Java Coding Standard

Arnaud Blandin [blandin@intalio.com]

Contributions

Assaf Arkin [arkin@intalio.com]
Keith Visco [kvisco@intalio.com]

Revision: **October 18, 2000**

Note to the reader

Before complaining about this document, please read the following:

Whenever possible, use the conventions from this document; do not reinvent them. Keep in mind that projects (whether it is a software project or not) are not developed in a vacuum and organizations do not work in a vacuum either.

A standard is never perfect and can not fit all situations: it is be possible that one day you find yourself in a situation where none of these conventions could be applied. Thus, you might want to break the conventions to fit your particular situation; if you do so, **document it**.
If you go against a standard, you must document why you broke it and the potential implications of breaking it.

The bottom line is that you need to understand each standard and understand fully when to apply it (and also when not to apply it).

1 Introduction and Contents

1.1 Introduction

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

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

1.2 Contents

Structure and Documentation

Standard ways to write and document constructions.

Naming conventions

Standard ways to name identifiers (class names, method names, variable names, etc).

Recommendations

Some rules of thumb that tend to avoid common errors and development obstacles. You can use these guidelines to make your own design and coding checklists to be used in retrospective code clean-up or when classes need to be used in new contexts or placed in reusable libraries.
But please do not forget to follow the last one.

Code Examples

'Example.java' uses the code conventions of this document. 'Package.html' outlines the purpose of a package.

Related Documents

References to other style guidelines etc.

2 Structure and Documentation

2.1 Packages

Create a new java `package` for each self-contained project or group of related functionality. Create and use directories in accord with java `package` conventions.

Consider writing a `package.html` file in each directory briefly outlining the purpose and structure of the package.

2.2 Program Files

Place each class in a separate file. This applies even to non-public classes (which are allowed by the Java compiler to be placed in the same file as the main class using them) except in the case of one-shot usages where the non-public class cannot conceivably be used outside of its context.

Begin each file with a comment including:

1. The copyright. (http://www.exolab.org/license.html)
2. The 'Id' tag
3. A history table listing dates, authors, and summaries of changes.
4. If the file contains more than one class, list the classes, along with a very brief description of each.
5. If the file introduces a principal entry point for a `package`, briefly describe the rationale for constructing the package.

Immediately follow each file header with:

- The `package` name
- Empty line
- The `import` list.

Example:

```
/**
  * Redistribution and use of this software and associated
  * documentation
  * ("Software"), with or without modification, are permitted provided
  * that the following conditions are met:
  *
  * 1. Redistributions of source code must retain copyright
  *    statements and notices.  Redistributions must also contain a
  *    copy of this document.
  *
  * 2. Redistributions in binary form must reproduce the
  *    above copyright notice, this list of conditions and the
  *    following disclaimer in the documentation and/or other
  *    materials provided with the distribution.
  *
  * 3. The name "Exolab" must not be used to endorse or promote
  *    products derived from this Software without prior written
  *    permission of Exoffice Technologies.  For written permission,
  *    please contact info@exolab.org.
  *
  * 4. Products derived from this Software may not be called "Exolab"
  *    nor may "Exolab" appear in their names without prior written
  *    permission of Exoffice Technologies. Exolab is a registered
  *    trademark of Exoffice Technologies.
  *
  * 5. Due credit should be given to the Exolab Project
  *    (http://www.exolab.org/).
  *
  * THIS SOFTWARE IS PROVIDED BY EXOFFICE TECHNOLOGIES AND CONTRIBUTORS
  * ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT
  * NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
  * FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL
  * EXOFFICE TECHNOLOGIES OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT,
  * INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
  * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
  * SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
  * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
  * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
  * OF THE POSSIBILITY OF SUCH DAMAGE.
  *
```

```
 * Copyright 1999 (C) Exoffice Technologies Inc. All Rights Reserved.
 * $Id: Example.java ,v 1.1 2000/08/03 22:11:13 authorname Exp $
 * Date       Author         Changes
 * Aug 2 2000 author-name    Created
 * Aug 2 2000 author-name    Added new methods
 */


package demo;

import java.util.NoSuchElementException;
```

2.3 Classes and Interfaces

Write all `/** ... */` comments using *javadoc* conventions.

