: Name An EJB Remote Home Interface Like [Name]Home (High)
Remote interface	[Name]	Product	JEN_003: Name An EJB Remote Interface Like [Name] (High)
Primary Key	[Name]PK	ProductPK	JEN_009: Name A Primary Key Class Like [Name]PK (High)
Local home interface	[Name]LocalHome	ProductLocalHome	JEN_004: Name An EJB Local Home Interface Like [Name]LocalHome (High)
Local interface	[Name]Local	ProductLocal	JEN_011: Name A Local Interface Like [Name]Local (High)
JMS Queue destination	[Name]Queue	productQueue	JEN_017: Use A Correct Name For A JMS Environment Reference Name (High)
JMS Topic destination	[Name]Topic	productTopic	JEN_017: Use A Correct Name For A JMS Environment Reference Name (High)
Transferable (Value) Object	[Name]TO	ProductTO	JEN_005: Name A Transfer Object Like [Name]TO (High)



### Note

Use of the 'Bean' suffix for the enterprise bean class name can lead to developers thinking they are dealing with a standard JavaBean and not an EJB. Use of the 'EJB' suffix clearly states that the class represents an enterprise java bean implementation.



### Note

The Transferable Object (TO) was previously known as Value Object (VO).

## Resource Adapter

Resource adapter modules contain all Java interfaces, classes, native libraries and other documentation, along with the resource adapter deployment descriptor. Together, these implement the Connector architecture for a particular EIS. Resource adapter modules are packaged as JAR files and must have a .rar (Resource adapter ARchive) extension.

## Resource Environment References

Feedback

[mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]namingJ2ee.resourceReferences]

Resource environment references are logical names associated to different types of administered objects (for example a JMS destination) configured within the deployment descriptor. The developer uses this

resource environment reference to lookup and bind to that specific administered object. Below you'll find the naming conventions for the different types of resources.

## Enterprise Java Beans

All deployment descriptor references to Enterprise Java Beans must be organized in the `ejb` subcontext of the application component's environment. See rule JEN\_015: Use A Correct Name For An EJB Environment Reference Name (High).

For example:

```
<ejb-ref>
  <description>
    This is a reference to the entity bean that
    encapsulates access to Foo.
  </description>
  <ejb-ref-name>ejb/Foo</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>be.vlaanderen.FooHome</home>
  <remote>be.vlaanderen.Foo</remote>
</ejb-ref>
```

## Java Messaging Service

All deployment descriptor references to Java Messaging Service's must be organized in the `jms` subcontext of the application component's environment. See rule JEN\_016: Name A JMS Destination Like [Name]Queue Or [Name]Topic (High).

For example:

```
<resource-env-ref>
  <description>
    This is a reference to a JMS queue
  </description>
  <resource-env-ref-name>jms/FooQueue</resource-env-ref-name>
  <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
</resource-env-ref>
```

## JDBC

All deployment descriptor references to JDBC related objects must be organized in the `jdbc` subcontext of the application component's environment. See rule JEN\_018: Use A Correct Name For A JDBC Environment Reference Name (High).

For example:

```
<resource-ref>
  <description>
    A data source for a Foo database
  </description>
  <res-ref-name>jdbc/FooDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

The above example uses a database name called `FooDB` following rule JEN\_019: Name A Database Like [Name]DB (High).



# Comments Conventions

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]comments]

## Java Comments

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]commentsJava]

### Introduction

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.

Discussion of nontrivial or not really obvious design decisions is appropriate, but avoid duplicating information that is present in the code. In general, avoid any comments that are likely to get out of date as the code evolves.

Java programs can have two kinds of comments: implementation comments and documentation comments.

- Implementation comments are those found in C++, which are delimited by `/* . . . */` and `//`.
- Documentation comments are meant to describe the specification of the code, from an implementation-free perspective to be read by developers who might not necessarily have the source code at hand. Documentation comments are Java-only, and are delimited by `/** . . . */`. These comments can be extracted to HTML files using the Javadoc tool.

### Implementation Comments

Implementation comments are means for commenting out code or for comments about the particular implementation.

Comments should not be enclosed in decorative large boxes drawn with asterisks or other characters. Also avoid special characters such as tabs and new lines.



#### Note

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

### Beginning comments

All source files should begin with a comment that lists the copyright notice. See rule JAD\_002: Provide A Correct File Comment For A File (High).

```
/*  
 * Copyright notice  
 *  
 */
```

### Documentation Comments

Feedback

[mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]commentsJava.javadocComments]

## What Is Javadoc?

The Javadoc tool parses the declarations and documentation comments in a set of Java source files and produces a corresponding set of HTML pages describing the `public` and `protected` classes, nested classes, interfaces, constructors, methods, and fields. You can use it to generate the API documentation or the implementation documentation for a set of source files.

You can run the Javadoc tool on entire packages, individual source files, or both.

## Format Of A Document Comment

A doc comment is written in HTML and must precede a class, field, constructor or method declaration. It is made up of two parts: a description followed by block tags. In this example, the block tags are `@param`, `@return`, and `@see`.

For example:

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The URL argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 * <p>
 *
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url argument
 * @return the image at the specified URL
 * @see Image
 */
public Image getImage(URL url, String name) {
    // ...
}
```

- The first line starts with the document-comment delimiter `/**`.
- Starting with Javadoc 1.4, the leading asterisks are optional.
- Each line above is indented to align with the code below the comment.
- Write the first sentence as a short summary of the method, as Javadoc automatically places it in the method summary table and index.
- If you have more than one paragraph in the doc comment, separate the paragraphs with a `<p>` paragraph tag, as shown.
- Insert a blank comment line between the description and the list of tags, as shown above. There is only one description block per comment; you cannot continue the description following block tags.
- The last line contains the end-comment delimiter `*/`. The end document comment contains only a single asterisk.
- Lines won't wrap, limit any document comment line to 80 characters. See rule JAC\_004: Do Not

Make A Line Longer Than 120 Characters (Normal).



### Tip

Please read the very interesting document from Sun Microsystems describing the style guide and Javadoc conventions used in the documentation comments for Java programs (<http://java.sun.com/j2se/javadoc/writingdoccomments/index.html>).



### Tip

You can always use Jalopy, an open source code formatter for the Sun Java programming language. It layouts any valid Java source code according to configurable rules. You can download an IDE plug-in from <http://jalopy.sourceforge.net/index.html>.

## Tag Comments

Feedback

[[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]commentsJava.javadocComments.tagComments](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]commentsJava.javadocComments.tagComments)]

The rule JAD\_003: Provide A Javadoc Comment For A Class (Enforced) checks whether Javadoc comments are provided for classes, interfaces, methods and attributes. In the list below you'll find the required and optional Javadoc tags.

### @author (Required)

Adds an *Author* entry with the specified name-text to the generated JavaDocs.

You can provide one or multiple @author tags. For example:

```
/**
 * ...
 * @author firstName lastName - companyName
 */
```



### Note

You need to use the Javadoc option `-author` to get the author information in the generated JavaDocs.

### @version (Required)

This Javadoc tag adds a *Version* subheading with the specified version-text-date to the generated docs

This tag is intended to hold the current version number of the software that this code is part of and the date from the last source check-in.

The version-text-date has no special internal structure but is dependent on the used versioning system. Below you can find an example for CVS:

```
/**
 * ...
 * @version $Revision: 1.2 $ $Date: 2004/02/25 14:21:03 $
```

\* /



## Note

You need to use the Javadoc option `-version` to add the version information in the generated Javadocs.

### @param (Required)

Adds a parameter to the *Parameters* section. The description may be continued on the next line. This tag is valid only in a doc comment for a method or constructor.

The `@param` tag is followed by the name (not data type) of the parameter, followed by a description of the parameter. By convention, the first noun in the description is the data type of the parameter. (Articles like *a*, *an* and *the* can precede the noun.) An exception is made for the primitive `int`, where the data type is usually omitted.

Additional spaces can be inserted between the name and description so that the descriptions line up in a block. Dashes or other punctuation should not be inserted before the description, as the Javadoc tool inserts one dash.

Parameter names are lowercase by convention. The data type starts with a lowercase letter to indicate an object rather than a class. The description begins with a lowercase letter if it is a phrase (contains no verb), or an uppercase letter if it is a sentence. End the phrase with a period only if another phrase or sentence follows it.

For example:

```
/**
 * ...
 * @param c the character to be tested
 * @param observer the image observer to be notified
 */
```

Do not bracket the name of the parameter after the `@param` tag with `<code>...</code>` since Javadoc 1.2 and later automatically do this. (Beginning with 1.4, the name cannot contain any HTML, as Javadoc compares the `@param` name to the name that appears in the signature and emits a warning if there is any difference.)

### @return (Required)

Adds a *Returns* section with the description text. This text should describe the return type and permissible range of values. This tag is valid only in a doc comment for a method.

Having an explicit `@return` tag makes it easier for someone to find the return value quickly. Whenever possible, supply return values for special cases (such as specifying the value returned when an out-of-bounds argument is supplied).

### @throws (Required)

A `@throws` tag should be included for any checked exceptions (declared in the `throws` clause), as illustrated below, and also for any unchecked exceptions that the caller needs to be aware of (for example `java.lang.IllegalArgumentException`).

```
/**
 * ...
```

```
*
* @throws IOException If an input or output exception occurred
*/
public void bar() throws IOException {
    // body
}

/**
 * ...
 * @throws IllegalArgumentException If <code>name</code> is <code>null</code>
 */
public void setName(String name) {
    if (name == null) {
        throw new IllegalArgumentException("name can not be null");
    }
    // ...
}
```

**Tip**

Multiple @throws tags should be listed alphabetically by the exception names.

**Tip**

Add multiple @throws tags for the same exception if it can be thrown under different circumstances instead of describing all circumstances in a single @throws tag.

**@see (Optional)**

Adds a *See Also* heading with a link or text entry that points to reference. A doc comment may contain any number of @see tags, which are all grouped under the same heading.

In the example below an @see tag refers to the equals method in the `String` class. The tag includes both arguments: the name `String#equals(Object)` and the label `equals`.

```
/**
 * ...
 * @see String#equals(Object) equals
 */
```

The standard Doclet produces HTML something like this:

```
<dl>
<dt><b>See Also:</b>
<dd><a href="../../../java/lang/String#equals(java.lang.Object)">
<code>equals</code></a>
</dd>
</dl>
```

**@deprecated (Optional)**

Adds a comment indicating that this API should no longer be used (even though it may continue to work). The Javadoc tool moves the deprecated-text ahead of the main description, placing it in italics and preceding it with a bold warning: *Deprecated*. This tag is valid in all doc comments: overview,

package, class, interface, constructor, method and field.

The `@deprecated` description in the first sentence should at least tell the user when the API was deprecated and what to use as a replacement. Only the first sentence will appear in the summary section and index. Subsequent sentences can also explain why it has been deprecated.

```
/**
 * ...
 *
 * @deprecated As of JDK 1.1, replaced by #setBounds(int,int,int,int)}
 */
```

## A JavaDoc Comments Example

```
/**
 * Graphics is the abstract base class for all graphics contexts
 * which allow an application to draw onto components realized on
 * various devices or onto off-screen images.
 * A Graphics object encapsulates the state information needed
 * for the various rendering operations that Java supports. This
 * state information includes:
 * <ul>
 * <li>The Component to draw on
 * <li>A translation origin for rendering and clipping coordinates
 * <li>The current clip
 * <li>The current color
 * <li>The current font
 * <li>The current logical pixel operation function (XOR or Paint)
 * <li>The current XOR alternation color
 * (see <a href="#setXORMode">setXORMode</a>)
 * </ul>
 * <p>
 * Coordinates are infinitely thin and lie between the pixels of the
 * output device.
 * Operations which draw the outline of a figure operate by traversing
 * along the infinitely thin path with a pixel-sized pen that hangs
 * down and to the right of the anchor point on the path.
 * Operations which fill a figure operate by filling the interior
 * of the infinitely thin path.
 * Operations which render horizontal text render the ascending
 * portion of the characters entirely above the baseline coordinate.
 * <p>
 * Some important points to consider are that drawing a figure that
 * covers a given rectangle will occupy one extra row of pixels on
 * the right and bottom edges compared to filling a figure that is
 * bounded by that same rectangle.
 * Also, drawing a horizontal line along the same y coordinate as
 * the baseline of a line of text will draw the line entirely below
 * the text except for any descenders.
 * Both of these properties are due to the pen hanging down and to
 * the right from the path that it traverses.
 * <p>
 * All coordinates which appear as arguments to the methods of this
 * Graphics object are considered relative to the translation origin
 * of this Graphics object prior to the invocation of the method.
 * All rendering operations modify only pixels which lie within the
 * area bounded by both the current clip of the graphics context
 * and the extents of the Component used to create the Graphics object.
 * <p>
 *
 * @author Sami Shaio - SunMicrosystems
 * @author Arthur van Hoff - SunMicrosystems
 * @version $Revision: 1.2 $ $Date: 2004/02/25 14:21:03 $
```

```

*/
public abstract class Graphics {

    /**
     * Draws as much of the specified image as is currently available
     * with its northwest corner at the specified coordinate (x, y).
     * This method will return immediately in all cases, even if the
     * entire image has not yet been scaled, dithered and converted
     * for the current output device.
     * <p>
     * If the current output representation is not yet complete then
     * the method will return false and the indicated {@link ImageObserver}
     * object will be notified as the conversion process progresses.
     *
     * @param img        the image to be drawn
     * @param x          the x-coordinate of the northwest corner of the
     *                  destination rectangle in pixels
     * @param y          the y-coordinate of the northwest corner of the
     *                  destination rectangle in pixels
     * @param observer    the image observer to be notified as more of the
     *                  image is converted. May be <code>null</code>
     * @return           <code>true</code> if the image is completely
     *                  loaded and was painted successfully;
     *                  <code>false</code> otherwise.
     * @see             Image
     * @see             ImageObserver
     */
    public abstract boolean drawImage(Image img, int x, int y,
                                     ImageObserver observer);

    /**
     * Dispose of the system resources used by this graphics context.
     * The Graphics context cannot be used after being disposed of.
     * While the finalization process of the garbage collector will
     * also dispose of the same system resources, due to the number
     * of Graphics objects that can be created in short time frames
     * it is preferable to manually free the associated resources
     * using this method rather than to rely on a finalization
     * process which may not happen for a long period of time.
     * <p>
     * Graphics objects which are provided as arguments to the paint
     * and update methods of Components are automatically disposed
     * by the system when those methods return. Programmers should,
     * for efficiency, call the dispose method when finished using
     * a Graphics object only if it was created directly from a
     * Component or another Graphics object.
     *
     * @see             #create(int, int, int, int)
     * @see             #finalize()
     * @see             Component#getGraphics()
     * @see             Component#paint(Graphics)
     * @see             Component#update(Graphics)
     */
    public abstract void dispose();

    /**
     * Disposes of this graphics context once it is no longer referenced.
     *
     * @see             #dispose()
     */
    public void finalize() {
        // ...
    }
}

```

## Documentation Checking Tool

Sun has developed a tool for checking JavaDoc comments, called the Sun Doc Check Doclet, or DocCheck. You run it on your source code and it generates a report describing what style and JavaDoc errors the comments have, and recommends changes.

You can download a free copy of the DocCheck tool from <http://java.sun.com/j2se/javadoc/doccheck/index.html>.

### The DocCheck report

The DocCheck report contains the following parts:

- The executive summary shows the number of missing comments and the percentage for package groups (e.g. `java.awt.*`). Also shows the number of minor errors for packages that have no major comment errors.
- The package summary shows the number of missing comments and the number of minor errors, along with percentages by package.
- The package statistics shows the kinds of errors found in each package, the number and kind of items that were inspected (classes, methods, etc.), and provides a statistical breakdown of the errors.

## Coding Conventions

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]coding](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]coding)]

### Introduction

There are so many very good Java and J2EE references available on the internet these days that we've decided not to rewrite how you should develop your first Java application or deploy a J2EE project. Instead we'll start with some great on-line references which will definitely help you in your Java quest.

In the sections below you'll also find some J2EE examples based on the coding and naming conventions described in Appendix A. Please re-use these examples as templates for your Java related projects. The complete J2EE source code, including the unit tests, can be downloaded from <https://jguidelines.dev.java.net/servlets/ProjectDocumentList?folderID=287>.

### Java Coding

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]codingJava](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]codingJava)]

### References

Before an author can write about a Java API, the API has gone through the Java Community Process (JCP) as a Java Specification Request (JSR). It's on the JCP website <http://www.jcp.org> that you can get your first contact with a potential J2EE API. You can download the specification once the JSR is in public draft, and that's also where the different authors (often already involved in the JSR expert group) start writing more reader friendly pages.

Some interesting Java specifications are:

- JSR-901: Java Language Specification, Second Edition [<http://www.jcp.org/en/jsr/detail?id=901>] includes all changes, clarifications and amendments made since the publication of the first edition of the language specification in 1996. See also JLANGSPEC .



- JSR-059: J2SE (1.4) Merlin Release [<http://www.jcp.org/en/jsr/detail?id=59>] defines the major features of the J2SE 1.4 release.
- JSR-041: A simple Assertion Facility [<http://www.jcp.org/en/jsr/detail?id=41>] introduces a new Java keyword allowing programmers to include assertions which can be enabled as programs execute to detect development bugs.
- JSR-047: Logging API Specification [<http://www.jcp.org/en/jsr/detail?id=47>] defines a standard API for error and trace logging.
- JSR-010: Preferences API Specification [<http://www.jcp.org/en/jsr/detail?id=10>] allows Java programs to manipulate user preference and configuration data.

If you don't like reading the core specifications you can always download some great on-line java books. My personal favorite is the Thinking in Java book [<http://www.bruceeckel.com>] from Bruce Eckel, which I recommend to any person starting with the Java language. Below you can find a few others:

- Essentials of the Java Programming Language, Part 1  
[<http://developer.java.sun.com/developer/onlineTraining/Programming/BasicJava1/>]
- Essentials of the Java Programming Language, Part 2  
[<http://developer.java.sun.com/developer/onlineTraining/Programming/BasicJava2/>]
- Advanced Programming for the Java 2 Platform  
[<http://developer.java.sun.com/developer/onlineTraining/Programming/JDCBook/>]
- The Sun Microsystems Tutorials and Short Courses  
[<http://developer.java.sun.com/developer/onlineTraining/Programming/JDCBook/>]

## Java Source Files

Feedback

[[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]codingJava.javaSourceFiles](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]codingJava.javaSourceFiles)]

Each Java source file contains a single `public` class or interface.

Java source files have the following ordering:

- Beginning file comments
- Package and `import` statements
- Class and interface declarations



### Note

Avoid too long Java files. According to the Sun Coding Conventions for Java, files longer than 2000 lines are cumbersome and should be avoided. See rule JAC\_003: Do Not Make A File Longer Than 2000 Lines (Enforced).

## Package And `import` Statements

All source files should start with a copyright notice. See rule JAD\_002: Provide A Correct File Comment For A File (High). The first non-comment line of a Java source file is the package statement. After that, import statements can follow.

For example:

```
/*
 * Copyright notice
 *
 */
package be.vlaanderen.projectname.subpackage;
```



### Note

Wildcard import-declarations must be replaced by a list of single import-declarations, meaning that import statements should not end with an asterisk. See rule JAC\_010: Do Not Use A Demand Import (Enforced).

## Class And Interface Declaration

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

**Table 1.3. Parts Of A Class Or Interface Declaration Notes**

Part Of A Class Or Interface	Declaration Notes
Class/Interface documentation comment with javadoc tags	This (javadoc) comment will contain more information about the class or interface but also the javadoc author and version information.
Class or interface statement	
Member and static variables	First the <code>public</code> class variables, then the <code>protected</code> , then package level (no access modifier), and then the <code>private</code> .
Instance variables	First the <code>public</code> , then the <code>protected</code> , then package level (no access modifier), and then the <code>private</code> .
Constructors	The constructor(s) are placed here.
Methods	These methods should be grouped by functionality rather than by scope or accessibility. For example, a <code>private</code> class method can be in between two <code>public</code> instance methods. The goal is to make reading and understanding the code easier.

For example:

```
/**
 * You can place more information about the class or interface here.
 *
 * Just add an additional javadoc author tag If more than one developer
 * has worked on the same class or Interface.
 *
 * @author firstName lastName - companyName
 * @version $Revision: 1.4 $   $Date: 2004/02/25 14:21:03 $
 */
public class MySingleton {
```

```

/**
 * The instance variable is static so we don't have
 * to make the getInstance method synchronized.
 */
private static final MySingleton INSTANCE = new MySingleton();

/**
 * More information about the constructor here.
 */
private MySingleton() {

/**
 * More information about the method here.
 *
 * @return The singleton reference
 */
public static MySingleton getInstance() {
    return INSTANCE;
}

/**
 * More information about the method here.
 *
 * @param value    A value that needs to be processed.
 */
public void doSomething(int value) {
}
}

```

## Change History (Optional)

Source files could end with an overview of the changes made to the source. The change history can be extracted from your version control system and added to the source file by adding the following comment block.

```

/*
 * $Log: codingJava.xml,v $
 * Revision 1.4  2004/02/25 14:21:03  gde
 * updated all feedback links
 *
 * Revision 1.3  2004/02/24 13:54:29  gde
 * removed title id
 *
 * Revision 1.2  2004/02/17 15:04:29  gde
 * more bib links
 *
 * Revision 1.1  2004/02/16 17:37:15  gde
 * First version with modulation
 *
 * Revision 1.16  2004/02/10 15:18:28  gde
 * fixed malformed sentence in JAC_027
 *
 * Revision 1.15  2004/02/10 15:02:34  gde
 * adjusted revision number
 *
 * Revision 1.14  2004/02/10 14:53:44  gde
 * fixed all small JAC issues
 *
 * Revision 1.13  2004/02/10 14:46:54  gde
 * finished methodname, classname, ...; fixed image problem, solves a few issues

```

```
*
* Revision 1.12  2004/02/09 18:18:23  gde
* fixed methodname/classname from conv.naming.j2ee till rules.conv.naming.java (i
*
* Revision 1.11  2004/02/06 18:06:04  gde
* all callouts done
*
* Revision 1.10  2004/02/06 17:18:01  gde
* AFTER callout for Servlet Example only
*
* Revision 1.9   2004/02/06 16:11:04  gde
* issues + programlisting no longer part of para
*
* Revision 1.8   2004/02/05 14:17:53  gde
* adjusted line numbering,
* before changing line numbering to callouts (if allowed)
*
* Revision 1.7   2004/02/05 13:46:36  gde
* before changing line numbering to callouts
*
* Revision 1.6   2004/02/04 14:01:00  gde
* working on table standarization
*
* Revision 1.5   2004/02/04 12:57:56  gde
* resolved several issues created by Stephan
*
* Revision 1.4   2004/02/03 16:53:52  gde
* bibliography's url are now ulinks
*
* Revision 1.3   2004/02/03 13:31:41  gde
* adjusted Overviews, links, xref
* Did not adjust severity links
*
* Revision 1.2   2004/02/03 09:16:05  dsc
* Stephan's merged version
*
* Revision 1.1   2004/02/02 20:13:10  sja
* Chapter One and Two merged + update biblio
*
* Revision 1.8   2004/02/01 10:32:36  dsc
* fixed issue 26
*
* Revision 1.7   2004/01/31 18:16:34  dsc
* continued development
*
* Revision 1.4   2004/01/29 09:50:14  sja
* Fixed issue #2567
*
* Revision 1.3   2004/01/28 10:47:10  sja
* Fixed issue #1245
*
* Revision 1.2   2004/01/27 16:37:15  sja
* continued development
*
* Revision 1.1   2004/01/27 01:07:44  sja
* creation
*/
```

## Methods

- Avoid too long methods ! A method should not be longer than 1 page (60 lines), if a method is

longer you can probably re-factor it to smaller methods resulting in more readable code. See rule JAC\_013: Do Not Make A Method Longer Than 60 Lines (Normal).

- A method should only have one point of entry and exit. So use only one return statement in your method making the method easier to read and debug. This convention will also allow you to easily introduce pre and post conditional assertions. See rule JAC\_016: Use A Single return Statement (Normal).
- Do not chain method calls, exceptionally a chain of 2 method calls can be used. This convention will avoid scenario's where one of the method returns null but because they're chained you don't know which one it is! See rule JAC\_015: Do Not Chain Multiple Methods (Normal).

## Wrapping Lines

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]Java Coding - wrapping lines]

Avoid lines longer than 80 characters since they're not handled well by many terminals and tools.

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

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

## Statements

Feedback  
[mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]codingJava.statements]

Every statement in the Java language has a preferred style and certain conventions that should be followed.

### Simple Statements

Each line should contain only one statement.

For example:

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

argv++; argc++; // AVOID
```



### Note

Don't place multiple statements on the same line. See rule JAC\_007: Do Not Place Multiple Statements On The Same Line (High).

## Compound Statements

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

- The enclosed statements should be indented one more level than the compound statement.
- The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.
- Braces are used around all statements, even single statements. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

## if else Statements

The if and else statements should have the following form:

```
if (condition) {
    statements;
}

if (condition) {
    statements;
} else {
    statements;
}

if (condition) {
    statements;
} else if (condition) {
    statements;
} else {
    statements;
}
```



### Note

if statements should always use { } braces. See rule JAC\_007: Do Not Place Multiple Statements On The Same Line (High).

## while Statements

A while statement should have the following form:

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

## do while Statements

A do while statement should have the following form:

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

## switch Statements

A switch statement should have the following form:

```
switch (condition) {  
    case ABC:  
        statements;  
        // falls through  
  
    case DEF:  
        statements;  
        break;  
  
    case XYZ:  
        statements;  
        break;  
  
    default:  
        statements;  
        break;  
}
```

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

Every switch statement should include a default case. See rule JAC\_011: Provide A default case In A switch Statement (Enforced). The break in the default case is redundant, but it prevents a fall-through error if later another case is added.



### Note

If the default case is empty then an *assertion* should be placed or an exception should be thrown. See rule ASSERT\_005: Use Assertions In A switch Statement's default Case Correct.



### Note

Use *constant* variables after each case instead of meaningless values.

From JDK 1.5 onwards enums will be supported within the Java Language this means the *constant* variables could be replaced with enum variables.

## try catch Statements

A try catch statement should have the following format:

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

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

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



### Note

Avoid empty catch blocks, as an alternative place a logging statement. See rule JAC\_038: Provide At Least One Statement In A catch Block (Normal).



### Note

A bad practice is catching a `java.lang.Exception` or `java.lang.Throwable`. See rule JAC\_039: Do Not Catch `java.lang.Exception` Or `java.lang.Throwable` (Normal).



### Tip

When opening a `Socket`, `Stream` or any other handler within the `catch` block make sure you close the handler within the `finally` block. See rule JAC\_068: Close A Connection Inside A finally Block (Enforced).

## J2EE Coding

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]codingJ2ee](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]codingJ2ee)]

### Introduction

In this section you'll find some interesting J2EE references and code examples following the different conventions. A full list of the convention rules are listed in Appendix A.

The complete J2EE source code, including the unit tests, can be downloaded from <https://jjguidelines.dev.java.net/servlets/ProjectDocumentList?folderID=287>.

### References

Some interesting J2EE 1.3 specifications are:

- JSR-058, Java 2 Platform, Enterprise Edition 1.3 Specification [<http://www.jcp.org/en/jsr/detail?id=58>] defines all the features of the J2EE version 1.3 release.
- JSR-019, Enterprise Java Beans 2.0 Specification [<http://www.jcp.org/en/jsr/detail?id=19>] introduces the integration with JMS but also improved support for entity bean persistence and relationships. The EJB query language for finder methods, additional home interface methods and support for server interoperability are also included in this release.
- JSR-053, Java Servlet 2.3 and JavaServer Pages 1.2 Specifications [<http://www.jcp.org/en/jsr/detail?id=53>] are platform independent Java based web components executed in a web server that generate dynamic content.
- JSR-914, Java Message Service API [<http://www.jcp.org/en/jsr/detail?id=914>] provides a common



way for Java programs to create, produce and consume messages in an asynchronous way.

- JSR-054, JDBC 3.0 Specification [<http://www.jcp.org/en/jsr/detail?id=54>] allows you to access a relational database.
- JSR-052, A standard Tag library for JavaServer Pages Specification [<http://www.jcp.org/en/jsr/detail?id=52>] helps JSP authors simplify their web pages.
- JSR-112, J2EE Connector Architecture 1.5 [<http://www.jcp.org/en/jsr/detail?id=112>] extends the existing 1.0 specification by adding new features like asynchronous integration with enterprise information systems (EIS) and JMS providers.

If the different J2EE specifications are written too cryptic for you, then point your browser to The Source for Java Technology: <http://java.sun.com>. This is the most complete web site covering the different Java platforms: J2ME, J2SE and J2EE. Below you can find some great on-line J2EE books that will help you step-by-step in developing your first J2EE application.

- The J2EE Developers Guide [[http://java.sun.com/j2ee/sdk\\_1.2.1/techdocs/guides/ejb/html/DevGuideTOC.html](http://java.sun.com/j2ee/sdk_1.2.1/techdocs/guides/ejb/html/DevGuideTOC.html)] helps you in how to develop and deploy EJBs.
- The J2EE 1.4 Tutorial (<http://java.sun.com/j2ee/1.4/docs/tutorial/doc/>)
- Designing Enterprise Applications with the J2EE Platform ([http://java.sun.com/blueprints/guidelines/designing\\_enterprise\\_applications\\_2e/titlepage.html](http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/titlepage.html))

You can also find very interesting on-line Java and J2EE related articles on the following sites:

- JavaWorld (<http://www.javaworld.com>)
- JGuru (<http://www.jguru.com>)
- The Server Side (<http://www.theserverside.com>)

## A Servlet Example

Below you can find an example of a Servlet implementing the Command and Controller Strategy.

```
/*
 * Copyright notice❶
 *
 */
package be.vlaanderen.examples.servlet;❷

import java.io.IOException;❸

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import be.vlaanderen.examples.common.Constants;
```

```

/**
 * This Servlet implements a simple Command and
 * Controller Strategy.
 *
 * @author Stephan Janssen - JCS Int.
 * @version $Revision: 1.3 $ $Date: 2004/03/15 15:08:27 $
 */
public class FooServlet extends HttpServlet {④

    /**
     * Catch the GET HTTP requests.
     *
     * @param req The HTTP servlet request
     * @param res The HTTP servlet response
     * @throws IOException thrown when an IO problem is encountered
     * @throws ServletException thrown when the Servlet encounters a difficulty
     */
    protected void doGet(HttpServletRequest req, HttpServletResponse res)
        throws IOException, ServletException {
        processRequest(req, res);
    }

    /**
     * Catch the PUT HTTP requests.
     *
     * @param req The HTTP servlet request
     * @param res The HTTP servlet response
     * @throws IOException thrown when an IO problem is encountered
     * @throws ServletException thrown when the Servlet encounters a difficulty
     */
    protected void doPut(HttpServletRequest req, HttpServletResponse res)
        throws IOException, ServletException {
        processRequest(req, res);
    }

    /**
     * Process all HTTP requests.
     *
     * @param req The HTTP servlet request
     * @param res The HTTP servlet response
     * @throws IOException thrown when an IO problem is encountered
     * @throws ServletException thrown when the Servlet encounters a difficulty
     */
    protected void processRequest(HttpServletRequest req,
                                HttpServletResponse res)
        throws IOException, ServletException {
        Command command = null;⑤
        String page = null;
        String param = null;
        param = req.getParameter(Constants.CMD);⑥

        if (null != param) {
            command = CommandFactory.createCommand(param);
            page = command.execute(req, res);
        } else {
            page = Constants.ERROR_PAGE;⑦
        }

        dispatch(req, res, page);
    }
}

```

```

    * Dispatch the request to the JSP or HTML page
    *
    * @param req The HTTP servlet request
    * @param res The HTTP servlet response
    * @param page The HTML or JSP page that will handle the view
    * @throws IOException thrown when an IO problem is encountered
    * @throws ServletException thrown when the Servlet encounters a difficulty
    */
    protected void dispatch(HttpServletRequest req, HttpServletResponse res,
                           String page)
        throws IOException, ServletException {
        ServletContext ctx = getServletContext();
        RequestDispatcher dispatcher = ctx.getRequestDispatcher(page);
        dispatcher.forward(req, res);
    }
}

```

The Servlet example is compliant to the following convention rules:

- Rule JAC\_004: Do Not Make A Line Longer Than 120 Characters (Normal)
- Rule JAC\_013: Do Not Make A Method Longer Than 60 Lines (Normal) and Rule JAC\_016: Use A Single return Statement (Normal).
- Rule JEC\_056: Do No Print Unnecessary Static Content From A Servlet (High)
- Rule JAD\_003: Provide A JavaDoc Comment For A Class (Enforced), rule JAD\_004: Provide A JavaDoc Comment For A Constructor (Enforced) and rule JAD\_007: Provide A JavaDoc comment For A Field (Enforced)
- Specifically:
  - ❶ Rule JAD\_002: Provide A Correct File Comment For A File (High)
  - ❷ Rule JAN\_007: Use A Correct Name For A Package (Enforced)
  - ❸ Rule JAC\_010: Do Not Use A Demand Import (Enforced) and JAC\_021: Do Not Import A Class Without Using It (Enforced)
  - ❹ Rule JAN\_001: Match A Class Name With Its File Name (High) and JEN\_008: Name A Servlet Like [Name]Servlet (High)
  - ❺ Rule JAC\_007: Do Not Place Multiple Statements On The Same Line (High) and JAC\_044: Explicitly Initialize A Local Variable (High)
  - ❻ Rule JAN\_005: Use A Correct Name For A Constant (Enforced)
  - ❼ Rule JAN\_005: Use A Correct Name For A Constant (Enforced)



### Tip

The above example shows a straight forward Servlet implementation of the Command and Controller strategy based on the Model 2 web-tier architecture.

Below you can find a simple Filter Servlet example.

```

/*
 * Copyright notice
 *
 */
package be.vlaanderen.examples.servlet;

import java.io.IOException;

```

```
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

import be.vlaanderen.examples.common.Constants;

/**
 * Example of a simple Filter that handles character encoding.
 *
 * @author Stephan Janssen - JCS Int.
 * @version $Revision: 1.3 $ $Date: 2004/03/15 15:08:27 $
 */
public class EncodingFilter implements Filter {

    /** Default encoding string */
    private String encoding = Constants.UTF8;

    /**
     * Initialization of the filter happens here.
     *
     * @param config The FilterConfig reference
     * @throws ServletException thrown when the Servlet encounters a problem
     */
    public void init(FilterConfig config) {
        String enc = config.getInitParameter(Constants.ENCODING);

        if (null != enc) {
            encoding = enc;
        }
    }

    /**
     * Dispatching specified request changing its parameter's character
     * encoding.
     *
     * @param req The ServletRequest reference
     * @param res The ServletResponse reference
     * @param chain The FilterChain reference
     * @throws IOException thrown when an IO problem is encountered
     * @throws ServletException thrown when the Servlet encounters a problem
     */
    public void doFilter(ServletRequest req, ServletResponse res,
        FilterChain chain) throws IOException, ServletException {
        req.setCharacterEncoding(encoding);
        chain.doFilter(req, res);
    }

    /**
     * Called when filter is destroyed
     *
     * @see javax.servlet.Filter#destroy()
     */
    public void destroy() {
    }
}
```

The Filter Servlet example is compliant to the following, not yet mentioned, convention rules:

- Rule JEN\_010: Name A Filter Servlet Like [Name]Filter (High)

**Tip**

See also JSPSPEC version 2.3 (or higher) - chapter 6 for more information on Servlet Filters.

## A Java ServerPage Example

Below you can find a simple JSP example that's using the JSP Tag Library.

```
<%@ taglib uri="/WEB-INF/jcs.tld" prefix="jcs" %>

<html>
  <body>
    <table>
      <jcs:employeeelist id="employee_list">
        <tr>
          <td><jcs:employee attribute="name"/></td>
          <td><jcs:employee attribute="profile"/></td>
        </tr>
      </jcs:employeeelist>
    </table>
  </body>
</html>
```

The JSP example is compliant to the following, not yet mentioned, convention rules:

- Rule JEC\_057: Use Custom JSP Tags Instead Of Scriptlets (High)
- Rule JEC\_058: Do Not Forward A Request From A JSP Page (High)

## Enterprise Java Bean Examples

Feedback

[mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]codingJ2ee.ejbExample]

### A Session Bean Example

Feedback

[mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]codingJ2ee.ejbExample.session]

### A Statefull Session Bean

Below you can find an example of a statefull session bean.

```
/*
 * Copyright notice
 *
 */
package be.vlaanderen.examples.session;

import java.util.logging.Logger;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

/**
 * A stateless session bean example
 */
```

```
* @author Stephan Janssen - JCS Int.
* @version $Revision: 1.3 $ $Date: 2004/03/15 15:08:27 $
*/
public class FooBean implements SessionBean {❶
    /** Initialize the logging API here and use it for any std. error output */
    private static final Logger logger =
        Logger.getLogger(FooBean.class.getName());

    /**
     * The session bean context variable
     */
    protected SessionContext ctx;

    /**
     * The constructor
     */
    public FooBean() {❷
    }

    /**
     * Sets the associated session context.
     *
     * @param ctx the session bean context
     */
    public void setSessionContext(SessionContext ctx) {
        this.ctx = ctx;
    }

    /**
     * The Session Bean create method
     */
    public void ejbCreate() {❸
    }

    /**
     * A container invokes this method before it ends the life
     * of the session object.
     */
    public void ejbRemove() {
    }

    /**
     * The passivate method is called before the instance
     * enters the "passive" state.
     */
    public void ejbPassivate() {
    }

    /**
     * The activate method is called when the instance is
     * activated from its "passive" state.
     */
    public void ejbActivate() {
    }

    /**
     * An example of a session business method
     *
     * @param value an example of a string parameter
     */
    public void bar(String value) {
        logger.info(value);❹
    }
}
```

```
    }  
}
```

The above example is compliant to the following, not yet mentioned, convention rules:

- Rule JEC\_017: Do Not Override The finalize Method For A EJB implementation class (High)
- Specifically:
  - ❶ Rule JEN\_001: Name An EJB Bean Class Like [Name]EJB or [Name]Bean (High), rule JEC\_013: Make All EJB interfaces and classes public (High) and rule JEC\_014: Do Not Make A EJB implementation class final (High)
  - ❷ Rule JEC\_016: Provide A public Default Constructor For A EJB implementation class (High)
  - ❸ Rule JEC\_015: Provide An ejbCreate Method For A Session Bean (High) and rule JEC\_018: Return void For An ejbCreate Method Of A Session Bean (High)
  - ❹ Rule JAC\_065: Do Not Unnecessary Use The System.out.print or System.err.print Methods (High)



### Tip

See also EJB20SPEC Chapter 7.10. for more details.

## The Home Interface

The Session home interface extends the `javax.ejb.EJBHome` interface and will define the create methods that a client can invoke. Every create method corresponds to an `ejbCreate` method in the session bean.

```
/*  
 * Copyright notice  
 */  
package be.vlaanderen.examples.session;  
  
import java.rmi.RemoteException;  
  
import javax.ejb.CreateException;  
import javax.ejb.EJBHome;  
  
/**  
 * An example of the Home Session interface  
 */  
 * @author Stephan Janssen - JCS Int.  
 * @version $Revision: 1.3 $ $Date: 2004/03/15 15:08:27 $  
 */  
public interface FooHome extends EJBHome {  
    /**  
     * A create example method  
     */  
     * @return the Bar object is returned  
     * @throws CreateException if the creation did not succeed  
     */  
    Foo create() throws CreateException, RemoteException;❶  
}
```

The above example is compliant to the following, not yet mentioned, convention rules:

- Specifically:

- ❶ Rule JEC\_019: Return The EJB Remote Interface For A create Method Of An EJB Remote Home Interface (Enforced), rule JEC\_020: Make A create Method Of An EJB Home Interface Throw An `javax.ejb.CreateException` (Enforced) and rule JEC\_055: Provide A Valid RMI Method Signature For An EJB Remote Home Interface (High)

## The Remote Interface

```
/*
 * Copyright notice
 *
 */
package be.vlaanderen.examples.session;

import java.rmi.RemoteException;

import javax.ejb.EJBObject;

/**
 * An example of the Remote Session interface
 *
 * @author Stephan Janssen - JCS Int.
 * @version $Revision: 1.3 $ $Date: 2004/03/15 15:08:27 $
 */
public interface Foo extends EJBObject {

    void bar(String value) throws RemoteException;
}
```

The Session remote interface extends the `javax.ejb.EJBObject` interface and will define the business methods that a client may invoke. The method definitions in a remote interface must follow these rules:

- Each method in the remote interface must match a method implemented in the enterprise bean class.
- The signatures of the methods in the remote interface must be identical to the signatures of the corresponding methods in the enterprise bean class.

## A Container Managed Persistency (CMP) Entity Example

Feedback

[mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]codingJ2ee.ejbExample.entity]

## Entity Bean Class

Below you can find an example of a simple container managed entity bean.

```
/*
 *
 * Copyright notice
 *
 */
package be.vlaanderen.examples.entity;

import java.util.logging.Level;
import java.util.logging.Logger;
```



```

import javax.ejb.CreateException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;

/**
 * A simple CMP Entity bean example
 *
 * @author Stephan Janssen - JCS Int.
 * @version $Revision: 1.3 $ $Date: 2004/03/15 15:08:27 $
 */
public abstract class FooBean implements EntityBean {❶
    /** Initialize the logging API here and use it for any std. error output */
    private static final Logger logger =
        Logger.getLogger(FooBean.class.getName());

    /** The entity context */
    private EntityContext context;

    /**
     * No-argument constructor
     */
    public FooBean() {❷
    }

    /**
     * Return the foo id
     * @return foo id
     */
    public abstract String getFooId();

    /**
     * Set the foo id
     *
     * @param fooId the foo id
     */
    public abstract void setFooId(String fooId);

    /**
     * Get the bar value
     *
     * @return returns the bar value
     */
    public abstract String getBar();

    /**
     * Set the bar value
     *
     * @param bar the given bar value
     */
    public abstract void setBar(String bar);

    /**
     * Create a bar record
     *
     * @param id the primary key
     * @param bar the bar value
     * @return return the primary key
     * @throws Create Exception
     */
    public String ejbCreate(String fooId, String bar) throws CreateException {

```

❸

```
        logger.info("FooBean.ejbCreate(" + fooId + "," + bar + ")");

        setFooId(fooId);
        setBar(bar);

        return fooId; ❹
    }

    /**
     * For each ejbCreate[METHOD](...) method, there is a matching
     * ejbPostCreate[METHOD](...) method that has the same input parameters
     * but whose return type is void.
     *
     * @param barId the primary key
     * @param bar the bar value
     * @throws Create Exception
     */
    public void ejbPostCreate(String fooId, String bar)
        throws CreateException { ❺
        logger.info("FooBean.ejbPostCreate(" + fooId + "," + bar + ")");
    }

    // business methods

    /**
     * Display the Bar value
     */
    public void doSomething() {
        logger.log(Level.INFO, getBar());
    }

    // life cycle methods

    /**
     * A container uses this method to pass a reference to the
     * EntityContext object.
     *
     * @param ctx the entity bean context
     */
    public void setEntityContext(EntityContext ctx) {
        context = ctx;
    }

    /**
     * A container invokes this method before terminating the life
     * of the instance.
     */
    public void unsetEntityContext() {
        context = null;
    }

    /**
     * Called in response to a client-invoked remove operation on
     * the entity bean's home or component interface or as the result
     * of a cascade-delete operation.
     */
    public void ejbRemove() {
        logger.info("FooBean.ejbRemove [" + getBar() + "]");
    }

    /**
     * When the container needs to synchronize the state of an enterprise
     * bean instance with the entity object's persistent state, the container
```

```

    * calls the ejbLoad() method.
    */
    public void ejbLoad() {
    }

    /**
     * When the container needs to synchronize the state of the entity
     * object's persistent state with the state of the enterprise bean
     * instance, the container first calls the ejbStore() method on the
     * instance.
     */
    public void ejbStore() {
    }

    /**
     * This method is called when the container decides to disassociate
     * the instance from an entity object identity, and to put the instance
     * back into the pool of available instances.
     */
    public void ejbPassivate() {
    }

    /**
     * This method is called when the container picks the instance from
     * the pool and assigns it to a specific entity object identity.
     */
    public void ejbActivate() {
    }
}

```

The above example is compliant to the following, not yet mentioned, convention rules:

- Rule JEC\_017: Do Not Override The finalize Method For A EJB implementation class (High)
- Specifically:
  - ❶ Rule JEN\_001: Name An EJB Bean Class Like [Name]EJB or [Name]Bean (High) and rule JEC\_013: Make All EJB interfaces and classes public (High)
  - ❷ Rule JEC\_016: Provide A public Default Constructor For A EJB implementation class (High)
  - ❸ Rule JEC\_024: Match An ejbPostCreate Method To An ejbCreate Method For An Entity Bean (High), rule JEC\_025: Declare An ejbCreate Method public For An Entity Bean (High), rule JEC\_027: Do Not Declare An ejbCreate Or ejbPostCreate Method As final Or static For An Entity Bean (High) and rule JEC\_043: Make An ejbCreate Method Of An Entity Bean Throw javax.ejb.CreateException (High)
  - ❹ Rule JEC\_026: Return The Primary Key For An ejbCreate[Name] Method Of An Entity Bean (High)
  - ❺ Rule JEC\_024: Match An ejbPostCreate Method To An ejbCreate Method For An Entity Bean (High), rule JEC\_027: Do Not Declare An ejbCreate Or ejbPostCreate Method As final Or static For An Entity Bean (High), rule JEC\_028: Return void For An ejbPostCreate Method Of An Entity Bean (High) and rule JEC\_036: Declae An ejbPostCreate Method Of An Entity Bean public (High)

## The Local Home Interface

```

/*
 *
 * Copyright notice
 *
 */
package be.vlaanderen.examples.entity;

```

```
import javax.ejb.CreateException;
import javax.ejb.EJBLocalHome;
import javax.ejb.FinderException;

/**
 * The Local Home interface for Enterprise Bean Foo
 *
 * @author Stephan Janssen - JCS Int.
 * @version $Revision: 1.3 $ $Date: 2004/03/15 15:08:27 $
 */
public interface FooLocalHome extends EJBLocalHome {❶

    /**
     * Creates an instance from a key for Entity Bean: Foo
     *
     * @param fooId the primary key
     * @param bar the bar value
     * @return the business interface Foo
     * @throws Create exception
     */
    FooLocal create(String fooId, String bar)
        throws CreateException;

    /**
     * Find by primary key
     *
     * @param fooId the primary key value
     * @return the business interface Foo
     * @throws Finder exception
     */
    FooLocal findByPrimaryKey(String fooId)
        throws FinderException;
}
```

The above example is compliant to the following, not yet mentioned, convention rules:

- Rule JEC\_038: Make A Method Of An EJB Remote Home Interface Throw `java.rmi.RemoteException` (Enforced)
- Specifally:
  - ❶ Rule JEN\_004: Name An EJB Local Home Interface Like `[Name]LocalHome` (High)

### The Local Interface

```
/*
 *
 * Copyright notice
 *
 */
package be.vlaanderen.examples.entity;

import javax.ejb.EJBLocalObject;

/**
 * The local interface for Enterprise Bean Foo
 *
 * @author Stephan Janssen - JCS Int.
 * @version $Revision: 1.3 $ $Date: 2004/03/15 15:08:27 $
 */
```

```
*/
public interface FooLocal extends EJBLocalObject { ❶

    /**
     * A Foo business method
     */
    void doSomething(); ❷
}
```

The above example is compliant to the following, not yet mentioned, convention rules:

- Specifically:

- ❶ Rule JEN\_011: Name A Local Interface Like [Name]Local (High)
- ❷ Rule JEC\_039: Do Not Make A Method Of An EJB Local Interface Throw java.rmi.RemoteException (Enforced)

## A Message Driven Bean Example

Message driven bean example:

```
/*
 * Copyright notice
 */
package be.vlaanderen.examples.messagebean;

import java.util.logging.Level;
import java.util.logging.Logger;

import javax.ejb.MessageDrivenBean;
import javax.ejb.MessageDrivenContext;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

/**
 * A simple Message Driven Bean example
 *
 * @author firstName lastName - companyName
 * @version $Revision: 1.3 $ $Date: 2004/03/15 15:08:27 $
 */
public class FooBean implements MessageDrivenBean, MessageListener { ❶

    /** Initialize the logging API here and use it for any std. error output */
    private static Logger logger =
        Logger.getLogger(FooBean.class.getName());

    /** The message driven context */
    private MessageDrivenContext ctx = null;

    /**
     * The default no-args constructor
     */
    public FooBean() { ❷
    }
}
```

```
/**
 * Sets the associated message driven bean context.
 *
 * @param ctx the message driven bean context
 */
public void setMessageDrivenContext(MessageDrivenContext ctx) {
    this.ctx = ctx;
}

/**
 * The Message Bean create method
 */
public void ejbCreate() {❸}

/**
 * A container invokes this method before it ends the life
 * of the message driven bean object.
 */
public void ejbRemove() {
}

/**
 * The onMessage method is called when a message
 * has been produced for
 *
 * @param msg The asynchronous received message.
 */
public void onMessage(Message msg) {
    String txtMessage = null;

    try {
        txtMessage = ((TextMessage) msg).getText();

        helperMethod(txtMessage);
    } catch (JMSEException e) {
        logger.log(Level.WARNING, "JMS Problem", e);
    }
}

/**
 * An example of a helper method which
 * can be called by the onMessage method.
 *
 * @param text A String value from the onMessage method
 */
public static void helperMethod(String text) {
    logger.info(text);
}
}
```

The above example is compliant to the following, not yet mentioned, rules:

- Rule JEC\_017: Do Not Override The finalize Method For A EJB implementation class (High)
- Specifically:
  - ❶ Rule JEC\_047: Make A Message Bean Implementation Implement `javax.jms.MessageListener` (High), rule JEC\_013: Make All EJB interfaces and classes public (High) and rule JEC\_014: Do Not Make A EJB implementation class final (High)
  - ❷ Rule JEC\_016: Provide A public Default Constructor For A EJB implementation class (High)

- ❸ Rule JEC\_052: Provide An ejbCreate Method For A Message Bean (High), rule JEC\_053: Return void For An ejbCreate Method Of A Message Bean (High) and rule JEC\_054: Do Not Declare Arguments For An ejbCreate Method Of A Message Bean (High)

**Tip**

The message driven bean class can have other helper methods which can be invoked by the onMessage method.

## Exceptions

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\] Exceptions](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines] Exceptions)]

### Exceptions Defined

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\] Exceptions Defined](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines] Exceptions Defined)]

Exceptions are used to report back on exceptional conditions that were encountered at runtime and which prevents the continuation of the method or scope you are in.

Exceptions are thrown from the location where the exception conditional was detected and caught (handled) by something capable of reacting to the exceptional condition.

Explicit use of throw statements provides an alternative to the old-fashioned style of handling error conditions by returning funny values, such as the integer value -1 where a negative value would not normally be expected. Experience shows that too often such funny values are ignored or not checked for by callers, leading to programs that are not robust, exhibit undesirable behavior, or both.

### When To Use Exceptions

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\] When To Use Exceptions](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines] When To Use Exceptions)]

It's important to distinguish an exceptional condition from a *normal* problem, in which you have enough information in the current context to somehow cope with the difficulty or problem.

With an exceptional condition, you can't continue processing because you don't have the information necessary to deal with the problem in the current context. All you can do is jump out of the current context and relegate the problem to a higher context where something is capable and qualified to make the proper decision.

For example a *normal* problem that can be handled without throwing an exception:

```
public int divide(int value, int denominator) {  
    int result = 0;  
    if (denominator != 0) {  
        result = value / denominator;  
    }  
    return result;  
}
```

In this example the method itself knows how to deal with the exceptional condition of trying to divide a value by 0.

For example the same problem but in this case the method itself does not know how to deal with the exceptional condition. The exception is reported back to the caller of the method.

```

public int divide(int value, int denominator)
    throws DivideByZeroException {
    if (denominator == 0) {
        throw new DivideByZeroException();
    }
    return value / denominator;
}

```



### Note

Don't use exceptions for every error condition.

Don't create a custom exception class for every possible error condition. Be pragmatic and try to use high level exception classes.

## Exception Types

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\] Exceptions Types](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines] Exceptions Types)]

`java.lang.Throwable` is the root class for all exceptions. The two direct subclasses are `java.lang.Error` and `java.lang.Exception`. Another important base class is `java.lang.RuntimeException`, which inherits from `java.lang.Exception`. These classes are the base classes in java's exception mechanism.

Together with these base classes java also defines two groups of exceptions: checked and unchecked exceptions. The table below describes the differences between the two groups.

**Table 1.4. Exception Types**

Group	Description
Checked	<ul style="list-style-type: none"> <li>Must be declared in the <code>throws</code> clause of the method from which they are thrown</li> <li>Must be handled (caught). The fact if they are handled is determined at compile time</li> <li>Inherit from <code>java.lang.Exception</code></li> </ul>
Unchecked	<ul style="list-style-type: none"> <li>Must not be declared in the <code>throws</code> clause of the method</li> </ul> <p><b>Remark:</b> Although it is a good practice to add it to the <code>throws</code> clause since it can be useful to the developer using the method informing him/her of what could possibly go wrong (for example:  <code>java.lang.IllegalArgumentException</code>)</p> <ul style="list-style-type: none"> <li>Must not be caught</li> </ul> <p><b>Remark:</b> it is a good practice to never catch unchecked exceptions.</p> <ul style="list-style-type: none"> <li>Inherit from <code>java.lang.RuntimeException</code> or <code>java.lang.Error</code></li> </ul>



## Throwing Exceptions

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines])] Throwing Exceptions]

Exceptions thrown by a method are declared in the `throws` clause of the method signature. A method can throw more than one exception.

Be specific and comprehensive with the `throws` clause. Don't be lazy and just include the super class in the `throws` clause if subclass exceptions are being thrown (see also Exception Matching).

Add a javadoc description for each exception being thrown and the reason why it could be thrown.

As stated earlier, java defines two groups of exceptions: checked and unchecked exceptions. Not every exception thrown by a method needs to be a checked exception (in other words: inherit from `java.lang.Exception`). The type of exception that needs to be thrown depends on the context in which it is generated.



### Tip

Try to limit the number of exceptions thrown by a method. Think about the poor developer having to catch and handle them all (see also Catching Exceptions).

## Catching Exceptions

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines])] Catching Exceptions]

Exceptions thrown by some method eventually need to be handled (caught) somewhere. The location where the exception must be caught is dependent of the context in which it is used.

You can catch an exception to:

- React to the exceptional condition being reported by, for example, presenting an error dialog to the user
- Take the exception and re-throw it again or wrap it in another exception (see also exception chaining).

The first scenario is quite straightforward. The second one can for example be used to limit the number of exceptions being thrown by a method or to give another meaning to an exception that was encountered.

Suppose you have a method that needs to call four other methods and that each of these four methods throw a different exception. If you are unable to handle these exceptions in the context of your method you will need to report them back to whoever called your method.

```
public void myMethod()  
    throws Exception1, Exception2,  
           Exception3, Exception4 {  
    otherMehod1();  
    otherMethod2();  
    otherMethod3();  
    otherMethod4();  
}
```

A better approach is to catch all of the individual exceptions and wrap them in one common exception.

```
public void myMethod() throws MyException {
    try {
        otherMehod1();
        otherMethod2();
        otherMethod3();
        otherMethod4();
    } catch (Exception1 e) {
        throw new MyException(e);
    } catch (Exception2 e) {
        throw new MyException(e);
    } catch (Exception3 e) {
        throw new MyException(e);
    } catch (Exception4 e) {
        throw new MyException(e);
    }
}
```

## Cleaning Up After An Exception Using A `finally` Clause

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\] Cleaning Up Exceptions](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines] Cleaning Up Exceptions)]

When handling (catching) exceptions there is often some piece of code that you want to execute whether or not an exception is thrown within the `try` block. This can be achieved by adding a `finally` block to the end of the `try/catch` block.

The fact if you need to use the `finally` clause entirely depends on the context in which the exception is being handled. There are however scenarios where you always have to include a `finally` clause:

- Release (non-memory) resources (streams, socket connections, JDBC resources, ...) that were created in the `try/catch` block.
- Restore the object state back to its original state before returning

For example a scenario in which a stream that was opened is always closed even if exceptions were raised.

```
public void readFile(String filename)
    throws IOException {
    char buffer[] = new char[1024];
    FileReader iStream = null;
    try {
        iStream = new FileReader(new File(filename));
        iStream.read(buffer);
    } catch (IOException e) {
        throw e;
    } finally {
        if (iStream != null) {
            iStream.close();
        }
    }
}
```

If the `finally` clause would not have been added in the example above you would run the risk that the input stream remains open when reading the file failed.



## Note

The finally clause must not always be preceded by a catch clause. The example above can be also be written as:

```
public void readFile(String filename)
    throws IOException {
    char buffer[] = new char[1024];
    FileReader iStream = null;
    try {
        iStream = new FileReader(new File(filename));
        iStream.read(buffer);
    } finally {
        if (iStream != null) {
            iStream.close();
        }
    }
}
```

## Exception Matching

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\] Exceptions Matching](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines] Exceptions Matching)]

When an exception is thrown the exception handling mechanism in the JVM looks for the nearest handlers (catch clause) in the order they are written. When it finds a match, the exception is considered handled, and no further searching occurs. This matching process does not require a perfect match between the exception that was thrown and the handler. A derived-class object will match a handler for the base class.

Take for example the `java.io.IOException`. The `IOException` acts as a base class for several other IO related exceptions (`FileNotFoundException`, `EOFException`, ...). Catching (handling) the root `IOException` involves catching all derived exception instances. See example below:

```
public char[] readFile(String filename)
    throws IOException {
    char buffer[] = new char[1024];
    FileReader iStream = null;
    try {
        iStream = new FileReader(new File(filename));
        iStream.read(buffer);
    } catch (IOException e) {
        throw e;
    } finally {
        if (iStream != null) {
            iStream.close();
        }
    }
    return buffer;
}
```

In some cases however you might want to react differently to a certain type of exceptional condition. The example shows the same `readFile` method but this time the method reacts differently to the fact that file could not be found.

```
public char[] readFile(String filename)
    throws IOException {
    char buffer[] = new char[1024];
    FileReader iStream = null;
    try {
```

```
        iStream = new FileReader(new File(filename));
        iStream.read(buffer);
    } catch (FileNotFoundException e) {
        // ok, simply return an empty buffer
    } catch (IOException e) {
        throw e;
    } finally {
        if (iStream != null) {
            iStream.close();
        }
    }
    return buffer;
}
```

There is no real guideline when it comes to determining the level at which exception matching should occur. It all depends on the context in which the exception is being handled.

## Exceptions And Error Messages

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines])] Exceptions And Error Messages]

All exceptions are capable of providing additional information describing the exceptional condition in more detail. More specific each exception can hold:

- A message
- A stack trace describing the location where the exception originated

For exceptions to be useful it is important that the message with which it was constructed provides sufficient information describing the exceptional condition. If your application is meant to be internationalized the internationalization must also be reflected in the exception's error messages. This is especially the case for exceptions that hold messages that eventually will need to be presented to a user.

There is nothing that stops you from adding additional attributes to your custom exceptions if these attributes will help you in providing more detailed feedback to whoever will have to handle the exception.

Additional information that could be added to your custom exception classes could for example be an error code. This last attribute is particularly handy when you have to deal with internationalized exceptions. If a user calls you to say he (or she) got an error message in French and your French is a bit rusty you can always ask him (or her) for the error code and continue your investigation of the problem based on this code.

You can control the messages that are used in your custom exceptions by providing specific constructors that only accept certain arguments and that will create the detailed messages themselves.

For example:

```
public class FileNotFoundException extends Exception {
    public FileNotFoundException(String filename) {
        super("The file " + filename
            + " could not be found");
    }
}
```

Another technique you could use is to make the exception constructor private and provide several,

factory like, methods that would throw the exceptions.

For example:

```
public class BusinessException extends Exception {  
    private BusinessException(String message) {  
        super(message);  
    }  
  
    public static void throwLoadException(String something)  
        throws BusinessException {  
        throw new BusinessException("Error loading "  
            + something);  
    }  
  
    public static void throwSaveException(String something)  
        throws BusinessException {  
        throw new BusinessException("Error saving "  
            + something);  
    }  
    // ...  
}
```

Alternatively you could create a `LoadException` and `SaveException` that inherit from a `BusinessException` and use the same approach as the `FileNotFoundException` from the first example.

## Exception Chaining

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\] Exceptions Chaining](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines] Exceptions Chaining)]

Often you want to catch one exception and throw another, but still keep the information about the original exception wrapped inside the new exception being thrown. This technique is called exception chaining.

Since JDK1.4 all `Throwable` subclasses may take a cause object in their constructor. For code that needs to run on an older JDK version the developers will have to write their own versions of chainable exceptions.



### Important

Be careful when using exception chaining in a multi-tiered environment. Sometimes the exception handler does not have all exception classes on its classpath. If a root exception has such an exception in its chain it's possible for a `ClassNotFoundException` to be thrown when the root exception is serialized between the tiers.

## Creating Custom Exceptions

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\] Custom Exceptions](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines] Custom Exceptions)]

The Java development environment provides a lot of exception classes that you could use. You should go to the trouble of writing your own exception classes if you answer yes to any of the following questions.

- Do you need an exception type that isn't represented by those in the Java development environment?

- Would it help your users if they could differentiate your exceptions from those thrown by classes written by other vendors?
- Does your code throw more than one related exception?
- If you use someone else's exceptions, will your users have access to those exceptions?
- Should your package be independent and self-contained?

## Exceptions And EJBs

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines])] Exceptions And EJBs]

### Use Of RemoteException

EJB implementation methods, corresponding to a method defined in the EJB's remote, local or home (both remote and local), should never throw an `java.rmi.RemoteException`.

If the EJB method itself uses code that throws a `RemoteException` the exception should either be wrapped in a `javax.ejb.EJBException` or an application specific exception (a checked exception in the methods throws clause).



#### Note

This restriction is enforced by the EJB spec 1.1 and higher. Some containers check for this specification violation, others don't.

For example consider someEjbMethod do be defined in the remote or local interface:

#### WRONG

```
public void someEjbMehod() throws RemoteException {
    doSomething();
}

private void doSomething() throws RemoteException {
    // ...
}
```

#### RIGHT

```
public void someEjbMehod() {
    try {
        doSomething();
    } catch (RemoteException e) {
        throw new EJBException(e);
    }
}

private void doSomething() throws RemoteException {
    // ...
}
```

## Rolling Back Transactions

Checked exceptions thrown by an EJB implementation do not rollback any pending transactions. If an EJB throws a checked exception and it decides that all changes already made in the ongoing transaction should be rolled back it should do so explicitly by calling `setRollbackOnly` on the EJB's context.

For example:

```
public void MySessionEJB implements SessionBean {  
    private SessionContext context;  
    // ...  
    public void doSomethingCritical()  
        throws SomeCheckedExcetion {  
        try {  
            doSomething();  
        } catch (SomeCheckedException e) {  
            context.setRollbackOnly();  
            throw e;  
        }  
    }  
}
```

## Exception Handling Strategies

What goes up must come down. Every exception that is thrown needs to be caught somewhere by something. Once an exception is caught it is time to determine what to do with it. What follows is a brief description of possible exception handling strategies and which strategies to avoid (see also [Catching exceptions](#)).

A strategy determines what will happen when an exception is caught, in other words, the code in the catch clause of a `try/catch` block.



### Important

Do not tackle exception handling at the end of the development cycle. Design your error handling strategy from the beginning.

## `printStackTrace` Is Not An Exception Handling Strategy

Although `printStackTrace` is not an exception handling strategy it is (unfortunately) the one that is encountered the most, especially in projects where exception handling was considered to be a low (or even no) priority.

The use of `printStackTrace` method should be avoided at all costs because:

- It only writes a stack trace to the standard error stream, usually the console. The only strategy that can be applied here is having someone sit down at the console and monitor it (or a file to which the console output is written).
- If the console output is not redirected to a file, the stack trace is lost.
- Once the stack trace is printed, the program merely continues on.

There are no scenarios (or excuses) in which the use of `printStackTrace` is justified. If the only option you have is to print the stack trace somewhere you should use a logging implementation (like the one provided in JDK 1.4 or Log4J).

## Don't Use Empty catch Clauses

Never use an empty catch clause. If the exception is somewhat expected (a rare condition) you can add a comment stating why the exception is not important.

Example that handles a `java.beans.PropertyVetoException`:

```
public void doSomething() {
    try {
        // ...
    } catch (PropertyVetoException e) {
        // property can not be changed, no problem just
        // continue
    }
}
```

If you think that an exception will never get thrown you are tempted to simply *eat* the exception; in other words: use an empty catch clause.

A better technique is to catch the *unexpected* exception and wrap it in a runtime exception. In this case you are sure that if the exception should occur anyway you will be aware of it.

For example:

```
public void doSomething() {
    try {
        callMethod();
    } catch (UnexpectedException e) {
        throw new Error("Unexpected exception", e);
    }
}
```



### Note

The example above uses the *exception chaining* feature available in JDK 1.4 to wrap the *unexpected* exception in an `Error` instance.

## Catch And Re-throw

This is the most common exception handling strategy in which an exception that is caught is re-thrown as is or translated into some other exception.

Eventually the exception reaches a method that does now what to do with it (or is unable to throw it further upwards). Once the exception has reached its final destination the following can be done with it:

- Log it with an appropriate severity level
- Present an error message to the user (only to be used in graphical clients, never on the server)



---

# Chapter 2. Guidelines

## Feedback

***[mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]guidelines]***

## Introduction

The Guidelines chapter includes different coding and testing recommendations which will unify different aspects within a J2EE project, for example the way we do logging or how and when we use assertions. The guidelines are based on re-usable existing (often open-source) components and Java language features. Where the conventions define a set of rules that must be adhered to, the guidelines can be seen as recommendations.



### Note

If necessary, guidelines can be promoted to become conventions.

## Project Directory Structure

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]project]

## Introduction

A project directory holds all files that are related to a specific project. Files can be categorized based on their purpose or role in the project. Typical file types include:

- Source files
- Documentation
- Configuration files
- ...

Sub-directories are added to the main project directory structure, each sub-directory holding a certain type of files. You end up with a sub-directory containing the sources, one containing the documentation ... In other words: you create a project directory structure.

Having a standardized project directory structure is important for the following reasons:

- It helps people to locate specific project related information. Example: sources can be found in the `src` directory.
- It helps people to determine where a certain type of file should be put.
- It allows you to create standardized build script templates (see the section called “Build Tools”)

This section puts forward a recommended directory structure.



## Note

Some IDEs (example: IBM's WebSphere Studio) somewhat force you to use a certain directory structure. What J2EEGuidelines tries to do is to define a tool-neutral way of structuring your project information.

## Proposed Directory Structure

What follows is a proposal for a project directory structure. The structure proposed here is tool neutral and assumes that some sort of external build script is used to build the project.

**Table 2.1. Project Directory Structure**

Directory	Content
<code>./</code>	The project's root directory. Holds the IDE's project files and the build script that should be used to build the project.
<code>./build</code>	<p>The staging directory used in the build process. The contents of this directory is cleaned upon each build.</p> <p>Can include various sub directories, each sub directory containing a specific build result. Examples:</p> <ul style="list-style-type: none"> <li>• <code>clientclasses</code> to hold all client related classes</li> <li>• <code>testclasses</code> to hold all test classes</li> <li>• <code>ejbs</code> to hold the EJB jar files</li> <li>• ...</li> </ul> <p>The contents of the build directory is used to create the project's distribution files. (see <code>./dist</code> directory)</p>
<code>./build/classes</code>	A common build subdirectory used as the destination directory when compiling all sources with your IDE.
<code>./conf</code>	<p>Holds the different project configuration files. Typical sub-directories include:</p> <ul style="list-style-type: none"> <li>• <code>web</code>: holds the web application's configuration files (example: <code>web.xml</code>, ...)</li> <li>• <code>ear</code>: holds the Enterprise Application configuration files (example: <code>application.xml</code>, ...)</li> <li>• <code>ejb</code>: holds the EJB deployment descriptors</li> </ul> <p>Add additional sub directories where needed.</p>
<code>./dist</code>	The directory in which the project's distributable files are placed. For example: the <code>.ear</code> file.
<code>./docs</code>	Holds all project related documentation
<code>./docs/api</code>	The generated JavaDoc.

Directory	Content
<code>./lib</code>	All external libraries this project depends upon.
<code>./src</code>	The project source directory.
<code>./src/java</code>	All the Java source files, logically grouped in several packages.
<code>./src/sql</code>	This directory holds SQL scripts which could build, index and populate the project database (if any).
<code>./src/web</code>	This directory holds the JSP and HTML files.
<code>./src/web/images</code>	This directory holds the web images used by the JSP and HTML files.
<code>./test</code>	Base directory for everything used in the automated testing process.
<code>./test/java</code>	Holds all the unit tests that are used in the automated testing process.
<code>./test/sql</code>	SQL scripts that are used to load the test data

## Build Tools

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]buildTools](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]buildTools)]

This section provides an overview of the various build tools that can be used to build your java application. The primary build tool for a java project is the IDE you use to develop your application.

Building your application with your IDE is done in different steps. First you might compile the code, then package the ear file after which the ear file is deployed on your application server. All these steps require user interaction, you have to instruct your IDE to perform a certain step. It's the developer that decides which steps should be performed at what stage. Each developer needs to be aware of these steps before he or she can build and deploy the application.

The tools described in this section differ from the IDE as a build tool in the sense that they allow for 'automating' your build process. They help you define the steps that need to be performed to build, test and deploy your application in a standard way, standard in the sense that the same build script is used by all developers. Another important feature of these tools is that they are not interactive which makes them suitable for nightly builds.

Tools described in this section include:

- Ant
- Maven

## ANT

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]ant](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]ant)]

### Introduction

Ant is a cross-platform build tool written entirely in java. It is an alternative for already existing build tools like *make* or other home grown build scripts. Ant has become the default build tool for building java applications and can also be used to automate non-java related tasks.

The acceptance of Ant in the java community is clearly demonstrated by the fact that all leading java IDEs provide standard Ant support allowing for Ant to be plugged in into the editor's environment.

Ant is downloadable from <http://ant.apache.org> under the Apache Software License (a very flexible open source license).



## Note

The JJGuidelines book is written in XML based on the DocBook DTD. An Ant build script is used to create its distribution, including generating the PDF and HTML version of the JJGuidelines book.

## Ant Concepts

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]ant.concepts](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]ant.concepts)]

### Build File

Ant operates on so a called build file. An Ant build file is written in XML. Each build file defines the build procedure for one project and defines a list of possible targets. The concept of targets is described in the section called “Targets”.

The default build filename is `build.xml`. It's possible to provide a specific build file using the `-f` command line option.

An example using a non-standard build file:

```
ant -f myBuildFile.xml
```



## Note

Although it's possible to provide a different build file it is recommend sticking with the default name (`build.xml`). See rule ANT\_001: Use the default `build.xml` build filename.

The root element for an Ant build file is `project`. For example:

```
<project name="MyProject" default="main">
  <!-- ... -->
</project>
```

The project has an optional name and required `default` attribute. The `name` attribute can be used to name the project for which the build file was written. The `default` attribute defines the default target that should be executed when ant was run with no specific target name (the target concept is described in the section called “Targets”).

Another project attribute is the optional `basedir` attribute. The `basedir` attribute defines the base directory where Ant should look for files and directories. If omitted, which is recommended, the directory containing the build file is used as the base directory.



## Note

In the recommended project directory structure, the ant build script is located in the project's root directory

## Targets

An Ant build script defines a set of targets. A target describes a set of tasks that need be performed (the concept of tasks is described in the section called “Tasks”). Each target has a unique name.

For example:

```
<project name="MyProject" default="deploy">
  <target name="compile">
    <javac ...>
      <!-- ... -->
    </javac>
  </target>

  <target name="deploy">
    <!-- ... -->
  </target>
</project>
```

This example defines a target named `compile` in which the `javac` task is executed and a target named `deploy`.

The project element is required to define a default target, which is the `deploy` in this example. Launching Ant without command line options will result in Ant executing the `deploy` target.

You can ask Ant to execute another target by providing its name on the command line.

For example:

```
ant compile
```

This command will instruct Ant to execute the `compile` target. It's possible to provide more than one target on the command line by providing a space or comma separated list of target names.

A target can depend on other targets. You might have a target for compiling and a target for creating a distribution. You can only build a distribution when you have compiled first, so the `distribution` target depends on the `compile` target. A target's dependencies are specified in target's `depends` attribute.

For example:

```
<target name="compile">
  <!-- ... -->
</target>

<target name="dist" depends="compile">
  <!-- ... -->
</target>
```

This example defines the 2 targets `compile` and `dist`. The `depends` attribute defined in the `dist` target tells Ant to do the `compile` target prior to performing the tasks specified in the `dist` target itself.

A target can have more than one dependency in this case dependencies are separated by a comma. Ant resolves a target's dependencies from left to right.

For example:

```
<target name="init">
  ...
</target>

<target name="compile">
  <!-- ... -->
</target>
```

```
<target name="dist" depends="init, compile">
  <!-- ... -->
</target>
```

In this example the `dist` target has 2 dependencies `init` and `compile`. When asked to execute the `dist` target Ant will first call the `init`, then the `compile` and finally perform the tasks defined in the `dist` target itself.



## Important

Once a target is executed it will not be executed again in the same run.

For example:

```
<target name="init">
  <!-- ... -->
</target>

<target name="compile" depends="init">
  <!-- ... -->
</target>

<target name="dist" depends="init, compile">
  <!-- ... -->
</target>
```

If you ask Ant to execute the `dist` target the `init` target will only be called once. The `init` target is called in the context of the `dist` dependency. Once Ant reaches the `compile` target (also included in the `dist` dependency list) the `init` target will already have been executed and will not be executed again.

## Tasks

A task describes a single unit of work that needs be executed by Ant. Ant defines 3 types of tasks:

- *Core tasks*: tasks that are available in the standard Ant distribution
- *Optional tasks*: tasks that are also available in the standard Ant distribution that need external libraries or tools to perform the task
- *Custom tasks*: tasks you develop yourself using the Ant API

Tasks are added to a target. Each task has a unique (tag)name and its own set of attributes and elements. Please refer to the Ant manual [<http://ant.apache.org/manual/index.html>] for an overview of all available core and optional tasks.

For example:

```
<target name="helloWorld">
  <echo message="Hello, world"/>
</target>
```

In this example the `helloWorld` target contains one task named `echo`. The `echo` task is one of Ant's core tasks.

Launch the helloWorld target using the following command:

```
ant helloWorld
```



### Note

As of Ant version 1.6 tasks can also be defined outside a target. Tasks that are defined on a project level are executed before any target is executed.

For example:

```
<echo message="Building hello world"/>

<target name="helloWorld">
  <echo message="Hello, world"/>
</target>
```

## Properties

Properties in Ant can be compared to constants in a java application. They are used to define constant values. Each property has a name and a value.

Properties are defined using the `property` task and referred to as `${propertyName}`. Where `propertyName` is the name of the property.

For example:

```
<property name="projectVersion" value="0.0.1"/>

<echo message="Building project version ${projectVersion}"/>
```

The first line defines the property `projectVersion`. The last line shows how to use the property in an echo task.



### Important

Properties are case-sensitive and immutable. Once set they can not be changed.

The `property` task allows you to be more specific when it comes to defining the property's value. A property pointing to a directory or file should use the `location` attribute instead of the `value` attribute. See rule ANT\_007: Use The location Attribute For A File Or Directory Based Property.

For example:

```
<property name="src.java" location="${basedir}/src/java"/>
```



### Note

Ant resolves the `/` character as the platform's directory separator.



## Note

If the `location` attribute is a non absolute path it is taken relative to the project's `basedir`. The example can also be written shorter like:

```
<property name="src.java" location="src/java"/>
```

Properties can also be loaded from a properties file. The file containing the properties should follow the same syntax as a standard java property file (name-value pairs).

The following example will retrieve the properties from a file named `project.properties` stored in the project's base directory.

```
<property file="project.properties"/>
```

The following example will retrieve the properties from the `project.properties` file served on a web server.

```
<property url="http://localhost/project.properties"/>
```

Ant also allows you to use OS environment variables as properties.

For example:

```
<property environment="env"/>
```

```
<echo message="OS system path ${env.PATH}"/>
```

The first line defines the prefix (or namespace) to use when you want to access OS environment variables. The second line prints out the OS's PATH environment variable.



## Important

Although the use of OS environment variables is supported by Ant it is not recommended to use them. Using OS environment variables introduces dependencies between Ant and the OS it is running on. Remember that Ant is platform independent.

Ant itself also defines a list of standard attributes that are always available.

**Table 2.2. Standard Ant Properties**

Attribute	Description
<code>basedir</code>	The absolute path of the project's <code>basedir</code>
<code>ant.file</code>	The absolute path of the build file
<code>ant.version</code>	The version of Ant
<code>ant.project.name</code>	The name of the project that is currently executing (as set with the <code>name</code> attribute of <code>&lt;project&gt;</code> )
<code>ant.java.version</code>	The JVM version Ant detected in the form of <code>major.minor</code> (for example <code>1.4</code> )



By default all standard JVM properties, the ones returned by the `java.lang.System.getProperties` method, are also available in the build script.

An example that will print out the user home directory based on the JVM `user.home` property:

```
<echo message="The user's home directory is ${user.home}"/>
```

## Directory-based Tasks

Most of the Ant tasks need to operate on files or directories. To make life easier for a developer Ant introduces a series of concepts that provide easy access to files and directories. The main concepts are:

- A pattern based set (`patternset`, `fileset`, `dirset`)
- A path-like structure (`path`)

These tasks can be used to provide files for other tasks:

```
<copy todir="${basedir}/src/web/images">  
  <fileset dir="${basedir}/build/server/war/images"/>  
</copy>
```



### Note

Directory-based tasks are usually defined inside another task, however they can be defined as children of the project element to be referenced to by other tasks.

## A Pattern Based Set

The `fileset` and `dirset` task define a list of files or directories based on selectors and patterns, starting from a certain directory. The `fileset` only includes files, the `dirset` task only includes directories and the `patternset` task defines a pattern which can be reused in the other 2 tasks.

A pattern selects files or directories on the filename and path, much like a very simple regular expression, starting from a directory. The following matches can be used:

- `*` matches zero or more characters (but not the directory separator)
- `?` matches exactly one character (but not the directory separator)
- `**` matches zero or more directories

A pattern can be used inside 2 tasks:

- An `include` task selects files or directories.
  - If no `include` task is defined, all files and directories are selected.
  - If one or more `include` tasks are defined, only those files and directories are selected.

- An `exclude` task deselects files or directories which would otherwise be selected by an `include` task or the lack of it.

For example:

```
<fileset dir="${basedir}">
  <include name="src/web/*.jsp"/>
  <!-- Includes all jsp files directly under src/web -->

  <include name="**/*.html"/>
  <!-- Includes all html files -->

  <exclude name="**/package.html"/>
  <!-- Excludes all package.html files -->

  <include name="src/java/**/*.java"/>
  <!-- Includes all java files directly or indirectly under src/java -->
</fileset>
```



## Important

A number of patterns are always excluded by default. These are temporary or VCS files recognized by the following patterns:

- `**/*~`, `**/#*#`, `**/*.#*`, `**/%**` and `**/._*`
- `**/CVS`, `**/CVS/**` and `**/.cvsignore`
- `**/SCCS`, `**/SCCS/**`, `**/vssver.scc`, `**/.svn`, `**/.svn/**` and `**/.DS_Store`

You can turn the default exclude off with the `defaultexcludes` attribute on the `fileset`, `dirset` or `patternset` task.

A selector, such as the `size` or `modified` task, selects files or directories based on other criteria than the name. See <http://ant.apache.org/manual>.

## A Path-like Structure

The `path` task defines a list of files or directories based on other directory-based tasks or `pathelement` tasks, across directories. For example:

```
<path>
  <pathelement
    path="${build.dir}/classes;${build.dir}/jar/myjar.jar"/>
  <fileset dir="${basedir}/lib">
    <include name="**/*.jar"/>
  </fileset>
</path>
```

## Referencing

Some tasks (such as directory-based tasks) can support an `id` attribute, which can be used to reference it with the `refid` attribute elsewhere in the build script:

```
<path id="classpath">
  <pathelement location="build/classes"/>
  <fileset dir="${basedir}/lib">
    <include name="**/*.jar"/>
  </fileset>
</path>

<target name="compile">
  <javac classpathref="classpath" ...>
    <!-- ... -->
  </javac>
</target>
```

## Common Targets

Most build scripts contain a subset of the following targets.

**Table 2.3. Common Ant Targets**

Target Name	Description
clean	Deletes all files that were generated during a build process allowing you to start a clean build.
fetch	Fetches the latest source from a VCS such as CVS
init	Initializes the build process. Can be used for instance to some required directories.
compile	Compiles all java sources
build	Does an incremental build
jar	Assembles the application jar
client	Assembles the client jar
war	Assembles the web tier war
ejb	Assembles the ejb jars
ear	Assembles the application ear
test	Runs all tests
docs	Generates all documentation, including javadocs
deploy	Deploys the war or ear file to an application server
dist	Generates the project's distributable files
publish	Publishes the source, binaries and docs to a site
all	Runs all the targets
main	The default build process (usually build and test)

Target names can also be used as "namespace" for a more specialized targets. For example:

```
<target name="clean" depends="clean.build, clean.dist"/>

<target name="clean.build"
  description="Cleans the build directory">
  <!-- ... -->
</target>
```

```
<target name="clean.dist"
      description="Cleans the distribution directory">
    <!-- ... -->
</target>
```

## Reusing The Build File Across Environments

It's important for your build files to be able to operate in different environments. Suppose your build file contains a target that will deploy a war file in Tomcat. Chances are high that each developer has installed tomcat in a different location on his local environment. One option would be to tell everybody to install tomcat in a specific location (good luck). But what do you do if development is done on Windows and the production deployment on Unix?

To solve these types of problems it is recommended to use properties that are kept in an external property file and to use 2 property files:

- `project.properties` the file containing the default project properties (this file is the same across all environments)
- `localProject.properties` the file in which the default project properties are tuned to the local environment.

Example:

```
# project.properties
tomcat.home=/opt/tomcat

# localProject.properties
tomcat.home=c:/apache/tomcat
```

To import the 2 properties file in your build file you would use the following in your build file

```
<project name="MyProject" default="all">
    <property file="localProject.properties"/>
    <property file="project.properties"/>
    ....
</project>
```

Note that the `localProject.properties` are loaded before the `project.properties`. Since Ant properties are immutable (once set they can not be changed) the properties defined in `localProject.properties` will overrule those defined in `project.properties`.

## Reusing Paths

A directory-based task can be reused by assigning it an `id` and reuse it through `refid`. It is a good practice to reuse classpaths.

For example:

```
<path id="classpath">
  <pathelement location="build/classes"/>
  <fileset dir="${basedir}/lib">
    <include name="**/*.jar"/>
  </fileset>
</path>

<target name="compile">
  <javac classpathref="classpath" ...>
    <!-- ... -->
  </javac>
</target>

<target name="test">
  <junit ...>
    <classpath>
      <path refid="classpath"/>
      <pathelement location="build/test"/>
    </classpath>
    <!-- ... -->
  </junit>
</target>
```

The classpath is only defined once in the example.

## A Build Script Example

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]ant.example](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]ant.example)]

Below you can find an example of an Ant build file for a J2EE project that uses the recommended directory structure.

```
<project name="example" default="main">
  <description>Build script for Example</description>

  <!-- version -->
  <property name="version" value="1_0_1"/>

  <!-- Project properties -->
  <property name="build.dir" location="build"/>
  <property name="dist.dir" location="dist"/>
  <property name="javadoc.dir" location="docs/api"/>

  <property name="src.dir" location="src"/>
  <property name="src.java.dir" location="${src.dir}/java"/>

  <property name="test.dir" location="test"/>
  <property name="test.java.dir" location="${test.dir}/java"/>
  <property name="test.results.dir" location="${test.dir}/results"/>

  <!-- The classpath -->
  <path id="classpath">
    <fileset dir="${basedir}/lib">
      <include name="**/*.jar"/>
    </fileset>
  </path>

  <target name="clean"
    description="Cleans up the build results">
    <delete includeEmptyDirs="true" quiet="true">
      <fileset dir="${build.dir}"/>
      <fileset dir="${dist.dir}"/>
    </delete>
  </target>
```

```
        <fileset dir="${javadoc.dir}"/>
        <fileset dir="${test.results.dir}"/>
    </delete>
</target>

<target name="init"
    description="Initialises the build process">
    <mkdir dir="${build.dir}"/>
    <mkdir dir="${dist.dir}"/>
    <mkdir dir="${test.results.dir}"/>
</target>

<!-- Compiles all source files -->
<target name="compile" depends="init"
    description="Compiles all source files">
    <mkdir dir="${build.dir}/classes"/>
    <javac destdir="${build.dir}/classes"
        source="1.4"
        classpathref="classpath">
        <src path="${src.java.dir}"/>
    </javac>
    <mkdir dir="${build.dir}/test"/>
    <javac destdir="${build.dir}/test"
        source="1.4"
        classpathref="classpath">
        <src path="${test.java.dir}"/>
    </javac>
</target>

<target name="dist" depends="compile, test, javadoc"
    description="Builds the distribution">

    <!-- binaries -->
    <jar destfile="${dist.dir}/${ant.project.name}.jar">
        <fileset dir="${build.dir}/classes"/>
    </jar>

    <!-- sources -->
    <jar destfile="${dist.dir}/src.jar">
        <fileset dir="${src.java.dir}"/>
        <fileset dir="${test.java.dir}"/>
    </jar>

    <!-- javadoc -->
    <jar destfile="${dist.dir}/docs.jar">
        <fileset dir="${javadoc.dir}"/>
    </jar>

</target>

<target name="test" depends="compile"
    description="Runs the unit tests" >
    <delete dir="${test.results.dir}" includeEmptyDirs="true"
        quiet="true"/>
    <mkdir dir="${test.results.dir}"/>
    <junit fork="yes" printsummary="yes" haltonfailure="no">
        <classpath>
            <path refid="classpath"/>
            <pathelement location="${build.dir}/classes"/>
            <pathelement location="${build.dir}/test"/>
        </classpath>
        <formatter type="plain"/>
        <batchtest todir="${test.results.dir}">
            <fileset dir="${build.dir}/test">
```

```
        <include name="**/*Test.class"/>
        <exclude name="**/Abstract*TestCase.class"/>
    </fileset>
</batchtest>
</junit>
</target>

<target name="javadoc" depends="init"
    description="Generates the javadocs" >
    <delete dir="${javadoc.dir}" includeEmptyDirs="true"
        quiet="true"/>
    <mkdir dir="${javadoc.dir}"/>
    <javadoc destdir="${javadoc.dir}"
        source="1.4"
        author="true"
        version="true"
        use="true"
        windowtitle="${ant.project.name}">
    <classpath>
        <path refid="classpath"/>
        <pathelement location="${build.dir}/classes"/>
        <pathelement location="${build.dir}/test"/>
    </classpath>
    <packageset dir="${src.java.dir}" defaultexcludes="yes"/>
    </javadoc>
</target>

<target name="main" depends="build,test"
    description="Builds the complete project"/>

</project>
```

## Maven

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]maven](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]maven)]

### Introduction

Maven is a uniform build system based on Ant and Jelly. It is freely downloadable from <http://maven.apache.org>.

Ant is currently the de facto standard build tool. It is very powerful, but has a few limitations:

- You need to write a `build.xml` file or at least configure a company template.
- Targets almost aren't standardized

Maven strives to solve these limitations by providing working targets (which maven calls *goals*) out of the box.

### Maven Concepts

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]maven.concepts](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]maven.concepts)]

### The Maven Command

After the installation, the **maven** command will be available on the command line. It is the primary way to use maven. Type **maven -h** to see the different arguments maven allows.

## POM

Unlike Ant, Maven does not initially expect you to write some sort of script or code. Instead it expects you to provide data: a POM or *Project Object Model*. This POM defines project information, such as the project name, project description, directory structure, dependencies, developer list, repository, etc.

The POM is described in an XML file called `project.xml` (similar in purpose to a subset of Ants `project.properties` file). For example:

```
<?xml version="1.0"?>
<project>
  <!-- the version of maven's project object model -->
  <pomVersion>3</pomVersion>

  <!-- a unique name for this project -->
  <id>myexample</id>

  <!-- a short but descriptive name for the project -->
  <name>MyExample</name>

  <!-- ... -->

  <!-- details about the organization that 'owns' the project -->
  <organization>
    <name>JCS</name>
    <url>http://www.jcs.be/</url>
  </organization>

  <!-- ... -->

  <!-- build information for the project -->
  <build>
    <sourceDirectory>src/java</sourceDirectory>
    <unitTestSourceDirectory>test/java</unitTestSourceDirectory>

    <!-- ... -->
  </build>
</project>
```

A full POM is actually a lot larger and can be a lot more dynamic. The POM is actually a Jelly script and can even be extended. However its recommended to keep scripting in the POM to a minimum.



### Note

Jelly has much in common with Ant and it can actually been as a more powerful front end to Ant. To start with Maven, there is no need for Jelly or Ant knowledge.



### Note

Maven also uses additional `.properties` files besides `project.xml`.

Maven can generate a `project.xml` and a skeleton directory structure. Try it out for yourself and type **maven -Dpackage=<yourpackage> genapp** in an empty directory:

```
C:\Temp\eval>maven -Dpackage=be.jcs.maveneval genapp
...
Enter a project template to use: [default]
default
```



```

Please specify an id for your application: [app]
maveneval
Please specify a name for your application: [Example Application]
MavenEval
Please specify the package for your application: [default.example.app]
be.jcs.maveneval
    [copy] Copying 1 file to C:\Temp\eval\src\java\be\jcs\maveneval
    [copy] Copying 3 files to C:\Temp\eval\src\test\be\jcs\maveneval
    [copy] Copying 1 file to C:\Temp\eval
    [copy] Copying 2 files to C:\Temp\eval
BUILD SUCCESSFUL
...

```

## Goals

### Introduction

Based on the POM, Maven can perform *goals* (similar to Ant's targets) such as:

- Build one or more jar, war or ear files.
- Perform unit tests or check their coverage.
- Generate documentation in various formats.
- Deploy Java archives to an application or web server.
- Generate a complete website with documentation, source reports (metrics, auditing, testing, etc), project information, etc. See Mavens website ???, generated by Maven.



### Important

The POM declares data for goals, however it does not declare the goals themselves. That is left over to the Maven plugins and the `maven.xml` file.

Goals are bundled into a plugin for Maven.

### Use

To call a maven goal, type:

```
maven <plugin-name>[:<goal-name>]
```

It's that simple. Each plugin has a default goal, so the goal name is optional. Try it out and type **maven site**, this will call the default goal for the site plugin and generate a website for your project, by default in the `target/docs` directory.

In the previous section you actually called the `genapp`'s plugin default goal. Unlike most other goals, that goal does not need a `project.xml`. Its possible to give the `project.xml` file another name and use the **-p** argument to specify the filename.

Type **maven -g** to see a list of available goals. This is a very small subset of it:

```

...
[site] : Generate the web site
        deploy ..... deploy the generated site docs

```

```
ear ..... Create an EAR File from the generated site
fsdeploy ..... Deploy the generated site by copying to the site
               directory
sshdeploy ..... Deploy the generated site docs using ssh
war ..... Create a WAR File from the generated site

[statcvs] : Generate CVS statistics for the current CVS project
  generate ..... Generate CVS statistics for the current CVS
                  project
...

```

Try it yourself. Impressive, isn't it? And more goals can be plugged in.



### Note

If the project or one of goals depend on an artifact (such as `xalan.jar`), Maven will automatically download the artifact and store it in Maven's local repository.

It is even possible to generate IDE configuration files, for example use the `idea` goal for IntelliJ IDEA.



### Important

An IDE currently doesn't know all the depended artifacts. Although Maven can compile for example a unit test (because the `unit` plugin has the `junit.jar` in its repository), the IDE won't recognize its classes, such as `TestCase`.

For now you must include each artifact from  
`${user.home}/.maven/repository/.../jars`. Hopefully this will be solved as more Maven plugins for IDE's appears.

## Configuring Goal Properties

A goal can be made available through:

- a Jelly/Ant script
- a plugin for Maven



### Note

Do not confuse a plugin for Maven, which provides a goal, with a Maven plugin for a specific IDE, which provides access to Maven inside the IDE.

Goals are not configured on the command line. Each plugin defines a set of properties, which its goals use to configure themselves.

For example to set the `Main-Class` attribute in the manifest of the jar, add the following line to `project.properties`:

```
maven.jar.mainclass=be.jcs.maveneval.MyMainClass
```

Now call **maven jar** and the jar plugin will generate a jar file with the main class set.

The properties files are processed in the following order:

1. `${project.home}/project.properties`
2. `${project.home}/build.properties`
3. `${user.home}/build.properties`

## Custom Goals

Its possible to create custom goals in the `maven.xml` file (which is similar in purpose to Ants `build.xml`) by use of:

- other goals
- Ant tasks and Ant scripting
- Jelly scripting



### Tip

As it is possible to reuse Ant tasks, its also possible to create a custom Ant task and call that from the maven script.

For example 2 custom goals, one using Ant scripting and another using both Ant and Jelly scripting:

```
<project default="ant-goal"
  xmlns:j="jelly:core" xmlns:u="jelly:util">

  <goal name="ant-goal">
    <echo message="This goal only uses Ant scripting" level="info"/>
    <mkdir dir="${maven.build.dir}/MyAntDirectory"/>
  </goal>

  <goal name="jelly-ant-goal">
    <echo message="This goal uses both Jelly and Ant scripting" level="info"/>
    <j:set var="goals" value="java:compile,test,ant-goal"/>
    <mkdir dir="${maven.build.dir}"/>
    <u:tokenize var="goals" delim=",">${goals}</u:tokenize>
    <j:forEach items="${goals}" var="goal" indexVar="goalNumber">
      Now attaining goal number ${goalNumber}, which is ${goal}
      <attainGoal name="${goal}"/><!-- Calls other goals -->
    </j:forEach>
  </goal>

</project>
```

With project's default attribute its possible to define a default goal for the project (similar to a default target for ant). The output for the command **maven** or **maven ant-goal** is:

```
...
ant-goal:
  [echo] Starting the Ant only goal
  [mkdir] Created dir: C:\Temp\eval\target\MyAntDirectory
BUILD SUCCESSFUL
```

...

**Tip**

For more information on Jelly script, see <http://jakarta.apache.org/commons/jelly>.

Its also possible to create custom goals by creating a plugin for Maven (similar in purpose to Ant tasks).

**Ant Versus Maven**

Lets compare Ant and Maven, which is intersting for users migrating from Ant to Maven:

**Table 2.4. Comparison Of Ant And Maven**

Concept	Ant	Maven
Script file(s)	<code>build.xml</code>	<code>maven.xml</code>
Project properties file(s)	<code>project.properties</code> and other properties files	<code>project.xml</code> , <code>project.properties</code> and other properties files
A build action	a target (scripted)	a goal (scripted or defined by plugin)
A building unit (Java)	a task	a plugin (defines goals), an ant task

Maven provides a configurable harness around Ant, standardizing a set of build actions and properties. Currently Ant has more maturity.

**Tip**

Do not exclude Ant because you are using Maven. Maven supports Ant tasks directly in `maven.xml` and allows to run targets in an outside `build.xml` file.

**Maven Guidelines**

Feedback

[mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]maven.guidelines]

**POM Naming**

Maven defines a set of naming conventions for the POM:

**Table 2.5. POM Naming**

POM Entity (XPath)	Naming Rule	For example
<code>/project/id</code>	only lowercase letters [a-z] and hyphens	<code>&lt;id&gt;foo&lt;/id&gt;</code>
<code>/project/name</code>	a human readable name	<code>&lt;name&gt;The Grand Master Foo&lt;/name&gt;</code>
<code>/project/groupId</code>	only lowercase letters [a-z] and dots	<code>&lt;groupId&gt;org.foo.bar&lt;/group</code>

POM Entity (XPath)	Naming Rule	For example
		Id>

## Logging

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]logging]

## Introduction

The `System.out.println` method is often misused to display, in a quick and dirty way, logging or debugging information. The problem with this approach is that often these print methods remain in the source code even when the project is running in production!

Some developers centralize these print methods in a debug class and use a `boolean` flag to turn the debugging on or off. This approach is already a lot better than the first one. Often this approach is only applicable for straight forward debugging purposes and the debug information is displayed in a proprietary format and priority levels are frequently not used.

Different logging APIs have been created to solve the above problems, the best known ones are the Apache Jakarta Log4J [<http://jakarta.apache.org/log4j/docs/>] and the IBM logging toolkit called JLog [<http://www.alphaworks.ibm.com/tech/loggingtoolkit4j>].

With the release of J2SE version 1.4 we finally have a consolidated logging API located in the `java.util.logging` package.

## Why Use Logging?

The J2SE 1.4 logging API can be used to improve problem diagnosis during the life-cycle of a J2EE project by either developers, system administrators and even end users. With the logging API you can capture information such as security failures, configuration errors, bugs etc.

The core logging package includes formatted log records using either plain text or XML. Through the use of *handlers* we can send our log information to the console, streams, console, files, sockets or even implement our own.



### Tip

In the SDK 1.4 documentation you can find an overview of the Logging API and more detailed information on the logging package.



### Tip

Maybe you're still using J2SE version 1.2 or 1.3? No problem because there is an open source implementation of the Logging API called Lumberjack available on SourceForge (<http://javalogging.sourceforge.net/>). This implementation uses the same package structure and class names and can be easily replaced once you've migrated to J2SE 1.4.

## Logging Example

Below you can find a shortened version of a Message Driven Bean using the logging API.

```
/*
```

```
* Copyright notice
*
* File : $RCSfile: logging.xml,v $
*/
package be.vlaanderen.examples.messagebean;

import java.util.logging.Level;
import java.util.logging.Logger;

// Other imports

/**
 * A simple Message Driven Bean example
 *
 * @author Stephan Janssen - JCS Int.
 * @version $Revision: 1.3 $ $Date: 2004/02/25 14:21:03 $
 */
public class FooBean implements MessageDrivenBean, MessageListener {

    /** Initialize the logging API here and use it for any std. error output */
    private static final Logger logger =
        Logger.getLogger(FooBean.class.getName()); ❶

    // Bean methods

    /**
     * The onMessage method is called when a message
     * has been produced for
     *
     * @param msg The asynchronous received message.
     */
    public void onMessage(Message msg) {
        String txtMessage = null;
        try {
            txtMessage = ((TextMessage) msg).getText();
            helperMethod(txtMessage);
        } catch (JMSEException e) {
            logger.log(Level.WARNING, "JMS Problem", e); ❷
        }
    }

    /**
     * An example of a helper method which
     * can be called by the onMessage method.
     *
     * @param text A String value from the onMessage method
     */
    public static void helperMethod(String text) {
        logger.info(text); ❸
    }
}
```

The above example is compliant to the following logging rules:

- ❶ Rule LOG\_001: Retrieve A Logger Based On The Fully Qualified Package And Class Name
- ❷ Rule LOG\_003: Log A Caught Exception
- ❸ Rule JAC\_065: Do Not Unnecessary Use The System.out.print or System.err.print Methods (High)

**Tip**

If you want to create your own a Log Handler using JDBC then have a look at <http://www.developer.com/db/article.php/1468351>.

## Logging Levels

The Logging API supports 7 standard log levels through the `java.util.logging.Level` class.

**Table 2.6. Logging Levels**

Level	Description
SEVERE	Use this log level for non-recoverable or caught runtime (unchecked) exceptions. The severe level has the highest importance. See also rule LOG_005: Use The Log Level SEVERE Only For Non Recoverable Problems.
WARNING	Use this log level for recoverable or checked exceptions. See also rule LOG_006: Use The Log Level WARNING Only For Recoverable Problems.
INFO	Use this level for informational logs. See also rule LOG_007: Use The Log Level INFO Only For Information Logs.
CONFIG	This log level should be used for configuration messages or related problems. See also rule LOG_008: Use The Log Level CONFIG Only For Configuration Problems.
FINE	Can be used for temporary debug info.
FINER	Can be used for temporary debug info.
FINEST	The finest level is has the lowest importance and can be used for temporary debug info.

Each of the above log levels has a related convenience method named after the level. For example:

```
logger.severe("This is a severe problem");
logger.warning("A warning log message");
logger.info("An information log message");
```

When using the SEVERE or WARNING log levels you might also want to display the related exception, this can be done as follows:

```
try {
    txtMessage = ((TextMessage) msg).getText();

    helperMethod(txtMessage);
} catch (JMSEException e) {
    logger.log(Level.WARNING, "JMS Problem", e);
}
```

**Tip**

If you need to collect a lot of information for your log message or create some specific handlers than use the `isLoggable` method within the `Logger` class. For example:

```
if (logger.isLoggable(Level.FINE)) {
    // prepare your log message here
}
```

## Localized Messages

The text used for each log message can be localized through the use of a `ResourceBundle`. The example below shows a localized warning message using the Dutch `ResourceBundle` for Belgium.

```
private static final Logger logger =
    Logger.getLogger(Demo.class.getName(),
        "be.vlaanderen.examples.MyResources");
// ...

Locale currentLocale = new Locale("nl", "BE");

ResourceBundle myResources =
    ResourceBundle.getBundle(
        "be.vlaanderen.examples.MyResources",
        currentLocale);

logger.warning("WARNING1");
```

The supported resource bundle for this localized logging example needs to look as follows:

```
package be.vlaanderen.examples;

import java.util.ListResourceBundle;

public class MyResources_nl extends ListResourceBundle {

    static final Object[][] contents = {
        { "WARNING1", "Voorbeeld van een nederlandstalige boodschap." },
    };

    public Object[][] getContents() {
        return contents;
    }
}
```

This simple logging program generates the following output:

```
22-okt-2003 14:19:04 be.vlaanderen.examples.Foo main
INFO: Voorbeeld van een nederlandstalige boodschap
```

## Configuration File

The project logging features can easily be maintained by using a centralized configuration file. In this file we can setup a default logging level or define a separate level per package or even just turn the logging off for a given package. The logging configuration file is based on the `java.util.Properties` format using key/value pairs. We'll name this configuration file `logging.properties` and store it in the `conf` directory.

Next to the log level we can also configure the different logging handlers which the project can use. The Logging API supports five handlers: `StreamHandler`, `ConsoleHandler`, `FileHandler`, `SocketHandler` and `MemoryHandler`. We simply add these handlers to the configuration file and the project will use the given handlers once it's restarted.



### Tip

Use the `readConfiguration` method in the `LogManager` class when the logging configuration is changed and you do not want to restart your application. This method



could also get triggered through the use of JMX. See the chapter about JMX.

The logging configuration example below sets the default log level to CONFIG, uses the File and Console handlers and will redirect all log records in the user home directory to the log file `guidelines%u.log`.