Preface each class with a `/** ... */` comment describing the purpose of the class, guaranteed invariants, usage instructions, and/or usage examples. Also include any reminders or disclaimers about required or desired improvements. Use HTML format, with added tags:

- `@author` *author-name*
- `@version` $Revision $ $Date$ (CVS will expand automatically)
- `@see` *string*
- `@see` *URL*
- `@see` *classname#methodname*

Example:

```
/**
 * A class representing a window on the screen.
 * For example:
 * <pre>
 *    Window win = new Window(parent);
 *    win.show();
 * </pre>
 *
```

```
 *
 * @author <a href="mailto:clown@exolab.org">Bozo the Clown</a>
 * @version $Revision: 1.4 $ $Date: 2000/02/29 22:11:13 $
 * @see          awt.BaseWindow
 * @see          awt.Button
 */

class Window extends BaseWindow {
      ...
}
```

Declaration order:

- member variables
- class constructors
- public methods
- protected and private methods
- inner classes when necessary (see 2.2)

2.4 Class Variables

First declare `public` variables, then `protected` ones , then `package` level ones and then the `private` ones.
Use *javadoc* conventions to describe nature, purpose, constraints, and usage of instances variables and static variables. Use HTML format, with added tags:

- `@see` *string*
- `@see` *URL*
- `@see` *classname#methodname*

Example:

```
    /**
     * The current number of elements.
     * must be non-negative, and less than or equal to capacity.
     */
    protected int _count;
```

2.5 Methods

Use *javadoc* conventions to describe nature, purpose, preconditions, effects, algorithmic notes, usage instructions, reminders, etc. Use HTML format, with added tags:

- `@param` *paramName* description.
- `@return` description of return value
- `@throws` *exceptionName* description (why and when)
- `@see` *string*
- `@see` *URL*
- `@see` *classname#methodname*

Be as precise as reasonably possible in documenting effects.

2.6 Local declarations, statements, and expressions

Use `/* ... */` comments to describe algorithmic details, notes, and related documentation that spans more than a few code statements.
You can also use // (see below).
Do not forget to document **why** something is being done, not just what. With some time every one can figure out what your code is doing but can hardly say why you choose to do that.

Example:

```
/*
 * Strategy:
 *    1. Find the node
 *    2. Clone it
 *    3. Ask inserter to add clone
 *    4. If successful, delete node
 */
```

Use running `//` comments to clarify non-obvious code. But don't bother adding such comments to obvious code; instead try to make code obvious!

Example:

```
int index = -1; // -1 serves as flag meaning the index isn't valid
```

Or, often better:

```
static final int INVALID = -1;
int index = INVALID;
```

2.7 Layout

- ### Indentation

  Avoid the 'Tab' key; four spaces should be used as the unit of indentation.
  Note: most editors have an option to convert tabs to spaces.

- ### Line Length

  Lines up to 70 characters are easier to read and print.
  Note: this is only a recommendation, lines can be longer, just don't abuse it.
  So think about it when writing comments.

- ### Left and Right braces

  At the beginning of classes, the left-brace ('{') could be placed at the beginning of a new line or at the end of the naming line.
  For internal code, it should be put at the end of the line.

  Right-brace ('}') starts a line by itself intended to match its corresponding opening statement, except when it is a null statement the '}' should appear immediately after the '{'.
  Try to comment a right-brace that closes a 10-line or more method.

- Put the 'extend' line under the 'naming' line if the 'extend line is too long.

Examples:

```
class Example1
       extends Class1,Class2,Class3,Class4,Class5
{
    if (condition) {
        ...
     } else {
       ...
    }

} //--Example1
```

```
class Example2 extends Class1 {

        ...
        if (condition) {
            ...
        }

         else {
           ...
        }


    } //--Example2
```

- A blank line should separate methods.
- Declare all class variables at the top of the class.
- Declare all method variables at the top of the method (except for loop initializers or local loop variables)

2.8 Blank Spaces

Blank spaces should be used in the following circumstances:

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

  Example:

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

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

- A blank space should appear after commas in parenthesis.