```
.level = CONFIG
handlers = java.util.logging.FileHandler, java.util.logging.ConsoleHandler
java.util.logging.FileHandler.pattern = %h\guidelines%u.log
be.vlaanderen.examples.level = WARNING
```

The application can only use the above configuration file when the JVM parameter `java.util.logging.config.file` is set correctly:

```
java -Djava.util.logging.config.file=..\..\conf\logging.properties
    be.vlaanderen.examples.Foo
```

## Database Independency

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]jdbc](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]jdbc)]

### Introduction

The JDBC API is a Java API for accessing virtually any kind of tabular data. The API consists of a set of classes and interfaces written in the Java programming language that provide a standard API for tool/database developers and makes it possible to write industrial strength database applications using an all-Java API.



#### Note

JDBC is the trademarked name and is not an acronym. Nevertheless, JDBC is often thought of as standing for *Java Database Connectivity*.

The JDBC API makes it easy to send SQL statements to relational database systems and supports all dialects of SQL.

The value of the JDBC API is that an application can access virtually any data source and run on any platform with a Java Virtual Machine. In other words, with the JDBC API, it isn't necessary to write one program to access a Sybase database, another program to access an Oracle database, another program to access an IBM DB2 database, and so on. One can write a single program using the JDBC API, and the program will be able to send SQL or other statements to the appropriate data source. And, with an application written in the Java programming language, one doesn't have to worry about writing different applications to run on different platforms. The combination of the Java platform and the JDBC API lets a programmer write once and run anywhere.

For an application to be platform and database independent some rules need to be taken into account. The sections that follow provide an overview of the things to look out for.

### JDBC Basics

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]jdbc.basics](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]jdbc.basics)]

### Packages

The standard JDBC classes can be found in the `java.sql` package. Since JDK 1.4 some JDBC extension are also included in the core J2SE distributions. These extensions are located in the `javax.sql`

package.

## Database Driver

If you want to use a database in your application the first thing you need to do is to establish a connection with it. Creating a database connection (`java.sql.Connection`) is delegated to a database driver (`java.sql.Driver`). The drivers themselves are managed by the `java.sql.DriverManager`. See the section about `DriverManager`.

The `java.sql.Driver` concept is the starting point for writing platform and database independent applications. All database vendors have specific `java.sql.Driver` implementations that can be used to connect to their database(s) using the JDBC API.



### Tip

The JDBC driver database (<http://servlet.java.sun.com/products/jdbc/drivers>) on the java sun website allows you to look for a specific driver for your database.

The JDBC API defines 4 driver categories (referred to as types). The types were defined based on:

- The fact if the driver uses native (OS specific) code or is written completely in java
- The fact if the connection to the database is direct or indirect

**Table 2.7. JDBC Driver Types**

Type	Description	All Java	Network connection
1	JDBC-ODBC bridge  Allows for JDBC database access via ODBC.  Requires for the ODBC binary code to be loaded on each client wanting to connect to the database.	No	Direct
2	Native-API partly-Java driver  Converts JDBC calls into calls on the client API for the target database.  Like the type 1 driver, this type of driver requires that some operating system-specific binary code be loaded on each client machine.	No	Direct
3	JDBC-Net pure Java driver  Translates JDBC calls into a DBMS-independent net protocol, which is then translated to a DBMS protocol by a server. This net server middleware is able to connect its pure Java clients to many different databases.	Client: Yes  Server: Maybe	Indirect (use server middleware)
4	Native-protocol pure Java driver	Yes	Direct

Type	Description	All Java	Network connection
	Converts JDBC calls directly into the network protocol used by the DBMS. This allows a direct call from the client machine to the DBMS server.		

**Tip**

The type of driver you choose only affects the level of platform independence. Types 4 and 3 are the preferred way to access databases using the JDBC API. Type 4 being completely platform independent and type 3 allowing for client platform independency.

## Database Driver Management

The `java.sql.DriverManager` is responsible for managing the available `java.sql.Drivers` and is used to create the actual database connection.

The only method in the `DriverManager` class that needs to be used is `getConnection`. As its name implies, this method gets a connection to the database.

**Tip**

An application may call the `DriverManager` methods `getDriver`, `getDrivers` and `registerDriver` as well as the `Driver.connect` method `connect`, but in most cases it is better to let the `DriverManager` class manage the details of establishing a connection.

One of the arguments in `DriverManager.getConnection` method is the database URL. This URL specifies which database to connect to. Each `java.sql.Driver` has its own URL syntax. The `DriverManager` takes the URL and tries to locate the `Driver` that needs to be used to establish the connection.

For the `DriverManager` to be able to create the connection, it must know which JDBC drivers are available. Registering JDBC drivers with the manager can be done in two ways:

- Setting the `jdbc.drivers` system property. The value for this property is a colon-separated list containing the driver class names that need to be loaded by the `DriverManager`.
- Loading the driver at runtime using `Class.forName("driverClassName")`

**Tip**

The `Class.forName` approach for registering JDBC drivers is the preferred one.

## Creating A Connection

Creating a connection to a database involves the following steps:

1. Loading the database driver

2. Asking the `DriverManager` to create a connection using a database URL that is conforming the syntax used by the JDBC driver.

Example that connects to a database using the JDBC\_ODBC bridge driver:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con = DriverManager.getConnection("jdbc:odbc:MyDb",
"myLogin", "myPassword");
```

## Using DataSource

Since JDBC 2.0 the use of `javax.sql.DataSource` provides an alternative to creating a database connection. If available, this technique is preferred over the `DriverManager.getConnection` approach.

The `DataSource` interface is implemented by a driver vendor, the same way the driver vendor has a specific JDBC driver implementation.

**Table 2.8. DataSource Types**

Type	Description
Basic implementation	Produces a standard <code>java.sql.Connection</code> object.
Connection pooling implementation	Produces a <code>javax.sql.PooledConnection</code> object that will automatically participate in connection pooling. This implementation works with a middle-tier connection pool manager.
Distributed transaction implementation	<p>Produces a <code>javax.sql.XAConnection</code> object that may be used for distributed transactions and almost always participates in connection pooling.</p> <p>This data source does not implement the standard <code>javax.sql.DataSource</code> interface but the <code>javax.sql.XADataSource</code>.</p> <p>This implementation works with a middle-tier transaction manager and almost always with a connection pooling manager.</p>



### Important

Not all data sources support all of these 3 implementations. Check the driver vendor's documentation.

An object that implements the `DataSource` interface will typically be registered with a naming service based on the *Java Naming and Directory* (JNDI) API. Clients that want to connect to the database simply have to lookup the correct `DataSource` in the JNDI tree and ask it for a connection.

## Executing SQL Statements

Once you have obtained a connection to the database you can start executing statements. The steps for executing a statement are:

1. Ask the connection to create a statement

## 2. Execute the statement

**Table 2.9. Statements Types**

Type	Description
<code>java.sql.Statement</code>	<p>The basic SQL statement. Accepts any SQL statement as a raw <code>String</code>.</p> <p>For example:</p> <pre>Statement stat =     connection.createStatement(); stat.executeQuery(     "SELECT name FROM person");</pre>
<code>java.sql.PreparedStatement</code>	<p>A SQL statement that is precompiled and stored in a <code>PreparedStatement</code> object. The same statement can then be executed multiple times.</p> <p>In a prepared statement the SQL is provided as a <code>String</code> with the exception that the input parameters are provided as <code>?</code>. The values for the input parameters are then provided using the various setters available on the <code>PreparedStatement</code> interface.</p> <p>For example:</p> <pre>PreparedStatement stat =     connection.prepareStatement(         "SELECT name FROM person "         + "WHERE dob=?"); stat.setDate(1, new Date()); stat.executeQuery();</pre>
<code>java.sql.CallableStatement</code>	<p>Provides access to stored procedures. Uses a SQL escape syntax that allows stored procedures to be called in a standard way for all databases. The input parameters are treated in the same way as with the <code>PreparedStatement</code>.</p> <p>For example:</p> <pre>CallableStatement stat =     connection.prepareCall(         "call findPerson(?)"); stat.setDate(1, new Date()); stat.execute();</pre>

**Tip**

It is recommended to use the `PreparedStatement` as much as possible. Not only for the fact that they are precompiled and can be reused but also because the `PreparedStatement` hides the complexity of formatting values like dates and strings that are used in the statement. The driver handles the formatting. This is very important when you want to keep your code database independent. The formatting of a date (or timestamp) can be (and probably is) database dependent.



### Important

Prepared statements are linked to the connection on which they were created. Closing the connection will render the use of prepared statements on that connection useless. JDBC 3.0 defines support for caching prepared statements. An application may find out whether a data source supports statement pooling by calling the `DatabaseMetaData.supportsStatementPooling`. If the return value is true, the application can then choose to use `PreparedStatement` objects knowing that they are being pooled.



### Important

In many cases, reusing SQL statements is a significant optimization. This is especially true for complex prepared statements. However, it should also be noted that leaving large numbers of statements open could have an adverse impact on the use of resources.

## Processing Results

The base interface for interpreting results that were returned after executing a query is the `java.sql.ResultSet`. After a result set is initially returned the cursor is positioned before the first result. Calling the `next` method the first time will give you access to the first result.

The `ResultSet` interface provides some other methods that can be used to navigate through the `ResultSet`'s contents. Other methods include:

- `afterLast`
- `beforeFirst`
- `previous`
- `first`
- `last`

The `next` method is always available the other navigation methods are optional and their availability depends on the JDBC driver being used. The availability of the optional navigation methods can be checked using the `getType` method on the `ResultSet`. If the return value is `ResultSet.TYPE_FORWARD_ONLY` none of the additional navigation methods are available.

Once you are positioned on a record in the `ResultSet` you can start retrieving values. For each type of value the `ResultSet` has two getters: one for accepting an index as an argument and one for accepting a column name as an argument:

- `getString(int index)` // numbering starts at 1
- `getString(String name)`

## Errors And Warnings

The JDBC API uses the `java.sql.SQLException` to report errors it has encountered. The `SQLException` is a chainable exception meaning that one exception can contain a link to another `SQLException`. See also rule JDBC\_010: Check For A Nested `SQLException`.

Besides exceptions, the JDBC API also knows the notion of warnings (`java.sql.SQLWarning`). Warnings are linked to almost every main JDBC interface, such as:

- `java.sql.Connection`
- `java.sql.Statement`
- `java.sql.ResultSet`

Warnings can be retrieved using the `getWarning` method on any of those interfaces. Just like the `SQLException`, the `SQLWarning` is also chainable.

Except for logging the warning there is not much you can do with it, unless you start examining the warning code (which is database dependant).

Once you have treated the warning you can clear it by calling the `clearWarnings` method.

## Use Of SQL

When you use the JDBC API to access a database you will end up having to write SQL statements. The most common SQL standard (today) is SQL-92 but often database vendors provide some database specific extensions to the standard SQL syntax.



### Important

Keep in mind that once you start using the extensions offered by the database vendor you are depended of their database.



### Note

JDBC 3.0 did implement a subset of the SQL-99 standard but the chances are that the database (or the driver) does not support it yet.

## Know Your Target Database(s)

Even if you follow all basic conventions mentioned here (not using vendor specific classes, sticking to standard SQL, etc.), your application may still not be portable across different database platforms.

Why? Each database has its own *special* features. In order to find out what these features are and how they can affect your code you must study the database. Unfortunately there are no strict guidelines that specify which topics you should examine in detail.

There are however some general topics that should be taken into account, such as:

- Is the database case sensitive when it comes to executing SQL? For example are column names case sensitive?
- How does the database deal with null values? For example does the database treat an empty string or zero number as being `null`?
- Which character set does the database use? This is important for internationalized applications.

- How does the database (and its driver) handle a CLOB or BLOB?
- Are there database specific performance issues?
- What about data conversions, such as double, float, BigDecimal, etc.?
- What about data precision? For example are the nanoseconds for a TimeStamp stored?

## Architecture

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]jaas.architecture](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]jaas.architecture)]

### Use An Existing Persistence Strategy

Using external data stores to store data is not something new. Almost every application needs it. Today a lot of existing persistence strategies exist. Some examples include:

- Container Managed Persistence Entity Beans (CMP EJB)
- JDO
- Hibernate
- TopLink

If possible try to use one of these existing implementations. It will make life a lot easier and allow you to concentrate on the application's logic and not so much on the plumbing.

### Introduce A Persistence Layer

Writing portable JDBC code implies that you don't write database specific code. Unfortunately this is not always possible. Sometimes you are forced to use vendor specific classes or SQL. In this case it is important that you isolate your database access code in a persistence layer.

If you ever you need to migrate to or support another database then you only have to change (or extend) the code in the persistence layer. The most common *design pattern* that is used in such a persistence layer is the Data Access Object (DAO). The DAO is one of the J2EE design patterns J2EEDP but can also be applied to non J2EE environments.

The DAO contains all code that is used to access stored data. The persistence layer can contain one or more DAOs. When you need to migrate to another database the only code you have to change is located in these DAOs. You could think of a scenario where you have more then one version of the same DAO but for different databases. In this last case you could determine at runtime which DAO to use depending on the database you need to connect to.

Even if you use an existing persistence strategy you might still be forced to do some manual SQL coding, for example DAOs in combination with CMP EJB.

## JMS

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]jms](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]jms)]

### Introduction

A number of different Message-Oriented Middleware (MOM) products exist. Java Message Service



(JMS) was introduced to abstract access to these products, in the same way JDBC abstracts access to relational databases and JNDI abstracts access to naming and directory services. A MOM product implementing the JMS API is called a JMS provider. Java applications can use the JMS API to access the JMS providers in a vendor neutral way, allowing them to run on more JMS providers without changing code. See rule JMS\_001: Do Not Use A Vendor Specific Class.

JMS provides an asynchronous messaging system so that different systems can be integrated without being tightly coupled. One system can send messages to a second system, without requiring that the second system is running. The second system can process the messages when it is convenient. The sender is not required to wait for the message being received by the recipient.

It is also possible to use JMS in a synchronous way, for instance to implement a request/reply mechanism. Be sure to investigate carefully if JMS is the best choice in this case.

## Overview Of JMS

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]jms.overview](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]jms.overview)]

A JMS application consists of one or more JMS servers and a number of JMS clients. A JMS client that produces messages is called a *producer*. A JMS client that consumes messages is called a *consumer*.

## Structure Of A JMS Message

A JMS message contains:

- a number of predefined headers
- a number of custom properties
- a body

The JMS API provides a number of messages, based on the body type of the message:

- `TextMessage`
- `BytesMessage`
- `ObjectMessage`
- `StreamMessage`
- `MapMessage`
- `XMLMessage`

Some providers define more messages. In order to have a portable application, you shouldn't use these proprietary messages. See rule JMS\_001: Do Not Use A Vendor Specific Class. It is also important to choose the message type carefully to avoid unnecessary memory or processing overhead.



### Tip

Watch out for persistent `ObjectMessages`. Make sure that the class of the object in an `ObjectMessage` at consuming time is compatible with the class of the object at producing time. This could happen when a new version of an application is deployed while some `ObjectMessages` are in the persistent store of the JMS server.

The predefined message headers are listed in the following table. A message provides get and set methods for each predefined header. The JMS client can set some of these headers, the vendor sets the other headers.

**Table 2.10. Header Types**

Header	Set by	Description
JMSMessageID	provider	The JMS provider attaches a unique message id to each produced message.
JMSDestination	provider	This header contains the destination of the message.
JMSDeliveryMode	provider	This header indicates if the message should be persistent or not.
JMSExpiration	provider	<p>This header is a calculated based on the time to live specified in the message producer. It indicates when a message should be destroyed on the JMS server. Be careful with this header.</p> <p>A JMS provider typically has a background thread to clean up all expired messages every once a while. Depending on how often this clean up is executed, messages that should have been destroyed, could still be alive. Executing the clean up with very short intervals will result in better results for the time to live aspect, but could have performance consequences.</p>
JMSPriority	provider	This header indicates the priority a message should have for delivery to consumers. Values go from 0 to 9. Messages with a higher priority will be delivered before messages with a lower priority.
JMSCorrelationID	application	This optional header can be used to indicate that the current message is a reply to another message. The message id of the original message is put in the correlation id header of the reply.
JMSReplyTo	application	This optional header can be used to specify where a reply to this message can be put. The value can be the name of an administered destination or a temporary destination.
JMSRedelivered	provider	This header is set when a message is redelivered.
JMSTimestamp	provider	When a message is sent, the JMS provider puts a timestamp on the message.
JMSType	application	Some JMS providers use a message repository that contains the definitions of messages sent by applications. The JMSType header field may reference a message's definition in the provider's repository

## Messaging Models

A distinction is made between two messaging models: point-to-point and publish-and-subscribe. A JMS Provider can implement one or both of these models:

- Point-to-point (PTP)

- Publish-and-subscribe (pub/sub)

## Point-to-point

Point-to-point (PTP or p2p) is used for one-to-one messaging, with one sender (producer) who puts messages on a queue and one receiver (consumer) who consumes messages from a queue.

A message is delivered to only one receiver, even if more than one receiver is consuming messages from a specific queue. If no receivers are taking messages from a queue, the messages stay on the queue until a receiver becomes active.



### Tip

It is possible to use more than one receiver to consume from a single queue. This could be a way of load balancing or concurrent processing of messages.



### Note

The JMS provider only deletes a message from a queue after one consumer has received and acknowledged the message.

## Publish-and-subscribe

Publish-and-subscribe (pub/sub) is used for one-to-many messaging or broadcasting, with one publisher (producer) who puts messages on a topic and any number of subscribers (consumers) who consume messages from a topic.

A message is copied and delivered to all subscribers of a topic. Only subscribers that are active get a copy of the message. A durable subscriber however can also consume messages that were produced when the subscriber was not active.



### Note

The JMS provider only deletes a message from a topic after all active consumers and all durable consumers have received and acknowledged a copy of the message.

## Administered Objects

JMS administered objects are created by a JMS administrator, not programmatically. JMS includes two types of administered objects:

- Connection factories
- Destinations



### Tip

There is a convention of registering these administered objects in the JNDI namespace, so JMS clients can look them up in the JNDI namespace. See rule JEN\_017: Use A Correct Name For A JMS Environment Reference Name (High). By using JNDI, JMS client applications remain portable across different JMS servers.

## Connection Factories

For each messaging model, there exists a connection factory: `QueueConnectionFactory` and `TopicConnectionFactory`. A connection factory is used to create connections to the JMS server.

## Destinations

In the context of PTP messaging, a destination is called a *queue*. In pub/sub, a destination is called a *topic*. These destination objects (`Queue` or `Topic`) support concurrent use. A JMS message is sent to a destination by a *producer*, and picked up from the destination by a *consumer*.

Temporary queues and topics are not administered objects. They can be created and deleted programmatically.



### Note

There are certain restrictions in using temporary queues however, so they should only be used in certain circumstances. The main restriction is that only the JMS session that created the temporary destination is able to consume messages from it.

## Connections

A JMS connection (`Connection`) represents a physical connection between the JMS client and the JMS server. JMS connections are rather heavyweight objects and allow concurrent use, so a JMS client application typically creates one connection for each messaging model it uses (`QueueConnection` and `TopicConnection`).



### Note

It is not possible to do both PTP and pub/sub messaging over the same JMS connection. This will be possible in a future release of the JMS specification.



### Tip

JMS provides an `ExceptionListener` interface to notify JMS clients of lost connections. It is the responsibility of the JMS provider to call the `onException` method of all registered `ExceptionListeners` after making reasonable attempts to reestablish the connection automatically. This is especially useful for a JMS client that only consumes messages using a `MessageListener`. In this case the JMS client won't detect the lost connection because it doesn't make any JMS calls itself.

## Sessions

A JMS session (`QueueSession` and `TopicSession`) is created from a JMS connection and represents a client's conversational state with a JMS server. A session is considered a lightweight object and was designed for single threaded use only. It supports a single series of transactions and defines a serial order for the messages it consumes and the messages it produces. A session is also a message factory.



### Note

JMS defines a serial order for messages only within a JMS session. Across different JMS sessions, this is no longer true. See rule JMS\_010: Do Not Rely On JMS To Deliver Message In The Same Order As They Were Sent.

## Delivery Mode

JMS messages can be persistent or non-persistent. Persistent messages are logged and stored to a stable storage. If messages are persistent, there is a guarantee that they will be delivered at least once.



### Tip

Use persistent message if messages may never be lost in transit. Non-persistent messages can be lost if the JMS server crashes.



### Note

Persistent messaging is slower than non-persistent messaging.

## Message Sending

Sending messages is done with a `QueueSender` or a `TopicPublisher`, which both extend `MessageProducer`.

The send and publish methods are synchronous. These methods only return successfully if the JMS server acknowledged the message.



### Tip

It is possible to send the same message object more than once, even to different destinations. The message producer will take a copy of the message before putting it on the destination.



### Tip

A `MessageProducer` holds default values for priority, time to live and delivery mode. The `MessageProducer` will set the according headers on the messages while they are being sent.

## Message Receiving

A client uses a `MessageConsumer` object to receive messages from a destination:

- `QueueReceiver` for PTP
- `TopicSubscriber` for pub/sub

A client may either synchronously receive a message consumer's messages or have the consumer asynchronously deliver them as they arrive.

For synchronous receipt, a client can request the next message from a message consumer using one of its receive methods that allow a client to poll or wait for the next message.

For asynchronous delivery, a client can register a `MessageListener` object with a message consumer. As messages arrive at the message consumer, it delivers them by calling the `MessageListener`'s `onMessage` method.

**Note**

The JMS server only considers a message delivered after it has received an acknowledgment from the client. This is important for message redelivery.

**Important**

The choice between synchronous or asynchronous is very important. Synchronous message receiving should be preferred in a normal JMS client application. In web, session and entity bean components, only synchronous message receiving should be used. Message-driven beans were designed for asynchronous message receiving in a J2EE environment.

## Message-driven Beans

Session beans, entity beans and web components can all act as JMS producers. However, they should only consume messages synchronously using the `MessageConsumer`'s receive methods, because they are driven by synchronous request-reply protocols. See rule JMS\_015: Do Not Receive Messages Asynchronously In A Web Component, A Session Bean Or An Entity Bean. Only the message-driven bean and application client components can both produce and consume asynchronous messages. Message-driven beans were introduced in EJB 2.0. A Message-driven bean implements the `MessageListener` interface, like a normal asynchronous message consumer, but the EJB container manages its lifetime and possibly the transactions.

**Note**

A message-driven bean doesn't have a business interface, so it cannot be called synchronously. It can only be triggered by JMS messages.

**Note**

Message-driven beans are like stateless session beans; they don't maintain state between requests.

**Tip**

A big advantage of message-driven beans is that they can be used for concurrent processing. This is the case when the EJB container decides to select different bean instances to process different messages. This leads to higher throughput and better scalability in a robust server environment.

**Note**

Message-driven beans should not attempt to use the JMS API for message acknowledgment. Message acknowledgment is automatically handled by the container.

If the message-driven bean uses container managed transaction demarcation, message acknowledgment is handled automatically as a part of the transaction commit (the `Required` transaction attribute must be used).

If bean managed transaction demarcation is used, the message receipt cannot be part of the bean-managed transaction, and, in this case, the receipt is acknowledged by the container.

## Selectors

Message selectors are used for message filtering. A consumer can be configured to receive only messages with properties matching the message selector expression.



### Tip

It is important to evaluate the choice of putting messages on several destinations without using selectors, or putting messages on a single destination and dispatching them using selectors. The performance of using selectors should also be evaluated, especially if the destination holds a lot of messages.

## Acknowledgement Modes For Message Receipt

The JMS provider considers a message to be consumed when it receives an acknowledgement for the message. A JMS session can be created with one of the following acknowledgement modes:

- `Session.AUTO_ACKNOWLEDGE`
- `Session.CLIENT_ACKNOWLEDGE`
- `Session.DUPS_OK_ACKNOWLEDGE`

The acknowledgement mode is only taken into account in a non-transactional session. Choosing the acknowledgement mode has serious consequences for message redelivery.

### AUTO\_ACKNOWLEDGE

With this acknowledgment mode, the session automatically acknowledges a client's receipt of a message either when the session has successfully returned from a call to receive or when the message listener the session has called to process the message successfully returns.



### Warning

If the `receive` or `onMessage` method throws an exception, the message will be redelivered automatically. It is also possible that the JMS provider fails to acknowledge the message. In these cases, the message will be redelivered (even if the message was successfully processed the first time), but the `JMSRedelivered` flag will be set. To guard against duplicate messages, the application should check this flag. A common solution to track duplicate messages is to keep message ids in a database table with a processed flag. This warning also holds for the `DUPS_OK_ACKNOWLEDGE` mode.

### DUPS\_OK\_ACKNOWLEDGE

This acknowledgment mode instructs the session to lazily acknowledge the delivery of messages. This means that the JMS provider is not obliged to acknowledge each message. This could result in more messages being redelivered after a system failure.



### Tip

Choose this mode if messages may be delivered more than once, but performance is more important than reliability. Under certain circumstance, such as a heavy load, there may be no performance gain.

## CLIENT\_ACKNOWLEDGE

With this acknowledgment mode, the client acknowledges a consumed message by calling the message's `acknowledge` method. This mode gives the JMS client application complete control over message acknowledgements. The `acknowledge` method informs the JMS provider that the message has been successfully received by the consumer. This method throws an exception in the case of a provider failure during the acknowledgement process. This results in redelivery of the message. The application should either undo the processing of this message or be prepared to ignore the redelivery of this message. The `acknowledge` method only has effect with this acknowledgement mode.



### Tip

Calling the `acknowledge` method results in acknowledging all messages that were received in the current JMS session since the previous call to `acknowledge` (all previously unacknowledged messages are acknowledged). This gives the application the ability to group messages and consume either the whole group or no messages at all. All unacknowledged messages will be redelivered when a JMS session is restarted (for instance after a failure) or after calling the `recover` method on the session.

## Transactions

Local transactions involve work performed on a single resource, like one JMS provider or one database. JMS also supports global transactions. These involve work performed across several different resources, for instance several databases and JMS providers.

### Local Transactions

JMS sessions can be either transactional or non-transactional. When a session is transacted, all messages sent or received using that session, are automatically grouped in a transaction. The transaction remains open until either the `rollback` or the `commit` method is called. At this point a new transaction is started. If a transaction is rolled back or a failure occurs, all messages sent in the transaction are discarded. All received messages in the transaction will be redelivered with the `redelivered` flag set.

Transacted producers and consumers can be in a single transaction only if they were created from the same session object. This allows a JMS client to produce and consume messages in one single transaction. This can be useful if you want to send some messages as a result of receiving one or more messages.



### Warning

It is important to realize that the production and the consumption of a JMS message can never be done in the same transaction. So waiting for a reply on a message that is sent in a transaction will always result in a deadlock. See rule JMS\_009: Do Not Produce A Message In A Transaction And Rely On It To Be Consumed Before The End Of The Transaction. The replier will never see the message until the transaction is committed.



### Note

For the moment, it is not possible to combine PTP and pub/sub messaging in a single transaction. This will be possible in future releases of the JMS specification.



**Tip**

Only use transactions if the functionality of acknowledgement modes is not sufficient.

**Table 2.11. Local Transactions**

Type	Description
Non-transactional Sending	Each sent message is automatically put on the destination.
Transactional Sending	Sent messages are only put on the destination when the transaction commits. A transaction includes all messages sent since the last commit or rollback. A commit or a rollback automatically starts a new transaction. A rollback of a transaction results in no messages being put on the destination.
Non-transactional Receiving	Message redelivery is done according to the acknowledgement mode.
Transactional Receiving	Messages are acknowledged when the transaction commits. A rollback of a transaction automatically causes message redelivery.

**Global Transactions**

The two-phase commit protocol is used by a transaction manager to coordinate the interactions of resources in a global transaction. A resource can only participate in a global transaction if it supports the 2PC protocol, which is usually implemented using the XA interface developed by the Open Group. In the Java enterprise technologies, the XA interface is implemented by the Java Transaction API and XA interfaces (`javax.transaction` and `javax.transaction.xa` packages). Any resource that implements these interfaces can be enrolled in a global transaction by a transaction manager that supports these interfaces.

The JMS specification provides XA versions of a number of JMS objects, like `XAConnectionFactory`, `XAQueueConnection`, `XAQueueSession`, ... Each of these objects works like its corresponding non-XA-compliant object. The transaction manager uses the XA interfaces directly, but a JMS client only sees the non-transactional versions.

**Dead Message Queue**

Some vendors have a notion of a Dead Message Queue. Messages are put on this queue when they are undeliverable. This could happen if they have expired, or if there is a deployment configuration problem. Without a Dead Message Queue, these messages would be lost. Messages from a Dead Message Queue can also be consumed, as from a normal queue. An application should provide some logic for handling dead messages, or at least make sure the Dead Message Queue doesn't fill up.

**JMS Examples**

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]jms.examples](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]jms.examples)]

**Sending Messages From A JMS Client**

There are a number of steps to be taken in order to send messages:

- ❶ Lookup connection factory (JMS administered object)
- ❷ Create connection: choose between PTP and pub/sub
- ❸ Lookup destinations (JMS administered objects)

- ④ Create sessions (single threaded context!)
- ⑤ Create senders
- ⑥ Create message
- ⑦ Send message
- ⑧ Close resources

```
public void testSend() {
    QueueConnection qcon = null;
    QueueSession qsession = null;
    QueueSender qsender = null;
    try {
        Hashtable env = new Hashtable();
        env.put(Context.INITIAL_CONTEXT_FACTORY,
            "weblogic.jndi.WLInitialContextFactory");
        env.put(Context.PROVIDER_URL, "t3://localhost:7001");
        InitialContext ctx = new InitialContext(env);
        QueueConnectionFactory qconFactory = (QueueConnectionFactory)
            ctx.lookup("jms/FooQueueConnectionFactory"); ①

        qcon = qconFactory.createQueueConnection(); ②
        Queue queue = (Queue) ctx.lookup("jms/FooQueue"); ③
        qsession = qcon.createQueueSession(false,
            Session.AUTO_ACKNOWLEDGE); ④
        qsender = qsession.createSender(queue); ⑤

        TextMessage msg = qsession.createTextMessage(); ⑥
        msg.setText("some sample text");
        qsender.send(msg); ⑦
    } catch (NamingException e) {
        // ...
    } catch (JMSEException e) {
        // ...
    } finally {
        try {
            if (qsender != null) {
                qsender.close();
            }
            if (qsession != null) {
                qsession.close();
            }
            if (qcon != null) {
                qcon.close();
            }
        } ⑧
    } catch (JMSEException e) {
        // ...
    }
}
```

## Receiving Messages From A JMS Client

There are a number of steps to be taken in order to receive messages. The following lines describe the steps to be taken, with a full example illustrating these steps.

- ① Lookup connection factory (JMS administered object)
- ② Create connection: choose between PTP and pub/sub
- ③ Lookup destinations (JMS administered objects)
- ④ Create sessions (single threaded context!)

- ⑤ Create receivers
- ⑥ Receive messages
- ⑦ Close resources

```

public void testReceive() {
    QueueConnection qcon = null;
    QueueSession qsession = null;
    QueueReceiver qreceiver = null;
    try {
        Hashtable env = new Hashtable();
        env.put(Context.INITIAL_CONTEXT_FACTORY,
            "weblogic.jndi.WLInitialContextFactory");
        env.put(Context.PROVIDER_URL, "t3://localhost:7001");
        InitialContext ctx = new InitialContext(env);
        QueueConnectionFactory qconFactory = (QueueConnectionFactory)
            ctx.lookup("jms/FooQueueConnectionFactory"); ①

        qcon = qconFactory.createQueueConnection(); ②
        Queue queue = (Queue) ctx.lookup("jms/FooQueue"); ③
        qsession = qcon.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);

        ④ qreceiver = qsession.createReceiver(queue); ⑤

        qreceiver.setMessageListener(new MessageListener() {
            public void onMessage(Message message) { ⑥
                // ...
            }
        });
        qcon.start();
    } catch (NamingException e) {
        // ...
    } catch (JMSEException e) {
        // ...
    } finally {
        try {
            if (qreceiver != null) {
                qreceiver.close();
            }
            if (qsession != null) {
                qsession.close();
            }
            if (qcon != null) {
                qcon.close();
            }
            ⑦
        } catch (JMSEException e) {
            // ...
        }
    }
}

```

## JMS In a Global Transaction

The following example shows how JMS work can be explicitly enlisted in a global transaction.

```

public void testGlobalTransaction() {
    try {
        Hashtable env = new Hashtable();
        env.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialContextFa
        env.put(Context.PROVIDER_URL, "t3://localhost:7001");
    }
}

```

```
InitialContext ctx = new InitialContext(env);
//lookup the transaction manager
TransactionManager transactionManager = (TransactionManager)
    ctx.lookup("tx/TransactionManager");
//start the global transaction
transactionManager.begin();
//get the transaction object that represents the global transaction
Transaction transaction = transactionManager.getTransaction();

XAQueueConnectionFactory qconFactory = (XAQueueConnectionFactory)
    ctx.lookup("jms/FooQueueConnectionFactory");
//create an XA-compliant queue connection
XAQueueConnection xaQueueConnection = qconFactory.createXAQueueConnection();
//create an XA-compliant queue session
XAQueueSession xaQueueSession = xaQueueConnection.createXAQueueSession();
//get the XAResource
XAResource xaResource = xaQueueSession.getXAResource();
//enlist the resource in the current transaction
transaction.enlistResource(xaResource);
// do some JMS work

// Enlist some other XAResources in the current transaction
// and do some work with them

//commit the transaction
transaction.commit();
} catch (NamingException e) {
    // ...
} catch (NotSupportedException e) {
    // ...
} catch (SystemException e) {
    // ...
} catch (JMSException e) {
    // ...
} catch (RollbackException e) {
    // ...
} catch (HeuristicMixedException e) {
    // ...
} catch (HeuristicRollbackException e) {
    // ...
}
}
```

**Tip**

When you are using container-managed transactions, all this is done by the EJB container.

## Assertions

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]Assertions](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]Assertions)]

## Introduction

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]Assertions - Introduction](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]Assertions - Introduction)]

Every programmer makes assumptions in the code he/she writes. When these assumptions are no longer true, you know there is a bug, or your assumption was not correct.

Code written to ensure that an assumption is correct is referred to as an *assertion*.

Using assertions in your code has huge advantages:

- Experience has shown that using assertions in your programs is one of the quickest and most effective ways to detect and correct bugs.
- Assertions provide a way to document your assumptions, enhancing the maintainability and readability of your programs.
- Using assertions during the development of your application allows you to ensure that all the assumptions you made are correct. This increases the quality of your code.

## Coding Assertions

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]Assertions - Coding Assertions](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]Assertions - Coding Assertions)]

All JVM versions 1.4 and higher have a built-in assertion facility. If you want to use assertions in code that needs to run in a pre 1.4 JVM you will have to write your own assertion *framework*.



### Important

Check your deployment platforms to make sure that all platforms have a 1.4 JVM at their disposal before using the 1.4 assertion facility. For example: at the time this document was written WebSphere 5 still used the 1.3 JVM meaning that code using the 1.4 assertions will not run on that platform.

## Before JDK 1.4

Creating an assertion facility is quite simple:

- Provide an `Assert` class with a set of static `assert` methods that are capable of assuring that a given assumption (condition) is correct.
- Add an `AssertionFailedException` exception that is thrown by the `assert` methods when an assumption is incorrect.



### Important

The fact that an assertion fails should be seen as a bug in the code. This means that the `AssertionFailedException` should be a unchecked exception (extending from `java.lang.RuntimeException`).

When writing your own assertion facility you should make sure that the overhead introduced by calling the `assert` methods is minimal. One way of doing this is by using the method-inlining feature available on most JVMs. One way of helping the compiler to determine if a method can be inlined or not is to make the method or the class containing the method `final`.



### Note

Method-inlining is not something that you can enable/or disable. The fact if a method call gets replaced with inline code is entirely up to the discretion of the JVM running the code.

Adding support for enabling and disabling assertions can be useful. Maybe you don't want the assertions to be checked in production code (because you already now, or think, that all assumptions are correct). Here you have two options:

- Add a constant boolean attribute to your `Assert` class indicating if assertions should be performed. A drawback to this approach is the fact that you will have to recompile your `Assert` class when enabling or disabling assertions. This approach was used in the example.
- Determine if assertions are enabled or disabled by evaluating a system property or the content of an external configuration file.

For example:

```
public final class Assert {  
    private static final boolean ENABLED = true;  
  
    public static void assert(boolean condition, String assumption) {  
        if ( (ENABLED) && (!condition) ) {  
            throw new AssertionError(assumption);  
        }  
    }  
  
    public static void assertNotNull(Object something, String name) {  
        if ( (ENABLED) && (something == null) ) {  
            throw new AssertionError(  
                "Expected " + name + " not to be null");  
        }  
    }  
}  
  
public class AssertionError extends RuntimeException {  
    public AssertionError(String message) {  
        super("Assertion failed:" + message);  
    }  
}
```

Then you can use in your code:

```
Assert.assert(voteCount <= population, "Vote count > then population");
```

## Since JDK 1.4

Since version 1.4, the java language has a built-in assertion facility. This has 3 serious advantages:

- You don't need to write your own assertion facility.
- To ensure that assertions are not a performance liability in deployed applications, assertions can be enabled or disabled at runtime.
- Disabling assertions eliminates their performance penalty entirely.

Syntax:

```
assert Expression1;  
assert Expression1 : Expression2;
```

For example:

```
assert (voteCount <= population) : "Vote count > then population";
```

Compiling sources containing 1.4 assertions requires an additional command line option `-source 1.4` when calling **javac**.

For example: **javac -source 1.4**

By default, assertions are switched off, so you must enable them if you want to use them. Enabling/disabling assertions can happen at various levels.

**Table 2.12. JVM Assertion Command Line Options**

Command line option(s)	Assertion behavior
<code>-ea</code> or <code>-enableassertions</code>	Enables assertions in your code
<code>-da</code> or <code>-disableassertions</code>	Disables assertions in your code
<code>-esa</code> or <code>-enablesystemassertions</code>	Enables assertions in the JRE libraries
<code>-dsa</code> or <code>-disablesystemassertions</code>	Disables assertions in the JRE libraries



### Tip

Enabling assertions in your code only is the most commonly used option. You can (usually) assume that the JRE libraries are bug free.

## Design By Contract

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]Assertions - Design By Contract](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]Assertions - Design By Contract)]

*Design by Contract* is a class design technique that shares some similarities with establishing a legal contract. A class contract is an explicit statement of rights and obligations between a client and a server (a client being a class that wants something and a server being a class that has something to offer). The contract states what both parties must do; independent of how it is accomplished.

Class contracts are typically written from the server's perspective: if a client promises to provide *x* when calling a method, then the server promises always to return *y* in response.

The server class can use assertions for:

- *Pre-condition*: ensures that the client lives up to its part of the contract by providing correct input to the server class
- *Post-condition*: ensure that the server lives up to its part of the contract by returning a valid response to the client and ensuring that its internal state is correct.
- *Invariants*: ensure that variable states and values are and remain valid during the course of fulfilling the client's request.

## Pre-conditions

By convention, preconditions on methods that are `public`, `protected` or package local are enforced by explicit checks that throw particular, specified exceptions (a commonly used exception is `java.lang.IllegalArgumentException`). We can't use assertions for this because the method must guarantee that these checks will ALWAYS be done, even if assertions are disabled.

For example:

### WRONG

```
public void setInitialCount(int count) {
    // Enforce specified precondition in public method
    assert count >= 0 : "Illegal initial count: " + count;
    initialCount = count;
}
```

### RIGHT

```
public void setInitialCount(int count) {
    // Enforce specified precondition in public method
    if (count <= 0) {
        throw new IllegalArgumentException(
            "Initial count must be > 0, got " + count);
    }
    initialCount = count;
}
```

You can use assertions to check the pre-conditions of `private` methods because these methods can only be called by methods in your class.



### Important

Remember to throw a specific exception if the method's scope changes.

## Invariants

In some cases you comment your assumptions. An `assert` statement can do the same and it validates your assumption.

For example:

### WRONG

```
if (val == 0) {
    // ...
} else if (val == 1) {
    // ...
} else { // We know that val is 2
    // ...
}
```

### RIGHT

```
if (val == 0) {
    // ...
}
```



```
} else if (val == 1) {  
    // ...  
} else {  
    assert val == 2 : val;  
    // ...  
}
```

## Post-conditions

Use assertions to check the post-condition upon leaving a method. Post-conditions that can be evaluated include:

- Ensuring that what you are returning to the caller lives up to your part contract.
- Ensuring that the object's internal state is correct prior to returning the result

For example:

```
public void deposit(double amount) {  
    if (amount < 0) {  
        throw new IllegalArgumentException("amount < 0");  
    }  
    // ...  
    assert getBalance() >= 0 : "balance < 0";  
}
```



### Tip

Writing message that have a single return statement makes it easier to check the post-conditions. See also rule JAC\_016: Use A Single return Statement (Normal).

## Testing

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]testing](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]testing)]

## Automated Regression Testing

Regression testing is a very important part of iterative software development. Whenever the source code was altered - either to fix a defect, refactor existing code or provide new functionality - regression tests should be run to check if the system has not been broken by the latest changes.

The set of required regression tests typically falls in 2 categories :

- Unit Tests - verify whether the code changes did not affect the correct behaviour of the other components (units).
- Scenario Tests - verify whether the code changes did not affect the other functionalities of the system.



### Tip

When your software is supported by a good coverage of automated regression tests, the developers have much more confidence in the developed code.

Automated regression testing allows the developers to make changes to the code in a controlled way. This is important *throughout the whole development cycle* to allow development of new functionality, to allow refactoring of existing code and to allow bugfixes with confidence. Therefore, automated regression testing should be done *from the very first day* you start implementing the system and should be *kept up to date all the time*.

To make this very explicit, the following guidelines should be followed :

- A component can not be considered to be "finished" until it's unit test is working.
- A use case can not be considered to be "finished" until all it's scenario tests are working.



### Tip

Before checking in a piece of source code into your version control system, make sure that there are unit tests that cover this code and that *ALL* unit tests (not only the ones that cover the code you are checking in) are succeeding.

## Unit Tests (White Box)

A unit test verifies whether a certain component (unit) does what it is supposed to do. It is written to test a particular small unit on its own so there should be no or little need for interaction with other classes or components to execute the test. Unit tests are essentially white box tests because the unit tests are designed together with the component they are testing, using the knowledge of the inner workings of that component.

## JUnit

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]testing.junit](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]testing.junit)]

The de facto standard tool for unit testing in Java is JUnit, a very simple open source unit testing framework. More information about JUnit can be found on <http://www.junit.org>.

The JUnit framework requires you to program your unit tests in Java, using specific JUnit functions and conventions.

To create a unit test in the JUnit framework, you need to design a class that extends the `junit.framework.TestCase` class.

The specific test code must be organized in test methods, i.e. methods that contain testing code. Actual testing is done by using the methods of the class you are testing and comparing the results with the expected result using JUnit assertion methods.

A test method should conform to the following JUnit standards : it's name should start with "test", the return type should be void, the accessibility should be public and it should take no parameters.

For example :

```
public void testMultiply() {  
    double result= value1 * value2;  
    assertTrue(result == 6.0);  
}
```

The `junit.framework.TestCase` base class also provides a `setUp()` and a `tearDown()` method that you can override to allow you to set up the necessary state for your testcase and clean up after it finishes.

The `setUp()` method is called each time before a test method is evaluated and the `tearDown()` method is called after a test method is finished. It is important that the test methods of the test case have no side effects. After the execution of the `tearDown()` method after a test method is finished, the system should be in exactly the same state as before the `setUp()` method was executed.

Example :

```
public class CalcTest extends TestCase {
    protected double value1;
    protected double value2;

    protected void setUp() {
        value1= 2.0;
        value2= 3.0;
    }
}
```

The tests to be run can be collected into a `TestSuite`. JUnit provides different test runners `JUnit4TextRunner` or `JUnit4SwingRunner` which can run a test suite and collect the results. A test runner either expects a static method `suite()` as the entry point to get a test to run or it will extract the suite automatically.

```
public static Test suite() {
    suite.addTest(new CalcTest("testDivide"));
    suite.addTest(new CalcTest("testMultiply"));
    return suite;
}
```

When this `suite()` is missing, JUnit relies on the Java reflection API to dynamically collect all your test methods in a test suite. This dynamic building of the test suite should be preferred over the static way because the static way is more error prone.

When executing a JUnit test suite, JUnit executes the methods in the test suite one by one in arbitrary order (the order is not specified in the JUnit specifications). As soon as an assert statement fails, an explicit fail statement is executed or another exception is thrown, the execution of that test method is aborted and the reason for the failure is logged.

## JUnit Best Practices

- Avoid code duplication in your unit tests.

When your unit tests often repeat the same blocks of testing code, factor out the duplicate code and maybe even move it to a common superclass.

- Do not use the test-case constructor to set up a test case

When something goes wrong during the initialization in the constructor and an exception is thrown, JUnit can not report this exception. Use the `setUp()` method instead.

- Don't rely on the order in which tests within a test case are executed.

When dynamically building the test suite, the JUnit framework offers no guarantees about the order of the tests. In any case, all tests should be totally independent.

- Avoid writing test cases with side effects

A test should leave no traces behind. E.g. a test that inserts data in the database should also delete this data, even when the test fails. Otherwise, other tests might fail due to an inconsistent state of the

system.

- Call a superclass's `setUp()` and `tearDown()` methods when subclassing

When subclassing `TestCase` for your project, allow the superclass to do the necessary setup and tear-down activities.

- Isolate your test data from your test code.

Separating test data allows you to change the test data without changing the test code.

- Do not load data from hard-coded locations on a filesystem

Tests often need external data. When loading this data, try using a more portable approach like retrieving the data from the classpath using `getResource()` or `getResourceAsStream()`.

- Name test methods properly

The name of a test method should not only start with "test" but also be very verbose e.g. `testDivisionByZero()`.

- Ensure that tests are time-independent

When your test data expires, it must be updated to keep the tests working.

- Consider locale when writing tests

E.g. if you have to test whether a certain `Date` is a saturday, don't do something like this :

```
// pattern "E" stands for the day of the week
DateFormat dateFormat = new SimpleDateFormat("E");

// check if the date is a saturday
assertEquals("Sat", dateFormat.format(date));
```

The `format` method returns a localized `String` and when this test is run on a computer with english locale, it will succeed. `format` the date to a `String` and look for "saturday". On the other hand, on a computer with french locale settings, the test will fail because the `format` method returns "sam" instead of "Sat".

- Avoid visual inspection

Developers are often tempted to print out result values to visually inspect them, especially when the printed result is not deterministic. This should be avoided. Instead, use the `assert()` methods to compare the result with the expected value. If necessary, use mock objects to make the test more deterministic.

- Keep tests small and fast

The complete set of tests need to be run as regression tests which means they need to be run often and fast.

- Use the reflection-driven JUnit API

The static `suite()` approach can be error prone. It's easy to forget to add a new test to your suite and you won't even notice that your test is not being executed.

- Build a test case for the entire system

The complete set of tests need to be run as regression tests. Building a `TestAll` testcase allows you to do this very easily.

## Unit testing regular Java components with JUnit

In an ideal situation, unit testing comes down to writing a JUnit `TestCase` class for each Java class.

The unit test should test the public and package public methods of the Java class it is testing.

The name of the JUnit testcase should be composed of the name of the class it tests, appended with the "Test" suffix. E.g. the `TestCase` testing the class named `HelloWorld` should be named `HelloWorldTest`.

The package of the JUnit `TestCase` should be the same as the package of the class it tests. E.g. the `TestCase` for `be.jcs.jjguidelines.samples>HelloWorld` should be `be.jcs.jjguidelines.samples>HelloWorldTest`.

The java sources for the testcases should be put under `/test/java` (see the section called "Project Directory Structure").

### Example

In the following example, we write a unit test for a certain class called `EuroAmount`. An object of class `EuroAmount` is immutable (just like `Integer`) and it keeps two numbers, one representing the amount of euros and the other representing the amount of eurocents. The class provides arithmetic methods and we need to verify if these methods are working correctly. Because this specific implementation (storing the amount in two numbers), addition is not as trivial as it could be. We should make sure that `30 eurocents + 85 eurocents` results in `1 euro and 15 eurocents` instead of `115 eurocents`.

The test class resides in the same package as the class it tests. This allows calling package-protected methods. To separate the tests from the production code, the tests are put in a separate directory as the production code.

```
package be.jcs.jjguidelines.examples.junit;
```

Import the JUnit framework classes (and all import statements required to compile the test program).

```
import junit.framework.TestCase;
```

Let the name of the test class be equal to the name of the class to be tested with as suffix `TestCase`. Create a public constructor with one argument: the name of the method.

```
/**
 * This test class tests the EuroAmount class
 */
public class EuroAmountTestCase extends TestCase {

    /**
     * The constructor should always take one parameter: the name of the testmethod
     * @param methodName the name of the test method to be called.
     */
    public EuroAmountTestCase(String methodName) {
        super(methodName);
    }
}
```

When writing a test method, the name of the test method *MUST* start with `test`, as the JUnit framework uses this convention to dynamically build the testsuite through the use of the Java reflection API. The test method should first create everything necessary so the method to be tested can be called, then call the method and verify the results with assert statements.

```
public void testAdd() {
    EuroAmount amount1 = new EuroAmount(0, 30);
    EuroAmount amount2 = new EuroAmount(0, 85);
    EuroAmount sum = amount1.add(amount2);
    assertEquals(sum, new EuroAmount(1, 15));
}
```

Optionally one can create the test data in the `setUp` method and cleanup all data in the `tearDown` method (typically when file, network or database connections are opened). This is typically useful when more test methods use the same test data and the construction of this data is not trivial.

```
public void setUp() {
    amount1 = new EuroAmount(0, 30); // amount1 is a member of the class.
    // ...
}
```

## Unit testing coupled Java components with JUnit and MockObjects

It is often very hard to test individual methods of individual classes without instantiating a lot of collaboration classes. In order to avoid that you test all these collaborating classes along with the class you are unit testing, you should stub out the collaborating classes in your test code. Stubbing out the collaborating classes can be done by using MockObjects.

### Mock Objects

A Mock Object is a substitute implementation to emulate other domain code. It should be simpler than the real code, not duplicate its implementation, and allow you to set up private state to aid in testing. The emphasis in mock implementations is on absolute simplicity, rather than completeness. For example, a mock collection class might always return the same result from an index method, regardless of the actual parameters.

A warning sign of a Mock Object becoming too complex is that it starts calling other Mock Objects. This usually means that the unit test is not sufficiently local. When using Mock Objects, only the unit test and the target domain code are real.

When do you use a MockObject instead of a real object ?

- To restrict the test-area. Since you want to test only the particular class and not it's collaborating classes, you want to avoid that errors are coming from these collaborating classes instead of the class to be tested.
- When the real object is slow. It is important that the tests run in a timely manner. Running all tests can take time, especially if you have a lot of tests and a lot of database calls are performed during the tests. Mocking the database will have the tests performed much quicker.
- To avoid expensive set-up or execution times. Each test should be independent and not fail if another test fails, for example because the failing testcase changed the fixture data (data present in the database that the test depends upon). In a J2EE-environment one needs to deploy the code in an EJB-container or web-container before being able to test it. This can be avoided by mocking functionality provided by these containers or even the database.

- To test conditions that are particularly difficult to set up, such as race conditions, a failing network connection, certain exceptions or other unusual situations that are not encountered during normal execution. With the use of mock objects, it is much easier to simulate these exceptional situations.
- To have deterministic behaviour. If the real object not always returns the same result, substituting this object with a mock object can make the test behave more deterministic (and thus repeatable).
- To stub out user intervention. During tests, the code might ask the user for input. This can be avoided through the introduction of a mock object for this user interface class.
- Because the real object is not available. It could be that another team, project or company develops the real object.

There are several mock object frameworks. The Mock Objects website <http://www.mockobjects.com> gives a nice overview and contains links to various JUnit extensions or add-ons.



### Tip

Typically mock objects are easily introduced when you create factories or other creational design patterns. Do not hesitate to change the design of the project slightly to make it easier unit test your code.



### Tip

Avoid the use of the singleton pattern, because an object of a singleton class can not be replaced with a mock object version of that class.



### Tip

A lot of open source mock objects already exist. Before you write your own mock object, check on the web if you can not reuse an existing mock object instead.

## Example

In this example, a simple class named `HelloWorld` is used. Objects of the class `HelloWorld` can only do one thing : printing "Hello" on a given `PrintWriter`.

```
package be.jcs.jjguidelines.examples.mockobjects;

import java.io.PrintWriter;

public class HelloWorld {
    public void printHello(PrintWriter printWriter) {
        printWriter.println("Hello");
    }
}
```

To test this class, we want to stub out the `PrintWriter` class using a self-made mock class named `Mock-PrintWriter` :

```
package be.jcs.jjguidelines.examples.mockobjects;

import java.io.PrintWriter;
import java.io.OutputStream;
```

```
import java.io.IOException;
import java.util.List;
import java.util.Vector;

public class MockPrintWriter extends PrintWriter {
    private List printedStrings;

    public MockPrintWriter() {
        super(new OutputStream() {
            public void write(int i) throws IOException {
            }
        });
        printedStrings = new Vector();
    }

    public void println(String string) {
        printedStrings.add(string);
    }

    public int countPrintedStrings() {
        return printedStrings.size();
    }

    public String getPrintedString(int index) {
        return (String)printedStrings.get(index);
    }

    public void clearPrintedStrings() {
        printedStrings.clear();
    }
}
```

The MockPrintWriter can pass for a PrintWriter because it extends the PrintWriter class. Apart from that, the MockPrintWriter class also has specific functionality to make it possible to check countPrintedStrings() and getPrintedString() and manipulate clearPrintedStrings() its state.

The HelloWorld class can now be tested in a JUnit test :

```
package be.jcs.jjguidelines.examples.mockobjects;

import junit.framework.TestCase;

public class HelloWorldTest extends TestCase {

    public void testPrintHello() {
        // create the mock object
        MockPrintWriter mockPrintWriter = new MockPrintWriter();

        HelloWorld helloWorld = new HelloWorld();
        // pass the mock object instead of the real object
        helloWorld.printHello(mockPrintWriter);

        // check if exactly one String was printed
        assertEquals(1, mockPrintWriter.countPrintedStrings());

        // check if the first printed String is equal to "Hello"
        assertEquals("Hello", mockPrintWriter.getPrintedString(0));
    }
}
```



## Unit testing the database tier

To aid in the testing of the database tier, another JUnit extension exists : DBUnit.

DBUnit is an open source tool which can help you with the following tasks :

- Putting a database into a known state between test runs. To do this, DBUnit has the ability to quickly export and import database data to and from XML datasets (even very large data sets).
- Verification whether your database data matches an expected set of values.

More information about DBUnit can be found on <http://www.dbunit.org>

## Unit testing Java EJB tier components

Unit testing EJB's is very difficult with the standard JUnit framework. The reason for this is the fact that EJB's can't be instantiated outside an EJB container and stubbing out the EJB container is so hard that it is simply not feasible.

The solution to this problem is testing the EJBs while they are inside their container.

Strictly speaking, such in-container tests are no longer unit tests but integration tests. An integration test verifies that the components of a module are collaborating correctly with each other and with the container itself.

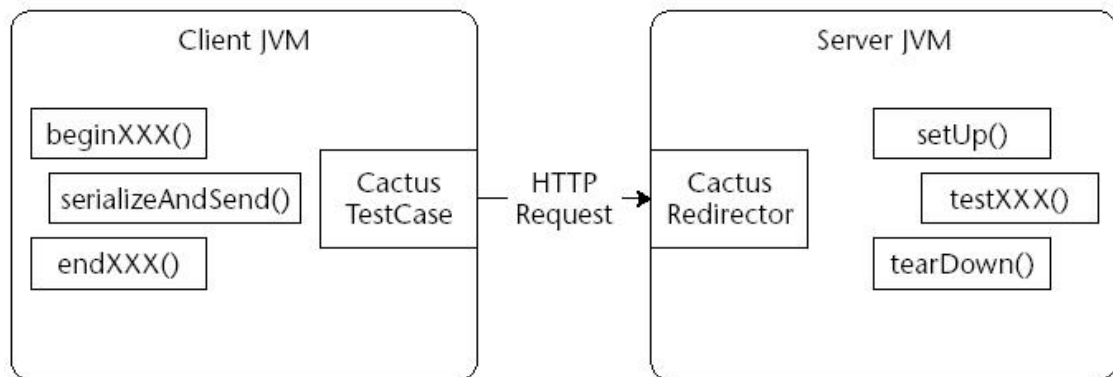
You could write a standard JUnit test case and view your code as client side code for the EJB to test. However, this approach has several problems :

- Since the test client is not running in the same JVM as the EJB, you can't test EJBs with local interfaces (EJB 2.0).
- In most cases, the production code that will call the EJBs is server side code (Servlets, JSPs, Tag libs or Filters). When you access the EJB from a standard JUnit test case, your tests will run in a different execution environment than the production code, which can lead to wrong test results.
- Testing the EJB from outside the EJB container gives extra overhead.

## In-container testing with Cactus

A solution to these problems is running the JUnit tests from inside the application server using Cactus.

Cactus is an open source JUnit extension that makes this possible. Cactus tests start in the client JVM and are sent to the application server's JVM to be executed there. Cactus has a clever way to communicate this information with the server. Just enough information is packaged so that the server side can find and execute the unit test. The package is sent via HTTP to one of the redirectors (ServletTestRedirector, FilterTestRedirector, or JSPTTestRedirector). The redirector then unpacks the info, finds the test class and method, and performs the test.



Cactus provides automated Ant tasks to automatically start your EJB server, run the tests and stop it, thus automating your entire test process and making it easy to implement continuous build and continuous integration of your J2EE project.

More information about this tool can be found on <http://jakarta.apache.org/cactus>.

## Unit testing Java web tier components

Testing Java web tier components adds extra complexity because the servlets or JSP's are executed in a webcontainer which is very hard to stub out using regular mock objects.

### In-container testing of servlets with Cactus

Cactus not only allows you to unit test EJBs (see above) but also Servlets and JSPs.

### Out-of-container Testing of servlets with ServletUnit

Contrary to the Cactus approach, ServletUnit allows you to test your servlets outside the container.

To accomplish this, ServletUnit emulates a servlet container. The framework not only allows you to do black-box tests of the servlet, but also lets you do complete white-box unit tests on your servlets.

More information about ServletUnit can be found on <http://httpunit.sourceforge.net/doc/servletunit-intro.html>

## Testing Struts Actions using StrutsTestCase

When using the Jakarta Struts framework (<http://jakarta.apache.org/struts>), testing the Struts actions poses specific problems.

Testing the Action class comes down to testing the `execute` method. The signature of the `execute` method :

```
public ActionForward execute(ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response);
```

In order to make the call you would have to supply four troublesome parameters that are dependent on the Struts setup and the servlet environment. To simplify this task, `StrutsTestCase` can be used.

`StrutsTestCase` is an open source project from SourceForge, which extends the JUnit framework to allow testing the Struts Action class.

The idea behind the `StrutsTestCase` framework is that it's not you that calls the `execute` method, but `StrutsTestCase`. `StrutsTestCase` supplies the four parameters to the `execute` method, and when you use the mock object approach, these parameters will be simulated. The request and response objects do not

come from a servlet container, they're set up by `StrutsTestCase`. The mapping and form parameters are created by `StrutsTestCase` by reading the Struts configuration file.

Example

```
package be.jcs.jjguidelines.examples.testing.struts;

import servletunit.struts.*;

public class TestStrutsAction extends MockStrutsTestCase {

    public void setUp() throws Exception {
        super.setUp();
    }

    public void tearDown() throws Exception {
        super.tearDown();
    }

    public TestStrutsAction(String testName) {
        super(testName);
    }

    public void testList() {
        setRequestPathInfo("/list");
        actionPerform();
        verifyForward("list");
        verifyNoActionErrors();
    }

    public static void main(String[] args) {
        junit.textui.TestRunner.run(TestStrutsAction.class);
    }
}
```

`MockStrutsTestCase` extends JUnit's `TestCase` class, so all of the "assert" methods from JUnit are available in the test methods.

There are four statements in the `testList` method :

- `setRequestPathInfo("/list")` : tells `StrutsTestCase` to test the "list" action defined in `struts-config`. The `struts-config` contains the information about which class to call.
- `actionPerform()` : tells `StrutsTestCase` to make the call to the execute method
- `verifyForward("list")` : tells `StrutsTestCase` to verify that a forward to the name "list" was attempted ("list" is also defined in `struts-config`).
- `verifyNoActionErrors()` : tells `StrutsTestCase` to verify that the execute method did not create any `ActionError` objects.

More information about `StrutsTestCase` can be found on <http://strutstestcase.sourceforge.net>.

## Unit testing Java Swing/AWT GUI components

When unit testing GUI components, it is difficult to make your tests robust. As soon as the layout of one component changes a little, the layout of many other components may be affected. A good unit test for GUI components should therefore never rely on the absolute positions or exact layout of a component.

JfcUnit is an open source extension of the JUnit framework. It provides following functionality to aid you with the development of unit tests for Swing/AWT components :

- Obtaining handles on Windows or Dialogs opened by the Java code.
- Locating components in a component hierarchy from the containers above.
- Raising events on the components, such as clicking buttons or typing text in a text component.
- Testing the components in a thread safe way.

## Example

An Online Shop application has a panel where the user can purchase a particular item from the shop. The customer needs to be able to enter the amount of items. The panel is made trivial to simplify the example :

```
public class PurchasePanel extends JPanel {
    public PurchasePanel() {
        setLayout(new GridLayout(2, 2, 5, 0));
        add(new JLabel("Amount"));
        add(new JTextField());
        add(new JButton("Purchase"));
        add(new JButton("Cancel"));
    }
}
```

We will write a unit test that verifies whether there is a purchase and a cancel button, whether we are able to enter the amount (of items to purchase) and whether it is possible to press the purchase button.

Start by creating a class inheriting from JFCTestCase and add as members the user interface panel, frame or dialog to be tested.

```
public class PurchasePanelTestCase extends JFCTestCase {
    private JFrame frame;
    private PurchasePanel purchasePanel;
    private TestHelper helper; // see further

    public PurchasePanelTestCase(String s) {
        super(s);
    }
}
```

Now construct the test by creating a JFCTestHelper and the user interface to test and show it (make it visible).

```
protected void setUp() throws Exception {
    super.setUp();
    helper = new JFCTestHelper();
    purchasePanel = new PurchasePanel();
    frame = new JFrame("Purchase test");
    frame.getContentPane().add(purchasePanel);
    frame.pack();
    frame.setVisible(true);
}
```

The cleanup routine is straightforward :

```
protected void tearDown() throws Exception {
    purchasePanel = null;
    helper.cleanup(this);
    super.tearDown();
}
```

The test routine :

```
public void testPresenceButtons() {
    // Look for the 1st button on the screen. Other methods: see further.
    JButton purchaseButton = (JButton) helper.findComponent(
        JButton.class, purchasePanel, 0);
    assertNotNull("Could not find the Purchase button", purchaseButton);

    // An alternative way to look for components is to find by name.
    // We look for the 1st button with name "Cancel"
    JButton cancelButton = (JButton) helper.findComponent(
        new AbstractButtonFinder("Cancel"), purchasePanel, 0);
    assertNotNull("Could not find the Cancel button", cancelButton);

    // We now search for the last component: the text field.
    JTextField amountField = (JTextField) helper.findComponent(
        JTextField.class, purchasePanel, 0);
    assertNotNull("Could not find the amountField", amountField);
    assertEquals("Username field is empty", "", amountField.getText());
}
```

When this test succeeds, this proves that the user interface has all the components it needs to, so the test case is finished.

More complicated components where clicking on one item should disable another component would be more complex to write, but probably more interesting as the change of failure is higher.

Tip: to avoid looking for components through the index or label, give the button a name. In this way the test case is independent on label changes. `JButton purchaseButton = new JButton("Purchase"); purchaseButton.setName("purchase");` This could be enforced through a client framework by inheriting from `JButton` or `AbstractAction`, possibly adding internationalisation (the same name could be used to lookup the translation in the resource bundle), tooltips and other features to such a framework depending on the requirements of the project.

Tip: Avoid testing logic from other classes in the user interface unit test case. This is still a unit test, testing only the user-interface unit. Have unit-tests on the components the user interface interacts with if you want to test their behaviour.

Tip: Sometimes it is necessary to add a sleep as the user interface might perform a time-consuming activity. You can add a call to `awtSleep()`, which will wait for user-interface to respond by waiting on AWT-events in the AWT event queue (Swing is build on top of AWT). Probably, this would not be inside a unit test case (see the previous tip), but rather an integration or scenario test case testing the user interface.

More information about `JfcUnit` can be found on <http://jfcunit.sourceforge.net>

## Scenario Tests (Black Box)

A scenario test verifies whether or not a certain piece of functionality is working correctly by testing a particular interaction of a primary actor with the system.

It is like a movie scenario that is played and checked if that scenario is correctly executed. It is the high-

est end-to-end test and probably tests almost all needed components. A scenario test simulates user input as if a human user would enter the data, only as this is a time-consuming (and boring) task, the scenario tests is used to cover each use case.

Writing automated scenario tests is very time-consuming. Especially because of the inherent brittleness of such tests, a lot of rework is to be expected during the development of a software system.



### Tip

Scenario tests take much longer to run than unit test. To ensure they are run often enough, run the full suite of scenario tests every night.

## Scenario tests for non-GUI Java applications

For regular, non-GUI Java applications (e.g. batch applications), scenario tests can be programmed in Java using the JUnit framework. Each test method does a full scenario test. Test scenarios should be grouped by use case.

This is in fact a misuse of the JUnit unit testing framework because in this case, the unit test treats the system as a black box and only interacts with the public boundaries of the system (playing the role of the primary actor of the scenario).

## Scenario tests for Java web applications

In use case scenarios for web applications, the primary actor is a human being using a browser. When you create an automated scenario test for a web application, it is desirable to get rid of the browser by emulating it.

The HttpUnit open source library offers a way to emulate a web browser and can be used together with JUnit to program automated scenario tests.

The center of HttpUnit is the WebConversation class, which takes the place of a browser talking to a single site. It is responsible for maintaining session context, which it does via cookies returned by the server. To use it, one must create a request and ask the WebConversation for a response.

For example:

```
WebConversation webConversation = new WebConversation();
WebRequest request = new GetMethodWebRequest(
    "http://httpunit.sourceforge.net");
WebResponse response = webConversation.getResponse(request);
```

The response may now be manipulated either as pure text (via the getText() method), as a DOM (via the getDOM() method), or by using the various other methods described below.

Because the above sequence is so common, it can be abbreviated to:

```
WebConversation webConversation = new WebConversation();
WebResponse response = webConversation.getResponse(
    "http://httpunit.sourceforge.net");
```

The simplest and most common form of navigation among web pages is via links. HttpUnit allows users to find links by the text within them, and to use those links as new page requests. For example, this page contains a link to the JavaDoc for the WebResponse class, above. That page could therefore be obtained as follows:

```
WebConversation webConversation = new WebConversation();
// read this page
WebResponse response = webConversation.getResponse(
    "http://httpunit.sourceforge.net");
// find the link
WebLink link = response.getLinkWith("JavaDoc");
// follow it
link.click();
// retrieve the selected page
WebResponse javaDocPage = webConversation.getCurrentPage();
```

The `WebResponse` class offers the possibility to examine the HTTP headers of the response, as well as the body. It is even possible to represent the body as an XML DOM with the `getDOM()` method. This allows you even to use XPath queries to assert the validity of the response page.

Using all these `HttpUnit` elements, it is possible to build fully automated scenario tests in `JUnit`.

`HttpUnit` also supports cookies, tables, forms and frames. More information about `HttpUnit` can be found on <http://httpunit.sourceforge.net>

## Scenario tests for Java Swing/AWT applications

To automate scenario tests for Java Swing or AWT applications, you can use `JfcUnit`. (See the section called “Unit testing Java Swing/AWT GUI components”) `JfcUnit` has XML scripting support. To do scenario testing, you can record the user interactions with a Swing/AWT application in a proprietary XML format. This XML script can then be edited and enriched with assertions about the state of the GUI components. To run your automated regression test, simply play back the XML script from `JfcUnit`.

`MarathonMan` is also an open source tool designed for the purpose of automated scenario testing. It allows you to record, edit and playback `Jython` scripts. `Jython` is a Java variant of the Python scripting language. The `Jython` scripting language allows more flexibility in the design of your tests.

## Integration of JUnit with development tools

### Integration of JUnit with Ant

The default Ant distribution provides two optional `JUnit` tasks : `<junit>` and `<junitreport>`.

The `<junit>` task executes a set of `JUnit` testcases

Example 1 :

```
<junit>
  <test name="my.test.TestCase"/>
</junit>
```

Runs the test defined in `my.test.TestCase` in the same VM. No output will be generated unless the test fails.

Example 2 :

```
<junit printsummary="yes" fork="yes" haltonfailure="yes">
  <formatter type="plain"/>
  <test name="my.test.TestCase"/>
</junit>
```

Runs the test defined in `my.test.TestCase` in a separate VM. At the end of the test, a one-line summary will be printed. A detailed report of the test can be found in `TEST-my.test.TestCase.txt`. The build process will be stopped if the test fails.

Example 3 :

```
<junit printsummary="yes" haltonfailure="yes">
  <classpath>
    <pathelement location="${build.tests}"/>
    <pathelement path="${java.class.path}"/>
  </classpath>

  <formatter type="plain"/>

  <test name="my.test.TestCase" haltonfailure="no" outfile="result">
    <formatter type="xml"/>
  </test>

  <batchtest fork="yes" todir="${reports.tests}">
    <fileset dir="${src.tests}">
      <include name="**/*Test*.java"/>
      <exclude name="**/AllTests.java"/>
    </fileset>
  </batchtest>
</junit>
```

Runs `my.test.TestCase` in the same VM, ignoring the given CLASSPATH; only a warning is printed if this test fails. In addition to the plain text test results, for this test a XML result will be output to `result.xml`. Then, for each matching file in the directory defined for `${src.tests}` a test is run in a separate VM. If a test fails, the build process is aborted. Results are collected in files named `TEST-name.txt` and written to `${reports.tests}`.

The `<junitreport>` task merges individual XML files generated by the `<junit>` task and applies a stylesheet on the resulting merged document. The result is a customized browsable report of the test-cases results.

Example :

```
<junitreport todir="./reports">
  <fileset dir="./reports">
    <include name="TEST-*.xml"/>
  </fileset>
  <report format="frames" styledir="./reports" todir="./report/html"/>
</junitreport>
```

Generates a `TESTS-TestSuites.xml` file in the directory reports and generates a framed report in the directory `report/html` using the stylesheet named "junit-frames.xsl" from the reports directory.

More information about the JUnit Ant tasks can be found on <http://ant.apache.org/manual>

## Integration of JUnit with Java IDEs

Nearly all recent versions of Java integrated development environments have built-in Ant support.

For more information on the JUnit integration with specific IDE, consult the manual of that IDE.

## Stress Tests (Volume Tests)



Volume or stress tests evaluate the performance of the system. This can be done using big record sets, lots of simultaneous users or many requests per second. These tests might be necessary if your project requires the treatment of huge amount of data or a huge amount of users. Typically, these tests are nothing more than performing the acceptance tests with a big data set or running the acceptance tests in parallel, as this is a very convenient way to ensure that the tests represents typical user behaviour.

Stress tests can give you confidence that the non functional performance requirements can be met.



### Tip

Do not use the same computer for running both the tests and the application or web server, as the simulation also takes up a lot of resources.

## Stress testing with JMeter

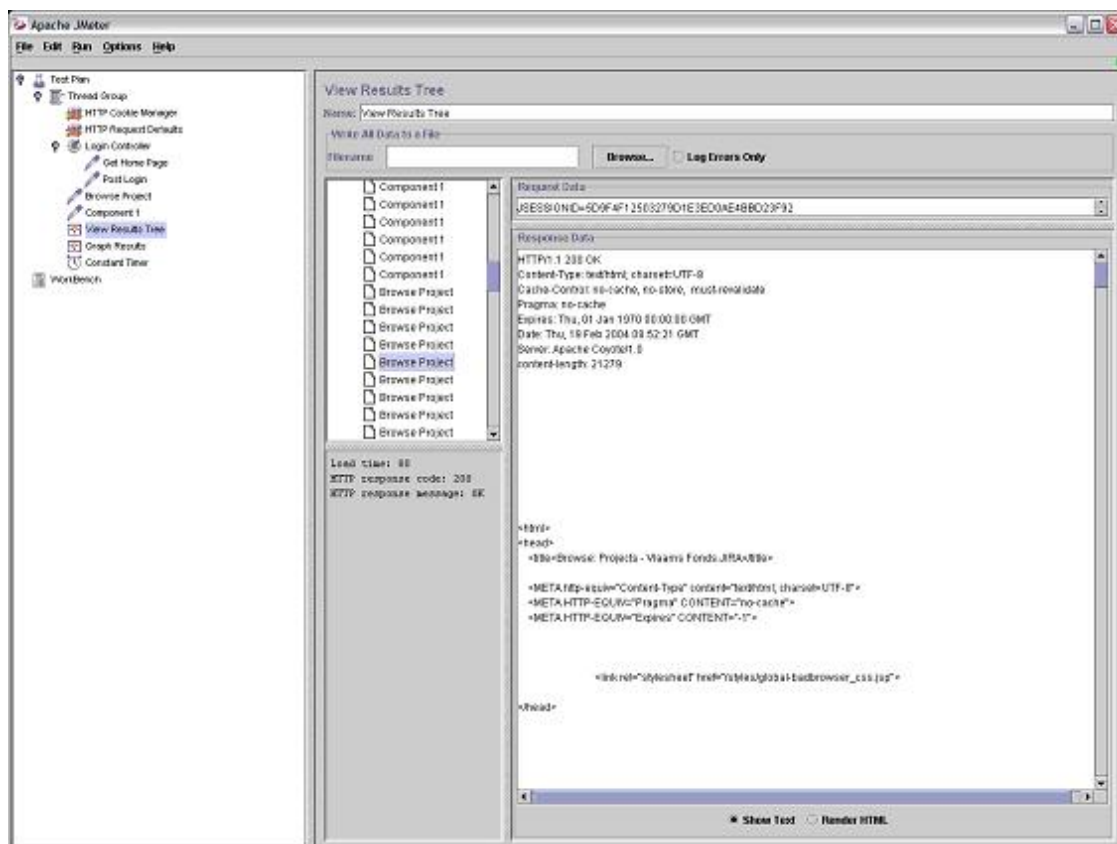
Apache JMeter is an open source Java desktop application designed to load test functional behavior and measure performance. It was originally designed for testing web applications but has since expanded to other test functions like FTP, JDBC and LDAP.

### Example

With JMeter, you can enter (or record) and play performance tests easily. In this example, we will check whether the performance of the issue-tracking software JIRA (<http://www.atlassian.com/software/jira>) is satisfying. Any web client would be similar.

- Add a ThreadGroup on a TestPlan. A ThreadGroup can be viewed as a group of threads, each simulating a user. Start with one user, disable the loop forever and enter as loop count 1 or a low number. If everything works fine, increase the number of simulated users and check the loop forever on. Know that if you increase the number of users, the client might become the bottleneck. In such a case, you can set up JMeter (or rather the JMeterEngine) on multiple machines connecting through RMI, which is easily configurable with JMeter.
- Start with adding a Config Element: "HTTP Requests Defaults" node to the ThreadGroup. This will save you from entering the servername (or IP-address), portnumber and protocol (HTTP) for each request. Fill in the above fields in the node. This allows you to switch from different machines to be tested, for example from a development machine to a test machine, by just changing one node.
- We need a HTTP Cookie Manager as the site uses cookies. Just adding this element to the thread group is sufficient for the cookies to be handled correctly.
- Now we will enter the sequence of actions we want to simulate. First things first, the user needs to login and this is only once required. Hence, we create the logic controller "Only Once Controller". We can rename the node to "Login Controller" or something more easily to remember for the users.
- Within this node, we want to request the home page. So we create a HTTP Request, and give it the name "Get Home Page". Fill in the path, possibly being empty.
- Within the same Login Controller, we now want to post our user and password information. Create a new HTTP Request, with as name "Post Login" for example. Make sure the path is correct, for JIRA 2.5 it is "/secure/Dashboard.jspa". To make sure the path is correct, just copy-paste the URL. Now add in the parameter table of this node the parameter "os\_username" with as value the id of the test user. Secondly, add "os\_password" with the appropriate password as value. To know this, one needs to do a "Show Code" in the browser or go into the JSP. As a last thing to do, set the Method (next to the protocol) to "POST".

- Now we're logged in, we can throw a few HTTP requests, such as browsing the project and selecting a component of the project and viewing it's information. For the "Browse Project", the path is `/secure/BrowseProject.jspa`, and as parameter one needs to pass "id" and a valid id, for example 10000. To know these parameters, one can walk through the test in a browser, and copy the links instead of clicking them (this is an option when you right-click on a link in your browser). There you will see for example:
- Add a listener to see what's happening. A "View Results Tree" is perfect for this job.
- Save this setting and start to run. Go to the "View Results Tree" and check the results (and the HTML generated or check for items in red (indicating the errors)).



- If everything is running as expected, you can now increase the number of users and the number of loops (or set this to forever). But to simulate real users, you need to a Timer. Set it to for example 3000 (= 3 seconds). There are different kinds of timers, for example a Guassian random distribution timer with an average of 3 seconds, but the Constant Timer will do nicely, especially if you want to have predictable results. Finally, you can add "Graph Results" listener where you can see the results in a nice graph (throughput, average, minimum and maximum response times of the site with the particular number of concurrent users connected). In the above picture, we have the total tree of this example.

## JMX

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]jmx](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]jmx)]

## Introduction

With the advent of Java Application Servers, Java has found its way to enterprise level, mission critical software systems. These systems have to be reliable, performing, scalable and they have to be **manageable**.

A Java application of any complexity will most likely have a management and administrative part to it. This might be incorporated into the application or might be separate – the implementation will however be specific to the application.

JMX provides an architecture to standardize the way in which Java applications are managed and monitored.

## JMX Basics

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]jmx.basics](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]jmx.basics)]

### What Is JMX?

JMX is a layer to instrument different application resources. Possible resources are: connections, connection pools, printers but also any component within the application itself.

JMX provides a simple yet powerful and generic framework, which allows Java applications to implement their manageability in a standard and portable way.

### How Is JMX Related To J2EE?

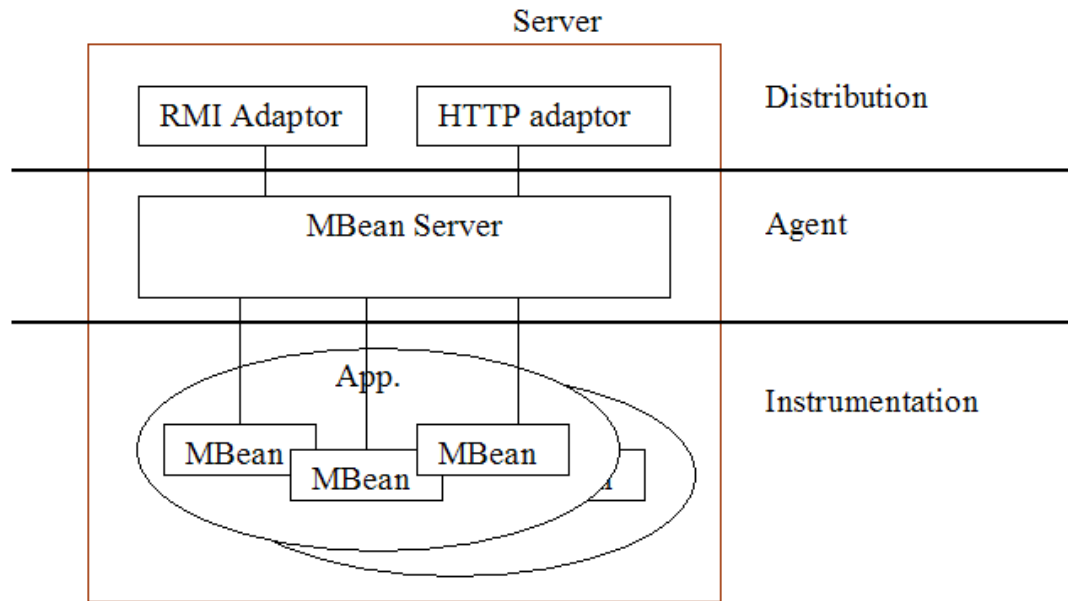
J2EE application servers come in many sizes and flavours. They all agree on what they offer (the J2EE specification), but not on how they accomplish it. Some of them are 100% Java, others contain a fair amount of native code. Because of this, and the lack of an agreement on the subject, they are all managed in different ways.

This is where JMX will play a key role, through JSR 77: the J2EE Management specification. JSR 77 builds upon the general JMX framework and defines its use to manage a specific application: the J2EE application server. The Standard J2EE management functions include: state management, event generation, performance monitoring and statistics.

However, the fact that future J2EE application servers will have to comply with the J2EE management specification not only implies a better manageability of the server itself, but also provides the J2EE application programmer himself with a powerful framework for management of the specifics of the application.

## JMX Architecture

JMX is layered into three levels: instrumentation (application), agent and distributed (connector) services.



## MBeans

At the application level MBeans are the components that provide the application management functionality.

MBeans come in different types. Each type offers more functionality and flexibility, which however comes at the price of higher complexity. The different types are: Standard MBeans, Dynamic MBeans, Model MBeans and Open MBeans.

MBeans are described by their interface and follow the Bean patterns (and closely resemble JavaBeans). In short:

- The interface must have the same name as the implementing class followed by `MBean` (case sensitive!)
- The interface must have `public` visibility
- The implementing class must contain at least one `public` constructor
- Getters and setters for the attributes follow strict naming conventions

## MBean Server

The MBean Server is the repository for all MBeans. For a MBean to be accessible via its management interface it must be registered with the MBean Server.

The MBean Server is available in most (if not all) major application servers and needs not to be developed.

## Why And When To Use JMX

An application deployed in a production environment will always have a management and administration part to it.

The key driving forces are:

- Monitoring availability
- Monitoring performance
- Minimizing down time
- Tracking of user transactions
- The need for pro-active response when monitoring indicates a problem or, for example, that a shortage of resources is eminent.

You can use JMX, instead of developing your own system, to enable above functionality in your business application.

The key benefits using JMX are:

- JMX is the emerging standard for managing Java and J2EE applications
- Enables Java applications to be managed without heavy investment
- Provides a scalable management architecture
- Integrates existing management solutions, including SNMP consoles
- Leverages existing standard java technology
- Can leverage future management concepts
- Makes management data available via a choice of protocols (SNMP, HTTP, SOAP, ...).
- Provides an easy-to-use Java API

## How To Use JMX

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]jmx.howToUse](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]jmx.howToUse)]

Consider an application component `MyCache`. This component caches some entities. We want to view how many entities are in the cache. If needed we want to alter the maximum items allowed in the cache and we want to be able to *reset* the cache, removing all entities.

## Define A Management Interface Enabling The Requirements

For example:

```
public interface MyCacheMBean {  
    // Maximum entities is read/write  
    public void setMaxEntities(long nbr);  
    public long getMaxEntities();  
  
    // the actual nbr of entities in cache is read only
```

```
    public long getEntitiesInCache();

    // empties the cache
    public void reset();

    // ...
}
```

## Provide An Implementing Class

For example:

```
public class MyCache implements MyCacheMBean {

    private long entitiesInCache = 0;
    private long maxEntities = 100; // programatically set default value

    public void setMaxEntities(long nbr) {
        // management code omitted
        // e.g. when new maximum is smaller than current nbr
        maxEntities = nbr;
    }
    public long getMaxEntities() {
        return maxEntities;
    }

    public long getEntitiesInCache() {
        return entitiesInCache;
    }

    public void reset() {
        // actual reset omitted

        setEntitiesInCache (0);
    }

    // actual caching methods omitted
}
```

## Reference The MBean Server

The MBeanServerFactory class allows you to create instances of MBeanServer implementations. In its simplest form this is done as follows:

```
MBeanServer mbs = MBeanServerFactory.createMBeanServer();
```

## Instantiate The MBean

MBeans should not be constructed by the application. See rule JMX\_011: Do Not Construct An MBean. The MBeanServer provides several methods to *instantiate* a MBean. For example:

```
Object myCache = mbs.instantiate("MyCache");
```

## Create An ObjectName

An ObjectName is unique within an MBean server and is used by the management applications and other MBeans to invoke operations and manipulate MBean components registered to the server.

The MBeanServer uses object names to refer to the MBeans it manages. Using an object name rather than a direct object reference to an MBean creates a level of indirection that allows the MBeanServer to help manage the MBean's lifecycle, control access to it, and associate additional information with the MBean.

The ObjectName consists of two parts:

- the domain name
- an unordered list of property-value pairs

The string representation of an ObjectName must follow the syntax *[DomainName]:property=value[,property=value]\**. For example:

```
be.vlaanderen.jmxdemo:name=MyCache
```

The ObjectName object can be constructed in different ways:

```
ObjectName cacheObjectName = new ObjectName(
    "be.vlaanderen.jmxdemo:name=MyCache");

Hashtable sqprops = new Hashtable();
sqprops.put("name", "MyCache");
ObjectName cacheObjectName = new ObjectName(
    "be.vlaanderen.jmxdemo", sqprops);

ObjectName cacheObjectName = new ObjectName(
    "be.vlaanderen.jmxdemo", "name", "MyCache");
```

## Register The MBean

```
mbs.registerMBean(myCache, cacheObjectName);
```

Now the MBean is ready to be used by the application or any other management system that has access to the MBeanServer.



### Note

The *instantiate, then register* approach just described is a natural one for Java programmers used to working with `Collection` classes and the like, but it has the disadvantage of leaving a *live* reference to the MBean in the management application. To avoid this it is recommended to use one of the `createMBean` methods provided by `MbeanServer`. See also the section called “MBeans”.

## JMX Pitfalls

**Inheritance Standard MBeans:** in the case where a class A that implements an interface `AMBean` extends a class B that implements an interface `BMBean` only the management functions in interface `AMBean` are available for the management of class A. In other words the management functions of interface `BMBean` are NOT inherited. When interface `AMBean` however extends interface `BMBean`, both interfaces are joined when managing class A.

**Inheritance Dynamic MBeans:** the management interface for a Dynamic MBean is the first interface en-

countered in the class hierarchy. So if MBean A does not implement a management interface the systems look at its superclass B etc. The nearest interface is used, no aggregate is made with the interfaces further up in the hierarchy. Also, if no management interface is found in the classes hierarchy a `NotCompliantMBeanException` is thrown.

Interface needs to be declared `public`. If you forget to declare the MBean interface `public` the `MBeanServer` will throw a runtime `javax.management.ReflectionException`. The compiler does not capture this mistake, nor will it prevent the MBean to be registered.

Getters and setters need to be declared according to the conventions.

## MBeans

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JGuidelines\]jmx.mbeans](mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]jmx.mbeans)]

### MBean Naming

#### Use A Domain Name Structured Like Java Packages Names To Avoid Collisions

MBeans need to register with the `MBeanServer`. When a MBean with the same name already exists, it will be impossible to register your MBean. Part of the MBean name is the domain name, which serves as a namespace. There are few restrictions in the JMX specification how to name the MBean domain name.

Using a domain name just like java package names will help to guarantee the uniqueness of your namespace.

`be.vlaanderen.jmxdemo`

#### Use A `name` Property In The Property List

The specification stipulates that there be at least one property. A good and standard way of dealing with this is to always have a `name` property, which takes the (unqualified) name of the MBean class as value. This will keep the MBean unique within the same domain.

#### Use An `instanceId` Property To Distinguish Different Instances Of 1 MBean Within 1 Domain

If you need to register multiple instances of the same MBean with the `MBeanServer` you need a further means of distinguishing the MBean. Using an `instanceId` property again might be a good and standard way of accomplishing this.



#### Tip

The `instanceId` will need a value that is unique. One way to accomplish this could be by using a key generator. This key generator can be based on the `System.currentTimeMillis` method.

### MBean Granularity

Management of an application should follow a *functional* route. You should not *as matter of fact* design your MBeans to map directly onto your implementation components. More than likely, multiple classes will cooperate to provide a certain functionality, design the MBean(s) so that the functionality is presented as 1 unit.



**Tip**

If you have similar resources, make the management interface similar.

## Instrumentation

Provide all *needed* info and functionality for adequate management and no more. You want the management system to provide whatever is necessary to monitor and regulate the behaviour of the application components.

When starting out with JMX and discovering how easy it is, the temptation might be to provide just all info and functionality to be found in a managed component. However, management comes at a cost.

An overload on information and functionality comes with an overhead on used resources and will overwhelm the operators of the management system.

Of course deciding what is needed, what is valuable and what is trivial will be specific to each application.

**Tip**

Keep the attributes and parameters simple. When accessing the MBean through a Web based user interface, complex objects might not be available.

## The MBeanRegistration interface

MBeanRegistration is an interface that can be implemented by an MBean. Implement this interface whenever is a need for:

- Processing when registering or de-registering
- A reference to the MBeanServer

## Dynamic MBeans

Dynamic MBeans are much more powerful and flexible than standard MBeans. There is however no such thing as a free lunch, this power and flexibility comes at a price: complexity.

Use Dynamic MBeans when there is a clear added value in doing so; don't climb the mountain just because it's there. Although this is probably sound advice in any case, pay additional attention to it when you decide to use Dynamic MBeans.

Good reasons might be:

- Wrapping a third party resource
- A changing management interface

**Tip**

When using Dynamic MBeans use a Dynamic MBean Facade Pattern. A lot of code in a dynamic MBean implementation is similar to that of another and are related to two areas:

- Creating metadata describing the features of the MBean
- Implementing the DynamicMBean interface

Basically it boils down to implementing an add/remove method for each feature (attribute, operation, notification) and implementing the DynamicMBean interface.

Do not create dynamic MBean feature metadata without a description. Externalise the feature description.

## Creating MBeans

Do not create an MBean using the Java *new* keyword.

Do not use the `newInstance` and `registerMBean` methods, which are provided through the `MBeanServer`. Use the `createMBean` method. This will perform the instantiating and registering without exposing the object reference to the MBean instance.

## Referencing MBeans

Do not manipulate MBeans through a direct object reference. See also the section called “Creating MBeans”. This will bypass JMX (1.2) security checks and should be avoided.

# JAAS

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]jaas](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]jaas)]

## Introduction

Today no single server-side application can be deployed rationally lacking security. Security should not be restricted to public networks like the internet, even on private networks business applications need to distinguish between well-defined user profiles. Security implies both authentication and authorization, often combined with encryption and digital signing.

On a Java platform it makes no sense anymore to develop a custom security-framework: since J2SE v1.4 a configurable security-layer is part of the core identified as JAAS, the Java Authentication and Authorization Service. This layer is in fact an upgrade of codesource to principal-based security. It is clear that Java security should rely at all times on this extremely pluggable framework both for authentication and authorization.

## Why Use JAAS

JAAS adds security to applications in a standard, portable fashion. Security motives should be obvious.



### Tip

JAAS is available as an optional package for J2SEv1.3.X

## JAAS Basics

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]jaas.basics](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]jaas.basics)]

## Authentication

In order to authenticate a user, you first need a `javax.security.auth.login.LoginContext`. Here is the basic way to instantiate a `LoginContext`:

```
import javax.security.auth.login.*;
// ...
LoginContext lc = new LoginContext(/* config file entry name */,
    /* CallbackHandler to be used for user interaction */);
```

The arguments are:

- The name of an entry in the JAAS login configuration file
- A `CallbackHandler` instance

## The Name Of An Entry In The JAAS Login Configuration File

This is the name for the `LoginContext` to use to look up an entry for this application in the JAAS login configuration file. Such an entry specifies the class(es) that implement the desired underlying authentication technology(ies). The class(es) must implement the `LoginModule` interface, which is in the `javax.security.auth.spi` package.

## A CallbackHandler Instance

When a `LoginModule` needs to communicate with the user, for example to ask for a user name and password, it does not do so directly. That is because there are various ways of communicating with a user, and it is desirable for `LoginModules` to remain independent of the different types of user interaction. Rather, the `LoginModule` invokes a `javax.security.auth.callback.CallbackHandler` to perform the user interaction and obtain the requested information, such as the user name and password.

An instance of the particular `CallbackHandler` to be used is specified as the second argument to the `LoginContext` constructor. The `LoginContext` forwards that instance to the underlying `LoginModule` (in our case `SampleLoginModule`). An application typically provides its own `CallbackHandler` implementation. There are two simple `CallbackHandlers` provided in the `com.sun.security.auth.callback` package as sample implementations.

- `TextCallbackHandler`
- `DialogCallbackHandler`

Once we have a `LoginContext lc`, we can call its `login` method to carry out the authentication process:

```
lc.login();
```

The `LoginContext` instantiates a new empty `javax.security.auth.Subject` object (which represents the user or service being authenticated). The `LoginContext` constructs the configured `LoginModule` and initializes it with this new `Subject` and `MyCallbackHandler`.

If authentication is successful, the `LoginModule(s)` populate(s) the `Subject` with a `Principal` representing the user. The actual `Principal` is a class implementing the `java.security.Principal` interface.

The calling application can subsequently retrieve the authenticated `Subject` by calling the `LoginContext`'s `getSubject` method.

## CallbackHandler

In some cases a LoginModule must communicate with the user to obtain authentication information. LoginModules use a `javax.security.auth.callback.CallbackHandler` for this purpose. An application can either use one of the sample implementations provided in the `com.sun.security.auth.callback` package or, more typically, write a `CallbackHandler` implementation. The application passes the `CallbackHandler` as an argument to the `LoginContext` instantiation. The `LoginContext` forwards the `CallbackHandler` directly to the underlying LoginModules.

`CallbackHandler` is an interface with one method to implement:

```
void handle(Callback[] callbacks)
    throws IOException, UnsupportedCallbackException;
```

The `LoginModule` passes the `CallbackHandler` handle method an array of appropriate `javax.security.auth.callback.Callbacks`, for example a `NameCallback` for the user name and a `PasswordCallback` for the password, and the `CallbackHandler` performs the requested user interaction and sets appropriate values in the `Callbacks`. The `handle` method is structured as follows:

```
public void handle(Callback[] callbacks)
    throws IOException, UnsupportedCallbackException {
    for (int i = 0; i < callbacks.length; i++) {
        if (callbacks[i] instanceof TextOutputCallback) {
            // display a message according to a specified type
            // ...
        } else if (callbacks[i] instanceof NameCallback) {
            // prompt the user for a username
            // ...
        } else if (callbacks[i] instanceof PasswordCallback) {
            // prompt the user for a password
            // ...
        } else {
            throw new UnsupportedCallbackException(callbacks[i],
                "Unrecognized Callback");
        }
    }
}
```

A `CallbackHandler` `handle` method is passed an array of `Callback` instances, each of a particular type (`NameCallback`, `PasswordCallback`, etc.). It must handle each `Callback`, performing user interaction in a way that is appropriate for the executing application.

This `CallbackHandler` handles three types of `Callbacks`:

- `NameCallback` to prompt the user for a user name
- `PasswordCallback` to prompt for a password
- `TextOutputCallback` to report any error, warning, or other messages the `SampleLoginModule` wishes to send to the user

The `handle` method handles a `TextOutputCallback` by extracting the message to be reported and then printing it to `System.out`, optionally preceded by additional wording that depends on the message type. The message to be reported is determined by calling the `TextOutputCallback`'s

getMessage method and the type by calling its getMessageType method. Here is the code for handling a TextOutputCallback:

```
if (callbacks[i] instanceof TextOutputCallback) {
    // display the message according to the specified type
    TextOutputCallback toc = (TextOutputCallback) callbacks[i];
    switch (toc.getMessageType()) {
        case TextOutputCallback.INFORMATION:
            System.out.println(toc.getMessage());
            break;
        case TextOutputCallback.ERROR:
            System.out.println("ERROR: " + toc.getMessage());
            break;
        case TextOutputCallback.WARNING:
            System.out.println("WARNING: " + toc.getMessage());
            break;
        default:
            throw new IOException("Unsupported message type: "
                + toc.getMessageType());
    }
}
```

The handle method handles a NameCallback by prompting the user for a user name. It then sets the name for use by the LoginModule by calling the NameCallback's setName method, passing it the name typed by the user:

```
if (callbacks[i] instanceof NameCallback) {
    // prompt the user for a username
    NameCallback nc = (NameCallback)callbacks[i];

    System.err.print(nc.getPrompt());
    System.err.flush();
    nc.setName((new BufferedReader(
        new InputStreamReader(System.in))).readLine());
}
```

Similarly, the handle method handles a PasswordCallback by printing a prompt to prompt the user for a password. It then sets the password for use by the LoginModule by calling the PasswordCallback's setPassword method, passing it the password typed by the user, which it reads by calling the readPassword method (shown below):

```
if (callbacks[i] instanceof PasswordCallback) {
    // prompt the user for sensitive information
    PasswordCallback pc = (PasswordCallback)callbacks[i];

    System.err.print(pc.getPrompt());
    System.err.flush();
    pc.setPassword(readPassword(System.in));
}
```

The readPassword method is a basic method for reading a password from an InputStream:

## Login Configuration

JAAS authentication is performed in a pluggable fashion, so applications can remain independent from underlying authentication technologies. A system administrator determines the authentication technologies, or LoginModules, to be used for each application and configures them in a login Configuration. The source of the configuration information (for example, a file or a database) is up to the current `javax.security.auth.login.Configuration` implementation. The default Configuration

implementation from Sun Microsystems reads configuration information from configuration files.

A login configuration file (`jaas.config`), could look like:

```
JJGuidelines {  
    be.jcs.jjguidelines.JjglsLoginModule required debug=true;  
};
```

The LoginModule defines a *debug* option that can be set to `true` as shown. If this option is set to `true`, the LoginModule outputs extra information about the progress of authentication. A LoginModule can define as many options as it wants. The LoginModule documentation should specify the possible option names and values you can set in your configuration file.

The following System property must refer to the login configuration file:

```
java.security.auth.login.config
```

## Authorization

JAAS authorization extends the existing Java security architecture that uses a security policy to specify what access rights are granted to executing code. That architecture, introduced in the Java 2 platform, is code-centric. That is, the permissions are granted based on code characteristics: where the code is coming from and whether it is digitally signed and if so by whom. We saw an example of this in the `jaasacn.policy` file used in the JAAS Authentication tutorial. That file contains the following:

```
grant codebase "file:./jjgls.jar" {  
    permission javax.security.auth.AuthPermission  
        "createLoginContext.JJGuidelines";  
};
```

This grants the code in the `jjgls.jar` file, located in the current directory, the specified permission. (No signer is specified, so it doesn't matter whether the code is signed or not.)

JAAS authorization augments the existing code-centric access controls with new user-centric access controls. Permissions can be granted based not just on what code is running but also on who is running it.

When an application uses JAAS authentication to authenticate the user (or another entity such as a service), a Subject is created as a result. The purpose of the Subject is to represent the authenticated user. A Subject is comprised of a set of Principals, where each Principal represents an identity for that user. For example, a Subject could have a name Principal (`"Bart Strubbe"`) and a Social Security Number Principal (`"987-65-4321"`), thereby distinguishing this Subject from other Subjects.

Permissions can be granted in the policy to specific Principals. After the user has been authenticated, the application can associate the Subject with the current access control context. For each subsequent security-checked operation, (a local file access, for example), the Java runtime will automatically determine whether the policy grants the required permission only to a specific Principal and if so, the operation will be allowed only if the Subject associated with the access control context contains the designated Principal.

## How Is JAAS Authorization Performed?

To make JAAS authorization take place, the following is required:

- The user must be authenticated.
- Principal-based entries must be configured in the security policy.

- The Subject that is the result of authentication must be associated with the current access control context.

## How Do You Make Principal-Based Policy File Statements?

Policy file grant statements can now optionally include one or more Principal fields. Inclusion of a Principal field indicates that the user or other entity represented by the specified Principal, executing the specified code, has the designated permissions.

Thus, the basic format of a grant statement is now:

```
grant <signer(s) field>, <codeBase URL>
    <Principal field(s)> {
        permission perm_class_name "target_name", "action";
        ...
        permission perm_class_name "target_name", "action";
    };
```

Each of the signer, codeBase and Principal fields is optional and the order between the fields doesn't matter.

A Principal field looks like the following:

```
Principal Principal_class "principal_name"
```

That is, it is the word *Principal* (where case doesn't matter) followed by the (fully qualified) name of a Principal class and a principal name.

A Principal class is a class that implements the `java.security.Principal` interface. All Principal objects have an associated name that can be obtained by calling their `getName` method. The format used for the name is dependent on each Principal implementation.

It is possible to include more than one Principal field in a `grant` statement. If multiple Principal fields are specified, then the permissions in that `grant` statement are granted only if the Subject associated with the current access control context contains all of those Principals.

To grant the same set of permissions to different Principals, create multiple grant statements where each lists the permissions and contains a single Principal field designating one of the Principals.

## How Do You Associate a Subject with an Access Control Context?

To create and associate a Subject with an access control context, you need the following:

- The user must first be authenticated
- The static `doAs` method from the Subject class must be called, passing it an authenticated Subject and a `java.security.PrivilegedAction` or `java.security.PrivilegedExceptionAction`. (See API for Privileged Blocks for a comparison of `PrivilegedAction` and `PrivilegedExceptionAction`.) The `doAs` method associates the provided Subject with the current access control context and then invokes the `run` method from the action. The `run` method implementation contains all the code to be executed as the specified Subject. The action thus executes as the specified Subject.

The static `doAsPrivileged` method from the Subject class may be called instead of the `doAs`

method. In addition to the parameters passed to `doAs`, `doAsPrivileged` requires a third parameter: an `AccessControlContext`. Unlike `doAs`, which associates the provided Subject with the current access control context, `doAsPrivileged` associates the Subject with the provided access control context.

## Complete JAAS Examples

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]jaas.examples](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]jaas.examples)]

Notice that logout raises potentially a `LoginException`.