  Example:

  ```
  public void foo(int a, int b, string c) {

  }
  ```

  ```
  for (expr1; expr2; expr3) {
      ...
  }
  ```

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

  Examples:

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

  while (condition) {
      n++;
  }
  printSize("size is " + foo + "\n");
  ```

- Casts **should** be followed by a blank space (this is only a recommendation).

  Examples:

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

- Concerning the blank space in the naming line of a method, do as you want:

  Examples:

```
public foo(int i, byte b){
}
public foo(int i, byte b) {
}
public foo (int i, byte b){
}
public foo (int i, byte b) {
}
```

2.9 Wrapping Lines

When an expression does not fit on a single line, break it in order to make your code as readable as possible.
Here are some simple principles you can follow:

- Break after a comma
- Break before an operator
- Prefer high-level breaks to lower-level breaks
- Align the new line with the beginning of the expression at the same level on the previous line.

  Examples:

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

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

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

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

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

3 Naming Conventions

packages

`lowercase`

Consider using the recommended domain-based conventions described in the *Java Language Specification*, page 107 as prefixes.

files

The java compiler enforces the convention that file names have the same base name as the public class they define.

classes

`CapitalizedWithInternalWordsAlsoCapitalized`

class (When necessary to distinguish from similarly named interfaces)**:**

`ClassNameEndsWithImpl` OR
`ClassNameEndsWithObject`

exception class

`ClassNameEndsWithException`.

constants (finals)

`UPPERCASE_WITH_UNDERSCORE_IF_MORE_THAN_ONE_WORD`

private or protected

`_firstWordLowerCaseWithUnderscoreAndInternalWordsCapitalized`

local variables

`firstWordLowerCaseButInternalWordsCapitalized`

methods

`firstWordLowerCaseButInternalWordsCapitalized()`

factory method for objects of type X:

```
newX OR
createX
```

converter method that returns objects of type X

```
toX
```

method that reports an attribute x of type X

```
X getX()
```

method that changes an attribute x of type X

```
void setX(X value)
```

4 Recommendations

4.1 Classes

- When sensible, consider writing a `main` for the principal class in each program file. The `main` should provide a simple unit test or demo.

  Rationale: Forms a basis for testing. Also provides usage examples.

- For self-standing application programs, the class with `main` should be separate from those containing normal classes.

  Rationale: Hard-wiring an application program in one of its component class files hinders reuse.

- Consider writing source code templates for the most common kinds of class files you create: Applets, library classes, application classes.

  Rationale: Simplifies conformance to coding standards.

- If you can conceive of someone else implementing a class's functionality differently, define an interface, not an abstract class. Generally, use abstract classes only when they are ``partially abstract''; i.e., they implement some functionality that must be shared across all subclasses.

  Rationale: Interfaces are more flexible than abstract classes. They support multiple inheritances and can be used as `mixins' in otherwise unrelated classes.

- Consider whether any class should implement `Cloneable` and/or `Serializable`.

  Rationale: These are ``magic'' interfaces in Java, that automatically add possibly-needed functionality only if so requested.

- Declare a class as `final` only if it is a subclass or implementation of a class or interface declaring all of its non-implementation-specific methods. (And similarly for final methods).

  Rationale: Making a class final means that no one ever has a chance to reimplement functionality. Defining it instead to be a subclass of a base that is not final means that someone at least gets a chance to subclass the base with an alternate implementation. Which will essentially always happen in the long run.

- Whenever reasonable, define a default (no-argument) constructor so objects can be created via Class.newInstance().

  Rationale: This allows classes of types unknown at compile time to be dynamically loaded and instantiated (as is done for example when loading unknown Applets from html pages).

4.2 Variables

- Never declare instance variables as public(that does not include constant variables (final static)).

  Rationale: The standard OO reasons. Making variables public gives up control over internal class structure. Also, methods cannot assume that variables have valid values.

- Minimize reliance on implicit initializers for instance variables (such as the fact that reference variables are initialized to `null`).

  Rationale: Minimizes initialization errors.

- Minimize statics (except for static final constants).

  Rationale: Static variables act like globals in non-OO languages. They make methods more context-dependent, hide possible side-effects, sometimes present synchronized access problems. and are the source of fragile, non-extensible constructions. Also, neither static variables nor methods are overridable in any useful sense in subclasses.

- Generally prefer `double` to `float` and use `int` for compatibility with standard Java constructs and classes (for the major example, array indexing, and all of the things this implies, for example about maximum sizes of arrays, etc).

  Rationale: fewer precision problems occur with `doubles` than `floats`. On the other hand, because of limitations in Java atomicity guarantees, use of `double(respectively int)` must be synchronized in cases where use of `floats(long)` sometimes would not be.

- Use `final` and/or comment conventions to indicate whether instance variables that never have their values changed after construction are intended to be constant (immutable) for the lifetime of the object (versus those that just so happen not to get assigned in a class, but could in a subclass).

  Rationale: Access to immutable instance variables generally does not require any synchronization control, but others generally do.

- Avoid unnecessary instance variable access and update methods. Write `get/set`-style methods only when they are intrinsic aspects of functionality. Always write these methods when coding an interface.

  Rationale: Most instance variables in most classes must maintain values that are dependent on those of other instance variables. Allowing them to be read or written in isolation makes it harder to ensure that consistent sets of values are always used.

- Minimize direct internal access to instance variables inside methods. Use `protected` access and update methods instead (or sometimes `public` ones if they exist anyway).

  Rationale: While inconvenient and sometimes overkill, this allows you to vary synchronization and notification policies associated with variable access and change in the class and/or its subclasses, which is otherwise a serious impediment to extensibility in concurrent OO programming. (Note: The naming conventions for instance variables serve as an annoying reminder of such issues.)

- Avoid giving a variable the same name as one in a superclass.

  Rationale: This is usually an error. If not, explain the intent.

- Declare a very local variable (such as loop variable or array index) only at that point in the code where you know what its initial value should be.

  Rationale: Minimizes bad assumptions about values of variables.

- Declare and initialize a new local variable rather than reusing (reassigning) an existing one whose value happens to no longer be used at that program point.

  Rationale: Minimizes bad assumptions about values of variables.

- Assign `null` to any reference variable that is no longer being used. (This includes, especially, elements of arrays.)

  Rationale: Enables garbage collection.

4.3 Methods

- Avoid cascading method calls

  Rationale: While convenient, the resulting method cascades (`a.meth1().meth2().meth3()`) can be the sources of synchronization problems and other failed expectations about the states of target objects.

- Avoid overloading methods on argument type. (Overriding on arity is OK, as in having a one-argument version versus a two-argument version). If you need to specialize behavior according to the class of an argument, consider instead choosing a general type for the nominal argument type (often `Object`) and using conditionals checking `instanceof`. Alternatives include techniques such as double-dispatching, or often best, reformulating methods (and/or those of their arguments) to remove dependence on exact argument type. However if overload is still convenient, just give the type in the function argument
  Example
  bind(object)
  bindInt(int)
  bindString(string)

  Rationale: Java method resolution is static; based on the listed types, not the actual types of argument. This is compounded in the case of non-Object types with coercion charts. In both cases, most programmers have not committed the matching rules to memory. The results can be counterintuitive; thus the source of subtle errors. For example, try to predict the output of this. Then compile and run.

```
class Classifier{
    String identify(Object x) { return "object"; }
    String identify(Integer x) { return "integer"; }

} //class Classifier

class Relay {

    String relay(Object obj) {
        return (new Classifier()).identify(obj);
    }

} //class Relay



public class App{

    public static void main(String[] args) {
        Relay relayer = new Relay();
        Integer i = new Integer(17);
        System.out.println(relayer.relay(i));
    }