### Authentication

JJGuidelinesAuthentication.java

```
/**
 * <p> This Sample application attempts to authenticate a user
 * and reports whether or not the authentication was successful.
 */
public class JJGuidelinesAuthentication {

    private static final Logger LOGGER = Logger.getLogger(JJGuidelinesAuthentication.class);

    public static void main(String[] args) {
        LoginContext lc = null;
        try {
            lc = new LoginContext(
                "JJGuidelines",
                new JJGuidelinesCallbackHandler());
        } catch (LoginException e) {
            LOGGER.log(Level.SEVERE, e.getMessage(), e);
        } catch (SecurityException e) {
            LOGGER.log(Level.SEVERE, e.getMessage(), e);
        }

        if(lc != null) {

            // the user has 3 attempts to authenticate successfully
            int i;
            boolean successful = false;
            for (i = 0; (!successful) && (i < 3); i++) {
                try {
                    lc.login();
                    successful = true;
                } catch (LoginException e) {
                    LOGGER.log(Level.SEVERE, e.getMessage(), e);
                }
            }
            if (i == 3) {
                LOGGER.log(Level.WARNING, "Failed 3 times");
            } else {
                LOGGER.log(Level.INFO, "Authentication succeeded!");
            }
        }
    }
}
```

JJGuidelinesCallbackHandler.java

```
/**
 * Text-based Callbackhandler implementation.
 */
```



```
*/
class JJGuidelinesCallbackHandler implements CallbackHandler {
    public void handle(Callback[] callbacks)
        throws IOException, UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof TextOutputCallback) {
                TextOutputCallback toc = (TextOutputCallback)callbacks[i];
                switch (toc.getMessageType()) {
                    case TextOutputCallback.INFORMATION:
                        System.out.println(toc.getMessage());
                        break;
                    case TextOutputCallback.ERROR:
                        System.out.println("<ERROR>" + toc.getMessage());
                        break;
                    case TextOutputCallback.WARNING:
                        System.out.println("<WARNING>" + toc.getMessage());
                        break;
                    default:
                        throw new IOException(
                            "Unsupported message type: "
                            + toc.getMessageType());
                }
            } else if (callbacks[i] instanceof NameCallback) {
                NameCallback nc = (NameCallback)callbacks[i];
                System.out.print(nc.getPrompt());
                System.out.flush();

                BufferedReader br = null;
                try {
                    Reader r = new InputStreamReader(System.in);
                    br = new BufferedReader(r);
                    nc.setName(br.readLine());
                } catch (IOException ioe) {
                    logger.log(Level.SEVERE, ioe.getMessage(), ioe);
                } finally {
                    if (br != null) {
                        IOUtil.close(br);
                    }
                }
            } else if (callbacks[i] instanceof PasswordCallback) {
                PasswordCallback pc = (PasswordCallback)callbacks[i];
                System.err.print(pc.getPrompt());
                System.err.flush();
                pc.setPassword(readPassword(System.in));
            } else {
                throw new UnsupportedCallbackException(
                    callbacks[i], "Unrecognized Callback");
            }
        }

        private char[] readPassword(InputStream in) {
            // ...
        }
    }
}
```

JJGuidelinesLoginModule.java

```
package be.jcs.jjguidelines;

import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;
```

```
import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.FailedLoginException;
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;
import sample.principal.SamplePrincipal;

/**
 * <p> This sample LoginModule authenticates users with a password.
 *
 * <p> This LoginModule only recognizes one user: testUser
 * <p> testUser's password is: testPassword
 *
 * <p> If testUser successfully authenticates itself,
 * a <code>SamplePrincipal</code> with the testUser's user name
 * is added to the Subject.
 */
public class JjglsLoginModule implements LoginModule {

    // logger
    private static final Logger LOGGER = Logger.getLogger(JjglsLoginModule.class.g

    // initial state
    private Subject subject;
    private CallbackHandler callbackHandler;
    private Map sharedState;
    private Map options;

    // the authentication status
    private boolean succeeded = false;
    private boolean commitSucceeded = false;

    // username and password
    private String username;
    private char[] password;

    // testUser's SamplePrincipal
    private SamplePrincipal userPrincipal;

    /**
     * <p> Initialize this <code>LoginModule</code>.
     *
     * <p>
     * @param subject the <code>Subject</code> to be authenticated. <p>
     * @param callbackHandler a <code>CallbackHandler</code> for communicating
     * with the end user (prompting for user names and
     * passwords, for example). <p>
     * @param sharedState shared <code>LoginModule</code> state. <p>
     * @param options options specified in the login
     * <code>Configuration</code> for this particular
     * <code>LoginModule</code>.
     */
    public void initialize(Subject subject, CallbackHandler callbackHandler,
        Map sharedState, Map options) {
```

---

```
        this.subject = subject;
        this.callbackHandler = callbackHandler;
        this.sharedState = sharedState;
        this.options = options;
    }

    /**
     * <p>Authenticate the user by prompting for a user name and password.
     *
     * <p>
     *
     * @return true in all cases since this <code>LoginModule</code>
     *         should not be ignored.
     *
     * @exception FailedLoginException if the authentication fails. <p>
     *
     * @exception LoginException if this <code>LoginModule</code>
     *         is unable to perform the authentication.
     */
    public boolean login() throws LoginException {

        // prompt for a user name and password
        if (callbackHandler == null) {
            throw new LoginException("Error: no CallbackHandler available "
                                     + "to garner authentication information from the user");
        }

        Callback[] callbacks = new Callback[2];
        callbacks[0] = new NameCallback("user name: ");
        callbacks[1] = new PasswordCallback("password: ", false);

        try {
            callbackHandler.handle(callbacks);
            username = ((NameCallback)callbacks[0]).getName();
            char[] tmpPassword = ((PasswordCallback)callbacks[1]).getPassword();
            if (tmpPassword == null) {
                // treat a null password as an empty password
                tmpPassword = new char[0];
            }
            password = new char[tmpPassword.length];
            System.arraycopy(tmpPassword, 0,
                            password, 0, tmpPassword.length);
            ((PasswordCallback)callbacks[1]).clearPassword();
        } catch (java.io.IOException e) {
            throw new LoginException(e.toString());
        } catch (UnsupportedCallbackException e) {
            throw new LoginException("Error: " + e.getCallback().toString()
                                     + " not available to garner authentication information "
                                     + "from the user");
        }

        // print debugging information
        LOGGER.log(Level.FINE, "user entered user name: " + username);

        // verify the username/password
        boolean usernameCorrect = false;
        boolean passwordCorrect = false;
        if (username.equals("testUser")) {
            usernameCorrect = true;
        }
        if (usernameCorrect
            && password.length == 12
            && password[0] == 't'
            && password[1] == 'e'
```

---

```
        && password[2] == 's'
        && password[3] == 't'
        && password[4] == 'P'
        && password[5] == 'a'
        && password[6] == 's'
        && password[7] == 's'
        && password[8] == 'w'
        && password[9] == 'o'
        && password[10] == 'r'
        && password[11] == 'd') {

    // authentication succeeded!!!
    passwordCorrect = true;
    LOGGER.log(Level.FINE, "authentication succeeded");

} else {

    // authentication failed -- clean out state
    LOGGER.log(Level.FINE, "authentication failed");

    succeeded = false;
    username = null;
    for (int i = 0; i < password.length; i++) {
        password[i] = ' ';
    }
    password = null;
    if (!usernameCorrect) {
        throw new FailedLoginException("User Name Incorrect");
    } else {
        throw new FailedLoginException("Password Incorrect");
    }
}
return succeeded;
}

/**
 * <p> This method is called if the LoginContext's
 * overall authentication succeeded
 * (the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL LoginModules
 * succeeded).
 *
 * <p> If this LoginModule's own authentication attempt
 * succeeded (checked by retrieving the private state saved by the
 * <code>login</code> method), then this method associates a
 * <code>SamplePrincipal</code>
 * with the <code>Subject</code> located in the
 * <code>LoginModule</code>. If this LoginModule's own
 * authentication attempted failed, then this method removes
 * any state that was originally saved.
 *
 * <p>
 *
 * @exception LoginException if the commit fails.
 *
 * @return true if this LoginModule's own login and commit
 *         attempts succeeded, or false otherwise.
 */
public boolean commit() throws LoginException {
    if (succeeded == false) {
        return false;
    } else {
        // add a Principal (authenticated identity)
        // to the Subject
    }
}
```

```
        // assume the user we authenticated is the SamplePrincipal
        userPrincipal = new SamplePrincipal(username);
        if (!subject.getPrincipals().contains(userPrincipal)) {
            subject.getPrincipals().add(userPrincipal);
        }

        LOGGER.log(Level.FINE, "added SamplePrincipal to Subject");

        // in any case, clean out state
        username = null;
        for (int i = 0; i < password.length; i++) {
            password[i] = ' ';
        }
        password = null;

        commitSucceeded = true;
        return true;
    }
}

/**
 * <p> This method is called if the LoginContext's
 * overall authentication failed.
 * (the relevant REQUIRED, REQUISITE, SUFFICIENT and OPTIONAL LoginModules
 * did not succeed).
 *
 * <p> If this LoginModule's own authentication attempt
 * succeeded (checked by retrieving the private state saved by the
 * <code>login</code> and <code>commit</code> methods),
 * then this method cleans up any state that was originally saved.
 *
 * <p>
 *
 * @exception LoginException if the abort fails.
 *
 * @return false if this LoginModule's own login and/or commit attempts
 *         failed, and true otherwise.
 */
public boolean abort() throws LoginException {
    if (succeeded == false) {
        return false;
    } else if (succeeded == true && commitSucceeded == false) {
        // login succeeded but overall authentication failed
        succeeded = false;
        username = null;
        if (password != null) {
            for (int i = 0; i < password.length; i++) {
                password[i] = ' ';
            }
            password = null;
        }
        userPrincipal = null;
    } else {
        // overall authentication succeeded and commit succeeded,
        // but someone else's commit failed
        logout();
    }
    return true;
}

/**
 * <p> Logout the user.
 *
 * <p> This method removes the <code>SamplePrincipal</code>
```

```
* that was added by the <code>commit</code> method.
*
* <p>
*
* @exception LoginException if the logout fails.
*
* @return true in all cases since this <code>LoginModule</code>
*         should not be ignored.
*/
public boolean logout() throws LoginException {

    subject.getPrincipals().remove(userPrincipal);
    succeeded = false;
    succeeded = commitSucceeded;
    username = null;
    if (password != null) {
        for (int i = 0; i < password.length; i++) {
            password[i] = ' ';
        }
        password = null;
    }
    userPrincipal = null;
    return true;
}
}
```

## Authorization

For example authorization in a class:

```
public class Foo {
    public static void main(final String[] args) {
        LoginContext foo = new LoginContext("JJGuidelines");
        foo.login();
        Subject subject = foo.getSubject();
        subject.doAsPriviledged(
            subject,
            new MyPriviledgedAction(),
            null);
    }
}

public class MyPriviledgedAction
    implements PriviledgedExceptionAction {
    public Object run() throws Exception {
        // ...
    }
}
```

---

# Appendix A. Conventions Rules

Feedback

[mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]conventionsRules]

This appendix bundles all convention rules. Each rule has severity level as defined in the section called “Convention Rules”.

## Java Naming Conventions Rules

Feedback

[mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]namingJavaRules]

### Overview

**Table A.1. Java Naming Conventions Overview**

Rules
JAN_001: Match A Class Name With Its File Name (High)
JAN_002: Group Operations With The Same Name Together (Low)
JAN_003: Use A Correct Name For A Class Or Interface (Enforced)
JAN_004: Use A Correct Name For A Non Final Field (Enforced)
JAN_005: Use A Correct Name For A Constant (Enforced)
JAN_006: Use A Correct Name For A Method (Enforced)
JAN_007: Use A Correct Name For A Package (Enforced)
JAN_008: Name An Exception Class Ending With Exception (High)
JAN_009: Use A Conventional Variable Name When Possible (Normal)
JAN_010: Do Not Use \$ In A Name (Enforced)
JAN_011: Do Not Use Names That Only Differ In Case (High)
JAN_012: Make A Constant private Field static final (Normal)
JAN_013: Do Not Declare Multiple Variables In One Statement (High)
JAN_014: Use A Correct Order For Class Or Interface Member Declarations (Normal)
JAN_015: Use A Correct Order For Modifiers (enforced)
JAN_016: Put The main Method Last (High)
JAN_017: Put The public Class First (High)

### JAN\_001: Match A Class Name With Its File Name (High)

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JAN\_001]

Checks whether the `public` class and interface has the same name as the file in which it resides.

WRONG

```
/**
 * Copyright Notice
 *
 * Filename: FooAutitor.java
```

```
*/  
public class Foo {  
}
```

RIGHT

```
/**  
 * Copyright Notice  
 *  
 * Filename: FooAutitor.java  
 */  
public class FooAutitor {  
}
```

## JAN\_002: Group Operations With The Same Name Together (Low)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAN\\_002](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAN_002)]

Enforces standard to improve readability.

WRONG

```
package be.vlaanderen.myproject.subpackage;  
  
public class Foo {  
    void operation() {  
    }  
  
    void function() {  
    }  
  
    void operation(int param) {  
    }  
}
```



### Tip

Group operations that differ only by their parameter list together. Good to order from least number of parameters to most.

RIGHT

```
package be.vlaanderen.myproject.subpackage;  
  
public class Foo {  
    void operation() {  
    }  
  
    void operation(int param) {  
    }  
  
    void function() {  
    }  
}
```



## JAN\_003: Use A Correct Name For A Class Or Interface (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAN\\_003](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAN_003)]

Class and interface names must contain only letters and start with an upper-case letter (See JLANGSPEC - section 6.8.2).

WRONG

```
class _Foo {  
}
```

RIGHT

```
class Foo {  
}
```

## JAN\_004: Use A Correct Name For A Non Final Field (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAN\\_004](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAN_004)]

Non-final field names that are not constants should contain only letters, starting with a lower-case letter (See JLANGSPEC , Section 6.8.4).

WRONG

```
int BAR = 100;
```

RIGHT

```
int bar = 100;
```

## JAN\_005: Use A Correct Name For A Constant (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAN\\_005](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAN_005)]

Names of constants (final fields that are assigned a constant value) should contain only upper-case letters and underscores (See JLANGSPEC - section 6.8.5). Makes Java code easier to read and maintain by enforcing the standard naming conventions.

WRONG

```
static final int bar = 100;
```

RIGHT

```
static final int BAR = 100;
```

## JAN\_006: Use A Correct Name For A Method (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAN\\_006](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAN_006)]

Method names should contain only letters, starting with a lower-case letter (See JLANGSPEC - section 6.8.3). Makes Java code easier to read and maintain by enforcing the standard naming conventions.

WRONG

```
int Foo(int x) {  
    // ...  
}
```

RIGHT

```
int foo(int x) {  
    // ...  
}
```

## JAN\_007: Use A Correct Name For A Package (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAN\\_007](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAN_007)]

Package names should contain only lower-case letters. Makes it easier to view a directory listing and be able to differentiate packages from classes. The package name can also not start with *java* or *sun*. Package names beginning with *javax* should also be avoided.

WRONG

```
package be.vlaanderen._myproject.subpackage;
```

RIGHT

```
package be.vlaanderen.myproject.subpackage;
```

## JAN\_008: Name An Exception Class Ending With **Exception** (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAN\\_008](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAN_008)]

Names of classes that directly or indirectly inherit from `java.lang.Exception` or `java.lang.RuntimeException` should end with `Exception`.

WRONG

```
class AuditEx extends Exception {  
}
```

RIGHT

```
class AuditException extends Exception {  
}
```

## JAN\_009: Use A Conventional Variable Name When Possible (Normal)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAN\\_009](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAN_009)]

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. Conventional variable names are:

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

To avoid potential conflicts, change the names of local variables or parameters that consist of only two or three uppercase letters and coincide with initial country codes and domain names, which could be used as first components of unique package names.

WRONG

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

RIGHT

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

## JAN\_010: Do Not Use \$ In A Name (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAN\\_010](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAN_010)]

Do not use the character \$ as part of any identifiers (even though the Java Language Specification allows this).

WRONG

```
String buffer$1;
```

## JAN\_011: Do Not Use Names That Only Differ In Case (High)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JAN\\_011](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JAN_011)]

Names that only differ in case makes it difficult for the reader to distinguish between e.g. `anSQL-Database` and `anSqlDatabase`.

## JAN\_012: Make A Constant `private` Field `static final` (Normal)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JAN\\_012](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JAN_012)]

`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

```
public class Foo {
    private int attr1 = 10;
    private int attr2 = 20;

    void bar() {
        attr1 = attr2;
        logger.info(attr1);
    }
}
```

RIGHT

```
public class Foo {
    private static final int ATTR2 = 20;
    private int attr1 = 10;

    void bar() {
        attr1 = ATTR2;
        logger.info(attr1);
    }
}
```

## JAN\_013: Do Not Declare Multiple Variables In One Statement (High)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JAN\\_013](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JAN_013)]

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

```
public class Foo {
    private int attr1;
    private int attr2, attr3;

    void bar() {
        int var1;
```

```
        int var2, var3;
    }
}
```

#### RIGHT

```
public class Foo {
    private int attr1;
    private int attr2;
    private int attr3;

    void bar() {
        int var1;
        int var2;
        int var3;
    }
}
```

## JAN\_014: Use A Correct Order For Class Or Interface Member Declarations (Normal)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAN\\_014](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAN_014)]

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

#### WRONG

```
public class Foo {
    public void bar() {
    }
    public Foo() {
    }
    public static void main(String[] args) {
    }
    private int bar;
    private static Logger logger =
        Logger.getLogger(Foo.class.getName());
}
```

#### RIGHT

```
public class Foo {

    private static Logger logger =
        Logger.getLogger(Foo.class.getName());

}
```

```
private int bar;

public Foo() {
}

public void bar() {
}

public static void main(String[] args) {
}
}
```

## JAN\_015: Use A Correct Order For Modifiers (enforced)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JGuidelines\]JAN\\_015](mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JAN_015)]

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.

WRONG

```
final public class Foo {
    public static final int attr1;
    static public int attr2;
}
```

RIGHT

```
public final class Foo {
    public static final int attr1;
    public static int attr2;
}
```

## JAN\_016: Put The main Method Last (High)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JGuidelines\]JAN\\_016](mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JAN_016)]

Tries to make the program comply with various coding standards regarding the form of class definitions.

WRONG

```
public class Foo {
    void bar1() {}

    public static void main(String args[]) {}

    void bar2() {}
}
```

RIGHT

```
public class Foo {  
    void bar1() {}  
    void bar2() {}  
  
    public static void main(String args[]) {}  
}
```

## JAN\_017: Put The `public` Class First (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAN\\_017](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAN_017)]

According to Sun coding conventions, the `public` class or interface should be the first class or interface in the file.

WRONG

```
class Helper {  
    // some code  
}  
  
public class Foo {  
    // some code  
}
```

RIGHT

```
public class Foo {  
    // some code  
}  
  
class Helper {  
    // some code  
}
```

## J2EE Naming Conventions Rules

Feedback  
[[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]namingJ2eeRules](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]namingJ2eeRules)]

### Overview

**Table A.2. J2EE Naming Conventions Overview**

Rules
JEN_001: Name An EJB Bean Class Like [Name]EJB or [Name]Bean (High)
JEN_002: Name An EJB Remote Home Interface Like [Name]Home (High)
JEN_003: Name An EJB Remote Interface Like [Name] (High)
JEN_004: Name An EJB Local Home Interface Like [Name]LocalHome (High)
JEN_005: Name A Transfer Object Like [Name]TO (High)
JEN_006: Name An EJB In The Deployment Descriptor Like [Name]EJB (High)
JEN_007: Name An EJB Display Name In The Deployment Descriptor Like [Name]JAR (High)
JEN_008: Name A Servlet Like [Name]Servlet (High)

Rules
JEN_009: Name A Primary Key Class Like [Name]PK (High)
JEN_010: Name A Filter Servlet Like [Name]Filter (High)
JEN_011: Name A Local Interface Like [Name]Local (High)
JEN_012: Name A Data Access Object Like [Name]DAO (High)
JEN_013: Use A Correct Name For An Enterprise Application Display Name (High)
JEN_014: Use A Correct Name For A Web Module Display Name (High)
JEN_015: Use A Correct Name For An EJB Environment Reference Name (High)
JEN_016: Name A JMS Destination Like [Name]Queue Or [Name]Topic (High)
JEN_017: Use A Correct Name For A JMS Environment Reference Name (High)
JEN_018: Use A Correct Name For A JDBC Environment Reference Name (High)
JEN_019: Name A Database Like [Name]DB (High)

## JEN\_001: Name An EJB Bean Class Like [Name]EJB or [Name]Bean (High)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JGuidelines\]JEN\\_001](mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JEN_001)]

A class that extends one of the enterprise bean types (SessionBean, EntityBean or MessageDrivenBean) must have a name that ends with EJB or Bean.



### Note

Use of the Bean suffix for the enterprise bean class name can lead to developers thinking they are dealing with a standard JavaBean and not an EJB. Use of the EJB suffix clearly states that the class represents an enterprise java bean implementation.

### WRONG

```
public class Foo implements EntityBean {
    public Foo() {
    }
    // ...
}
```

### RIGHT

```
public class FooBean implements EntityBean {
    public FooBean() {
    }
    // ...
}
```

## JEN\_002: Name An EJB Remote Home Interface Like [Name]Home (High)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JGuidelines\]JEN\\_002](mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JEN_002)]



An interface that extends `EJBHome` must have a name that ends with `Home`.

WRONG

```
import java.rmi.RemoteException;
import javax.ejb.EJBHome;

public interface Foo extends EJBHome {
    // ...
}
```

RIGHT

```
import java.rmi.RemoteException;
import javax.ejb.EJBHome;

public interface FooHome extends EJBHome {
    // ...
}
```

## JEN\_003: Name An EJB Remote Interface Like [Name] (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEN\\_003](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEN_003)]

A remote interface that extends `EJBObject` should be the name of the EJB.

WRONG

```
public interface FooRemote extends EJBObject {

    void bar(String value) throws RemoteException;

}
```

RIGHT

```
public interface Foo extends EJBObject {

    void bar(String value) throws RemoteException;

}
```

## JEN\_004: Name An EJB Local Home Interface Like [Name]LocalHome (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEN\\_006](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEN_006)]

Only classes that extend the `EJBLocalHome` interface should have a suffix `LocalHome`.

WRONG

```
public interface FooHome extends EJBLocalHome {
    // ...
}
```

RIGHT

```
public interface FooLocalHome extends EJBLocalHome {  
    // ...  
}
```

## JEN\_005: Name A Transfer Object Like [Name]TO (High)

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEN\_007]

Implementations of the J2EE pattern *Transfer Object* must have a suffix TO.



### Note

The *Transfer Object* pattern was previously called *Value Object* and normally used a name suffix VO.

WRONG

```
public class AccountData implements Serializable {  
    // ...  
}
```

RIGHT

```
public class AccountTO implements Serializable {  
    // ...  
}
```

## JEN\_006: Name An EJB In The Deployment Descriptor Like [Name]EJB (High)

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEN\_008]

RIGHT

```
<enterprise-beans>  
  <entity>  
    <ejb-name>FooEJB</ejb-name>  
    <home>be.vlaanderen.myproject.subpackage.FooHome</home>  
    <remote>be.vlaanderen.myproject.subpackage.Foo</remote>  
    <ejb-class>be.vlaanderen.myproject.subpackage.FooBean</ejb-class>  
    <!-- ... -->  
  </entity>  
  <!-- ... -->  
</enterprise-beans>
```

## JEN\_007: Name An EJB Display Name In The Deployment Descriptor Like [Name]JAR (High)

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEN\_009]

```
<ejb-jar>  
  <description>A Session EJB Example</description>  
  <display-name>SessionJAR</display-name>  
  <enterprise-beans>  
    <!-- ... -->
```

```
</enterprise-beans>
<assembly-descriptor>
  <!-- ... -->
</assembly-descriptor>
</ejb-jar>
```

## **JEN\_008: Name A Servlet Like [Name]Servlet (High)**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JEN\_010]

The name of a Servlet should always end with `Servlet`, for example when implementing a Front Controller Servlet the name should be `FrontControllerServlet`.

## **JEN\_009: Name A Primary Key Class Like [Name]PK (High)**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JEN\_011]

The name of primary key class used by Entity Beans must end with `PK`.

WRONG

```
public class FooKey implements java.io.Serializable {
    // ...
}
```

RIGHT

```
public class FooPK implements java.io.Serializable {
    // ...
}
```

## **JEN\_010: Name A Filter Servlet Like [Name]Filter (High)**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JEN\_012]

The name of a Filter Servlet should always end with `Filter`, for example `EncodingFilter`.

## **JEN\_011: Name A Local Interface Like [Name]Local (High)**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JEN\_013]

An interface that extends `EJBLocalObject` must have a name that ends with `Local`.

WRONG

```
import java.rmi.RemoteException;
import javax.ejb.EJBLocalObject;

public interface Foo extends EJBLocalObject {
    // ...
}
```

RIGHT

```
import java.rmi.RemoteException;
import javax.ejb.EJBLocalObject;

public interface FooLocal extends EJBLocalObject {
    // ...
}
```

## JEN\_012: Name A Data Access Object Like [Name]DAO (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEN\\_007](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEN_007)]

Implementations of the J2EE pattern *Data Access Object* must follow the naming convention [Name]DAO.

WRONG

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

RIGHT

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

## JEN\_013: Use A Correct Name For An Enterprise Application Display Name (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEN\\_015](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEN_015)]

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>
```

## JEN\_014: Use A Correct Name For A Web Module Display Name (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEN\\_016](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEN_016)]

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>mywebproject</display-name>
```

RIGHT

```
<display-name>MyWebProjectWAR</display-name>
```

## JEN\_015: Use A Correct Name For An EJB Environment Reference Name (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEN\\_017](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEN_017)]

All references to Enterprise Java Beans must be organized in the `ejb` subcontext of the application component's environment.

WRONG

```
<ejb-ref>
  <description>
    This is a reference to the entity bean that
    encapsulates access to Foo.
  </description>
  <ejb-ref-name>Foo</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>be.vlaanderen.FooHome</home>
  <remote>be.vlaanderen.Foo</remote>
</ejb-ref>
```

RIGHT

```
<ejb-ref>
  <description>
    This is a reference to the entity bean that
    encapsulates access to Foo.
  </description>
  <ejb-ref-name>ejb/Foo</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>be.vlaanderen.FooHome</home>
  <remote>be.vlaanderen.Foo</remote>
</ejb-ref>
```

## JEN\_016: Name A JMS Destination Like [Name]Queue Or [Name]Topic (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEN\\_018](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEN_018)]

JMS destinations must either have a suffix `Queue` or `Topic` depending on their `jms` type.

WRONG

```
<resource-env-ref>
  <description>
    This is a reference to a JMS queue
  </description>
  <resource-env-ref-name>jms/Foo</resource-env-ref-name>
  <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
</resource-env-ref>
```

RIGHT

```
<resource-env-ref>
  <description>
    This is a reference to a JMS queue
  </description>
  <resource-env-ref-name>jms/FooQueue</resource-env-ref-name>
  <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
</resource-env-ref>
```

## JEN\_017: Use A Correct Name For A JMS Environment Reference Name (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEN\\_019](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEN_019)]

All references to Java Messaging Service's must be organized in the `jms` subcontext of the application component's environment.

WRONG

```
<resource-env-ref>
  <description>
    This is a reference to a JMS queue
  </description>
  <resource-env-ref-name>FooQueue</resource-env-ref-name>
  <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
</resource-env-ref>
```

RIGHT

```
<resource-env-ref>
  <description>
    This is a reference to a JMS queue
  </description>
  <resource-env-ref-name>jms/FooQueue</resource-env-ref-name>
  <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
</resource-env-ref>
```

## JEN\_018: Use A Correct Name For A JDBC Environment Reference Name (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEN\\_020](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEN_020)]

All references to JDBC resources must be organized in the `jdbc` subcontext of the application component's environment.

WRONG

```
<resource-ref>
  <description>
    A data source for a Foo database
  </description>
  <res-ref-name>FooDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

RIGHT

```
<resource-ref>
  <description>
    A data source for a Foo database
  </description>
  <res-ref-name>jdbc/FooDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

## JEN\_019: Name A Database Like [Name]DB (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEN\\_020](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEN_020)]

The database name used by JDBC resource references must have a suffix DB.

WRONG

```
<resource-ref>
  <description>
    A data source for a Foo database
  </description>
  <res-ref-name>jdbc/Foo</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

RIGHT

```
<resource-ref>
  <description>
    A data source for a Foo database
  </description>
  <res-ref-name>jdbc/FooDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

## Java Comments Conventions Rules

Feedback

[[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]commentsJavaRules](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]commentsJavaRules)]

### Overview

**Table A.3. Java Comments Conventions Overview**

Rules
JAD_001: Do Not Use An Invalid Javadoc Comment Tag (High)
JAD_002: Provide A Correct File Comment For A File (High)
JAD_003: Provide A Javadoc Comment For A Class (Enforced)

Rules
JAD_004: Provide A JavaDoc Comment For A Constructor (Enforced)
JAD_005: Provide A JavaDoc Comment For A Method (Enforced)
JAD_007: Provide A JavaDoc comment For A Field (Enforced)
JAD_008: Provide a package.html per package (Low)

## JAD\_001: Do Not Use An Invalid Javadoc Comment Tag (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAD\\_001](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAD_001)]

This rule verifies code against accidental use of improper Javadoc tags. Replace misspelled tags.



### Note

An exception to this rule are javadoc tags that are used by third party tools like, for example XDoclet or Together.

## JAD\_002: Provide A Correct File Comment For A File (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAD\\_002](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAD_002)]

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.

## JAD\_003: Provide A JavaDoc Comment For A Class (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAD\\_003](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAD_003)]

The JavaDoc comments for a class are missing

WRONG

```
public class Foo {
    // ...
}
```

RIGHT

```
/**
 * More information about the class.
 */
```



```
* @author firstName lastName - companyName
* @version 1.2.2 - 25/09/2003
*/
public class Foo {
    // ...
}
```

## JAD\_004: Provide A JavaDoc Comment For A Constructor (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAD\\_004](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAD_004)]

The JavaDoc comments for a constructor is missing.

WRONG

```
public class Foo {
    public Foo(char c) {
    }

    /**
     * More information about the method here
     */
    public boolean isValid() {
    }
}
```

RIGHT

```
public class Foo {
    /**
     * More information about the constructor here
     *
     * @param c the character to be tested
     */
    public Foo(char c) {
    }

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

## JAD\_005: Provide A JavaDoc Comment For A Method (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAD\\_005](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAD_005)]

The JavaDoc comments are missing for a method.

WRONG

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

RIGHT

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

## JAD\_007: Provide A JavaDoc comment For A Field (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAD\\_007](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAD_007)]

The JavaDoc comments for a public, friendly or protected field are missing

WRONG

```
public class Foo {  
    public static final int FOO_CONSTANT = 1;  
}
```

RIGHT

```
public class Foo {  
    /**Foo constant value (set to 1)*/  
    public static final int FOO_CONSTANT = 1;  
}
```

## JAD\_008: 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.

## Java Coding Conventions Rules

Feedback

[[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]codingJavaRules](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]codingJavaRules)]

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. See JLANGSPEC .

## Overview

**Table A.4. Java Coding Conventions Overview**

Rules
JAC_001: Do Not Make An Attribute Non final And static (Low)
JAC_002: Do Not Reference A static Member Through An Object Instance (High)
JAC_003: Do Not Make A File Longer Than 2000 Lines (Enforced)
JAC_004: Do Not Make A Line Longer Than 120 Characters (Normal)
JAC_005: Do Not Use Complex Variable Assignment (High)
JAC_006: Do Not Code Numerical Constants Directly (Low)
JAC_007: Do Not Place Multiple Statements On The Same Line (High)
JAC_008: Parenthesize The Conditional Part Of A Ternary Conditional Expression (High)
JAC_009: Provide An Incremental In A for Statement (High)
JAC_010: Do Not Use A Demand Import (Enforced)
JAC_011: Provide A default case In A switch Statement (Enforced)
JAC_012: Use The Abbreviated Assignment Operator When Possible (Normal)
JAC_013: Do Not Make A Method Longer Then 60 Lines (Normal)
JAC_014: Do Not Make A switch Statement With More Than 256 Cases (Normal)
JAC_015: Do Not Chain Multiple Methods (Normal)
JAC_016: Use A Single return Statement (Normal)
JAC_017: Do Not Duplicate An import Declaration (Enforced)
JAC_018: Do Not Import A Class Of The Package To Which The Source File Belongs (Enforced)
JAC_019: Do Not Import A Class From The Package java.lang (Enforced)
JAC_020: Do Not Use An Equality Operation With A boolean Literal Argument (Enforced)
JAC_021: Do Not Import A Class Without Using It (Enforced)
JAC_022: Do Not Unnecessary Parenthesize A return Statement (Normal)
JAC_023: Do Not Declare A private Class Member Without Using It (Enforced)
JAC_024: Do Not Use Unnecessary Modifiers For An Interface Method (Low)
JAC_025: Do Not Use Unnecessary Modifiers For An Interface Field (Low)
JAC_026: Do Not Use Unnecessary Modifiers For An Interface (High)
JAC_027: Do Not Declare A Local Variable Without Using It (Enforced)
JAC_028: Do Not Do An Unnecessary Typecast (High)
JAC_029: Do Not Do An Unnecessary instanceof Evaluation (High)
JAC_030: Do Not Hide An Inherited Attribute (High)
JAC_031: Do Not Hide An Inherited static Method (High)
JAC_032: Do Not Declare Overloaded Constructors Or Methods With Different Visibility Modifiers (High)
JAC_033: Do Not Override A Non abstract Method With An abstract Method (High)
JAC_034: Do Not Override A private Method (High)
JAC_035: Do Not Overload A Super Class Method Without Overriding It (High)
JAC_036: Do Not Use A Non final static Attribute For Initialization (High)

Rules
JAC_037: Do Not Use Constants With Unnecessary Equal Values (High)
JAC_038: Provide At Least One Statement In A catch Block (Normal)
JAC_039: Do Not Catch java.lang.Exception Or java.lang.Throwable (Normal)
JAC_040: Do Not Give An Attribute A public Or Package Local Modifier (Enforced)
JAC_041: Provide At Least One Statement In A Statement Body (Enforced)
JAC_042: Do Not Compare Floating Point Types (Low)
JAC_043: Enclose A Statement Body In A Loop Or Condition Block (Enforced)
JAC_044: Explicitly Initialize A Local Variable (High)
JAC_045: Do Not Unnecessary Override The finalize Method (High)
JAC_046: Parenthesize Mixed Logical Operators (High)
JAC_047: Do Not Assign Values In A Conditional Expression (High)
JAC_048: Provide A break Statement Or Comment For A case Statement (Normal)
JAC_049: Use equals To Compare Strings (Enforced)
JAC_050: Use L Instead Of l At The End Of A long Constant (Enforced)
JAC_051: Do Not Use The synchronized Modifier For A Method (Normal)
JAC_052: Declare Variables Outside A Loop When Possible (Normal)
JAC_053: Do Not Append To A String Within A Loop (High)
JAC_054: Do Not Make Complex Calculations Inside A Loop When Possible (Low)
JAC_055: Provide At Least One Statement In A try Block (Enforced)
JAC_056: Provide At Least One Statement In A finally Block (Enforced)
JAC_057: Do Not Unnecessary Jumble Loop Incrementors (High)
JAC_058: Do Not Unnecessary Convert A String (High)
JAC_059: Override The equals And hashCode Methods Together (Enforced)
JAC_060: Do Not Use Double Checked Locking With Lazy Initialization (Enforced)
JAC_061: Do Not Return From Inside A try Block (High)
JAC_062: Do Not Return From Inside A finally Block (High)
JAC_063: Do Not Use A try Block Inside A Loop When Possible (Normal)
JAC_064: Do Not Use An Exception For Control Flow (High)
JAC_065: Do Not Unnecessary Use The System.out.print or System.err.print Methods (High)
JAC_066: Do Not Return In A Method With A Return Type of void (High)
JAC_067: Do Not Reassign A Parameter (Enforced)
JAC_068: Close A Connection Inside A finally Block (Enforced)
JAC_069: Do Not Declare A Method That Throws java.lang.Exception or java.lang.Throwable (Normal)
JAC_070: Do Not Use An instanceof Statement To Check An Exception (High)
JAC_071: Do Not Catch A RuntimeException (High)
JAC_072: Do Not Use printStackTrace (High)
JAC_073: Package Declaration Is Required (Enforced)

## JAC\_001: Do Not Make An Attribute Non final And

## static (Low)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_001](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_001)]

This rule helps you to avoid non final static attributes.

WRONG

```
static ArrayList InterfaceListeners = new ArrayList();
```

RIGHT

```
static final ArrayList InterfaceListeners = new ArrayList();
```

Exception(s):

- Using an attribute that represents the instance in a singleton can not be made final. In this case this rule is not applicable.
- Is not applicable when the static instance is used in a lazy initialization mechanism.

## JAC\_002: Do Not Reference A static Member Through An Object Instance (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_002](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_002)]

static members should be referenced through class names rather than through objects.

WRONG

```
class Fool {
    void func() {
        Foo2 obj = new Foo2();

        int i = obj.CONSTANT; // don't access through an object reference

        obj.operation();
    }
}

class Foo2 {
    static final int CONSTANT = 10;

    static void operation() {}
}
```

RIGHT

```
class Fool {
    void func() {
        int i = Foo2.CONSTANT;

        Foo2.operation();
    }
}
```

```
class Foo2 {  
    static final int CONSTANT = 10;  
  
    static void operation() {}  
}
```

## JAC\_003: Do Not Make A File Longer Than 2000 Lines (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_003](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_003)]

According to the Sun coding conventions, files longer than 2000 lines are cumbersome and should be avoided.

## JAC\_004: Do Not Make A Line Longer Than 120 Characters (Normal)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_004](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_004)]

According to Sun coding conventions, lines longer than 80 characters should be avoided, since they're not handled well by many terminals and tools.

## JAC\_005: Do Not Use Complex Variable Assignment (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_005](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_005)]

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;
```

## JAC\_006: Do Not Code Numerical Constants Directly (Low)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_006](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_006)]

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.



### Tip

Add `static final` attributes for numeric constants. See also rule JAN\_005: Use A Correct Name For A Constant (Enforced).

## JAC\_007: Do Not Place Multiple Statements On The Same Line (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_007](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_007)]

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++;
```

## JAC\_008: Parenthesize The Conditional Part Of A Ternary Conditional Expression (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_008](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_008)]

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;
```

## JAC\_009: Provide An Incremental In A for Statement (High)

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC\_009]

This rule checks whether the third argument of a `for` statement is missing.

WRONG

```
for (Enumeration enum = getEnum(); enum.hasMoreElements();) {  
    Object o = enum.nextElement();  
    doSomeProc(o);  
}
```



### Tip

Either provide the incremental part of the `for` structure, or cast the `for` statement into a `while` statement.

RIGHT

```
Enumeration enum = getEnum();  
while (enum.hasMoreElements()) {  
    Object o = enum.nextElement();  
    doSomeProc(o);  
}
```

## JAC\_010: Do Not Use A Demand Import (Enforced)

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC\_010]

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 be.vlaanderen.myproject.mypackage.*;
```

RIGHT

```
import java.util.Date;  
import be.vlaanderen.myproject.mypackage.MyClass;
```

## JAC\_011: Provide A default case In A switch Statement (Enforced)

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC\_011]

According to Sun coding conventions, every `switch` statement should include a default case.



### Tip

Put an assertion in the default case if you expect the default case never to be called.



## JAC\_012: Use The Abbreviated Assignment Operator When Possible (Normal)

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC\_012]

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;  
}
```

## JAC\_013: Do Not Make A Method Longer Then 60 Lines (Normal)

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC\_013]

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.

## JAC\_014: Do Not Make A `switch` Statement With More Than 256 Cases (Normal)

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC\_014]

Switch statements should not have more than 256 cases.



### Note

This rule is redundant if you consider rule JAC\_013: Do Not Make A Method Longer Then 60 Lines (Normal)

## JAC\_015: Do Not Chain Multiple Methods (Normal)

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC\_015]

Do not chain method calls. Exceptionally, a chain of 2 method calls can be used.

Take the following line of code:

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

If the return value from one of the methods is `null` you would get a `NullPointerException`. But

which one is it?

It's better to use:

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



### Note

See also the "Law of Demeter" [[http://www.wikipedia.org/wiki/Law\\_of\\_Demeter](http://www.wikipedia.org/wiki/Law_of_Demeter)] which is a design guideline for developing software, explaining that *an object should assume as little as possible about the structure or properties*.

## JAC\_016: Use A Single return Statement (Normal)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_016](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_016)]

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 < collection.size(); i++) {
        if (collection.get(i).equals(something)) {
            return i;
        }
    }
    return -1;
}
```

RIGHT

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

## JAC\_017: Do Not Duplicate An import Declaration (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_017](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_017)]

There should be at most one import declaration that imports a particular class/package.

WRONG

```
package be.vlaanderen.myproject.mypackage.subpackage;

import java.io.IOException;
import java.io.Reader;
import java.sql.Time;
```

```
import java.sql.Time;

public class Foo {
}
```

RIGHT

```
package be.vlaanderen.myproject.mypackage.subpackage;

import java.io.IOException;
import java.io.Reader;
import java.sql.Time;

public class Foo {
}
```

## JAC\_018: Do Not Import A Class Of The Package To Which The Source File Belongs (Enforced)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_018](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JAC_018)]

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

```
package be.vlaanderen.myproject.mypackage;

import be.vlaanderen.myproject.mypackage.OtherFoo;

public class Foo {
    public void setOtherFoo(OtherFoo other) {
        // ...
    }
}
```

RIGHT

```
package be.vlaanderen.myproject.mypackage;

public class Foo {
    public void setOtherFoo(OtherFoo other) {
        // ...
    }
}
```

## JAC\_019: Do Not Import A Class From The Package `java.lang` (Enforced)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_019](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JAC_019)]

Explicit import of classes from the package `java.lang` should not be performed.

WRONG

```
package be.vlaanderen.myproject.mypackage.subpackage;
```

---

```
import java.lang.String;

public class Foo {
    public void setName(String str) {
        // ...
    }
}
```

RIGHT

```
package be.vlaanderen.myproject.mypackage.subpackage;

public class Foo {
    public void setName(String str) {
        // ...
    }
}
```

## JAC\_020: Do Not Use An Equality Operation With A `boolean` Literal Argument (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_020](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_020)]

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 isLoading = true;

if (isLoading == true) {
    // ...
}
```

RIGHT

```
boolean isLoading = true;

if (isLoading) {
    // ...
}
```

## JAC\_021: Do Not Import A Class Without Using It (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_021](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_021)]

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.

WRONG

```
import java.awt.Button;
```

```
import java.util.Dictionary;
import java.util.Hashtable;
import java.util.Stack;
import java.util.Vector;

public class Foo {
    Dictionary dict;

    void func(Vector vec) {
        Hashtable ht;

        // ...
    }
}
```

RIGHT

```
import java.util.Dictionary;
import java.util.Hashtable;
import java.util.Vector;

public class Foo {
    Dictionary dict;

    void func(Vector vec) {
        Hashtable ht;

        // ...
    }
}
```

## JAC\_022: Do Not Unnecessary Parenthesize A return Statement (Normal)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_022](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_022)]

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();
```

## JAC\_023: Do Not Declare A private Class Member Without Using It (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_023](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_023)]

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).

**Tip**

Examine the program. If the given member is really unnecessary, remove it (or at least comment it out).

## JAC\_024: Do Not Use Unnecessary Modifiers For An Interface Method (Low)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_024](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_024)]

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 (See JLANGSPEC - section 9.4). Some day, this may be disallowed by Java. It is far less painful to clean up your code before this happens.

WRONG

```
interface Foo {  
    public abstract void operation();  
}
```

RIGHT

```
interface Foo {  
    void operation();  
}
```

## JAC\_025: Do Not Use Unnecessary Modifiers For An Interface Field (Low)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_025](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_025)]

All interface fields are implicitly `public`, `static` and `final`. However, the language permits the use of these modifiers with interface fields. Such use is not recommended (See JLANGSPEC - section 9.3). Some day, this may be disallowed by Java. It is far less painful to clean up your code before this happens.

WRONG

```
interface Foo {  
    public final static int ATTR = 10;  
}
```

RIGHT

```
interface Foo {  
    int ATTR = 10;  
}
```

**Note**

If the above interface `Foo` is used to collect all the constant then access the constant variable `ATTR` through the interface and not by implementing the interface ! To avoid this design misuse just place all your constants in a class.

## JAC\_026: Do Not Use Unnecessary Modifiers For An Interface (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_026](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_026)]

The modifier `abstract` is considered obsolete and should not be used.

WRONG

```
abstract interface Foo {  
}
```

RIGHT

```
interface Foo {  
}
```

## JAC\_027: Do Not Declare A Local Variable Without Using It (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_027](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_027)]

Local variables should be used.

WRONG

```
int bar(int param) {  
    int unused_var;  
  
    return 2 * param;  
}
```

## JAC\_028: Do Not Do An Unnecessary Typecast (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_028](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_028)]

Checks for the use of type casts that are not necessary, including redundant upcasts.

WRONG

```
public class FooA {  
}  
  
public class FooB extends FooA {  
    public void bar(FooB b) {  
        FooA a = (FooA) b; // VIOLATION  
    }  
}
```

RIGHT

```
public class FooA {  
}  
  
public class FooB extends FooA {  
    public void bar(FooB b) {
```

```
        FooA a = b;
    }
}
```

**Tip**

- Delete redundant upcasts to improve readability.
- In THINKING from Bruce Eckel, chapter 3 *Controlling Program Flow* has a section on casting operators and chapter 7 *Polymorphism* writes about up casting.

## JAC\_029: Do Not Do An Unnecessary instanceof Evaluation (High)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_029](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JAC_029)]

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 FooA {
    public FooA() {}

    public void bar() {}
}

class FooB extends FooA {
    public void barbar(FooB b) {
        if (b instanceof FooA) { // VIOLATION
            b.bar();
        }
    }
}
```

**RIGHT**

```
public class FooA {
    public FooA() {}

    public void bar() {}
}

class FooB extends FooA {
    public void barbar(FooB b) {
        b.bar();
    }
}
```



## JAC\_030: Do Not Hide An Inherited Attribute (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_030](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_030)]

This rule detects when attributes declared in child classes hide inherited attributes.

WRONG

```
class Foo {
    int attrib = 10;
}

class Fool extends Foo {
    int attrib = 0;

    // ...
}
```

RIGHT

```
class Foo {
    int attrib = 10;
}

class Fool extends Foo {
    int foolAttrib = 0;

    // ...
}
```

## JAC\_031: Do Not Hide An Inherited static Method (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_031](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_031)]

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

WRONG

```
class Fool extends Foo {
    void oper1() {
    }

    static void oper2() {
    }
}

class Foo {
    static void oper1() {
    }

    static void oper2() {
    }
}
```

RIGHT

```
class Fool extends Foo {
    static void anOper2() {
    }
}
```

```
        void anOper1() {  
        }  
    }  
  
    class Foo {  
        static void oper1() {  
        }  
  
        static void oper2() {  
        }  
    }
```

## JAC\_032: Do Not Declare Overloaded Constructors Or Methods With Different Visibility Modifiers (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_032](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_032)]

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.

WRONG

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

RIGHT

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

## JAC\_033: Do Not Override A Non abstract Method With An abstract Method (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_033](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_033)]

Checks for the overriding of non-abstract methods by abstract methods in a subclass.

WRONG

```
public class Animal {
    void func() {
    }
}

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



### Tip

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.

RIGHT

```
public class Animal {
    void func() {
    }
}

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

## JAC\_034: Do Not Override A private Method (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_034](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_034)]

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() {
    }
}
```

RIGHT

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

class Elephant extends Animal {
    private void extFunc() {
    }
}
```

```
}
```

## JAC\_035: Do Not Overload A Super Class Method Without Overriding It (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_035](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_035)]

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.

### WRONG

```
class FooB {
    public void bar(int i) {
    }

    public void bar(Object o) {
    }
}

public class FooA extends FooB {
    // additional overloaded method
    public void bar(char c) {
    }

    public void bar(Object o) {
    }
}
```

### RIGHT

```
class FooB {
    public void bar(int i) {
    }

    public void bar(Object o) {
    }
}

public class FooA extends FooB {
    // additional overloaded method
    public void bar(char c) {
    }

    public void bar(int i) {
    }

    public void bar(Object o) {
    }
}
```

## JAC\_036: Do Not Use A Non final static Attribute For Initialization (High)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_036](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JAC_036)]

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;
}
```

#### 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;
}
```

## JAC\_037: Do Not Use Constants With Unnecessary Equal Values (High)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_037](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JAC_037)]

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";
    }
}
```

**Note**

Enumerations are finally supported in JDK 1.5, making the above example a lot simpler and eliminating the mentioned problem!

## JAC\_038: Provide At Least One Statement In A catch Block (Normal)

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC\_038]

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.

**Tip**

When an empty catch block is specified it usually means that the exception being caught 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.

**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");
}
```

**Tip**

An interesting article on the usage of throw-object/catch-object exception-handling was written and published on the Java World website. More info at <http://www.javaworld.com/javaworld/jw-04-2002/jw-0405-java101.html>.

## JAC\_039: Do Not Catch `java.lang.Exception` Or `java.lang.Throwable` (Normal)

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC\_039]

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 rules 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.

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
}
```



### Tip

An interesting article on the usage of throw-object/catch-object exception-handling was written and published on the JavaWorld website. More info at <http://www.javaworld.com/javaworld/jw-04-2002/jw-0405-java101.html>.

## JAC\_040: Do Not Give An Attribute A `public` Or Package Local Modifier (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_040](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_040)]

Declare the attributes either `private` or `protected` and provide operations to access or change them. Also referred to as full encapsulation.

There might be one exception when some class is used just as struct in C language: it just holds some values, and thus has no methods.

WRONG

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

RIGHT

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

    int getAttr1() {
        return attr1;
    }
}
```

```
    public int getAttr2() {
        return attr2;
    }

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

## JAC\_041: Provide At Least One Statement In A Statement Body (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_041](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_041)]

As far as possible, avoid using statements with empty bodies.

WRONG

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



### Tip

Provide a statement body or change the logic of the program (for example, use a while statement instead of a for statement).

RIGHT

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

## JAC\_042: Do Not Compare Floating Point Types (Low)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_042](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_042)]

Avoid testing floating point numbers for equality. Floating-point numbers that should be equal are not always equal due to rounding problems.

WRONG

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



**Tip**

Replace direct comparison with estimation of absolute value of difference.

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) {
            // ...
        }
    }
}
```

## JAC\_043: Enclose A Statement Body In A Loop Or Condition Block (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_043](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_043)]

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();
}
```

## JAC\_044: Explicitly Initialize A Local Variable (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_044](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_044)]

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;  
    // ...  
}
```

## JAC\_045: Do Not Unnecessary Override The `finalize` Method (High)

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC\_045]

As mentioned in JLANGSPEC, 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 EFFECJ 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();  
    }  
    */  
}
```

## JAC\_046: Parenthesize Mixed Logical Operators (High)

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC\_046]

An expression containing multiple logical operators together should be parenthesized properly.

WRONG

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

**Tip**

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`

**RIGHT**

```
void bar() {
    boolean a = false;
    boolean b = true;
    boolean c = false;

    // ...

    if (a || (b && c)) {
        // ...
    }
}
```

## JAC\_047: Do Not Assign Values In A Conditional Expression (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_047](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_047)]

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()) {
    // ...
}
```

## JAC\_048: Provide A break Statement Or Comment For A case Statement (Normal)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_048](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_048)]

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.

**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;

    // ...
}
```

**Tip**

Add the comment `// falls through` where the `break` statement would normally be.

**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;

    // ...
}
```

## JAC\_049: Use `equals` To Compare Strings (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_049](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_049)]

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.

**WRONG**

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

**Tip**

Replace the '==' operator with the equals method.

RIGHT

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

## JAC\_050: Use L Instead Of 1 At The End Of A long Constant (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_050](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_050)]

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 = 0x00011111;  
}
```

RIGHT

```
void bar() {  
    long var = 0x00011111L;  
}
```

## JAC\_051: Do Not Use The synchronized Modifier For A Method (Normal)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_051](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_051)]

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.

WRONG

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

**Tip**

Use synchronized statements instead of synchronized methods.

RIGHT

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



### Note

See rule JAC\_061: Do Not Return From Inside A try Block (High).

## JAC\_052: Declare Variables Outside A Loop When Possible (Normal)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_052](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_052)]

This rule recommends declaring local variables outside the loops. The reason: as a rule, declaring variables inside the loop is less efficient.

WRONG

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

while (true) {
    int var2 = 0;
    // ...
}

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



### Tip

Move variable declarations out of the loop

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);
```

## JAC\_053: Do Not Append To A String Within A Loop (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_053](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_053)]

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.

### WRONG

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



### Tip

Use `StringBuffer` class instead of `String`

### 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();  
    }  
}
```

## JAC\_054: Do Not Make Complex Calculations Inside A Loop When Possible (Low)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_054](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_054)]

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.

WRONG

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



### Tip

Assign the expression to a variable before the loop and use that variable instead.

RIGHT

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

## JAC\_055: Provide At Least One Statement In A try Block (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_055](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_055)]

Empty try blocks should be removed because they bring no added value to your code.

WRONG

```
try {
    // Please remove this try-catch block !
} catch (SQLException e) {
    logger.log(Level.WARNING, "SQL Problem", e);
}
```

## JAC\_056: Provide At Least One Statement In A finally Block (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_056](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_056)]

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);
    }
```

## JAC\_057: Do Not Unnecessary Jumble Loop Incrementors (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_057](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_057)]

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);
        }
    }
}
```

## JAC\_058: Do Not Unnecessary Convert A string (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_058](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_058)]

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);
}
```

## JAC\_059: Override The equals And hashCode Methods Together (Enforced)

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JAC\_059]

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
    }
}
```

RIGHT

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

    public int hashCode() {
        // return some hash value
    }
}
```

## JAC\_060: Do Not Use Double Checked Locking With Lazy Initialization (Enforced)

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JAC\_060]

Lazy initialization code could return a `NullPointerException` when the method is used in a multi-threaded application, however the double checked lazy initialization does not work for all cases (See Effective Java).

WRONG

```
public class Foo {

    private static Resource resource = null;

    public static Resource getResource() {
        if (resource == null) {
            synchronized (Foo.class) {
                if (resource == null) {
                    resource = new Resource();
                }
            }
        }
        return resource;
    }
}
```

**Tip**

Read Item 48: Synchronize access to shared mutable data in EFPECJ .

## JAC\_061: Do Not Return From Inside A `try` Block (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_061](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_061)]

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.

**Note**

See also rule JAC\_016: Use A Single `return` Statement (Normal).

## JAC\_062: Do Not Return From Inside A `finally` Block (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_062](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_062)]

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

WRONG

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

RIGHT

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

## JAC\_063: Do Not Use A `try` Block Inside A Loop When Possible (Normal)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_063](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_063)]

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);
        }
    }
}
```

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);
    }
}
```

## JAC\_064: Do Not Use An Exception For Control Flow (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_064](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_064)]

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
    }
}
```

RIGHT

```
public void bar(int numbers[]) {
```

---

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

## JAC\_065: Do Not Unnecessary Use The `System.out.print` or `System.err.print` Methods (High)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_065](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JAC_065)]

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 (JMSEException e) {  
        System.out.println("exception occurred");  
        e.printStackTrace();  
    } finally {  
        System.out.println("cleaning up");  
    }  
}
```

RIGHT

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

## JAC\_066: Do Not Return In A Method With A Return Type of `void` (High)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_065](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JAC_065)]

Avoid unnecessary return statements, just remove this statement.

WRONG

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

RIGHT

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

## JAC\_067: Do Not Reassign A Parameter (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_067](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_067)]

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";  
}
```

## JAC\_068: Close A Connection Inside A `finally` Block (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_068](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_068)]

Ensure that a connection object is always closed within the `finally` block.

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);  
    }  
}
```

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);
    }
}
```

**Tip**

Use a helper method to close the connection, this makes the `finally` block small and readable.

## JAC\_069: Do Not Declare A Method That Throws `java.lang.Exception` or `java.lang.Throwable` (Normal)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_069](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_069)]

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 JAC\_039: Do Not Catch `java.lang.Exception` Or `java.lang.Throwable` (Normal).

**WRONG**

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

**RIGHT**

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

## JAC\_070: Do Not Use An `instanceof` Statement To Check An Exception (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAC\\_072](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAC_072)]

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, "NumberFormatException", nfe);
    }
    if (e instanceof IllegalArgumentException) {
        IllegalArgumentException iae = (IllegalArgumentException) e;
        logger.log(Level.WARNING, "Illegal argument", iae);
    }
}
```

```
}
```

RIGHT

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

## JAC\_071: 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.

## JAC\_072: 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.

## JAC\_073: 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 Rules

Feedback

[mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]codingJ2eeRules]

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.

## Overview

**Table A.5. J2EE Coding Conventions Overview**

Rules
JEC_001: Do Not Create A Class Loader (High)
JEC_002: Do Not Read Or Write A File Descriptor (High)
JEC_003: Do Not Use A AWT, Swing Or Other UI API In An EJB (High)
JEC_004: Do Not Attempt To Load A Native Library In An EJB (High)
JEC_005: Do Not Attempt To Obtain The Security Policy Information In An EJB (High)
JEC_006: Do Not Attempt To Set The Socket Factory In An EJB (High)
JEC_007: Do Not Attempt To Listen On A Socket In An EJB (High)
JEC_008: Do Not Attempt To Use The Subclass Or Object Substitution Features Of The Java Serializa- tion Protocol (High)
JEC_009: Do Not Attempt To Manage A Thread (High)
JEC_010: Do Not Use The java.io Package To Access The File System (Normal)
JEC_011: Do Not Pass An EJB's this Reference As An Argument Or Method Result (Enforced)
JEC_012: Do Not Throw A RemoteException From An EJB Implementation Method (High)
JEC_013: Make All EJB interfaces and classes public (High)
JEC_014: Do Not Make A EJB implementation class final (High)
JEC_015: Provide An ejbCreate Method For A Session Bean (High)
JEC_016: Provide A public Default Constructor For A EJB implementation class (High)
JEC_017: Do Not Override The finalize Method For A EJB implementation class (High)
JEC_018: Return void For An ejbCreate Method Of A Session Bean (High)
JEC_019: Return The EJB Remote Interface For A create Method Of An EJB Remote Home Interface (Enforced)
JEC_020: Make A create Method Of An EJB Home Interface Throw An javax.ejb.CreateException (Enforced)
JEC_022: Do Not Declare A Finder Method For A CMP EJB (High)
JEC_024: Match An ejbPostCreate Method To An ejbCreate Method For An Entity Bean (High)

Rules
JEC_025: Declare An ejbCreate Method public For An Entity Bean (High)
JEC_026: Return The Primary Key For An ejbCreate[Name] Method Of An Entity Bean (High)
JEC_027: Do Not Declare An ejbCreate Or ejbPostCreate Method As final Or static For An Entity Bean (High)
JEC_028: Return void For An ejbPostCreate Method Of An Entity Bean (High)
JEC_029: Declare An ejbFindByPrimaryKey Method For A BMP EJB (High)
JEC_030: Declare A ejbFind Method public For A BMP EJB (High)
JEC_031: Do Not Declare A ejbFind Method final Or static (High)
JEC_032: Make A find Method Of An EJB Home Interface throw javax.ejb.FinderException (High)
JEC_033: Declare An ejbSelect Method Of An Entity Bean public (High)
JEC_034: Make An ejbSelect Method Of An Entity Bean throw javax.ejb.FinderException (High)
JEC_035: Return The EJB Local Interface For A create Method In An EJB Local Home Interface (Enforced)
JEC_036: Declae An ejbPostCreate Method Of An Entity Bean public (High)
JEC_037: Make A Method Of An EJB Remote Interface Throw java.rmi.RemoteException (Enforced)
JEC_038: Make A Method Of An EJB Remote Home Interface Throw java.rmi.RemoteException (Enforced)
JEC_039: Do Not Make A Method Of An EJB Local Interface Throw java.rmi.RemoteException (Enforced)
JEC_040: Do Not Make A Method Of An EJB Local Home Interface Throw java.rmi.RemoteException (Enforced)
JEC_042: Declare An ejbSelect Method Of A CMP EJB abstract (High)
JEC_043: Make An ejbCreate Method Of An Entity Bean Throw javax.ejb.CreateException (High)
JEC_044: Declare An ejbHome Method Of An Entity Bean public (High)
JEC_045: Do Not Declare An ejbHome Method Of An Entity Bean final Or static (High)
JEC_046: Do Not Make An ejbHome Method Of An Entity Bean Throw java.rmi.RemoteException (High)
JEC_047: Make A Message Bean Implementation Implement javax.jms.MessageListener (High)
JEC_052: Provide An ejbCreate Method For A Message Bean (High)
JEC_053: Return void For An ejbCreate Method Of A Message Bean (High)
JEC_054: Do Not Declare Arguments For An ejbCreate Method Of A Message Bean (High)
JEC_055: Provide A Valid RMI Method Signature For An EJB Remote Home Interface (High)
JEC_056: Do No Print Unnecessary Static Content From A Servlet (High)
JEC_057: Use Custom JSP Tags Instead Of Scriptlets (High)
JEC_058: Do Not Forward A Request From A JSP Page (High)

## JEC\_001: Do Not Create A Class Loader (High)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JGuidelines\]JEC\\_001](mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JEC_001)]

*Allowing the enterprise bean to access information about other classes and to access the classes in a manner that is normally disallowed by the Java programming language could compromise security. The enterprise bean must not attempt to create a class loader; obtain the current class loader; set the context class loader; set security manager; create a new security manager; stop the JVM; or change the input, output, and error streams. (See EJB20SPEC - C.1.2 - Programming restrictions)*

These functions are reserved for the EJB Container. Allowing the enterprise bean to use these functions could compromise security and decrease the Container's ability to properly manage the runtime environment.

## **JEC\_002: Do Not Read Or Write A File Descriptor (High)**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JEC\_002]

*The enterprise bean must not attempt to directly read or write a file descriptor. Allowing the enterprise bean to read and write file descriptors directly could compromise security. (See EJB20SPEC - C.1.2 - Programming restrictions)*

## **JEC\_003: Do Not Use A AWT, Swing Or Other UI API In An EJB (High)**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JEC\_003]

*An enterprise bean must not use the AWT functionality to attempt to output information to a display, or to input information from a keyboard. Most servers do not allow direct interaction between an application program and a keyboard/display attached to the server system. (See EJB20SPEC - C.1.2 - Programming restrictions)*

## **JEC\_004: Do Not Attempt To Load A Native Library In An EJB (High)**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JEC\_004]

*The enterprise bean must not attempt to load a native library. This function is reserved for the EJB Container. Allowing the enterprise bean to load native code would create a security hole. (See EJB20SPEC - C.1.2 - Programming restrictions)*

## **JEC\_005: Do Not Attempt To Obtain The Security Policy Information In An EJB (High)**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JEC\_005]

*The enterprise bean must not attempt to obtain the security policy information for a particular code source. Allowing the enterprise bean to access the security policy information would create a security hole. (See EJB20SPEC - C.1.2 - Programming restrictions)*

## **JEC\_006: Do Not Attempt To Set The Socket Factory In An EJB (High)**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JEC\_006]

*The enterprise bean must not attempt to set the socket factory used by ServerSocket, Socket, or the stream handler factory used by URL. These networking functions are reserved for the EJB Container. Allowing the enterprise bean to use these functions could compromise security and decrease the Container's ability to properly manage the runtime environment. (See EJB20SPEC - C.1.2 - Programming restrictions)*

## **JEC\_007: Do Not Attempt To Listen On A Socket In An EJB (High)**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JEC\_007]

*An enterprise bean must not attempt to listen on a socket, accept connections on a socket, or use a socket for multicast. The EJB architecture allows an enterprise bean instance to be a network socket client, but it does not allow it to be a network server. Allowing the instance to become a network server would conflict with the basic function of the enterprise bean-- to serve the EJB clients. (See EJB20SPEC - C.1.2 - Programming restrictions)*

## **JEC\_008: Do Not Attempt To Use The Subclass Or Object Substitution Features Of The Java Serialization Protocol (High)**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JEC\_008]

*The enterprise bean must not attempt to use the subclass and object substitution features of the Java Serialization Protocol. Allowing the enterprise bean to use these functions could compromise security. (See EJB20SPEC - C.1.2 - Programming restrictions)*

## **JEC\_009: Do Not Attempt To Manage A Thread (High)**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JEC\_009]

*The enterprise bean must not attempt to manage threads. The enterprise bean must not attempt to start, stop, suspend, or resume a thread; or to change a thread's priority or name. The enterprise bean must not attempt to manage thread groups. These functions are reserved for the EJB Container. Allowing the enterprise bean to manage threads would decrease the Container's ability to properly manage the run-time environment. (See EJB20SPEC - C.1.2 - Programming restrictions)*

## **JEC\_010: Do Not Use The `java.io` Package To Access The File System (Normal)**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JEC\_010]

*An enterprise bean must not use the `java.io` package to attempt to access files and directories in the file system. The file system APIs are not well-suited for business components to access data. Business components should use a resource manager API, such as JDBC API, to store data. (See EJB20SPEC - C.1.2 - Programming restrictions)*

## **JEC\_011: Do Not Pass An EJB's *this* Reference As An Argument Or Method Result (Enforced)**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JEC\_011]

*The enterprise bean must not attempt to pass 'this' as an argument or method result. The enterprise bean must pass the result of the methods `SessionContext.getEJBObject()` or `EntityContext.getEJBObject()` instead. (See EJB20SPEC - C.1.2 - Programming restrictions)*

## **JEC\_012: Do Not Throw A `RemoteException` From An EJB Implementation Method (High)**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JEC\_012]

The EJB 2.0 specification no longer allows for methods in the EJB implementation to throw a `java.rmi.RemoteException`. This only goes for methods that are defined in the EJBs remote or home interface.

If the enterprise bean method encounters a system-level exception or error that does not allow the method to successfully complete, the method should throw a suitable non-application exception that is

compatible with the method's `throws` clause. While the EJB specification does not prescribe the exact usage of the exception, it encourages the Bean Provider to follow these guidelines:

If the bean method encounters a `RuntimeException` or error, it should simply propagate the error from the bean method to the container (i.e., the bean method does not have to catch the exception).

If the bean method performs an operation that results in a checked exception that the bean method cannot recover, the bean method should throw the `javax.ejb.EJBException` that wraps the original exception.

## JEC\_013: Make All EJB interfaces and classes **public** (High)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_013](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JEC_013)]

Enterprise bean classes must be defined as `public`. (See EJB20SPEC - 7.10.2, 10.6.2, 12.2.2, 15.7.2)

WRONG

```
class FooBean implements SessionBean {  
    // ...  
}
```

RIGHT

```
public class FooBean implements SessionBean {  
    // ...  
}
```

## JEC\_014: Do Not Make A EJB implementation class **final** (High)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_014](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JEC_014)]

Enterprise bean implementation classes must not be declared `final`. (See EJB20SPEC - 7.10.2, 15.7.2)

WRONG

```
public final class FooBean implements SessionBean {  
    // ...  
}
```

RIGHT

```
public class FooBean implements SessionBean {  
    // ...  
}
```

## JEC\_015: Provide An **ejbCreate** Method For A Session Bean (High)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_015](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JEC_015)]

*Enterprise session beans must implement one or more `ejbCreate` method) whose signatures must also be `public` and return `void`. (See EJB20SPEC - 7.10.3.)*

WRONG

```
public class FooBean implements SessionBean {  
    // ...  
}
```

RIGHT

```
public class FooBean implements SessionBean {  
    public void ejbCreate() throws CreateException {  
        // ...  
    }  
    // ...  
}
```



### Note

See also rule JEC\_018: Return void For An ejbCreate Method Of A Session Bean (High).

## JEC\_016: Provide A `public` Default Constructor For A EJB implementation class (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_016](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEC_016)]

Enterprise bean implementation classes must have a public default constructor. The Container uses this constructor to create the bean instances. (See EJB20SPEC - 7.10.2, 10.6.2, 12.2.2 and 15.7.2).

WRONG

```
public class FooBean implements SessionBean {  
    public FooBean(int bar) {  
    }  
    // ...  
}
```

RIGHT

```
public class FooBean implements SessionBean {  
}
```



### Note

It is recommended not to provide any constructor at all.

## JEC\_017: Do Not Override The `finalize` Method For A EJB implementation class (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_017](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEC_017)]

Enterprise bean implementation classes must not override the `finalize` method. (See EJB20SPEC -

7.10.2, 10.6.2, 12.2.2 and 15.7.2)

WRONG

```
public class FooBean implements SessionBean {  
    // ...  
    protected void finalize() {  
        super.finalize();  
    }  
}
```

RIGHT

```
public class FooBean implements SessionBean {  
    // ...  
}
```



### Note

See also rule JAC\_045: Do Not Unnecessary Override The finalize Method (High)

## JEC\_018: Return void For An ejbCreate Method Of A Session Bean (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_018](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEC_018)]

*Enterprise session beans must declare one or more ejbCreate methods of which the return type must be void. (See EJB20SPEC - 7.10.3.)*

WRONG

```
public class FooBean implements SessionBean {  
    public String ejbCreate() throws CreateException {  
        // ...  
    }  
    // ...  
}
```

RIGHT

```
public class FooBean implements SessionBean {  
    public void ejbCreate() throws CreateException {  
        // ...  
    }  
    // ...  
}
```

## JEC\_019: Return The EJB Remote Interface For A create Method Of An EJB Remote Home Interface

## (Enforced)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_019](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JEC_019)]

*The create methods declared in a remote home interface must return a remote interface of type `ejbObject`. (See EJB20SPEC - 7.10.6, 10.6.10 and 12.2.9.)*

### WRONG

```
public interface FooHome extends EJBHome {  
  
    Foo create() throws CreateException, RemoteException;  
}  
  
// In the file Foo.java  
// ...  
public interface Foo {  
    // ...  
}
```

### RIGHT

```
public interface FooHome extends EJBHome {  
  
    Foo create() throws CreateException, RemoteException;  
}  
  
// In the file Foo.java  
// ...  
public interface Foo extends EJBObject {  
    // ...  
}
```

## JEC\_020: Make A create Method Of An EJB Home Interface Throw An `javax.ejb.CreateException` (Enforced)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_020](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JEC_020)]

*The create methods declared in a home interface (remote or local) must specify `javax.ejb.CreateException` in the throws clause. (See EJB20SPEC - 7.10.6, 10.6.10, 10.6.12, 12.2.9 and 12.2.11.)*

### WRONG

```
import java.rmi.RemoteException;  
  
public interface FooHome extends EJBHome {  
  
    public abstract Foo create() throws RemoteException;  
}
```

### RIGHT

```
import java.rmi.RemoteException;  
import javax.ejb.CreateException;  
  
public interface FooHome extends EJBHome {
```



```
    public abstract Foo create() throws CreateException, RemoteException;
}
```

## JEC\_022: Do Not Declare A Finder Method For A CMP EJB (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_022](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEC_022)]

Enterprise entity beans that are declared abstract use container managed persistency. Such beans must not declare finder methods (ejbFindXXX) as they are provided by the EJB container. (See EJB20SPEC - 10.5.2.)

### WRONG

```
public abstract class FooBean implements EntityBean {
    public FooBean() {
    }

    public Collection ejbFindBar(int index) {
        // ...
    }
}
```

### RIGHT

```
public abstract class FooBean implements EntityBean {
    public FooBean() {
    }
}

// In the home interface source file
// ...
public interface FooHome extends EJBHome {

    public abstract Collection findBar(int index)
        throws FinderException, RemoteException;
}
```

## JEC\_024: Match An ejbPostCreate Method To An ejbCreate Method For An Entity Bean (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_024](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEC_024)]

Each ejbPostCreate method in an enterprise entity bean must match an ejbCreate method with the same arguments. (See EJB20SPEC - 10.6.5 and 12.2.4.)

### WRONG

```
public String ejbCreate(String fooId, String bar) throws CreateException {
    // ...
}

public void ejbPostCreate(String fooId) throws CreateException {
    // ...
}
```

### RIGHT

```
public String ejbCreate(String fooId, String bar) throws CreateException {  
    // ...  
}  
  
public void ejbPostCreate(String fooId, String bar) throws CreateException {  
    // ...  
}
```

## JEC\_025: Declare An `ejbCreate` Method `public` For An Entity Bean (High)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_025](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JEC_025)]

Enterprise beans must declare `ejbCreate` methods with `public` access. (See EJB20SPEC - 7.10.3, 10.6.4, 10.6.5, 12.2.3, 12.2.4 and 15.7.3.)

WRONG

```
private String ejbCreate(String fooId, String bar) throws CreateException {  
    // ...  
}
```

RIGHT

```
public String ejbCreate(String fooId, String bar) throws CreateException {  
    // ...  
}
```

## JEC\_026: Return The Primary Key For An `ejbCreate[Name]` Method Of An Entity Bean (High)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_026](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JEC_026)]

Enterprise entity beans must declare `ejbCreate` methods to return the bean's primary key type. (See EJB20SPEC - 10.6.4 and 12.2.3.)



### Caution

CMP Entity beans should return `null` as their primary key value (See EJB20SPEC - 10.6.2.)

WRONG

```
private void ejbCreate(String fooId, String bar) throws CreateException {  
    // ...  
}
```

RIGHT

```
public String ejbCreate(String fooId, String bar) throws CreateException {  
    // ...  
  
    return fooId; // java.lang.String is the primary key type  
}
```

## JEC\_027: Do Not Declare An `ejbCreate` Or `ejbPostCreate` Method As `final` Or `static` For An Entity Bean (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_027](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEC_027)]

The `ejbCreate` and `ejbPostCreate` methods defined in an enterprise bean must not be declared `final` or `static`. (See EJB20SPEC - 7.10.3, 10.6.4, 10.6.5, 12.2.3, 12.2.4 and 15.7.3.)

WRONG

```
public static String ejbCreate(String fooId, String bar) throws CreateException {
    // ...
}

public final void ejbPostCreate(String fooId, String bar) throws CreateException {
    // ...
}
```

RIGHT

```
public String ejbCreate(String fooId, String bar) throws CreateException {
    // ...
}

public void ejbPostCreate(String fooId, String bar) throws CreateException {
    // ...
}
```

## JEC\_028: Return `void` For An `ejbPostCreate` Method Of An Entity Bean (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_028](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEC_028)]

Enterprise entity beans must declare `ejbPostCreate` methods that return `void`. (See EJB20SPEC - 10.6.5 and 12.2.4.)

WRONG

```
public String ejbPostCreate(String fooId, String bar) throws CreateException {
    // ...
}
```

RIGHT

```
public void ejbPostCreate(String fooId, String bar) throws CreateException {
    // ...
}
```

## JEC\_029: Declare An `ejbFindByPrimaryKey` Method For A BMP EJB (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_029](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEC_029)]

Enterprise entity beans with Bean Managed Persistence must implement the `ejbFindByPrima-`

ryKey method. (See EJB20SPEC - 12.2.5.)

RIGHT

```
import javax.ejb.ObjectNotFoundException;
// ...
public String ejbFindByPrimaryKey(String pk) throws ObjectNotFoundException {
    // ...
    return pk;
}
// ...
```

## JEC\_030: Declare A ejbFind Method public For A BMP EJB (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_030](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEC_030)]

Enterprise entity beans with Bean Managed Persistence must implement finder methods (ejbFindXXX) with public access. (See EJB20SPEC - 12.2.5.)

WRONG

```
Collection ejbFindBigFoos(double greaterThan) {
    // ...
    return collection;
}
```

RIGHT

```
public Collection ejbFindBigFoos(double greaterThan) {
    // ...
    return collection;
}
```

## JEC\_031: Do Not Declare A ejbFind Method final Or static (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_031](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEC_031)]

Finder methods (ejbFindXXX) defined in an enterprise entity bean with Bean Managed Persistence must not be declared as final or static. (See EJB20SPEC - 12.2.5.)

WRONG

```
public static String ejbFindByPrimaryKey(String pk) throws ObjectNotFoundException {
    // ...
    return pk;
}

public final Collection ejbFindBigFoos(double greaterThan) {
    // ...
    return collection;
}
```

RIGHT

```
public String ejbFindByPrimaryKey(String pk) throws ObjectNotFoundException {
```

```
        // ...
        return pk;
    }

    public Collection ejbFindBigFoos(double greaterThan) {
        // ...
        return collection;
    }
}
```

## JEC\_032: Make A find Method Of An EJB Home Interface throw `javax.ejb.FinderException` (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_032](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEC_032)]

Finder methods (findXXX) defined in a home interface (remote or local) must include `javax.ejb.FinderException` in the throws clause. (See EJB20SPEC - 10.6.10, 10.6.12, 12.2.9 and 12.2.11.)

WRONG

```
public interface FooHome extends EJBHome {
    // ...

    public Foo findByPrimaryKey(String pk) throws RemoteException;

    // ...
}
```

RIGHT

```
public interface FooHome extends EJBHome {
    // ...

    public Foo findByPrimaryKey(String pk) throws FinderException, RemoteException;

    // ...
}
```

## JEC\_033: Declare An `ejbSelect` Method Of An Entity Bean `public` (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_033](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEC_033)]

The `ejbSelectXXX` methods defined in an enterprise entity bean must be declared as `public` and `abstract`. (See EJB20SPEC - 10.6.7.)

WRONG

```
import javax.ejb.FinderException;
// ...
private Set ejbSelectAfterDateFoo(Date givenDate) throws FinderException;
// ...
```

RIGHT

```
import javax.ejb.FinderException;
// ...
```

```
    public abstract Set ejbSelectAfterDateFoo(Date givenDate) throws FinderException;
// ...
```

## JEC\_034: Make An `ejbSelect` Method Of An Entity Bean throw `javax.ejb.FinderException` (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_034](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEC_034)]

Select methods (`ejbSelectXXX`) defined in an enterprise entity bean must include `javax.ejb.FinderException` in the `throws` clause. (See EJB20SPEC - 10.6.7.)

WRONG

```
// ...
    private Set ejbSelectAfterDateFoo(Date givenDate);
// ...
```

RIGHT

```
import javax.ejb.FinderException;
// ...
    public abstract Set ejbSelectAfterDateFoo(Date givenDate) throws FinderException;
// ...
```

## JEC\_035: Return The EJB Local Interface For A `create` Method In An EJB Local Home Interface (Enforced)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_035](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEC_035)]

The `create` methods declared in a local home interface must return a local interface type (a type that implements *ejbLocalObject*). (See EJB20SPEC - 10.6.12 and 12.2.11.)

## JEC\_036: Declae An `ejbPostCreate` Method Of An Entity Bean `public` (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_036](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEC_036)]

Enterprise entity beans must declare `ejbPostCreate` methods with `public` access. (See EJB20SPEC - 10.6.5 and 12.2.4.)

WRONG

```
private void ejbPostCreate(String fooId, String bar) throws CreateException {
    // ...
}
```

RIGHT

```
public void ejbPostCreate(String fooId, String bar) throws CreateException {
    // ...
}
```

## JEC\_037: Make A Method Of An EJB Remote Interface

## Throw `java.rmi.RemoteException` (Enforced)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_037](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JEC_037)]

All methods declared in a remote interface type must include `java.rmi.RemoteException` in the throws clause. (See EJB20SPEC - 7.10.5, 10.6.9 and 12.2.8.)

### WRONG

```
import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface Foo extends EJBObject {

    void doSomething();

}
```

### RIGHT

```
import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface Foo extends EJBObject {

    void doSomething() throws RemoteException;

}
```

## JEC\_038: Make A Method Of An EJB Remote Home Interface Throw `java.rmi.RemoteException` (Enforced)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_038](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JEC_038)]

All methods declared in a remote home interface type must include `java.rmi.RemoteException` in the throws clause. (See EJB20SPEC - 7.10.6, 10.6.10 and 12.2.9.)

### WRONG

```
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
// ...
public interface FooHome extends EJBHome {

    Foo create(String fooId, String bar) throws CreateException;

}
```

### RIGHT

```
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
// ...
public interface FooHome extends EJBHome {

    Foo create(String fooId, String bar)
        throws CreateException, RemoteException;

}
```

## JEC\_039: Do Not Make A Method Of An EJB Local Inter-

## face Throw `java.rmi.RemoteException` (Enforced)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_039](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JEC_039)]

No method declared in a local bean interface type should include `java.rmi.RemoteException` in the throws clause. (See EJB20SPEC - 7.10.7 and 12.2.1)

WRONG

```
import javax.ejb.CreateException;
import javax.ejb.EJBLocalHome;
// ...
public interface FooHome extends EJBLocalHome {

    Foo create(String fooId, String bar) throws CreateException;

}
```

RIGHT

```
import javax.ejb.CreateException;
import javax.ejb.EJBLocalHome;
// ...
public interface FooHome extends EJBLocalHome {

    Foo create(String fooId, String bar)
        throws CreateException, RemoteException;

}
```

## JEC\_040: Do Not Make A Method Of An EJB Local Home Interface Throw `java.rmi.RemoteException` (Enforced)

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_040](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JEC_040)]

No method declared in a local home interface type should include `java.rmi.RemoteException` in the throws clause. (See EJB20SPEC - 7.10.8 and 12.2.11.)

WRONG

```
import javax.ejb.EJBLocalObject;
import javax.ejb.RemoteException;
// ...
public interface Foo extends EJBLocalObject {

    public String getFooId() throws RemoteException;

}
```

RIGHT

```
import javax.ejb.EJBLocalObject;
// ...
public interface Foo extends EJBLocalObject {

    public String getFooId();

}
```

## JEC\_042: Declare An `ejbselect` Method Of A CMP EJB



## abstract (High)

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JEC\_042]

Enterprise entity beans with container-managed persistence must declare each `ejbSelectXXX` method as `abstract` as they will be implemented by the EJB container. (See EJB20SPEC - 10.6.2.)

WRONG

```
import javax.ejb.FinderException;
// ...
public Set ejbSelectAfterDateFoo(Date givenDate) throws FinderException;
// ...
```

RIGHT

```
import javax.ejb.FinderException;
// ...
public abstract Set ejbSelectAfterDateFoo(Date givenDate) throws FinderException;
// ...
```



### Note

See also rule JEC\_033: Declare An `ejbSelect` Method Of An Entity Bean `public` (High).

## JEC\_043: Make An `ejbCreate` Method Of An Entity Bean Throw `javax.ejb.CreateException` (High)

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JEC\_043]

The `ejbCreate` methods defined in an enterprise entity beans must include `javax.ejb.CreateException` in the `throws` clause. (See EJB20SPEC - 10.6.4.)

WRONG

```
public String ejbCreate(String fooId, String bar) {
    // ...
}
```

RIGHT

```
public String ejbCreate(String fooId, String bar) throws CreateException {
    // ...
}
```

## JEC\_044: Declare An `ejbHome` Method Of An Entity Bean `public` (High)

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JEC\_044]

Entity beans must declare home methods (`ejbHomeXXX`) with `public` access. (See EJB20SPEC - 10.6.6 and 12.2.6.)

## JEC\_045: Do Not Declare An `ejbHome` Method Of An En-

## Entity Bean final Or static (High)

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JEC\_045]

Home methods (ejbHomeXXX) declared in an enterprise entity bean must not be declared static. (See EJB20SPEC - 10.6.6 and 12.2.6.)

## JEC\_046: Do Not Make An ejbHome Method Of An Entity Bean Throw java.rmi.RemoteException (High)

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JEC\_047]

Home methods (ejbHomeXXX) declared in an enterprise entity bean must not include java.rmi.RemoteException in the throws clause. (See EJB20SPEC - 10.6.6 and 12.2.6.)

## JEC\_047: Make A Message Bean Implementation Implement javax.jms.MessageListener (High)

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JEC\_047]

Enterprise beans that implement javax.ejb.MessageDrivenBean must also, directly or indirectly, implement the javax.jms.MessageListener interface. (See EJB20SPEC - 15.7.2.)

WRONG

```
// ...
import javax.ejb.MessageDrivenBean;

public class FooMDB implements MessageDrivenBean {
    // ...
}
```

RIGHT

```
// ...
import javax.ejb.MessageDrivenBean;
import javax.jms.MessageListener;

public class FooMDB
    implements MessageDrivenBean
        MessageListener {
    // ...
}
```

## JEC\_052: Provide An ejbCreate Method For A Message Bean (High)

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JEC\_052]

Enterprise message beans must define one ejbCreate method. (See EJB20SPEC - 15.7.2.)

WRONG

```
public class FooBean extends AbstractMessageBean
    implements MessageListener {

    public FooBean() {
    }
}
```

```
        public void onMessage(Message msg) {  
        }  
    }
```

**RIGHT**

```
public class FooBean extends AbstractMessageBean  
    implements MessageListener {  
  
    public FooBean() {  
    }  
  
    public void ejbCreate() {  
    }  
  
    public void onMessage(Message msg) {  
    }  
}
```

## **JEC\_053: Return void For An ejbCreate Method Of A Message Bean (High)**

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_053](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEC_053)]

The ejbCreate method in an enterprise message bean must be declared to return void. (See EJB20SPEC - 15.7.3.)

**WRONG**

```
public class FooBean extends AbstractMessageBean  
    implements MessageListener {  
  
    public FooBean() {  
    }  
  
    public int ejbCreate() {  
    }  
  
    public void onMessage(Message msg) {  
    }  
}
```

**RIGHT**

```
public class FooBean extends AbstractMessageBean  
    implements MessageListener {  
  
    public FooBean() {  
    }  
  
    public void ejbCreate() {  
    }  
  
    public void onMessage(Message msg) {  
    }  
}
```

## JEC\_054: Do Not Declare Arguments For An ejbCreate Method Of A Message Bean (High)

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JEC\_054]

The ejbCreate method in an enterprise message bean must not declare any arguments. (See EJB20SPEC - 15.7.3.)

WRONG

```
public class FooBean extends AbstractMessageBean
    implements MessageListener {

    public FooBean() {

    }

    public void ejbCreate(String value) {

    }

    public void onMessage(Message msg) {

    }
}
```

RIGHT

```
public class FooBean extends AbstractMessageBean
    implements MessageListener {

    public FooBean() {

    }

    public void ejbCreate() {

    }

    public void onMessage(Message msg) {

    }
}
```

## JEC\_055: Provide A Valid RMI Method Signature For An EJB Remote Home Interface (High)

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JEC\_055]

*The methods defined in the Home interface must follow the rules for RMI/IIOP. This means that their argument and return values must be of valid types for RMI/IIOP, and that their throws clauses must include the java.rmi.RemoteException. (See EJB20SPEC - 7.10.6.)*

## JEC\_056: Do No Print Unnecessary Static Content From A Servlet (High)

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JEC\_056]

Following the Model-View-Controller design pattern, the Servlet should be used as a (Front) Controller and dispatches the Model (in J2EE this could be one or more Transfer Objects) to the Java ServerPages. The JSP handles all View functionality and might get some assistance from some Helper classes. This web design is also known as the model 2 (See MODEL2 ).

WRONG

```
public class FooServlet extends HttpServlet {
    // ...
    public void doGet(HttpServletRequest req,
                      HttpServletResponse res) throws IOException {

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("Hello World");
        out.println("</body>");
        out.println("</html>");
        out.close();
    }
}
```

#### RIGHT

```
public class FooServlet extends HttpServlet {
    // ...
    public void service(HttpServletRequest req,
                      HttpServletResponse res) {
        processRequest(req, res);
    }

    // ...
    protected void processRequest(HttpServletRequest req,
                                HttpServletResponse res) throws IOException {

        Command cmd = null;
        String page = null;
        String param = null;

        param = req.getParameter(CMD);

        if (null != param) {
            cmd = CommandFactory.createCommand(param);
            page = command.execute(req, res);
        } else {
            page = ERROR_PAGE;
        }

        dispatch(req, res, page);
    }

    // ...
    protected void dispatch(HttpServletRequest req,
                          HttpServletResponse res, String page) throws IOException {

        ServletContext ctx = getServletContext();
        RequestDispatcher dispatcher = ctx.getRequestDispatcher(page);

        dispatcher.forward(req, res);
    }
}
```

helloWorld.jsp

```
<html>
  <body>
    HelloWorld
```

```
        </body>
    </html>
```

## JEC\_057: Use Custom JSP Tags Instead Of Scriptlets (High)

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JEC\\_057](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JEC_057)]

Avoid the usage of scriptlets in JavaServer Pages because:

- Scriptlet code is not reusable.
- Scriptlets encourage copy and paste coding.
- Scriptlets combines Controller with View code. See also rule JEC\_056: Do Not Print Unnecessary Static Content From A Servlet (High).
- Scriptlets make JSP pages difficult to read and maintain
- Debugging scriptlet code is not straight forward.
- Scriptlet code is also not easy to test.

### WRONG

```
<html>
    <body>
        <table>
            <% for (int i=0; i < employees.length; i++) { %>
                out.println("<TD>" + employees.getName(i) + "</TD>");
                out.println("<TD>" + employees.getProfile(i) + "</TD>");
            <% } %>
        </table>
    </body>
</html>
```

### RIGHT

```
<%@ taglib uri="/WEB-INF/jcs.tld" prefix="jcs" %>

<html>
    <body>
        <table>
            <jcs:employeelist id="employee_list">
                <tr>
                    <td><jcs:employee attribute="name"/></td>
                    <td><jcs:employee attribute="profile"/></td>
                </tr>
            </jcs:employeelist>
        </table>
    </body>
</html>
```

## JEC\_058: Do Not Forward A Request From A JSP Page (High)

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JEC\_058]

A JSP page should not implement the Command and Controller strategy, this is the responsibility of the Front Controller Servlet.

WRONG

```
<%
String param = request.getParameter("cmd");

if (param.equals("helloWorld")) { %>
    <jsp:forward page="/helloWorld.jsp"/>
<%
} else if (param.equals("Goodbye")) { %>
    <jsp:forward page="/goodbye.jsp"/>
<% } %>
```

RIGHT - See example in rule JEC\_056: Do Not Print Unnecessary Static Content From A Servlet (High).

## Exceptions Conventions Rules

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]exceptionsRules]

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.

## Overview

The following table provides an overview of the exception related conventions:

**Table A.6. Exception Overview**

Rules
JAC_039: Do Not Catch java.lang.Exception Or java.lang.Throwable (Normal)
JAC_055: Provide At Least One Statement In A try Block (Enforced)
JAC_056: Provide At Least One Statement In A finally Block (Enforced)
JAC_061: Do Not Return From Inside A try Block (High)
JAC_062: Do Not Return From Inside A finally Block (High)
JAC_063: Do Not Use A try Block Inside A Loop When Possible (Normal)
JAC_064: Do Not Use An Exception For Control Flow (High)
JAC_068: Close A Connection Inside A finally Block (Enforced)
JAC_069: Do Not Declare A Method That Throws java.lang.Exception or java.lang.Throwable (Normal)
JAC_070: Do Not Use An instanceof Statement To Check An Exception (High)
JAC_071: Do Not Catch A RuntimeException (High)
JAC_072: Do Not Use printStackTrace (High)
JEC_012: Do Not Throw A RemoteException From An EJB Implementation Method (High)
JEC_020: Make A create Method Of An EJB Home Interface Throw An javax.ejb.CreateException (Enforced)
JEC_032: Make A find Method Of An EJB Home Interface throw javax.ejb.FinderException (High)

Rules
JEC_034: Make An ejbSelect Method Of An Entity Bean throw javax.ejb.FinderException (High)
JEC_037: Make A Method Of An EJB Remote Interface Throw java.rmi.RemoteException (Enforced)
JEC_038: Make A Method Of An EJB Remote Home Interface Throw java.rmi.RemoteException (Enforced)
JEC_039: Do Not Make A Method Of An EJB Local Interface Throw java.rmi.RemoteException (Enforced)
JEC_040: Do Not Make A Method Of An EJB Local Home Interface Throw java.rmi.RemoteException (Enforced)
JEC_043: Make An ejbCreate Method Of An Entity Bean Throw javax.ejb.CreateException (High)
JEC_046: Do Not Make An ejbHome Method Of An Entity Bean Throw java.rmi.RemoteException (High)



---

# Appendix B. Guidelines Rules

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]guidelinesRules]

## Introduction

This appendix provides an overview of the various guidelines rules.

Each guidelines rule is a recommendation and therefore has no priority.

Below you can find an overview of the different guidelines rules.

## Ant Rules

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]antRules]

## Overview

**Table B.1. Ant Overview**

Rules
ANT_001: Use the default build.xml build filename
ANT_002: Do Not Hard Code An Absolute Directory Or File Path
ANT_003: Suffix The Name Of A Property That Represents A Directory With .dir
ANT_004: Use Package Like Names To Namespace A Property Or Target
ANT_005: Depend Only On Direct Dependencies
ANT_006: Make Each Target Directly Runnable
ANT_007: Use The location Attribute For A File Or Directory Based Property
ANT_008: Use The zipfileset Task To Create Archives
ANT_009: Keep Your Build File Platform Independent When Possible
ANT_010: Set A Needed Environment Variable In The Build Script

### ANT\_001: Use the default build.xml build filename

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]ANT\_001]

Use the default build.xml filename for your Ant build files. See it as a Ant build file naming convention.

### ANT\_002: Do Not Hard Code An Absolute Directory Or File Path

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]ANT\_002]

Do not hard code a directory or file path which starts from the root of the filesystem. This includes `c :`, `d :`, etc on windows and `/` on unix. There are 2 well know techniques to comply to this rule:

- Code relative to the base directory by using Ants build-in variable `${basedir}` or a property

task's `location` attribute. This is almost always appropriate for files or directories directly or indirectly under the base directory.

- Place the absolute paths in a separate global properties file which can be overwritten by a local properties file. See the section called “Reusing The Build File Across Environments”. This is almost always appropriate for files or directories not under the base directory.

#### WRONG

```
<property name="tomcat.home" value="c:\tomcat"/>

<property name="build.dir" location="c:\MyProjects\MyProject\build"/>

<fileset dir="c:\MyProjects\MyProject\docs">
  <!-- ... -->
</fileset>
```

#### RIGHT

```
<property file="${basedir}/localProject.properties"/>
<property file="${basedir}/project.properties"/>
<!-- if tomcat.home is defined in localProject.properties, -->
<!-- it is ignored in project.properties -->

<property name="build.dir" location="build"/>

<fileset dir="${basedir}/docs">
  <!-- ... -->
</fileset>
```

## ANT\_003: Suffix The Name Of A Property That Represents A Directory With `.dir`

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]ANT\\_003](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]ANT_003)]

End the name of a property that points to a directory with `.dir`.

#### WRONG

```
<property name="dist" location="dist"/>
<property name="dist.ear" location="${dist}/ear"/>

<target name="dist"
  description="Builds distribution">
  <!-- ... -->
</target>
```

#### RIGHT

```
<property name="dist.dir" location="dist"/>
<property name="dist.ear.dir" location="${dist.dir}/ear"/>

<target name="dist"
  description="Builds distribution">
  <!-- ... -->
</target>
```

## ANT\_004: Use Package Like Names To Namespace A Property Or Target

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]ANT\_004]

Use the dots/lowercase notation to namespace properties and targets. Environment properties can only be named spaced with dots.

### WRONG

```
<property environment="env"/>
  <!-- Environment properties will be env.path, etc-->
<property name="distDir" location="dist"/>
<property name="distJarDir" location="${dist.dir}/jar"/>
<property name="distSrcDir" location="${dist.dir}/src"/>
<property name="version" value="1_0_0"/>

<target name="distJar"
  description="Builds the binary distribution jar">
  <!-- ... -->
</target>
<target name="distSrc"
  description="Builds the source distribution zip">
  <!-- ... -->
</target>
```

### RIGHT

```
<property environment="env"/>
<property name="dist.dir" location="dist"/>
<property name="dist.jar.dir" location="${dist.dir}/jar"/>
<property name="dist.src.dir" location="${dist.dir}/src"/>
<property name="project.version" value="1_0_0"/>

<target name="dist.jar"
  description="Builds the binary distribution jar">
  <!-- ... -->
</target>
<target name="dist.src"
  description="Builds the source distribution zip">
  <!-- ... -->
</target>
```

## ANT\_005: Depend Only On Direct Dependencies

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]ANT\_005]

Do not make a target depend on targets which it does not depend directly on, even if one or several targets it depends on directly do depend on it directly. Instead make those targets depend on it.



### Tip

If several targets which will be run depend on the same target, it will only be run once: before the first target that depends on it is run.

### WRONG

```
<target name="compile"
        description="Compiles the source files into class files">
    <!-- ... -->
</target>

<target name="war" depends="compile"
        description="Makes a war file with the class files">
    <!-- ... -->
</target>

<target name="ear" depends="compile, war"
        description="Makes a ear file with the war file">
    <!-- ... -->
</target>
```

#### RIGHT

```
<target name="compile"
        description="Compiles the source files into class files">
    <!-- ... -->
</target>

<target name="war" depends="compile"
        description="Makes a war file with the class files">
    <!-- Can now be run -->
    <!-- ... -->
</target>

<target name="ear" depends="war"
        description="Makes a ear file with the war file">
    <!-- Does not directly depends on the compile target -->
    <!-- ... -->
</target>
```



#### Note

However if the buildEar target directly uses one of the classes build with the build-Classes target, then it must depend on it.

## ANT\_006: Make Each Target Directly Runnable

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]ANT\\_006](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]ANT_006)]

Do not create a target that does not *depend* on another target but does need the others target's build output in order to work. Instead make it *depend* on the target which creates build output which it needs.

#### WRONG

```
<target name="compile">
    <!-- ... -->
</target>

<target name="war">
    <!-- Depends on compile. -->
    <!-- Can only be run as part of the ear target -->
    <!-- ... -->
</target>

<target name="ejb">
```

```
    <!-- Depends on compile. -->
    <!-- Can only be run as part of the ear target -->
    <!-- ... -->
</target>

<target name="ear" depends="compile, war, ejb">
    <!-- Does not directly depend on buildClasses. -->
    <!-- ... -->
</target>
```

#### RIGHT

```
<target name="compile">
    <!-- ... -->
</target>

<target name="war" depends="compile">
    <!-- ... -->
</target>

<target name="ejb" depends="compile">
    <!-- ... -->
</target>

<target name="ear" depends="war, ejb">
    <!-- ... -->
</target>
```



#### Tip

If a project build becomes too long, one might be inclined to provide *expert* targets: targets that only rebuild part of the project and rely on parts still being available without depending on the targets that generate those other parts.

Instead of using *expert* targets, it is a better practice to divide your project in modules (with their own `build.xml` files) and build short term releases of modules to be used by other modules.

## ANT\_007: Use The `location` Attribute For A File Or Directory Based Property

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]ANT\\_007](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]ANT_007)]

A property representing a single file or directory should be referenced through the `location` attribute instead of the commonly used `value` attribute.

#### WRONG

```
<property name="src.java.dir" value="${basedir}/src/java"/>
```

#### RIGHT

```
<property name="src.java.dir" location="src/java"/>
```

**Note**

If the `location` attribute is a non absolute path it is taken relative to the project's `basedir`.

## ANT\_008: Use The `zipfileset` Task To Create Archives

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]ANT\\_008](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]ANT_008)]

An archive file such as a WAR or EAR file needs to follow a certain directory structure. Do not create a temporary build directory to order all files according to the required directory structure, prior to archiving. Instead use the `zipfileset` task to copy the files directly into the correct path in the archive. This will avoid any copy overhead and make the build script gain noticable performance.

**WRONG**

```
<mkdir dir="${build.dir}/ear"/>
<mkdir dir="${build.dir}/ear/lib"/>
<copy todir="${build.dir}/ear/lib">
  <fileset dir="${basedir}/lib/log"/>
  <fileset dir="${basedir}/lib/xml"/>
</copy>
<mkdir dir="${build.dir}/ear/META-INF"/>
<copy todir="${build.dir}/ear/META-INF">
  <fileset dir="${conf.dir}/ear">
    <include name="weblogic-application.xml"/>
  </fileset>
</copy>
<!-- ... -->
<ear earfile="${dist.dir}/ear/${ant.project.name}.ear"
  appxml="${conf.dir}/ear/application.xml">
  <fileset dir="${build.dir}/ear"/>
</ear>
```

**RIGHT**

```
<ear earfile="${dist.dir}/ear/${ant.project.name}.ear"
  appxml="${conf.dir}/ear/application.xml">
  <zipfileset dir="${basedir}/lib/log" prefix="lib"/>
  <zipfileset dir="${basedir}/lib/xml" prefix="lib"/>
  <zipfileset dir="${conf.dir}/ear" prefix="META-INF">
    <include name="weblogic-application.xml"/>
  </zipfileset>
  <!-- ... -->
</ear>
```

## ANT\_009: Keep Your Build File Platform Independent When Possible

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]ANT\\_009](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]ANT_009)]

Avoid using platform dependent tasks such as:

- `exec` (executes a system command)
- `chmod` (works only on unix)



### Tip

Check the Ant documentation for a task to verify that its platform independent. Very few Ant tasks are platform dependent.



### Note

The `exec` target can specify the operation system(s) on which it should be run with the `os` attribute. This allows the build script to be run on multiple platforms, but doesn't make it platform independent.

#### WRONG

```
<exec executable="copy">
  <arg line="fileSrc.txt fileDest.txt"/>
</exec>
```

#### RIGHT

```
<copy file="fileSrc.txt" tofile="fileDest.txt"/>
```

## ANT\_010: Set A Needed Environment Variable In The Build Script

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]ANT\\_010](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]ANT_010)]

If a build script uses a platform depend program, an environment variable might need to be set. Do not depend on the local OS to be configured to contain it, instead define it in the build script, possibly based on a property fetched from the local properties file.

#### WRONG

```
<echo message="The environment variable SGML_CATALOG_FILES must be set"
      level="info"/>
<exec dir="${src.dir}/xml" executable="xmllint.exe">
  <arg value="--catalogs"/>
  <!-- ... -->
</exec>
```

#### RIGHT

```
<exec dir="${src.dir}/xml" executable="xmllint.exe">
  <env key="SGML_CATALOG_FILES"
      path="${basedir}/conf/dtd/catalog.xml"/>
  <arg value="--catalogs"/>
  <!-- ... -->
</exec>
```

## Logging Rules

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]loggingRules](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]loggingRules)]

### Overview

**Table B.2. Logging Overview**

Rules
LOG_001: Retrieve A Logger Based On The Fully Qualified Package And Class Name
LOG_002: Declare A Logger Instance <i>private final static</i>
LOG_003: Log A Caught Exception
LOG_004: Use A Correct FileHandler Log Name
LOG_005: Use The Log Level SEVERE Only For Non Recoverable Problems
LOG_006: Use The Log Level WARNING Only For Recoverable Problems
LOG_007: Use The Log Level INFO Only For Information Logs
LOG_008: Use The Log Level CONFIG Only For Configuration Problems

## LOG\_001: Retrieve A Logger Based On The Fully Qualified Package And Class Name

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]LOG\\_001](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]LOG_001)]

When retrieving a Logger the name should be the fully qualified package and class name. The easiest way to retrieve this information is by calling the `getName` method on the `class` reference.

WRONG

```
package be.vlaanderen.myproject.mypackage;

import java.util.logging.Logger;

public class Foo {
    private static final Logger logger =
        Logger.getLogger("example");

    // ...
}
```

RIGHT

```
package be.vlaanderen.myproject.mypackage;

import java.util.logging.Logger;

public class Foo {
    private static final Logger logger =
        Logger.getLogger(Foo.class.getName());

    // ...
}
```

## LOG\_002: Declare A Logger Instance *private final static*

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]LOG\\_002](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]LOG_002)]

A Logger instance should be declared *private final static*.



**WRONG**

```
package be.vlaanderen.myproject.mypackage;

import java.util.logging.Logger;

public class Foo {
    public Logger logger =
        Logger.getLogger(Foo.class.getName());

    // ...
}
```

**RIGHT**

```
package be.vlaanderen.myproject.mypackage;

import java.util.logging.Logger;

public class Foo {
    private static final Logger logger =
        Logger.getLogger(Foo.class.getName());

    // ...
}
```

## LOG\_003: Log A Caught Exception

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]LOG\\_003](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]LOG_003)]

A caught exception must be reported using the logging API.

**WRONG**

```
public class Foo {

    public void bar(Message msg) {
        String txtMessage = null;

        try {
            txtMessage = ((TextMessage) msg).getText();

            helperMethod(txtMessage);
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }
}
```

**RIGHT**

```
public class Foo {

    public void bar(Message msg) {
        String txtMessage = null;

        try {
            txtMessage = ((TextMessage) msg).getText();

            helperMethod(txtMessage);
        }
    }
}
```

```
        } catch (JMSEException e) {
            logger.log(Level.WARNING, "JMS Problem", e);
        }
    }
}
```

## LOG\_004: Use A Correct FileHandler Log Name

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]LOG\\_004](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]LOG_004)]

The naming convention for the log file, when using the logging `FileHandler` class, should be `%h/project_name%u.log`. `%h` is the user `.home` system property and `%u` is a unique rotating number when the limit size of the log file is reached.

### RIGHT

```
/** Initialize the logging API here and use it for any std. error output
 */
private static final Logger logger =
    Logger.getLogger(Foo.class.getName());

static {
    FileHandler fileHandler = null;
    try {
        fileHandler = new FileHandler("%h" +
            File.separator +
            "guidelines%u.log");
    } catch (SecurityException e) {
        logger.log(Level.SEVERE, "Security Problem", e);
    } catch (IOException e) {
        logger.log(Level.CONFIG, "IO Problem", e);
    }
    logger.addHandler(fileHandler);
}

public static void main(String[] args) {
    logger.info("info log message");
}
```

The above example will create a `guidelines0.log` file in the user home directory with following data:

```
<?xml version="1.0" encoding="windows-1252" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
  <record>
    <date>2003-10-22T11:48:37</date>
    <millis>1066816117500</millis>
    <sequence>0</sequence>
    <logger>be.vlaanderen.examples.Foo</logger>
    <level>INFO</level>
    <class>be.vlaanderen.examples.Foo</class>
    <method>main</method>
    <thread>10</thread>
    <message>info log message</message>
  </record>
</log>
```

## LOG\_005: Use The Log Level SEVERE Only For Non Re-

## coverable Problems

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]LOG\_005]

The log level `SEVERE` should be used for non-recoverable problems or runtime exceptions.

WRONG

```
try {  
    // ...  
} catch (SecurityException e) {  
    logger.log(Level.WARNING, "Security Problem", e);  
}
```

RIGHT

```
try {  
    // ...  
} catch (SecurityException e) {  
    logger.log(Level.SEVERE, "Security Problem", e);  
}
```

## LOG\_006: Use The Log Level `WARNING` Only For Recoverable Problems

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]LOG\_006]

The log level `WARNING` should be used for recoverable problems or checked exceptions.

WRONG

```
try {  
    // ...  
} catch (JMSEException e) {  
    logger.info("JMS Problem" + e);  
}
```

RIGHT

```
try {  
    // ...  
} catch (JMSEException e) {  
    logger.log(Level.WARNING, "JMS Problem", e);  
}
```

## LOG\_007: Use The Log Level `INFO` Only For Information Logs

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]LOG\_007]

The log level `INFO` should be used for informational logs.

RIGHT

```
try {  
    logger.info("got here");  
} catch (JMSEException e) {  
    logger.log(Level.WARNING, "JMS Problem", e);  
}
```

```
}
```

## LOG\_008: Use The Log Level CONFIG Only For Configuration Problems

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]LOG\\_008](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]LOG_008)]

The log level CONFIG should be used for configuration messages or related problems.

### WRONG

```
static {
    FileHandler fileHandler = null;
    try {
        fileHandler = new FileHandler("%h" +
            File.separator +
            "guidelines%u.log");
    } catch (SecurityException e) {
        logger.log(Level.SEVERE, "Security Problem", e);
    } catch (IOException e) {
        logger.log(Level.INFO, "IO Problem", e);
    }
    logger.addHandler(fileHandler);
}
```

### RIGHT

```
static {
    FileHandler fileHandler = null;
    try {
        fileHandler = new FileHandler("%h" +
            File.separator +
            "guidelines%u.log");
    } catch (SecurityException e) {
        logger.log(Level.SEVERE, "Security Problem", e);
    } catch (IOException e) {
        logger.log(Level.CONFIG, "IO Problem", e);
    }
    logger.addHandler(fileHandler);
}
```

## Database Independency Rules

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]jdbcRules](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]jdbcRules)]

### Overview

**Table B.3. Database Independency Overview**

Rules
JDBC_001: Do Not Hard Code A JDBC Driver Class Name
JDBC_002: Do Not Hard Code A JDBC Connection URL
JDBC_003: Do Not Use A Driver Managment Method Of <code>java.sql.DriverManager</code>
JDBC_004: Do Not Import A JDBC Vendor Specific Class

Rules
JDBC_005: Close Connection, Statement And ResultSet
JDBC_006: Close JDBC Resources In The Correct Order
JDBC_007: Use PreparedStatement Instead Of Statement When Possible
JDBC_008: Check The Return value Of A Navigation Method Of ResultSet
JDBC_009: Use Standard SQL Only
JDBC_010: Check For A Nested SQLException
JDBC_011: Check For A SQLWarning

## JDBC\_001: Do Not Hard Code A JDBC Driver Class Name

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JDBC\_001]

Don't hard code the database driver class name when loading it using the `Class.forName` method. It is better store the driver class name in an external configuration file that is read when the database connection must be created.

## JDBC\_002: Do Not Hard Code A JDBC Connection URL

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JDBC\_002]

Don't hard code the database connection URL. It is better to store this URL in an external configuration file that is read when the database connection must be created.

If possible try not include the username and password in the URL but store them separately. This allows you to encrypt your password and to reuse the same username password for different URLs (in the case where you have to support more than one database).

## JDBC\_003: Do Not Use A Driver Management Method Of `java.sql.DriverManager`

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JDBC\_003]

The `java.sql.DriverManager` is responsible for managing the known JDBC driver classes. Although the `DriverManager` has some method related to the management of drivers (the `register` and `deregister` methods) they should never be called directly.

The JDBC API states that all driver implementation should have a `static` initializer that calls the `DriverManager.register` method when the driver class is loaded. This means that when a driver is loaded there is no need to register it with the `DriverManager`.

Calling `DriverManager.deregister` is even more dangerous since this can seriously disrupt the `DriverManager`'s internal bookkeeping.

## JDBC\_004: Do Not Import A JDBC Vendor Specific Class

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JDBC\_004]

If your code needs to be database independent it is important that you do not import classes that are specific to one JDBC driver vendor. Using these imports means that you probably use database specific features that will prevent you from moving to another type of database without changing your code.

Stick to the classes in the `java.sql` and `javax.sql` packages.

## JDBC\_005: Close Connection, Statement And ResultSet

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JDBC\\_005](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JDBC_005)]

All JDBC resources (Connection, Statement, ResultSet) must be closed when they are no longer used. If you don't do this you can introduce memory leaks and possibly disrupt the functionality of the database. For example you can have too many open connections on your database, too many open cursors in your database, etc.

Always close your resources in a finally clause of a try block so you know that they always get closed. See rule JAC\_068: Close A Connection Inside A finally Block (Enforced).

### WRONG

```
public String[] getPersons(DataSource ds) throws SQLException {
    Connection conn = datasource.getConnection();
    Statement stat = conn.createStatement();
    ResultSet rs = stat.executeQuery("Select * from Person");
    while (rs.hasNext()) {
        // ...
    }
    rs.close();
    stat.close();
    conn.close();
    // return something
}
```

### RIGHT

```
public String[] getPersons(DataSource ds) throws SQLException {
    Connection conn = null;
    Statement stat = null;
    ResultSet rs = null;
    try {
        conn = datasource.getConnection();
        stat = conn.createStatement();
        rs = stat.executeQuery("Select * from Person");
        while (rs.hasNext()) {
            // ...
        }
    } finally {
        if (rs != null) {
            rs.close();
        }
        if (stat != null) {
            stat.close();
        }
        if (conn != null) {
            conn.close();
        }
    }
    // return something
}
```



### Tip

The code in the finally clause can become messy if you also have to catch the SQLException which is thrown when you close a resource. One technique to get rid of the messy code in the finally block is to use provide methods that will take care of closing

these resources for you.

For example:

```
public void close(Connection conn, Statement stat,
                  ResultSet rs) throws SQLException {
    if (rs != null) {
        rs.close();
    }
    if (stat != null) {
        stat.close();
    }
    if (conn != null) {
        conn.close();
    }
}
```

This method can be made available in for example a JDBC util class. This method also guarantees that the resources are closed in the correct order (see JDBC\_006: Close JDBC Resources In The Correct Order).

## JDBC\_006: Close JDBC Resources In The Correct Order

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JDBC\\_006](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JDBC_006)]

The JDBC resources must be closed in the correct order:

1. `ResultSet`
2. `Statement`
3. `Connection`

## JDBC\_007: Use `PreparedStatement` Instead Of `Statement` When Possible

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JDBC\\_007](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JDBC_007)]

Always try to use `java.sql.PreparedStatement` instead of the standard `java.sql.Statement`. Especially when executing SQL statements accepting arguments. Besides the fact that prepared statements get precompiled there is the fact the prepared statements handle the formatting of argument values for you. For example one database could format its string using single quotes where another one may use double quotes.

Another added value is the fact that you don't have to bother about writing long, error prone and complex string concatenation code.

WRONG

```
Date date = new Date();
// Format the date using the correct pattern (Note: the pattern
// used by SimpleDateFormat is not equal to that used by SQL-99)
SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy");
String dateString = sdf.format(date);

// Build the SQL statement (using a StringBuffer is more
// efficient)
StringBuffer sql = new StringBuffer("update Person ");
```

```
sql.append("set dob=to_date('");
sql.append(dateString);
sql.append("'", 'dd-mm-yyyy') where name=");
sql.append("'");
sql.append(pName);
sql.append("'");

//execute the statement
connection.createStatement();
statement.executeUpdate(sql.toString());
```

Using a `PreparedStatement`, we don't need that much code and we delegate the formatting to the driver.

#### RIGHT

```
Date date = new Date();

//prepare the statement
statement = connection.prepareStatement(
    "update person set dob=? where name=?");

//bind the parameters
statement.setDate(1, date);
statement.setString(2, pName);

//execute the statement
statement.executeUpdate();
```

## JDBC\_008: Check The Return value Of A Navigation Method Of ResultSet

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JDBC\\_008](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JDBC_008)]

Always look at the return value after calling one of the `ResultSet` navigation methods; never assume that results will be available.

`ResultSet` navigation methods include: `next`, `first`, `last`, `previous`

#### WRONG

```
Statement stat = conn.createStatement();
ResultSet rs = Stat.executeQuery("SELECT name FROM person");
rs.next();
String firstName = rs.getString(1);
```

#### RIGHT

```
Statement stat = conn.createStatement();
ResultSet rs = Stat.executeQuery("SELECT name FROM person");
if (rs.next()) {
    String firstName = rs.getString(1);
} else {
    // error handling
}
```

## JDBC\_009: Use Standard SQL Only



Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JDBC\\_009](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JDBC_009)]

Stick to the most common SQL standard when writing SQL statements. Don't use vendor specific extensions. If you do so you will be stuck to the vendor's database.

The most common standard (at the time this document was written) is SQL-92.

## JDBC\_010: Check For A Nested SQLException

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JDBC\\_010](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JDBC_010)]

The `java.sql.SQLException` has always been chainable (even before the exception chaining introduced in JDK1.4). The exception chain can help you in understanding what went wrong when some JDBC API method threw a `SQLException`. Walking through the exception chain is done using the `SQLException.getNextException` method.

### WRONG

```
try {
    Statement stat = connection.createStatement();
    ResultSet rs = stat.executeUpdate("SELECT name FROM person");
    // process rs
} catch (SQLException e) {
    // printStackTrace used as an example
    e.printStackTrace();
}
```

### RIGHT

```
try {
    Statement stat = connection.createStatement();
    ResultSet rs = stat.executeUpdate("SELECT name FROM person");
    // process rs
} catch (SQLException e) {
    // printStackTrace used as an example
    while (e != null) {
        e.printStackTrace();
        e = e.getNextException();
    }
}
```

## JDBC\_011: Check For A SQLWarning

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]JDBC\\_011](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JDBC_011)]

The main JDBC API interfaces provide access to a possible warning (`SQLWarning`) that was generated. Besides logging the warning there is not much you can do with it.

A suggestion is to check for warnings when you close the resources. See rule JDBC\_006: Close JDBC Resources In The Correct Order. This way logging possible warnings could be added to the helper method that can be used to close the `Connection`, `Statement` and `ResultSet`.

## JMS Rules

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JJGuidelines\]jmsRules](mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]jmsRules)]

## Overview

**Table B.4. JMS Overview**

Rules
JMS_001: Do Not Use A Vendor Specific Class
JMS_002: Do Not Hard Code An Initial Context Factory
JMS_003: Do Not Hard Code A Provider URL
JMS_004: Do Not Hard Code A Connection Factory Name For JNDI Lookup
JMS_005: Do Not Hard Code A Destination Name For JNDI Lookup
JMS_006: Use JNDI Lookups To Get A Connection Factory Or Destination
JMS_007: Close A Resource When It's No Longer Needed
JMS_008: Always Close Resources In The Correct Order
JMS_009: Do Not Produce A Message In A Transaction And Rely On It To Be Consumed Before The End Of The Transaction
JMS_010: Do Not Rely On JMS To Deliver Message In The Same Order As They Were Sent
JMS_011: Start A Producer Connection After Start A Consumer
JMS_012: Use A separate Transactional Session For transactional Messages And A Non-transactional Session For Non-transactional Messages
JMS_013: Set An Optimal Message Time To Live
JMS_014: Choose A Correct Message Type
JMS_015: Do Not Receive Messages Asynchronously In A Web Component, A Session Bean Or An Entity Bean
JMS_016: Do Not Use JMS Sessions In A Multi-threaded Context

## JMS\_001: Do Not Use A Vendor Specific Class

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JMS\\_001](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JMS_001)]

If you want your code to be vendor independent, you should not use vendor specific classes. You should only use classes from the `javax.jms` package.

WRONG

```
QueueConnectionFactory queueConnectionFactory
    = new progress.message.jclient.QueueConnectionFactory();
```

## JMS\_002: Do Not Hard Code An Initial Context Factory

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JMS\\_002](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JMS_002)]

Don't hard code the initial context factory class name when connecting to the server. It is better to store the initial context factory class name in an external configuration file that is read when the connection must be created..

WRONG

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "weblogic.jndi.WLInitialContextFactory");
// ...
InitialContext ctx = new InitialContext(env);
```

## JMS\_003: Do Not Hard Code A Provider URL

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JMS\_003]

Don't hard code the provider URL when connecting to the server. It is better store the provider URL in an external configuration file that is read when the connection must be created.

WRONG

```
Hashtable env = new Hashtable();
env.put(Context.PROVIDER_URL, "t3://localhost:7001");
// ...
InitialContext ctx = new InitialContext(env);
```

## JMS\_004: Do Not Hard Code A Connection Factory Name For JNDI Lookup

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JMS\_004]

Don't hard code the connection factory name for a JNDI lookup. It is better to store the connection factory name in an external configuration file that is read when the JNDI lookup must take place.

WRONG

```
InitialContext ctx = new InitialContext(env);
QueueConnectionFactory qconFactory = (QueueConnectionFactory)
    ctx.lookup("jms/FooQueueConnectionFactory");
```

## JMS\_005: Do Not Hard Code A Destination Name For JNDI Lookup

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JMS\_005]

Don't hard code the destination name for a JNDI lookup. It is better to store the destination name in an external configuration file that is read when the JNDI lookup must take place.

WRONG

```
InitialContext ctx = new InitialContext(env);
Queue queue = (Queue) ctx.lookup("jms/FooQueue");
```

## JMS\_006: Use JNDI Lookups To Get A Connection Factory Or Destination

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JMS\_006]

Never instantiate connection factories or destinations yourself. This would be an automatic violation of rule JMS\_001: Do Not Use A Vendor Specific Class.

WRONG

```
QueueConnectionFactory queueConnectionFactory
    = new progress.message.jclient.QueueConnectionFactory();
```

## JMS\_007: Close A Resource When It's No Longer

## Needed

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JMS\\_007](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JMS_007)]

All JMS resources (Connection, Session, Producer, Consumer) must be closed when they are no longer used.

Always close you resources in a finally clause of a try block. See rule JAC\_068: Close A Connection Inside A finally Block (Enforced).

### WRONG

```
private void doWrongClose(QueueConnectionFactory qconFactory,
    InitialContext ctx) throws JMSEException, NamingException {
    QueueConnection qcon = null;
    QueueSession qsession = null;
    QueueSender qsender = null;
    qcon = qconFactory.createQueueConnection();
    Queue queue = (Queue) ctx.lookup("jms/FooQueue");
    qsession = qcon.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
    qsender = qsession.createSender(queue);

    // ...
}
```

### RIGHT

```
private void doWrightClose(QueueConnectionFactory qconFactory,
    InitialContext ctx) throws JMSEException, NamingException {
    QueueConnection qcon = null;
    QueueSession qsession = null;
    QueueSender qsender = null;
    try {
        qcon = qconFactory.createQueueConnection();
        Queue queue = (Queue) ctx.lookup("jms/FooQueue");
        qsession = qcon.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
        qsender = qsession.createSender(queue);

        // ...
    } finally {
        if (qsender != null) {
            qsender.close();
        }
        if (qsession != null) {
            qsession.close();
        }
        if (qcon != null) {
            qcon.close();
        }
    }
}
```

## JMS\_008: Always Close Resources In The Correct Order

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JMS\\_008](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JMS_008)]

The JMS resources must be closed in the correct order:

1. Producer or Consumer
2. Session

### 3. Connection

See rule JMS\_007: Close A Resource When It's No Longer Needed for code examples.

## **JMS\_009: Do Not Produce A Message In A Transaction And Rely On It To Be Consumed Before The End Of The Transaction**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JMS\_009]

Sending messages in a transaction is possible. It is important to realize that these messages will not be seen by consumers until the transaction is committed. This means that it is impossible to process replies on the sent messages within the original transaction.

## **JMS\_010: Do Not Rely On JMS To Deliver Message In The Same Order As They Were Sent**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JMS\_010]

When using different JMS sessions for producing and consuming messages, JMS doesn't guarantee that messages are delivered to the consumer in the same order as they were produced.

If an application wants to consume messages in the same order as they were produced, a sequence number could be attached to each message. Instead of putting them on a JMS destination, they could be stored in a database table. The consumer could then read the messages from the table and sort them by the sequence number, in order to process them in the initial order.

## **JMS\_011: Start A Producer Connection After Start A Consumer**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JMS\_011]

If you start a connection before starting the consumer, then the messages have to wait in the JMS server. This is an unnecessary overhead, so first start consumers and then start the producer connection.

## **JMS\_012: Use A separate Transactional Session For transactional Messages And A Non-transactional Session For Non-transactional Messages**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JMS\_012]

If you want to send 100 messages, and only 10 of them in a transaction, it is better to use a transactional session to send the transactional messages and a non-transactional session to send the non-transactional messages.

## **JMS\_013: Set An Optimal Message Time To Live**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]JMS\_013]

Choose an optimal value for the time to live property of a message, to avoid unnecessary memory overhead. By default a message never expires.

## **JMS\_014: Choose A Correct Message Type**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JMS\_014]

The message size depends on the type of message you choose which in turn has an impact on the performance.

## **JMS\_015: Do Not Receive Messages Asynchronously In A Web Component, A Session Bean Or An Entity Bean**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JMS\_015]

Web, session or entity components should only consume messages synchronously using the Message-Consumer's receive methods, because they are driven by synchronous request-reply protocols, not asynchronous messages.

## **JMS\_016: Do Not Use JMS Sessions In A Multi-threaded Context**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JMS\_016]

JMS sessions were designed for single-threaded use. This is also true for all object created by the session, like producers and consumers.

If a client desires to have one thread producing messages while another thread asynchronously consumes messages at the same time, the client should use a separate session for its producing thread.

It is also forbidden to attempt to combine both synchronous and asynchronous message receiving in the same session. Either the session is dedicated to the thread of control used for delivery to message listeners or it is dedicated to a thread of control initiated by client code.

It is allowed however, to combine producing and consuming messages in the same thread.

## **Assertions Rules**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]assertionsRules]

### **Overview**

**Table B.5. Assertions Overview**

Rules
ASSERT_001: Check A Method's Arguments
ASSERT_002: Do Not Use Assertions For Argument Checking In A public Method
ASSERT_003: Do Not Do Any Processing In An Assertion's Condition
ASSERT_004: Do Not Catch Assertion Related Exceptions
ASSERT_005: Use Assertions In A switch Statement's default Case Correct
ASSERT_006: Do Not Evaluate More Than One Condition In An Assertion
ASSERT_007: Make An Assertion Descriptive

## **ASSERT\_001: Check A Method's Arguments**

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]ASSERT\_001]

Always check argument values in accessible (public, protected and package local) methods.

WRONG

```
public void setName(String name) {  
    this.name = name.trim();  
}
```

RIGHT

```
public void setName(String name) {  
    if (name == null) {  
        throw new IllegalArgumentException("name can not be null");  
    }  
    this.name = name.trim();  
}
```

## ASSERT\_002: Do Not Use Assertions For Argument Checking In A public Method

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]ASSERT\\_002](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]ASSERT_002)]

By convention, preconditions on methods that are public, protected or package local are enforced by explicit checks that throw particular, specified exceptions (a commonly used exception is `java.lang.IllegalArgumentException`). We can't use assertions for this because the method must guarantee that these checks will ALWAYS be done, even if assertions are disabled.

WRONG

```
public void setInitialCount(int count) {  
    // Enforce specified precondition in public method  
    assert count >= 0 : "Illegal initial count: " + count;  
    initialCount = count;  
}
```

RIGHT

```
public void setInitialCount(int count) {  
    // Enforce specified precondition in public method  
    if (count <= 0) {  
        throw new IllegalArgumentException(  
            "Initial count must be > 0, got " + count);  
    }  
    initialCount = count;  
}
```



### Important

Remember to throw a specific exception if the method's scope changes.

## ASSERT\_003: Do Not Do Any Processing In An Assertion's Condition

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]ASSERT\_003]

When assertions are disabled they are not executed by the JVM. This means that the condition, stated in the assertion, will not be evaluated and any processing done in the condition will not be executed.

WRONG

```
// the item will not be removed if assertions are disabled!
assert set.remove(item) : "The item was not in the set."
```

RIGHT

```
boolean modified = set.remove(item);
assert modified : "The item was not in the set."
```

## ASSERT\_004: Do Not Catch Assertion Related Exceptions

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]ASSERT\_004]

Don't catch any assertion related exceptions. Catching these exceptions bypasses the whole assertion and *Design By Contract* mechanism. Exceptions include:

- `AssertionError`
- `IllegalArgumentException`

Your own assertion failed exception in case you are using your own assertion implementation.

WRONG

```
public void setData() {
    try {
        setName(null);
    } catch (IllegalArgumentException e) {
        // some error handling
    }

    public void setName(String name) {
        if (name == null) {
            throw new IllegalArgumentException("name can not be null");
        }
        this.name = name;
    }
}
```

RIGHT

Don't catch the `IllegalArgumentException`.

## ASSERT\_005: Use Assertions In A switch Statement's default Case Correct

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]ASSERT\_005]



switch statements with no default case could use assertions to ensure that only known cases are handled. The way in which the assertion failure is reported to the outside world depends on the scope of the method in which the statement is used and the type of value that is evaluated by the switch statement.

**Table B.6. What Kind Of Assertion To Use**

Use	When
assert statement	If the method containing the switch statement is private If the value evaluated in the switch statement is a class variable
IllegalArgumentException	If the value evaluated in the switch statement is a value that was provided as an argument to the non-private method containing the switch statement

## RIGHT

Using IllegalArgumentException

```
public String getDescription(int code) {
    switch (code) {
        case NAME:
            description = "name";
            break;
        case ADDRESS:
            description = "address";
            break;
        default:
            throw new IllegalArgumentException("Unexpected code " + code);
    }
    assert (description != null) : "Expected a description";
    return description;
}
```

Using assert

```
public String getDescription() {
    switch (this.code) {
        case NAME:
            description = "name";
            break;
        case ADDRESS:
            description = "address";
            break;
        default:
            assert false : "Unknown code " + code;
    }
    assert (description != null) : "Expected a description";
    return description;
}
```

## ASSERT\_006: Do Not Evaluate More Than One Condition In An Assertion

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]ASSERT\\_006](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]ASSERT_006)]

Evaluate only one condition in a an assertion (or assertion like) expression. If you evaluate more than one condition you don't know which specific assumption is invalid and you will not be able to provide sufficient feedback.

#### WRONG

```
public void setName(String first, String last) {
    if ( (first == null) || (last == null) ) {
        throw new IllegalArgumentException("first or last can not be null");
    }
    // ...
}

private void setName(String first, String last) {
    assert (first != null) && (last != null) : "first or last can not be null";
    // ...
}
```

#### RIGHT

```
public void setName(String first, String last) {
    if (first == null) {
        throw new IllegalArgumentException("first can not be null");
    }
    if (last == null) {
        throw new IllegalArgumentException("last can not be null");
    }
    // ...
}

private void setName(String first, String last) {
    assert (first != null) : "first can not be null";
    assert (last != null) : "last can not be null";
    // ...
}
```

## ASSERT\_007: Make An Assertion Descriptive

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]ASSERT\\_007](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]ASSERT_007)]

Make your assertions descriptive by adding a message stating the assumption you are making. If the assertion then fails the description can be presented is to the user. If no description is provided to only signal you get is that something went wrong.

#### WRONG

```
public void setName(String name) {
    if (name == null) {
        throw new IllegalArgumentException();
    }
    // ...
}

private void setName(String name) {
    assert (name != null);
    // ...
}
```

RIGHT

```
public void setName(String name) {
    if (name == null) {
        throw new IllegalArgumentException("name can not be null");
    }
    // ...
}

private void setName(String name) {
    assert (name != null) : "name can not be null";
    // ...
}
```

## Testing Rules

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]testingRules](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]testingRules)]

### Overview

**Table B.7. Testing Overview**

Rules
TEST_001: Provide A Unit Test For Each Utility Class, Library Or Base Layer
TEST_002: Name Test Methods Properly
TEST_003: Provide Mock Objects For A Unit Which Collaborates With Other Objects
TEST_004: Only Test What Can Possibly Break
TEST_005: Automate Running Unit Tests In The Build Process
TEST_006: Do Not Write Business Logic In Mock Objects
TEST_007: Run Scenario Tests As Part Of The Delivery Procedure Of A Product
TEST_008: Keep Mock Objects Independent From Each Other
TEST_009: Do Not Rely On The Order Of Tests Within A Testcase
TEST_010: Avoid Visual Inspection In Unit Tests
TEST_011: Avoid Code Duplication In Unit Tests
TEST_012: Call setUp() and tearDown() Methods Of A Testcase's Superclass
TEST_013: Isolate Test Data From Test Code
TEST_014: Do Not Load Data From Hardcoded Locations
TEST_015: Make Tests Locale Independent
TEST_016: Keep Unit Tests Small And Fast

## TEST\_001: Provide A Unit Test For Each Utility Class, Library Or Base Layer

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]TEST\\_001](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]TEST_001)]

Each utility class, library or base layer should be covered by a unit test

## TEST\_002: Name Test Methods Properly

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]TEST\_002]

Name the test method of a testcase properly. The name should not only start with "test", it should also be an accurate description of what the test does.

WRONG

```
public class CalcTest extends TestCase {
    public void test1() {
        // ...
    }

    public void test2() {
        // ...
    }
}
```

RIGHT

```
public class CalcTest extends TestCase {
    public void testDivisionByZero() {
        // ...
    }

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

## TEST\_003: Provide Mock Objects For A Unit Which Collaborates With Other Objects

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]TEST\_003]

Provide mock objects when the unit collaborates with other objects.

## TEST\_004: Only Test What Can Possibly Break

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]TEST\_004]

Only test what can possibly break. Do not test trivial code like most accessors and mutators (get/set methods).

## TEST\_005: Automate Running Unit Tests In The Build Process

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]TEST\_005]

Automate running unit tests in the build process.

## TEST\_006: Do Not Write Business Logic In Mock Objects

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]TEST\_006]

Don't write business logic in mock objects.

## TEST\_007: Run Scenario Tests As Part Of The Delivery Procedure Of A Product

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]TEST\_007]

Running scenario tests should be a part of the delivery procedure of the product. Preferably run the automated scenario tests at least every night. Make sure all scenario tests work before delivering production code.

## TEST\_008: Keep Mock Objects Independent From Each Other

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]TEST\_008]

Keep your mock objects independent from each other. When mock objects start to call other mock objects, this is a sign that your mock objects try to be real objects (which they shouldn't).

## TEST\_009: Do Not Rely On The Order Of Tests Within A Testcase

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]TEST\_009]

Do not assume that the tests will be performed in the order they appear in your code. When JUnit uses the Java reflection API to dynamically build the testsuite, it offers no guarantees about the order of the tests.

### WRONG

```
public class ExampleTest extends TestCase {
    public ExampleTest (String testName) {
        super (testName);
    }

    public void testThisFirst () {
        // ...
    }

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

### RIGHT

Tests should be independent from each other. In the rare case that ordering tests actually makes sense (e.g. for performance reasons), use the static `suite()` method to order your tests :

```
public static Test suite() {
    suite.addTest(new ExampleTest ("testThisFirst"));
    suite.addTest(new ExampleTest ("testThisSecondly"));
    return suite;
}
```

## TEST\_010: Avoid Visual Inspection In Unit Tests

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]TEST\_010]

Avoid visual inspection in a unit test. Instead use the `assert` methods to compare the results with expected values.

WRONG

```
public void testMessage() {
    // ...
    System.out.println("Message body : " + message.getBody());
    System.out.println("Message size : " + message.getSize());
    // ...
}
```

RIGHT

```
public void testMessage() {
    // ...
    assertNotNull(message.getBody());
    assertEquals(1000, message.getSize());
    // ...
}
```

## TEST\_011: Avoid Code Duplication In Unit Tests

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]TEST\_011]

Avoid code duplication in unit tests. A possible solution to this is factoring out the duplicate code and maybe even moving it to a common superclass.

WORSE

```
public class EuroAmountTest extends TestCase {
    public void testEquals() {
        // identical initialization code
        EuroAmount fiveEuroFiftyCents = new EuroAmount(5, 50);
        EuroAmount threeEuroEightyCents = new EuroAmount(3, 80);

        Assert.assertTrue(!fiveEuroFiftyCents.equals(null));
        Assert.assertEquals(fiveEuroFiftyCents, fiveEuroFiftyCents);
        Assert.assertEquals(threeEuroEightyCents, new EuroAmount(3, 80));
        Assert.assertTrue(!fiveEuroFiftyCents.equals(threeEuroEightyCents));
    }

    public void testAddition() {
        // identical initialization code
        EuroAmount fiveEuroFiftyCents = new EuroAmount(5, 50);
        EuroAmount threeEuroEightyCents = new EuroAmount(3, 80);

        EuroAmount expected = new EuroAmount(9, 30);
        EuroAmount result = fiveEuroFiftyCents.add(threeEuroEightyCents);
        Assert.assertTrue(expected.equals(result));
    }
}
```

BETTER

```
public class EuroAmountTest extends TestCase {
    private EuroAmount fiveEuroFiftyCents;
    private EuroAmount threeEuroEightyCents;

    // identical initialization code is no longer duplicated
    protected void setUp() {
        fiveEuroFiftyCents = new EuroAmount(5, 50);
        threeEuroEightyCents = new EuroAmount(3, 80);
    }

    public void testEquals() {
        Assert.assertTrue(!fiveEuroFiftyCents.equals(null));
        Assert.assertEquals(fiveEuroFiftyCents, fiveEuroFiftyCents);
        Assert.assertEquals(threeEuroEightyCents, new EuroAmount(3, 80));
        Assert.assertTrue(!fiveEuroFiftyCents.equals(threeEuroEightyCents));
    }

    public void testAddition() {
        EuroAmount expected = new EuroAmount(9, 30);
        EuroAmount result = fiveEuroFiftyCents.add(threeEuroEightyCents);
        Assert.assertTrue(expected.equals(result));
    }
}
```

## TEST\_012: Call setUp() and tearDown() Methods Of A Testcase's Superclass

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]TEST\\_012](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]TEST_012)]

When you subclass for your testcase

WRONG

```
public class ExampleTest extends AbstractExampleTest {
    public void testFeatureX () {
        // ...
    }

    public void setUp () {
        // set up fixture specific for ExampleTestCase
        // ...
    }

    public void tearDown () {
        // tear down fixture specific for ExampleTestCase
        // ...
    }
}
```

RIGHT

```
public class ExampleTest extends AbstractExampleTest {
    public void testFeatureX () {
        // ...
    }

    public void setUp () {
        super.setUp() // allow super class to set up it's fixture
        // set up fixture specific for ExampleTestCase
    }
}
```

```
        // ...
    }

    public void tearDown () {
        // tear down fixture specific for ExampleTestCase
        // ...
        super.tearDown(); // allow super class to tear down it's fixture
    }
}
```

## TEST\_013: Isolate Test Data From Test Code

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]TEST\\_013](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]TEST_013)]

Isolate your test data from your test code. Separating test data allows you to change the test data without changing the test code.

## TEST\_014: Do Not Load Data From Hardcoded Locations

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]TEST\\_014](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]TEST_014)]

Do not read data from hardcoded locations on the filesystem. Reading from hardcoded absolute filepaths makes the tests less portable.

Instead, put your data files on your test classpath and read the data using the static methods `getResource()` or `getResourceAsStream()` of `Class`. This will make the test data as portable as the test code.

## TEST\_015: Make Tests Locale Independent

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]TEST\\_015](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]TEST_015)]

Make sure your tests are locale independent. Running a test on a system with different locale settings should not break your tests.

## TEST\_016: Keep Unit Tests Small And Fast

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]TEST\\_016](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]TEST_016)]

Unit tests should be fast and small since you need to run them very often.

## JMX Rules

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]jmxRules](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]jmxRules)]

## Overview

**Table B.8. JMX Overview**

Rules
JMX_001: Name A MBean Interface Like <code>[ImplementingClassName]MBean</code>
JMX_002: Declare A MBean Interface public
JMX_003: Provide A public No Arg Constructor For A MBean Implementing Class



Rules
JMX_004: Make A Getter Or Setter Follow The Naming Conventions
JMX_005: Return The Property Type For A Getter
JMX_006: Declare A Getter Without Parameters
JMX_007: Declare A Setter With At Least 1 Parameter
JMX_008: Declare A Setter With At Most 1 Parameter
JMX_009: Do Not Define The Same Getter Twice
JMX_010: Match Getter And Setter

## JMX\_001: Name A MBean Interface Like [ImplementingClassName]MBean

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JMX\\_001](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JMX_001)]

The management interface must have the same name followed by MBean as the resource it provides the interface for.

WRONG

```
public interface CacheMBean {
    // ...
}

public class MyCache implements CacheMBean {
}
```

WRONG

```
public interface MyCacheMbean {
    // ...
}

public class MyCache implements CacheMbean {
}
```

RIGHT

```
public interface MyCacheMBean {
    // ...
}

public class MyCache implements MyCacheMBean {
}
```



### Note

The naming of the interface and the implementing class are case sensitive. MyCacheMBean is not the same as MyCacheMbean.

If you declare the interface as in the second wrong example the compiler will of course not complain. However when you try to register your MBean with the MBean server a `NotCompliantMBeanException` will be thrown.

## JMX\_002: Declare A MBean Interface `public`

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JMX\\_002](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JMX_002)]

The interface must be `public`.

WRONG

```
interface CacheMBean {  
    // ...  
}
```

RIGHT

```
public interface CacheMBean {  
    // ...  
}
```



### Note

The MBean can be created and consequently registered with the MBean server. However when trying to use the management interface a `javax.management.ReflectionException` will be thrown.

## JMX\_003: Provide A `public` No Arg Constructor For A MBean Implementing Class

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JMX\\_003](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JMX_003)]

The implementeting class needs a `public` constructor.

WRONG

```
public class MyCache implements MyCacheMBean {  
    // a private constructor  
    // so that programmers cannot instantiate this class  
    private MyCache() {  
    }  
    //There is no means for the MBeanServer to instantiate the class  
  
    // ...  
}
```

RIGHT

```
public class MyCache implements MyCacheMBean {  
    // a default no-argument constructor is provided by the compiler  
    // ...  
}
```



### Note

Also constructors with arguments can be used.

**Tip**

Although the system will generate a no-argument constructor for you if you not define any constructor, it is recommended that you explicitly declare the no-arg constructor.

## JMX\_004: Make A Getter Or Setter Follow The Naming Conventions

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JMX\\_004](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JMX_004)]

A getter or setter needs to follow the naming convention.

**WRONG**

```
public interface MyCacheMBean {  
    public void setmaxEntities(long nbr); // Wrong capitalisation  
    public long getMax(); // Does not use the name of the arguments it gets  
}
```

**RIGHT**

```
public interface MyCacheMBean {  
    public void setMaxEntities(long nbr);  
    public long getMaxEntities();  
    // ...  
}
```

**Note**

For Getters returning a boolean value the `isProperty` form is also accepted, e.g. `isEmpty`.

## JMX\_005: Return The Property Type For A Getter

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JMX\\_005](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JMX_005)]

A getter needs to return the type of the property it is getting.

**WRONG**

```
public interface MyCacheMBean {  
    public void getMaxEntities(); // Will be viewed as an operation  
    // ...  
}
```

## JMX\_006: Declare A Getter Without Parameters

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JMX\\_006](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JMX_006)]

A getter shouldn't have any parameters.

WRONG

```
public interface MyCacheMBean {  
    public long getMaxEntities(long anArg); // Will be viewed as an operation  
    // ...  
}
```

## JMX\_007: Declare A Setter With At Least 1 Parameter

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JGuidelines\]JMX\\_007](mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JMX_007)]

A setter should have at least 1 parameter.

WRONG

```
public interface MyCacheMBean {  
    public void setMaxEntities(); // Will be viewed as an operation  
    // ...  
}
```



### Tip

In fact such operations will be regularly used for setting boolean values., e.g. `enableCaching()` in stead of `setCaching(boolean arg)`.

## JMX\_008: Declare A Setter With At Most 1 Parameter

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JGuidelines\]JMX\\_008](mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JMX_008)]

A setter may only have one parameter for an MBean.

WRONG

```
public interface MyCacheMBean {  
    public void setMaxEntities(long nbr, boolean allowShrinkage);  
    // Will be viewed as an operation  
    // ...  
}
```

## JMX\_009: Do Not Define The Same Getter Twice

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JGuidelines\]JMX\\_009](mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JMX_009)]

A getter must not be defined twice or more.

WRONG

```
public interface MyCacheMBean {
```

```
public boolean isEmpty();  
public boolean getEmpty();  
// ...  
}
```

## JMX\_010: Match Getter And Setter

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JMX\_010]

Getters and setters need to match.

WRONG

```
public interface MyCacheMBean {  
  
    // Maximum entities is read/write  
    public void setMaxEntities(int nbr);  
    public long getMaxEntities();  
  
    // ...  
}
```



### Note

The setter takes precedence over the getter (although this could be different for different MBean server implementations). In other words: there is NO getter defined for `maxEntities`. The MBean server had to choose between long and int and has given preference to the setter. There is now an operation `getMaxEntities` but not a getter.

## JMX\_011: Do Not Construct An MBean

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JMX\_011]

Use the `instantiate` method instead.

WRONG

```
Object myCache = new MyCache();
```

RIGHT

```
Object myCache = mbs.instantiate("MyCache");
```

## JAAS Rules

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]jaasRules]

### Overview

Table B.9. JAAS Overview

Rules
JAAS_001: Do Not Add A Principal Or Credential To A Subject At Login Time In A Login Module
JAAS_002: Only Clean A Principal Or Credential Previously Added To A Non Readonly Subject At Logout Time In A Login Module
JAAS_003: Return false From A login Method Only To Indicate To Ignore The Login Module
JAAS_004: Do Not Interact With A User Directly In A Login Module
JAAS_005: Ask For Credentials Only Once When Combining Multiple Login Modules
JAAS_006: Add A Debug Option To A Login Module
JAAS_007: Make A Principal Serializable
JAAS_008: Use Principals To Name Groups And Roles
JAAS_009: Use A PrincipalComparator To Structure Principals Hierarchically
JAAS_010: Use A PrivilegedExceptionAction To Raise Checked Exceptions From Secured Code Fragments
JAAS_011: Use The doAsPrivileged Method Of A Subject Instead Of The doAs Method

## JAAS\_001: Do Not Add A Principal Or Credential To A Subject At Login Time In A Login Module

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAAS\_001]

If the application instructs JAAS to login a 2-phase process is triggered:

- First, one by one, on each LoginModule the login method is invoked. The outcome of login should be stored by each login module privately.
- If all login invocations returned true, the distinct login modules are instructed to commit: this is the correct time to link relevant Principal and Credential objects to the Subject.

### WRONG

```
public class MyLoginModule {
    private Subject s;

    public boolean login() throws LoginException {
        // Declare and initialize boolean trustedUser
        if (trustedUser) {
            Set principals = Subject.getPrincipals();
            principals.add(new MyPrincipal("jcs"));
        }
        return true;
    }

    public boolean commit() {
    }
}
```

### RIGHT

```
public class MyLoginModule {
    private Subject s;
```

```
private boolean trustedUser;

public boolean login() throws LoginException {
    trustedUser = true;

    return true;
}

public boolean commit() throws LoginException {
    if (trustedUser) {
        Set principals = Subject.getPrincipals();
        principals.add(new MyPrincipal("jcs"));
    }
}
}
```

## JAAS\_002: Only Clean A Principal Or Credential Previously Added To A Non Readonly Subject At Logout Time In A Login Module

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAAS\\_002](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAAS_002)]

Each LoginModule uses well-defined Principals. LoginModules should not destroy each other's Principals.

WRONG

```
public class MyLoginModule {
    private Subject s;

    public boolean logout() throws LoginException {
        s.getPrincipals().clear();
    }
}
```

RIGHT

```
public class MyLoginModule {
    private Subject s;

    public boolean logout() {
        Set mine = s.getPrincipals(MyPrincipal.class);
        s.getPrincipals().removeAll(mine);
    }
}
```

## JAAS\_003: Return false From A login Method Only To Indicate To Ignore The Login Module

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAAS\\_003](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAAS_003)]

If a user is not trusted by a login module, the latter should not return false but raise a `LoginException` at login-time.

WRONG

```
public class MyLoginModule {
    public boolean login() throws LoginException {
        // ...
        return trustedUser;
    }
}
```

RIGHT

```
public class MyLoginModule {
    public boolean login() throws LoginException {
        // ...
        throw new LoginException("User not trusted");
    }
}
```

## JAAS\_004: Do Not Interact With A User Directly In A Login Module

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAAS\\_004](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAAS_004)]

At no single moment a login module may directly contact the subject to authenticate: at all times a callback should be used to do so. Numerous callback-implementations are available, even generic `TextInputCallback` and `TextOutputCallback`.

WRONG

```
public class MyLoginModule {
    public boolean login() throws LoginException {
        System.out.println("Please provide VISA");
        BufferedReader stdin = new BufferedReader(
            new InputStreamReader(System.in));
        String visa = stdin.readLine();
    }
}
```

RIGHT

```
public class MyLoginModule {
    private CallbackHandler cbh;

    public boolean login() throws LoginException {
        Callback visaCb = new TextInputCallback("Please provide VISA");
        cbh.handle(new Callback[] {visaCb});
        String visa = visaCb.getText();
    }
}
```

## JAAS\_005: Ask For Credentials Only Once When Combining Multiple Login Modules

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAAS\\_005](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAAS_005)]

`LoginModule` should verify that credentials were not yet prompted for by another other modules. This dramatically enhances user-experience. The shared options `Map` has well defined entries to contain user-



name and password:

- `javax.security.auth.login.name`
- `javax.security.auth.login.password`

#### WRONG

```
public class MyLoginModule {
    private CallbackHandler cbh;

    public boolean login() throws LoginException {
        cbh.handle(/* ... */);
    }
}
```

#### RIGHT

```
public class MyLoginModule {
    private CallbackHandler cbh;
    private Map sharedState;

    public boolean login() throws LoginException {
        String uid = (String)sharedState.get("javax.security.auth.login.name");
        if ((uid == null)) {
            cbh.handle(new Callback[] { /* ... */ });
        }
    }
}
```

## JAAS\_006: Add A Debug Option To A Login Module

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAAS\\_006](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAAS_006)]

Login modules function in the background, hiding a lot of detail. Both when doing development and debugging authentication-issues later on it, it is convenient to be able to turn on detailed debugging info. Consider this as lightweight debugging, serious bugs should always be cleared by attaching a JVMDI-enabled debugger to the application.

#### RIGHT

jaas.config

```
Foo {
    be.jcs.auth.MyLoginModule REQUIRED debug=true;
}
```

MyLoginModule.java

```
public class MyLoginModule {
    private Map options;

    public boolean login() throws LoginException {
        if (Boolean.valueOf((String)options.get("debug")).booleanValue()) {
            logger.debug("Authenticated " + username);
        }
    }
}
```

```
}  
}
```

## JAAS\_007: Make A Principal Serializable

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAAS\\_007](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAAS_007)]

At all times custom `Principal` implementations should implement `Serializable`.

WRONG

```
public class MyPrincipal implements Principal {  
    private String name;  
  
    public String getName() {  
        return this.name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

RIGHT

```
public class MyPrincipal implements Principal, Serializable {  
    private String name;  
  
    public String getName() {  
        return this.name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

## JAAS\_008: Use Principals To Name Groups And Roles

Feedback [[mailto:feedback@jjguidelines.dev.java.net?Subject=\[JJGuidelines\]JAAS\\_008](mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAAS_008)]

It is convenient to associate a `Subject` with `Principals` modeling *groups* and/or *roles*. Authorization will treat these accordingly.

RIGHT

```
jaas.config  
  
grant Principal foo.Role "administrator" {  
    permission java.io.FilePermission "/etc/passwd", "read, write";  
}  
  
grant Principal foo.Team "slamDunk" {  
    permission be.jcs.auth.DunkPermission "dunk";  
}
```

## JAAS\_009: Use A `PrincipalComparator` To Structure Principals Hierarchically

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAAS\_009]

A *Principal* implementation can in fact implement the interface `PrincipalComparator` which can be used to support role hierarchies.

### RIGHT

Following sample grants user permissions to admin.

```
public class MyPrincipalComparator {
    public boolean implies(Subject subject) {
        // Verify hierarchy, eg admin is granted all user-permissions
    }
}

public class MyPrincipal extends MyPrincipalComparator
    implements Principal, Serializable {
    // ...
}

jaas.config

grant Principal MyPrincipal "user" {
    // ...
}
```

## JAAS\_010: Use A `PrivilegedExceptionAction` To Raise Checked Exceptions From Secured Code Fragments

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]JAAS\_010]

A common `PrivilegedAction` cannot raise checked Exceptions (do not abuse the return `Object` to report an `Exception`). One should use a `PrivilegedExceptionAction` instead.

### WRONG

```
public class MyPrivilegedAction
    implements PrivilegedAction {
    public Object run() {
        try {
            // ...
        } catch (CheckedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

### RIGHT

```
public class MyPrivilegedAction
    implements PrivilegedExceptionAction {
    public Object run() throws Exception {
```

```
    }  
}
```

## JAAS\_011: Use The `doAsPrivileged` Method Of A Subject Instead Of The `doAs` Method

Feedback [[mailto:feedback@jguidelines.dev.java.net?Subject=\[JGuidelines\]JAAS\\_011](mailto:feedback@jguidelines.dev.java.net?Subject=[JGuidelines]JAAS_011)]

Make sure execution of `PrivilegedActions` starts on a blank `AccessControllerContext`. Avoid to pass the `currentThread`'s `AccessControllerContext`. Invoking `Subject.doAsPrivileged` is sufficient to force that a temporary `AccessControllerContext` is used to trace privileges for a given `PrivilegedAction`.

RIGHT

```
public class MyPrivilegedAction  
    implements PrivilegedAction {  
    public Object run() {  
        // ...  
    }  
}  
  
public class Foo {  
    public static void main(final String[] args) {  
        LoginContext foo = new LoginContext("Foo");  
        foo.login();  
        Subject subject = foo.getSubject();  
        subject.doAsPrivileged(  
            subject,  
            new MyPrivilegedAction(),  
            null);  
    }  
}
```

---

# Appendix C. CheckStyle

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]checkstyle]

## Introduction

Checkstyle is an open source Java auditing tool. See <http://checkstyle.sourceforge.net/>.

## Integration With ANT

Apache Ant is a free, open source, Java-based build tool. See <http://ant.apache.org>.

## Installation

1. Unzip the contents of `JJGuidelines-CheckStyle-Ant.zip` (which you can find in the directory `tools/checkstyle` as part of the JJGuidelines distribution) to an appropriate directory, for example in `c:\checkstyle`.
2. The CheckStyle distribution contains an ANT task implementation. To use it, you have to add the following XML fragment to your `build.xml` file:

```
<property name="checkstyle.dir" location="c:\checkstyle"/>
<property name="build.reports.dir" location="build/reports"/>

<property name="src.java.dir" location="src/java"/>

<target name="qa" depends="jjguidelines"
  description="Project Quality Assurance"/>

<target name="jjguidelines" depends="jjguidelines.checkstyle"
  description="JJGuidelines Quality Assurance"/>

<target name="jjguidelines.checkstyle"
  description="Enforce JJGuidelines rules with CheckStyle">
  <taskdef resource="checkstyletask.properties">
    <classpath>
      <fileset dir="${checkstyle.dir}/lib">
        <include name="**/*.jar"/>
      </fileset>
    </classpath>
  </taskdef>

  <mkdir dir="${build.reports.dir}"/>

  <checkstyle
    config="${checkstyle.dir}/conf/checkstyle-jjguidelines.xml"
    packageNamesFile="${checkstyle.dir}/conf/packages.xml"
    failOnViolation="true">
    <formatter type="xml"
      toFile="${build.reports.dir}/checkstyle-report.xml"/>
    <formatter type="plain"/>
    <fileset dir="${src.java.dir}">
      <include name="**/*.java"/>
    </fileset>
  </checkstyle>
</target>
```

```

        <classpath>
            <pathelement location="build/classes"/>
        </classpath>
    </checkstyle>
</target>

```

3. Change the properties in the ANT file to match your configuration:

checkstyle.dir	The directory where you unzipped the checkstyle files
build.reports.dir	The directory where the report should be generated
src.java.dir	The directory where your source files are located



### Important

The compiled classes and required libraries of your project must be included on checkstyle's classpath. You can include them in Ant's classpath or configure either the classpathref attribute or the nested classpath task of the checkstyle task appropriately. The above example adds all files in the 'build/classes' directory to the classpath.



### Tip

You might want to process the XML report, for example to automatically merge reports from several projects or transform it into HTML.

## Usage

- Start your ANT build script with `jjguidelines` as the target.

```
ant jjguidelines
```

## Integration With IntelliJ IDEA 4.x

IntelliJ IDEA is a java IDE with strong support for refactoring. You can find more information about IDEA at <http://www.intellij.com>.

## Installation

1. Unzip the contents of `JJGuidelines-CheckStyle-IntelliJ.zip` (which you can find in the directory `tools/checkstyle` as part of the JJGuidelines distribution) in an appropriate di-

rectory, for example in `c:\checkstyle`.

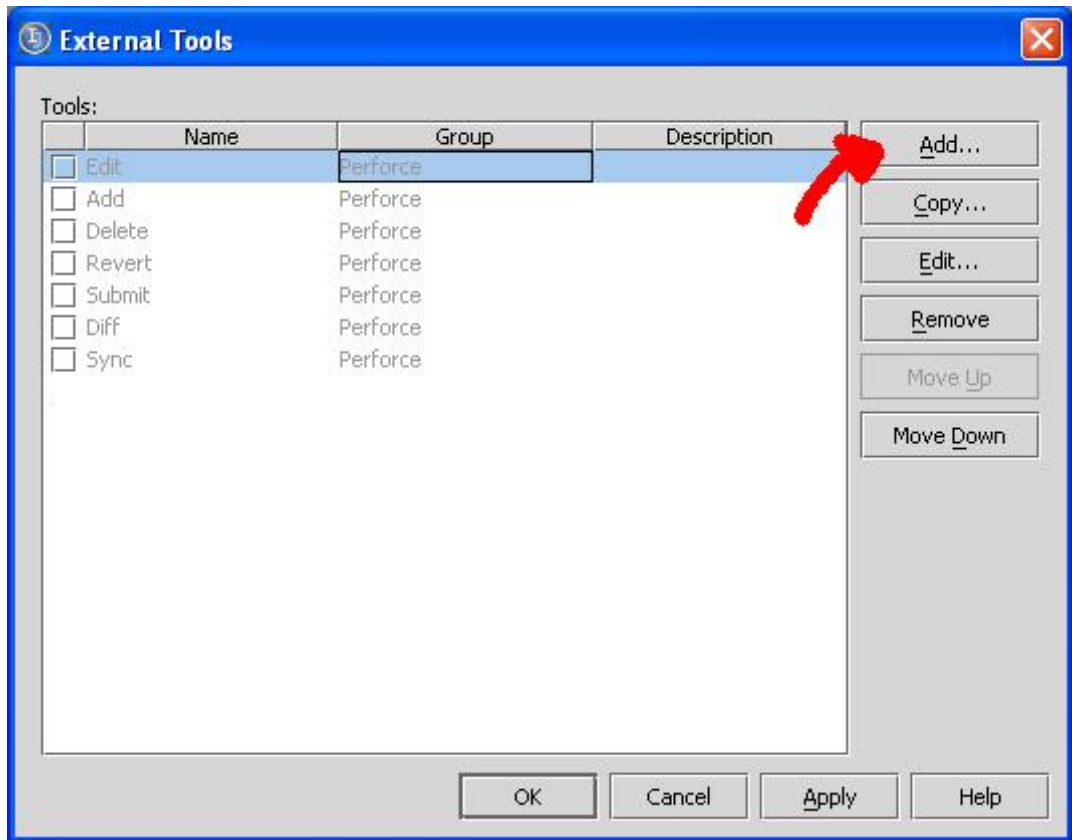
2. Open IntelliJ IDEA.
3. Open the menu **File, Settings**. Press the button **External Tools** under *IDE Settings* in the *Settings* window.



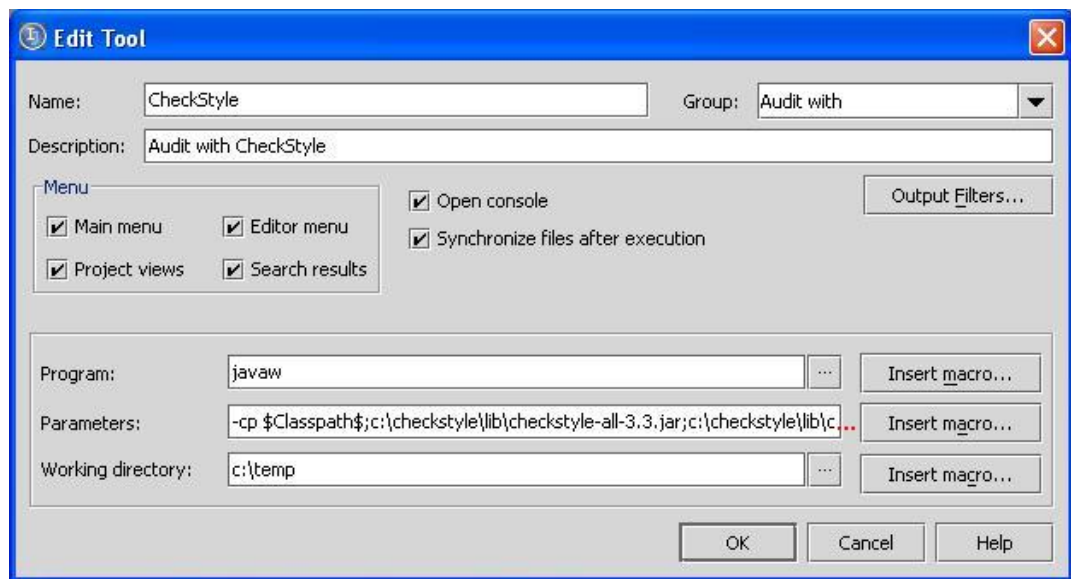
### Note

In IntelliJ IDEA 3.x, open the menu **Options, External Tools**.

4. Press the button **add** to add a new external tool.



5. Set the parameters for CheckStyle in the *Edit Tool* window:

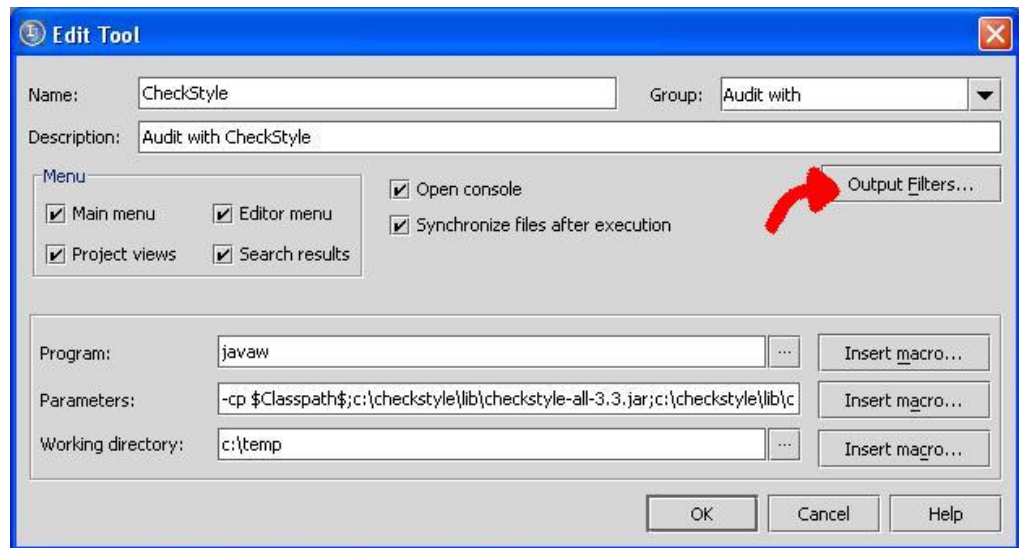


Name	CheckStyle
Group	Audit with
Description	Audit with CheckStyle
Program	javaw
Parameters	-cp \$Classpath\$c:\checkstyle\lib\checkstyle-all-3.3 .jar;c:\checkstyle\lib\checkstyle-optional-3.3.j ar com.puppcrawl.tools.checkstyle.Main -c c:\checkstyle\conf\checkstyle-jjguidelines.xml -n c:\checkstyle\conf\packages.xml -r \$FilePath\$
Working directory	c:\temp

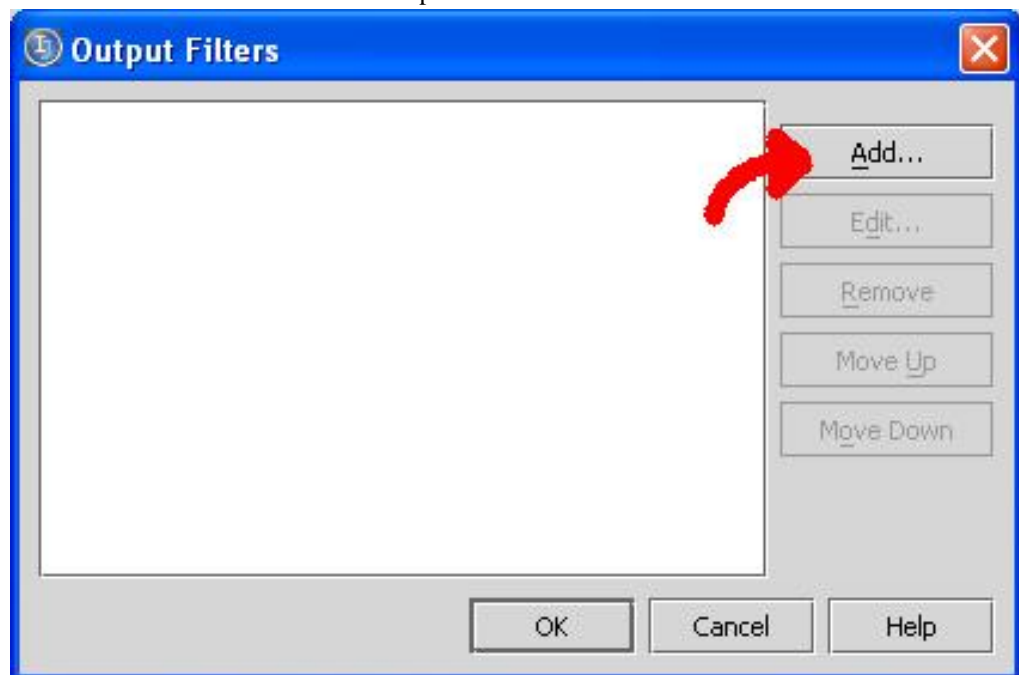
Change the *parameters* arguments to match the directory where you unzipped the files.

6. Add an output filter.
  - a. Press the button **Output Filters** to create links from the output to the source files.

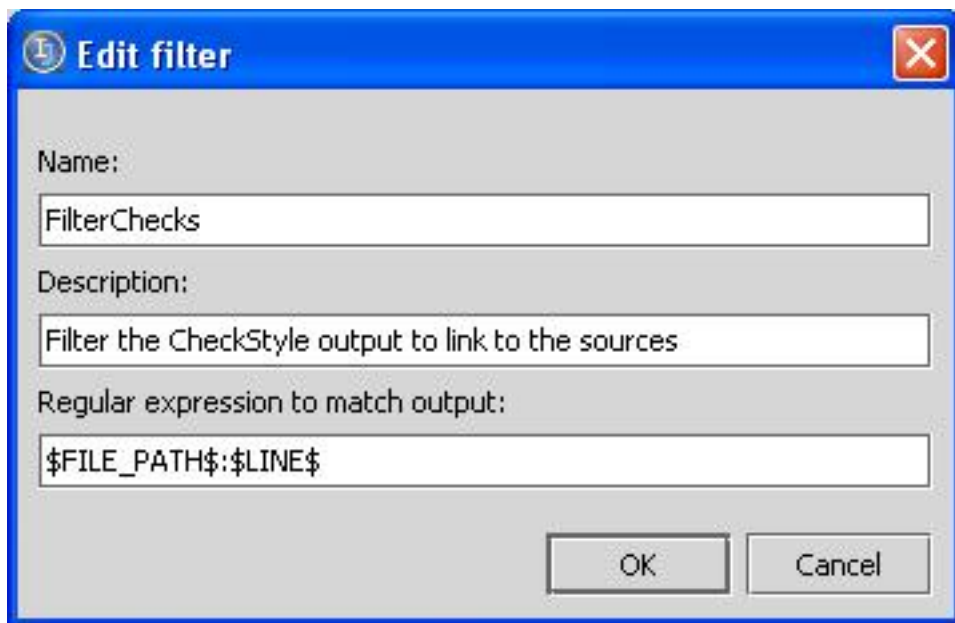




- b. Press the button **Add** to add a new output filter.



- c. Set the filter information in the *Add Filter* window:

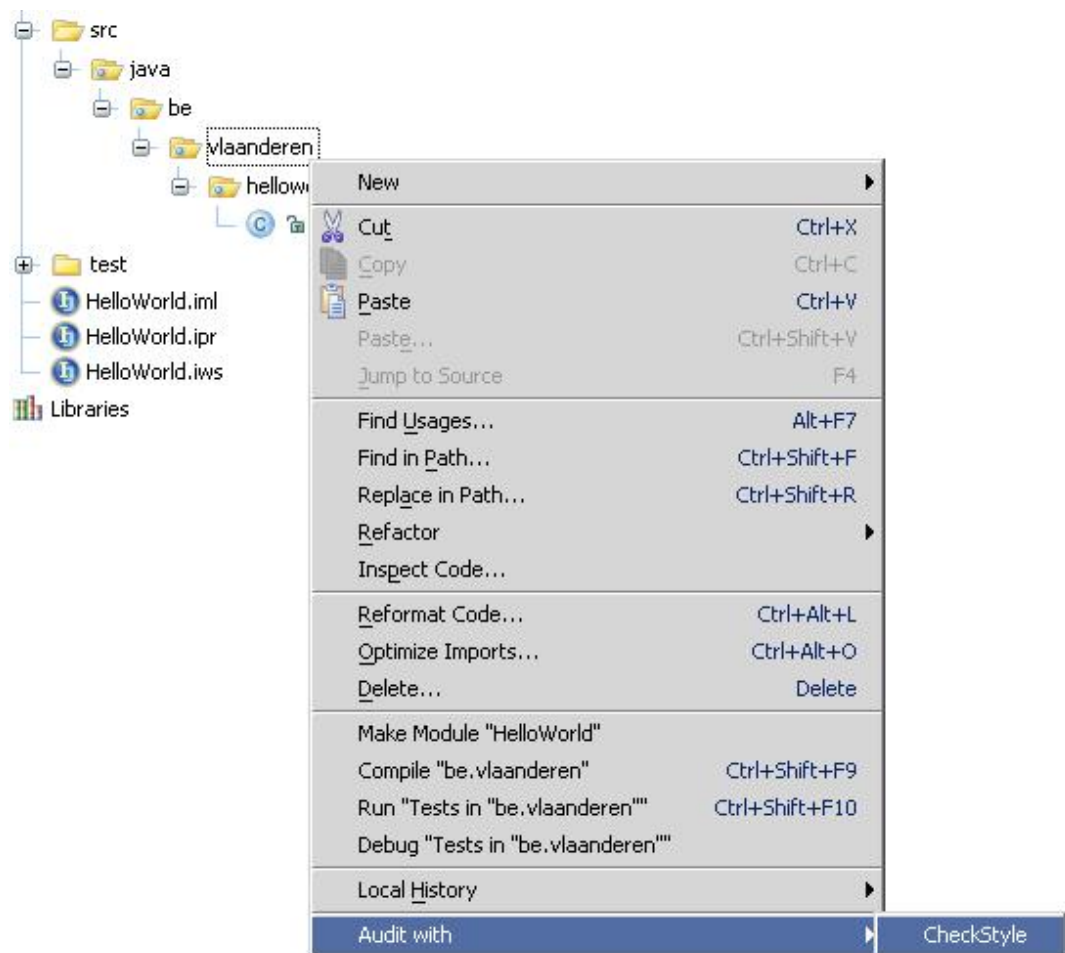


Name	FilterChecks
Description	Filter the CheckStyle output to link to the sources
Regular expression to match output	\$FILE_PATH\$: \$LINE\$

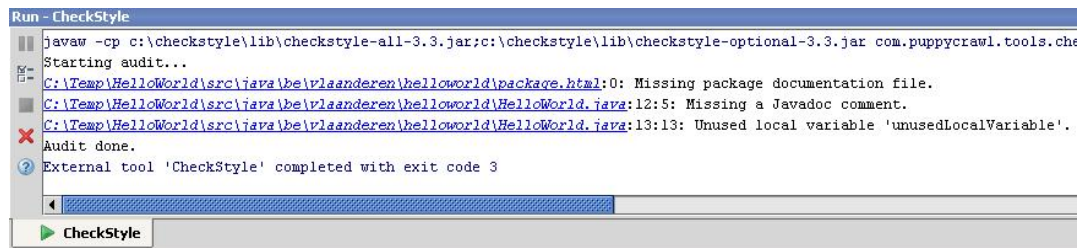
- d. Press the button **Ok** to close the *Add filter* window.
- e. Press the button **Ok** to close the *Output Filters* window.
7. Press the button **Ok** to close the *Edit Tool* window.
8. Press the button **Ok** to close the *External Tools* window.
9. Press the button **Close** to close the *Settings* window.

## Usage

1. Right click on a folder or file you want to audit.
2. Open the menu **Audit with, CheckStyle**.



3. The results of the audit are shown in the *Run* window at the bottom of the screen.



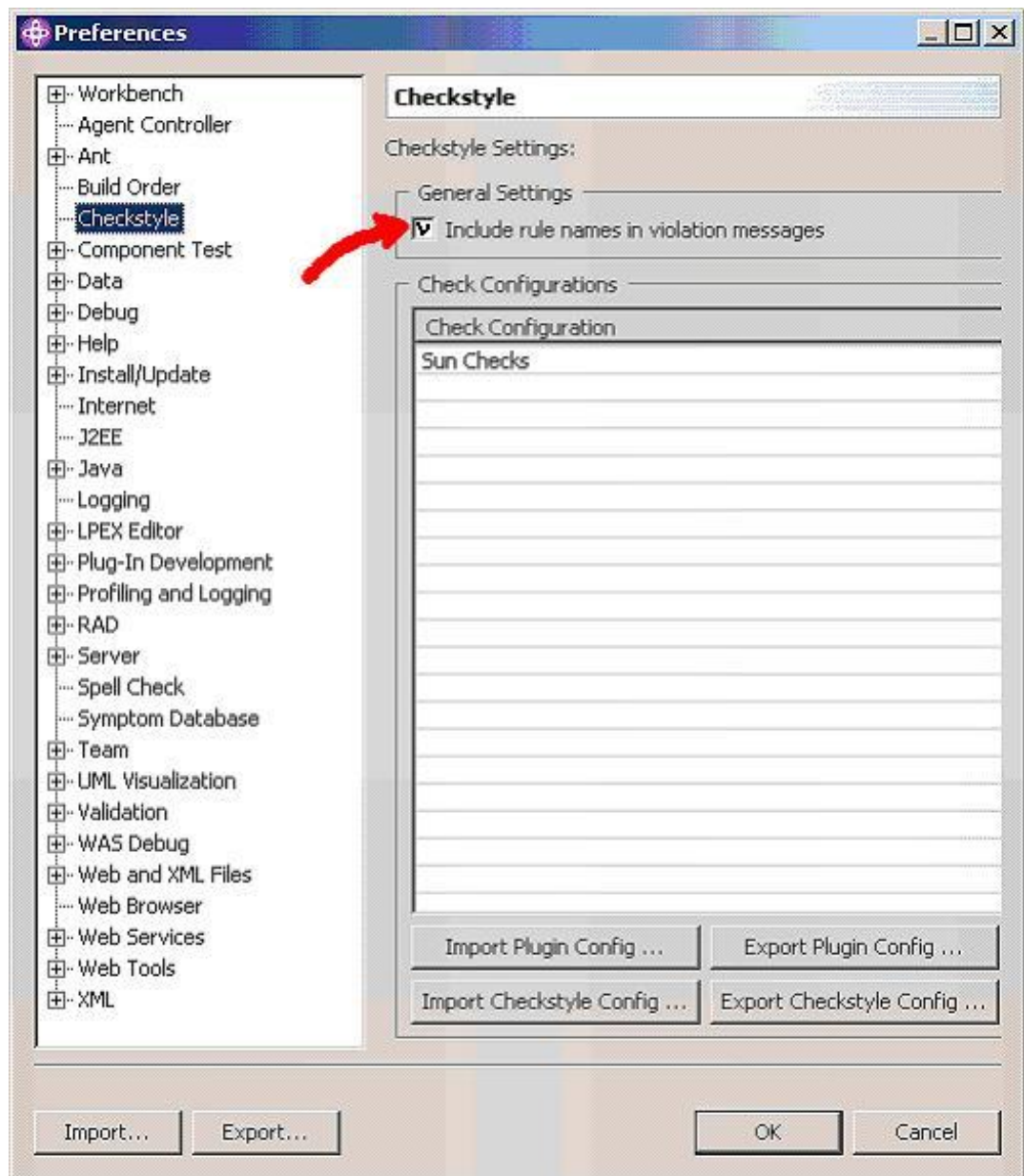
4. Click on a line to jump to the source code.

## Integration With Eclipse/WSAD

Eclipse is an open source java IDE which is very open ended. See <http://www.eclipse.org>. WSAD (WebSphere Studio Application Developer) is the core IBM development environment for creating and maintaining web services and J2EE applications. It also incorporates Eclipse. See <http://www-306.ibm.com/software/awdtools/studioappdev>. The Eclipse Checkstyle plugin is an open source project. See <http://eclipse-cs.sourceforge.net>.

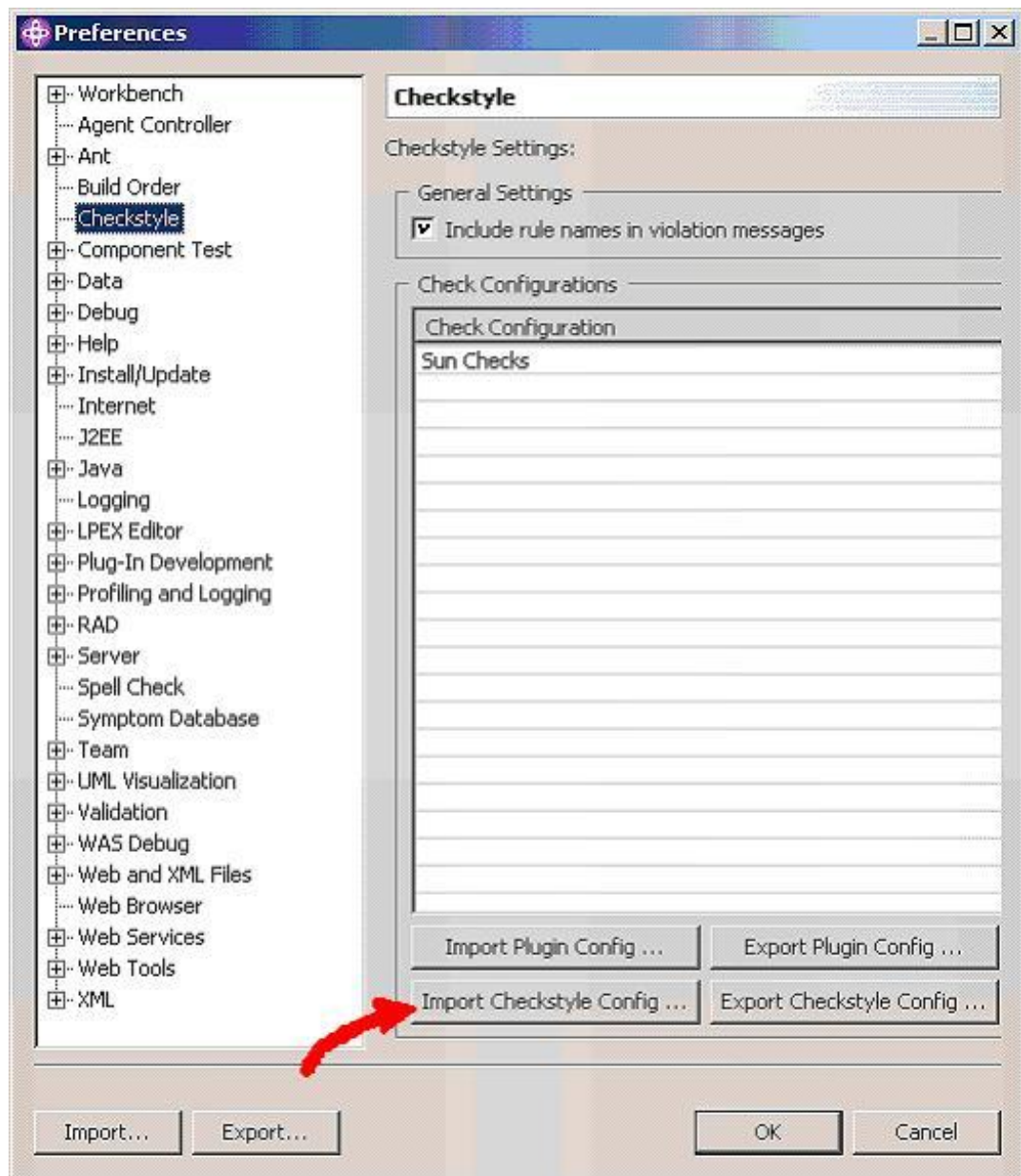
## Installation

1. Close Eclipse/WSAD if it is open.
2. Unzip the contents of `JJGuidelines-CheckStyle-Eclipse.zip` (which you can find in the directory `tools/checkstyle` as part of the JJGuidelines distribution) to an appropriate directory, for example `c:\checkstyle`.
3. Copy the unzipped directory `c:\checkstyle\com.atlassw.tools.eclipse.checkstyle_3.3.2.0` to the `plugins` directory of Eclipse.
4. Start Eclipse/WSAD.
5. Open the menu **Window, Preferences**.
6. Open the item **Checkstyle** in the tree and configure the CheckStyle plugin.

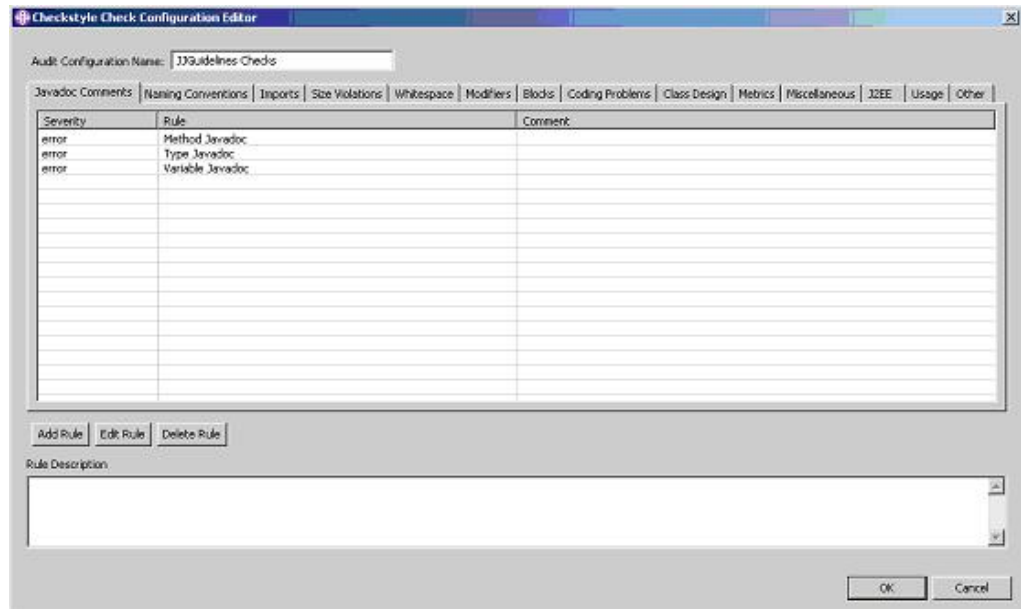


Enable Include rule names in violation messages.

7. Press the button **Import CheckStyle Configuration**.

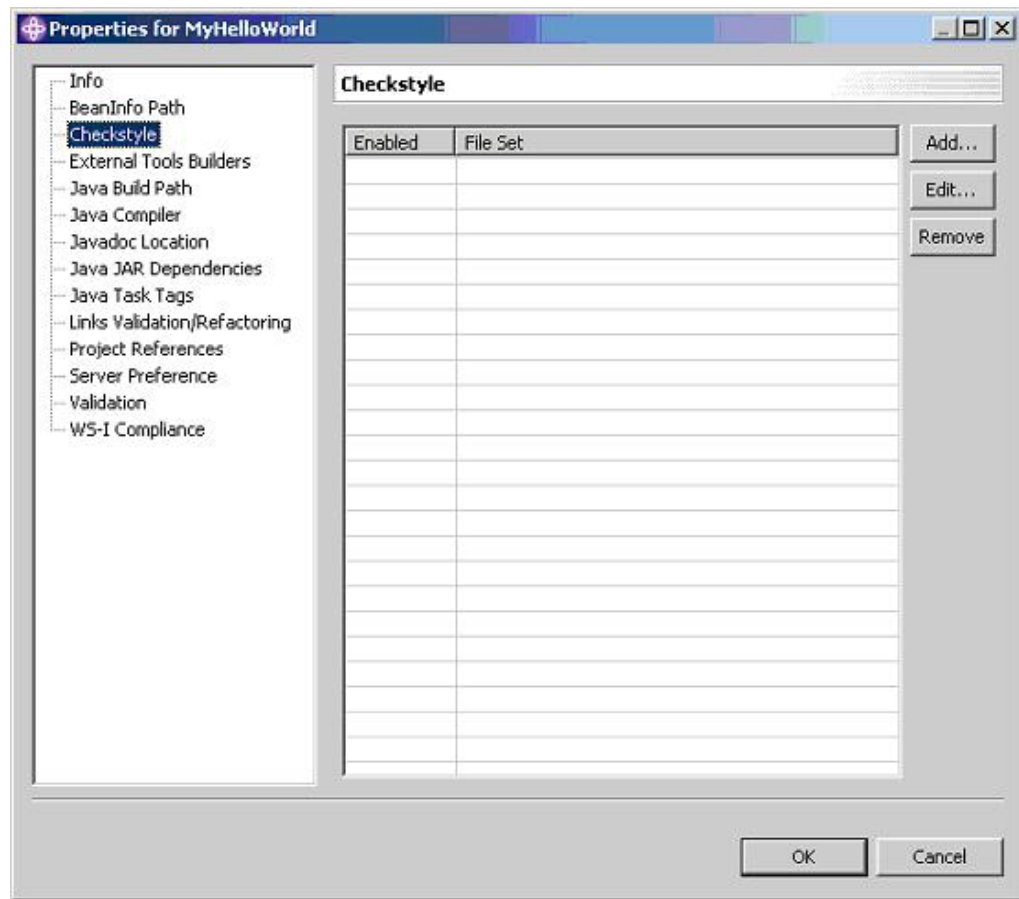


- a. Choose the `c:\checkstyle\conf\checkstyle-jjguidelines.xml` file you unzipped previously.
- b. Configure the rules.



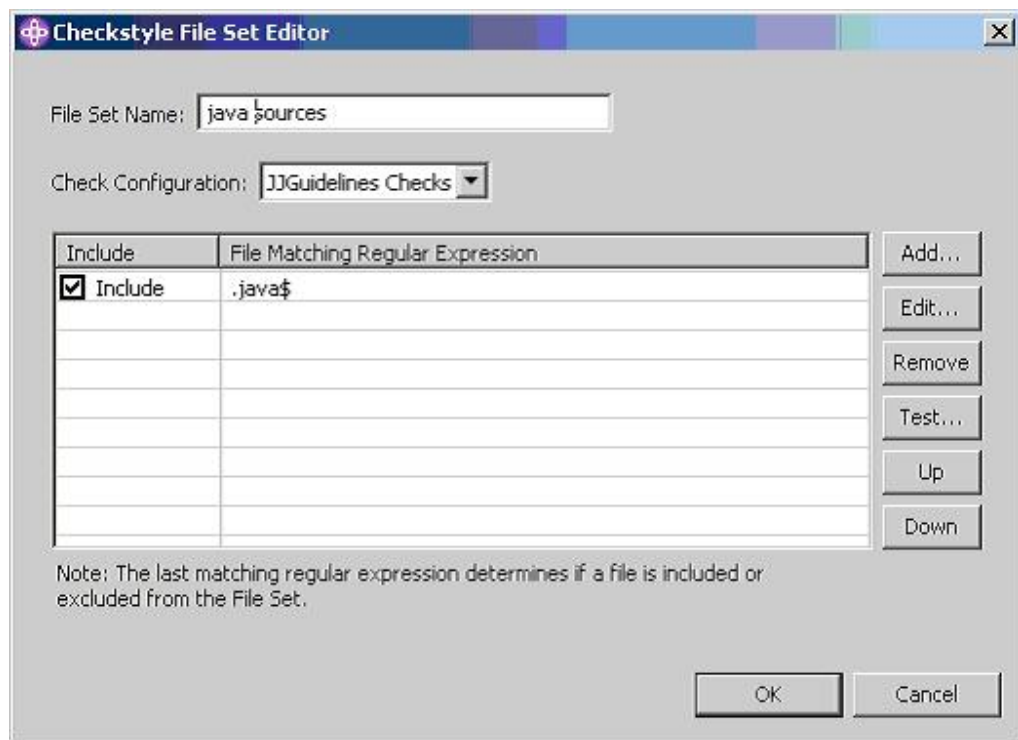
Audit Configuration Name	JJGuidelines	Checks
--------------------------	--------------	--------

- c. Press the button **Ok** to close the *Checkstyle Check Configuration Editor* window.
8. Press the button **Ok** to close the *Preferences* window.
9. Enable JJGuidelines CheckStyle for your project:
  - a. Open the menu **Project, Properties**.



- b. Open the item **Checkstyle** in the tree and configure the CheckStyle plugin.
- c. Press the button **add**.





File Set Name                      Java Sources

Check Configuration              JjGuidelines Checks

The default file set matches all . java source files.

- d. Press the button **Ok** to close the *Checkstyle File Set Editor* window.

10. Press the button **Ok** to close the *Properties for ...* window.



## Caution

The checkstyle release (3.3) included in the tools distribution has a JDK1.4 dependency which results in a `NoClassDefFoundException` for the `com.puppycrawl.tools.checkstyle.checks.usage.transmogrify.ASTManager` class. This causes problems in WSAD that uses a JRE1.3. To solve this problem you can either:

- Disable the 'usage' checks in the checkstyle configuration
- Download the latest checkstyle CVS version from SourceForge [[http://sourceforge.net/project/showfiles.php?group\\_id=29721](http://sourceforge.net/project/showfiles.php?group_id=29721)], build it and update the jar files in the checkstyle plugin directory.

## Usage

As soon as you enabled Checkstyle for your project, the JJGuidelines rules will be enforced on your project.

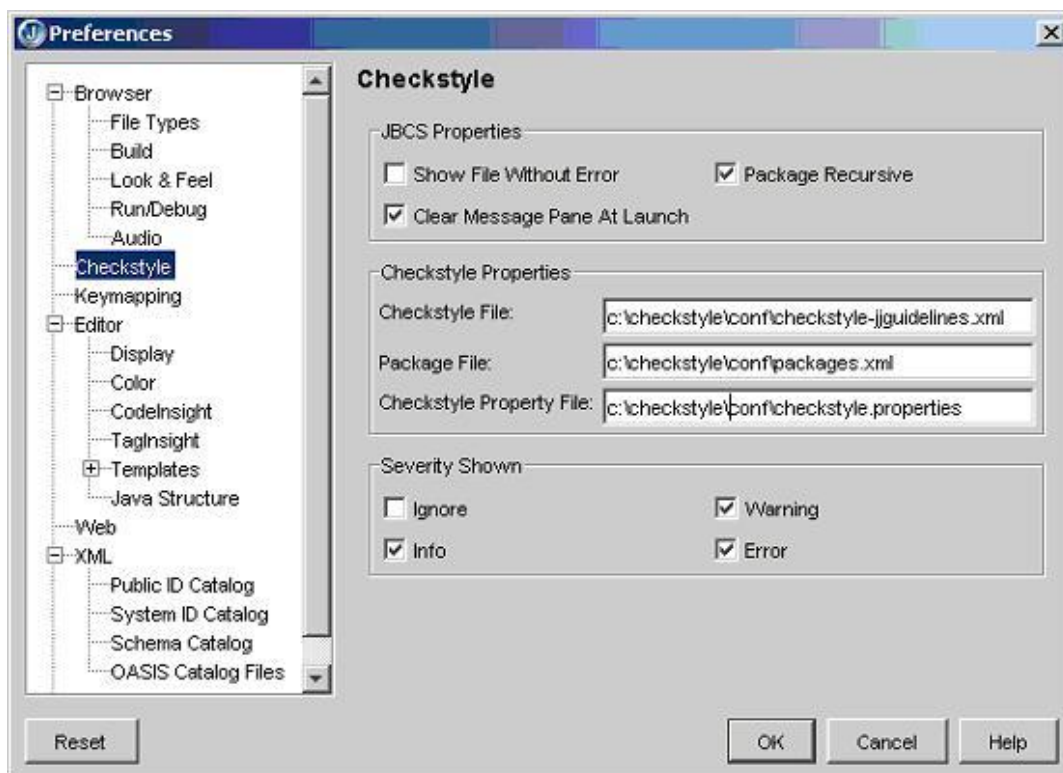


## Integration With JBuilder X

JBuilder is a popular java IDE from Borland with J2EE support. See <http://www.borland.com/jbuilder>. The JBuilder Checkstyle plugin is an open source project. See <http://sourceforge.net/projects/jbcheckstyle-pg>.

## Installation

1. Close JBuilder if it is open.
2. Unzip the contents of JJGuidelines-CheckStyle-JBuilderX.zip (which you can find in the directory `tools/checkstyle` as part of the JJGuidelines distribution) to an appropriate directory, for example `c:\checkstyle`.
3. Copy the contents of the `c:\checkstyle\lib` directory which you just unzipped to the `lib/ext` directory of your JBuilder installation.
4. Copy the contents of the `c:\checkstyle\home` directory you just unzipped to your home directory (On windows type `echo %USERPROFILE%` to find your home directory).
5. Start JBuilder.
6. Open the menu **Tools, Preferences**.
7. Open the item **Checkstyle** in the tree and configure the CheckStyle plugin in the *Preferences* window.



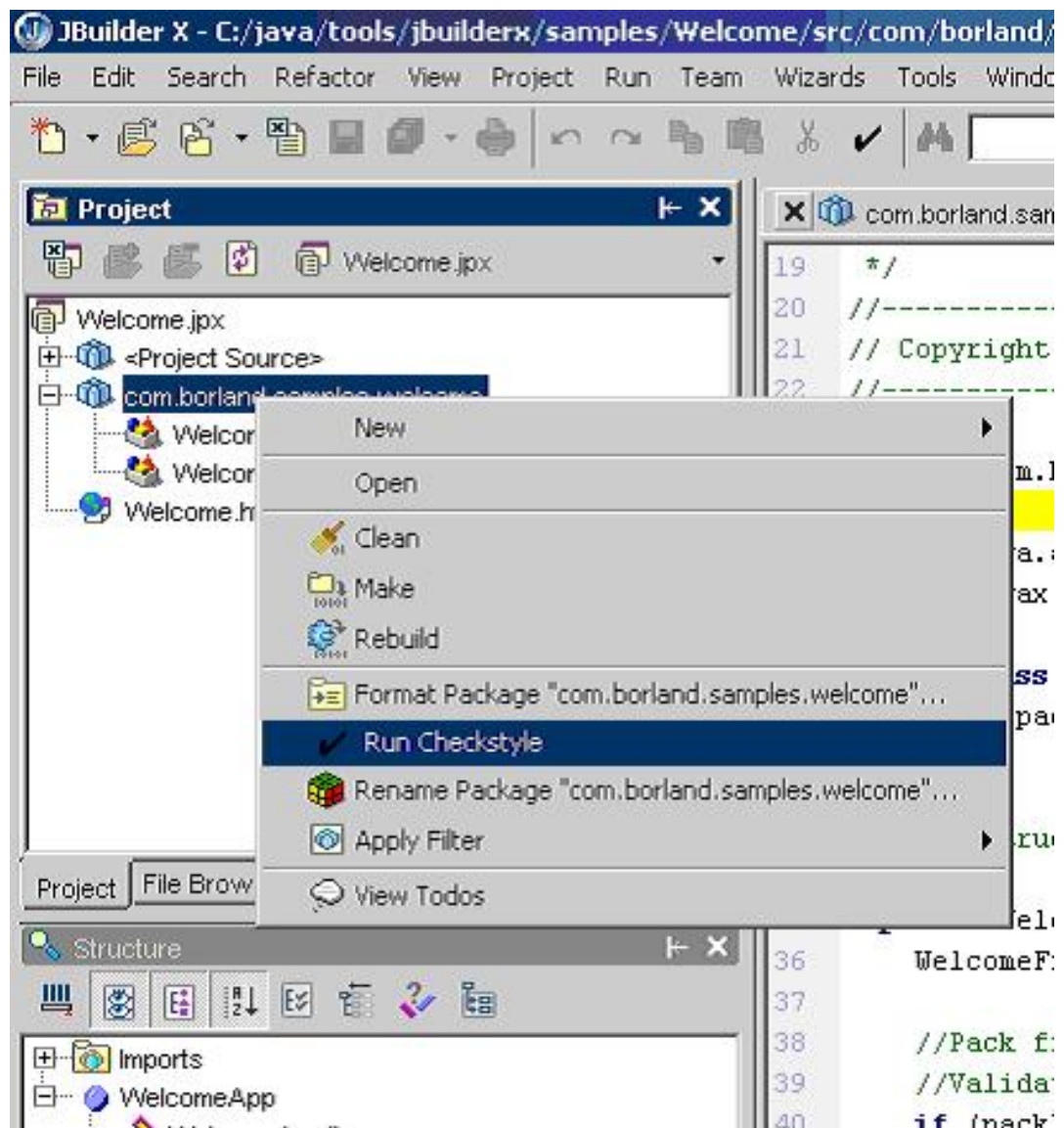
Checkstyle File	c:\checkstyle\conf\checkstyle-jjguidelines.xml
Package File	c:\checkstyle\conf\packages.xml
Checkstyle Property File	c:\checkstyle\conf\checkstyle.properties

Change the parameter arguments to match the directory where you unzipped the files.

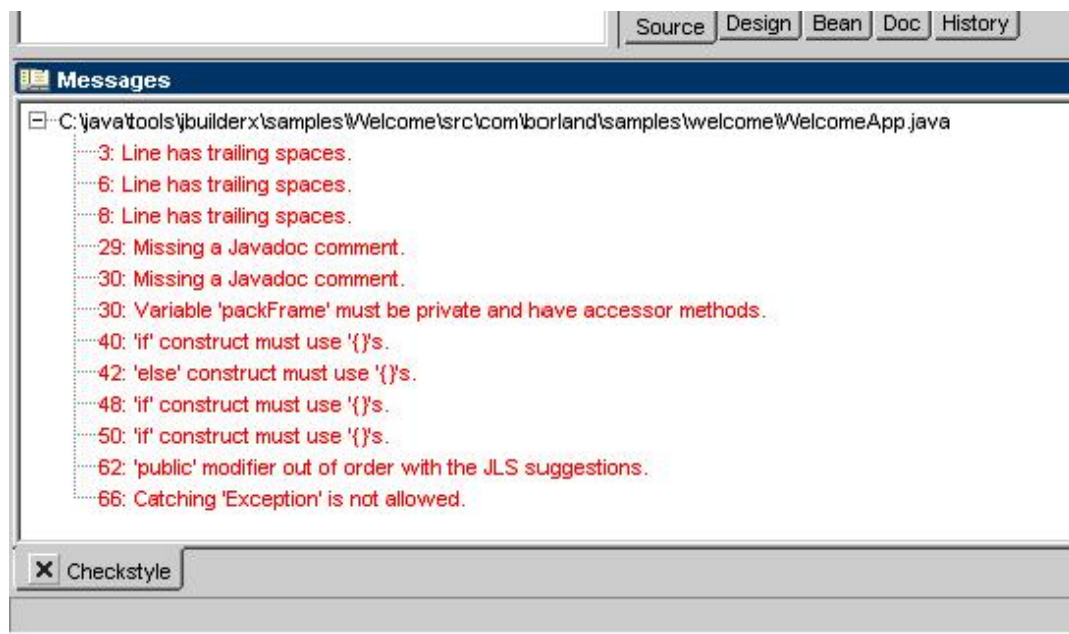
8. Press the button **Ok** to close the *Preferences* window.

## Usage

1. Right click on a folder or file you want to audit in your *Project* window.
2. Open the menu **Run Checkstyle**.



3. The results are shown in a Checkstyle tab in your *Messages* window.



## Integration With JDeveloper

Oracle JDeveloper is an integrated development environment with end-to-end support for modeling, developing, debugging, optimizing, and deploying Java applications and web services. See <http://otn.oracle.com/products/jdev>.

## Installation And Usage

There is currently no Checkstyle plugin available for JDeveloper. The JDeveloper tool architecture supports the creation of such a plugin but this has not yet been implemented for Checkstyle.

You can use the Ant support of JDeveloper to run CheckStyle using Ant. See the section called “Integration With ANT”.

---

# Glossary

Feedback [mailto:feedback@jjguidelines.dev.java.net?Subject=[JJGuidelines]glossary]

I've misused this glossary to group all the acronyms used in the Java world. With this list you'll (finally) be able to understand what other Senior Java and J2EE developers are saying to each other.



## Tip

For a complete Java and related terms glossary have a look at the <http://java.sun.com/docs/glossary.nonjava.html> web site, maintained by Sun Microsystems.

## A

### ACID

Acronym used for the four properties guaranteed by transactions: atomicity, consistency, isolation, and durability.

### API

Application Programming Interface. The specification of how a programmer writing an application accesses the behavior and state of classes and objects.

### APM

Application Programming Model. A programming model that defines how to use and combine the features of the J2EE platform to create solutions for common application domains in the enterprise.

### AVK

Application Verification Kit. A Sun Microsystems tool to verify if your J2EE implementation is compliant to the J2EE platform specification.  
See Also CTS.

### AWT

Abstract Windowing Toolkit. The very first collection of Java graphical user interface (GUI) components that were implemented using native-platform versions of the components. Today the Swing components (included in JFC) have mostly replaced the AWT usage.  
See Also JFC.

## B

### BMP

Bean Managed Persistency. This an Entity EJB that needs to take care of its own persistency code.  
See Also CMP, EJB.

## C

CMP

Container Managed Persistency. This an Entity EJB where the persistency strategy is enforced by the EJB container.  
See Also BMP, EJB.

CORBA

Common Object Request Broker Architecture. A language independent, distributed object model specified by the Object Management Group (OMG).  
See Also IIOP.

CTS

Compatibility Test Suite. A suite of compatibility tests for verifying that a J2EE product complies with the J2EE platform specification.  
See Also AVK.

## D

DAO

Data Access Object. This acronym is a J2EE pattern which is responsible for encapsulating all database access code.  
See Also TO.

DD

Deployment Descriptor. An XML file provided with each module and application that describes how they should be deployed. The deployment descriptor directs a deployment tool to deploy a module or application with specific container options and describes specific configuration requirements that a deployer must resolve.  
See Also XML.

DOM

Document Object Model. A tree of objects with interfaces for traversing the tree and writing an XML version of it, as defined by the W3C specification.  
See Also SAX.

DTD

Document Type Definition. A description of the structure and properties of a class of XML files.  
See Also XML.

## E

EAI

Enterprise Application Integration  
See Also SOA, MOM, JMS.

EIS

Enterprise Information System. Applications that provide an information infrastructure for an enterprise. Examples of EISs include: an ERP system, a mainframe transaction processing system, and a legacy database system.

EJB

## H

### HTTP

Enterprise Java Bean. A component architecture for the development and deployment of object-oriented, distributed, enterprise-level applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, and multi-user and secure. See Also J2EE.

HyperText Transfer Protocol. The Internet protocol, based on TCP/IP, used to fetch hypertext objects from remote hosts. See also TCP/IP.  
See Also URL, TCPIP.

## I

### I18N

Internationalization. I18N support through the use of uni-code allows Java developers to support other languages than English.

### IIOP

Internet Inter-ORB Protocol. A protocol used for communication between CORBA object request brokers.  
See Also CORBA.

## J

### J2EE

Java 2 Enterprise Edition. An environment for developing and deploying enterprise applications. The J2EE platform consists of a set of services, application programming interfaces (APIs), and protocols that provide the functionality for developing multi-tiered, Web-based applications.  
See Also J2ME, J2SE, JDK.

### J2ME

Java 2 Micro Edition Platform. Is the Java platform for consumer and embedded devices such as Java enabled mobile phones, PDAs, TV set-top boxes, in-vehicle telematics systems, and other embedded devices.  
See Also J2EE, J2SE.

### J2SE

Java 2 Standard Edition. The core Java technology platform.  
See Also J2ME, J2EE, JDK.

### JAR

Java Archive is a platform-independent file format that aggregates and compresses many files into one. Similar to ZIP and TAR.

### JAX

Java API for XML. A collection of APIs for handling XML data.



See Also XML.

JCA

J2EE Connector Architecture. The J2EE Connector architecture defines a standard architecture for connecting the J2EE platform to heterogeneous EIS's.

See Also J2EE.

JCP

Java Community Process. It's an open organization of international Java developers and licensees who develop and revise Java technology specifications, reference implementations, and technology compatibility kits.

See Also JSR.

JDBC

Java Database Connectivity API. An industry standard for database-independent connectivity between the Java platform and a wide range of relational databases.

See Also J2EE, DAO.

JDK

Java Development Kit. A free and open software development environment for writing applets and applications in the Java programming language.

See Also JRE.

JFC

Java Foundation Classes. An extension that adds graphical user interface class libraries to the Abstract Windowing Toolkit (AWT).

See Also AWT.

JIT

Just-in-time compiler. A compiler that converts all of the byte code into native machine code during the execution of a Java program.

See Also JVM.

JMS

Java Messaging Service. An API for implementing a-synchronous message oriented clients.

See Also SOA, MOM, EAI.

JMX

Java Management Extension

See Also J2EE.

JNDI

Java Naming Directory Interface. An API that assists with the interfacing to multiple naming and directory services.

JNLP

Java Network Launch Protocol. An XML file used by to launch and configure the Java WebStart client.

JRE

Java Runtime Environment. A subset of the Java Developer Kit for end-users and developers who want to redistribute the runtime environment.

See Also JDK.

JSP	JavaServer Pages. An extensible Web technology that uses template data, custom elements, scripting languages, and server-side Java objects to return dynamic content to a client.
JSR	Java Specification Request. Java Specification Requests are the actual descriptions of proposed and final specifications for the Java platform. See Also JCP.
JTA	Java Transaction API. An API that allows applications and J2EE servers to access transactions. See Also JTS.
JTS	Java Transaction Service. Specifies the implementation of a transaction manager which supports JTA and implements the Java mapping of the OMG Object Transaction Service (OTS) 1.1 specification at the level below the API. See Also JTA.
JVM	Java Virtual Machine. A software engine that safely and compatibly executes the Java byte code on a micro-processor.

## M

MDB	Message Driven Bean. A message driven enterprise bean is an asynchronous consumer that can listen to a JMS topic or queue but lives in the EJB container. See Also JMS.
MOM	Message Oriented Architecture. A message oriented architecture is client/server infrastructure which is loosely coupled and has guaranteed message delivery. The JMS API is used to develop such systems. See Also JMS, SOA.
MVC	Model View Controller. The most important pattern to introduce separation of concerns. Separate the GUI (View) from the business logic (Controller) and the data (Model).

## P

POJO	Plain Old Java Object. Martin Fowler introduced this acronym as a competitor against EJBs. See Also EJB.
------	---

## R

### RMI

Remote Method Invocation. A distributed object model for Java programs in which the methods of remote objects can be invoked from other Java virtual machines, possibly on different virtual machines. See Also IIOP.

## S

### SAX

Simple API for XML. SAX is a common, event-based API for parsing XML documents. See Also DOM, XML.

### SOA

Service Oriented Architecture. An architecture based on a systems environment specifically architected to leverage free-standing units of functional code, each of which corresponds to a specific service. See Also MOM, JMS.

### SQL

Structured Query Language. The standardized relational database language for defining database objects and manipulating data. See Also JDBC, DAO.

## T

### TCPIP

Transmission Control Protocol based on IP. This is an Internet protocol that provides for the reliable delivery of streams of data from one host to another. See Also HTTP.

### TO

Transfer Object. Transfer Object is a J2EE pattern name, previously known as Value Object, which allows model information to be transferred between different J2EE tiers. See Also DAO.

## U

### URI

Uniform Resource Identifier. A compact string of characters for identifying an abstract or physical resource. A URI is either a URL or a URI. URLs and URIs are concrete entities that actually exist; A URI is an abstract super class. See Also URL.

### URL

Uniform Resource Locator. A standard for writing a text reference

to an arbitrary piece of data in the WWW. A URL looks like "protocol://host/localinfo" where protocol specifies a protocol to use to fetch the object (like HTTP or FTP), host specifies the Internet name of the host on which to find it, and localinfo is a string (often a file name) passed to the protocol handler on the remote host.

See Also HTTP, URI.

## X

### XML

Extensible Markup Language. A markup language that allows you to define the tags (markup) needed to identify the data and text in XML documents.

See Also JAX.

### X-Path

XML Path Language. X-Path is an expression language used by XSLT to access or refer to parts of an XML document.

See Also XSLT.

### XSL

eXtensible Style Language. XSL is a family of recommendations for defining XML document transformation and presentation, it consists of XSLT, XPath and XSL-FO.

See Also XSLT, XSL-FO.

### XSL-FO

eXtensible Style Language Formatting Objects. XSL-FO is an XML vocabulary for specifying formatting semantics.

### XSLT

eXtensible Style Language Transformation. XSLT is a language for transforming XML to other formats like HTML, WML etc.

---

# Bibliography

Feedback [mailto:feedback@jguidelines.dev.java.net?Subject=[JJGuidelines]bibliography]

[JLANGSPEC] James Gosling, Bill Joy, Guy L. Steele Jr., and Gilad Bracha. 0201310082. Addison Wesley. *The Java Language Specification*. Second Edition. 1-532.  
<http://java.sun.com/docs/books/jls/index.html>.

[MARTINDP] Martin Fowler. 0-321-12742-0. Addison Wesley. *Patterns of Enterprise Application Architecture*. 1-533.

[J2EDP] Deepak Alur, John Crupi, and Dan Malks. 0-13-142246-4. Prentice Hall. *Core J2EE Patterns*. Best Practices and Design Strategies, Second Edition. 1-650.  
<http://www.corej2eepatterns.com>.

[EFFECJ] Joshua Bloch. 0-201-310054. Addison Wesley. *Effective Java*. Programming Language Guide. 1-252. <http://java.sun.com/books/Series>.

[PRACTJ] Peter Hagggar. 0-201616-46-7. Addison Wesley. March, 2000. *Practical Java*. Programming Language Guide. March, 2000. 1-279. <http://www.aw.com/cseng>.

[STRACTION] Ted Husted, Cedric Dumoulin, George Franciscus, and David Winterfeldt. 1-930110-50-2. Manning. *Struts in action*. Building web applications with the leading Java framework. 1-630. <http://www.manning.com/husted>.

[THINKING] Bruce Eckel. 0-131002-87-2. Prentice Hall. *Thinking in Java, 3rd edition*. The definitive introduction to Object-Orientated programming in the language of the world-wide web. 1-1119.  
<http://www.bruceeckel.com>.

[DSGJ2EE] Inderjeet Singh, Beth Stearns, and Mark Johnson. 0-201-78790-3. Addison Wesley. *Designing Enterprise Applications with the J2EE Platform*. 2nd Edition. 1-352.  
[http://java.sun.com/blueprints/guidelines/designing\\_enterprise\\_applications\\_2e/titlepage.html](http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/titlepage.html).

[JDOOREIL] David Jordan and Craig Russel. 0-596-00276-9. O'Reilly. *Java Data Objects*. Store objects with ease. 1-356. <http://www.oreilly.com/catalog/jvadaobj>.

[BESTPRAC] Robert Eckstein and The O'Reilly Java Authors. 0-596-00384-6. O'Reilly. *Java Enterprise Best Practices*. 1-288. <http://www.oreilly.com/catalog/javaebp/>.

[AMBY] Scott W. Ambler. *Writing Robust Java Code*. The AmbySoft Inc. Coding Standards for Java v17.01d. 1-76. <http://www.ambysoft.com/javaCodingStandards.pdf>.

[J2EE14TUT] Eric Armstrong, Stephanie Bodofoff, Debbie Carson, Ian Evans, Maydenne Fisher, Dale Green, Kim Haase, Eric Jendrock, Monica Pawlan, and Beth Stearns. May 30, 2003. *The J2EE 1.4 Tutorial*. 1-902. <http://java.sun.com/j2ee/1.4/docs/tutorial/doc/>.

[J2EE13SPEC] Bill Shannon. *Java™ 2 Platform Enterprise Edition Specification, v1.3*. July 27, 2001. 1-174. <http://java.sun.com/j2ee/1.3/docs/>.

[EJB20SPEC] Linda G. DeMichiel, L. Ümit Yalçinalp, and Sanjeev Krishnan. *Enterprise JavaBeans™ Specification, Version 2.0*. August 14, 2001. 1-572. <http://java.sun.com/products/ejb/docs.html>.

[JSPSPEC] Danny Coward. *Java™ Servlet 2.3 and JavaServer Pages™ 1.2 Specification*. August 13, 2001. 1-257. <http://jcp.org/aboutJava/communityprocess/first/jsr053/index.html>.

[ASSERT] Paul Rogers. *J2SE 1.4 premieres Java's assertion capabilities*. Understand the mechanics of Java's new assertion facility. November 9, 2001. 6100 words.  
<http://www.javaworld.com/javaworld/jw-11-2001/jw-1109-assert.html>.

- [MODEL2] Govind Seshadri. *Understanding JavaServer Pages Model 2 architecture*. Exploring the MVC design pattern. December, 1999. 2000 words.  
<http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html>.
- [EXCEPT] Jeff Friesen. *Exceptions to the programming rules*. Incorporate Java's throw-object/catch-object exception-handling technique into your Java programs. April 5, 2002. 5775 words. <http://www.javaworld.com/javaworld/jw-04-2002/jw-0405-java101.html>.  
<http://www.javaworld.com/javaworld/jw-04-2002/jw-0405-java101.html>.
- [JMX2] J. Steven Perry. 0596002459. O'Reilly. *Java Management Extensions*. June 27, 2002. 312 pages.
- [ARCHJMX] Marc Fleury and Juha Lindfors. *Enabling Component Architectures with JMX*. February 1, 2001. 1 page. <http://www.onjava.com/pub/a/onjava/2001/02/01/jmx.html>.
- [J2EEJMX] Tony G. Thomas. *JMX Boosts J2EE Application Management*. 2002. 11 pages.  
[http://download.adventnet.com/products/manageengine/j2ee\\_application\\_management.pdf](http://download.adventnet.com/products/manageengine/j2ee_application_management.pdf).
- [JAVAJMX] Heather Kreger, Ward Harold, and Leigh Williamson. 0-672-32408-3. The Java Series, Addison Wesley. *Java™ and JMX: Building Manageable Systems*.  
<http://java.sun.com/products/JavaManagement/>.
- [JUNITACT] Vincent Massol. 1930110995. Manning. *JUnit in Action*. October 1, 2003. 384 pages.
- [REFACTOR] Martin Fowler. 0201485672. Addison Wesley. *Refactoring*. Improving the Design of Existing Code. July 1999. 464 pages.
- [XPINTRO] Kent Beck. 0201616416. Addison Wesley. *Extreme Programming Explained*. Embracing Change. October 1999. 224 pages.
- [J2EEGUIDE] Sun Microsystems. *J2EE Developers Guide*.  
[http://java.sun.com/j2ee/sdk\\_1.2.1/techdocs/guides/ejb/html/DevGuideTOC.html](http://java.sun.com/j2ee/sdk_1.2.1/techdocs/guides/ejb/html/DevGuideTOC.html).
- [JMXSITE] Sun Microsystems. *JMX*. <http://java.sun.com/products/JavaManagement/>.
- [JMXSPEC] Sun Microsystems. *J2EE Management reference*.  
<http://java.sun.com/j2ee/tools/management/reference/docs/index.html>.