} //class App
```

- Declare when your class or method is thread-safe.

  Rationale: it helps you to master what you are coding.

- Prefer `synchronized` methods to `synchronized` blocks.

  Rationale: Better encapsulation; less prone to subclassing snags; can be more efficient.

- Always document the fact that a method invokes `wait`

  Rationale: Clients may need to take special actions to avoid nested monitor calls.

- Prefer `abstract` methods in base classes to those with default no-op implementations. (Also, if there is a common default implementation, consider instead writing it as a `protected` method so that subclass authors can just write a one-line implementation to call the default.)

  Rationale: The Java compiler will force subclass authors to implement `abstract` methods, avoiding problems occurring when they forget to do so but should have.

4.4 Technical points

- A constructor or method must explicitly declare all unchecked (i.e runtime) exceptions it expects to throw. The caller can use this documentation to provide the proper arguments.

  Rationale: Constructors and methods may place restrictions on arguments passed to them or the order in which they are being called in order to preserve consistency (often they call *throw IllegalArgumentException*). Documentation allows the caller to avoid runtime exceptions.

- Do not use unchecked exceptions instead of code that checks for an exceptional condition.

  Rationale: Comparing an index with the length of an array is faster to execute and better documented than catching *ArrayOutOfBoundsException*

- If you override Object.equals, also override Object.hashCode, and vice-versa.

  Rationale: Essentially all containers and other utilities that group or compare objects in ways depending on equality rely on hashcodes to indicate possible equality.

- Override `readObject` and `writeObject` if a `Serializable` class relies on any state that could differ across processes, including, in particular, `hashCodes` and transient fields.

  Rationale: Otherwise, objects of the class will not transport properly.

- If you think that clone() may be called in a class you write, then explicitly define it (and declare the class as `implements Cloneable`).

  Rationale: The default shallow-copy version of `clone` might not do what you want.

- Always use method `equals` instead of operator == when comparing objects. In particular, do not use == to compare `Strings` (unless comparing memory locations)

  Rationale: If someone defined an `equals` method to compare objects, then they want you to use it. Otherwise, the default implementation of `Object.equals` is just to use ==.

- Always embed `wait` statements in `while` loops that re-wait if the condition being waited for does not hold.

  Rationale: When a `wait` wakes up, it does not know if the condition it is waiting for is true or not.

- Use `notifyAll` instead of `notify` or `resume` when you do not know exactly the number of threads which are waiting for something.

  Rationale: Classes that use only `notify` can normally only support at most one kind of wait condition across all methods in the class and all possible subclasses. And unguarded `suspends/resumes` are even more fragile.

- Embed casts in conditionals. For example:

  ```
  C cx = null;
  if (x instanceof C) cx = (C) x;
  else evasiveAction();
  ```

  Rationale: This forces you to consider what to do if the object is *`not`* an instance of the intended class rather than just generating a `ClassCastException.`
- When throwing an exception, do not refer to the name of the method which has thrown it but specify instead some explanatory text

  Rationale: the name of the method could be given by the stack trace and good exception messages are always useful.

- Document fragile constructions used solely for the sake of optimization.

  Rationale: See Jonathan Hardwick's Java Optimization pages (see Chapter 6).

4.5 Common Sense

- Always avoid assignments ("=") inside `if` and `while` conditions.

  Rationale: There are almost always typos. The java compiler catches cases where ("=") should have been ("==") *except* when the variable is a `boolean`.

- Document cases where the return value of a called method is ignored.

  Rationale: These are typically errors. If it is by intention, make the intent clear. A simple way to do this is:
  `int unused = obj.methodReturningInt(args);`

- Ensure that there is ultimately a `catch` for all *unchecked* exceptions that can be dealt with.

  Rationale: Java allows you to not bother declaring or catching some common easily-handlable exceptions, for example `java.util.NoSuchElementException`. Declare and catch them anyway.

- Minimize `*` forms of `import`. Be precise about what you are importing. Check that all declared `imports` are actually used.

  Rationale: Otherwise readers of your code will have a hard time understanding its context and dependencies.

- Prefer declaring arrays as `Type[] arrayName` rather than `Type arrayName[]`.

  Rationale: The second form is just for incorrigible C programmers.

- Ensure that non-private `static` variables have sensible values even if no instances are ever created. (Similarly ensure that `static` methods can be executed sensibly.) Use static intitializers (`static { ... }`) if necessary.

  Rationale: You cannot assume that non-private statics will be accessed only after

And above all,

- Do not require 100% conformance to rules of thumb such as the ones listed here!

  Rationale: Java allows you program in ways that do not conform to these rules for good reason. Sometimes they provide the only reasonable ways to implement things. And some of these rules make programs less efficient than they might otherwise be, so are meant to be conscientiously broken when performance is an issue.

5 Code Examples

5.1 Example.java

```
/**
 * Copyright 2000 © Exolab Group Inc. All Rights Reserved
 * $Id: Example.java ,v 1.1 2000/08/03 22:11:13 authorname Exp $
 * File: Example.java
 * Date       Author         Changes
 * Aug 2 2000 author-name    Created
 * Aug 2 2000 author-name    Added new classes
 */

package com.intalio.n3.xml;

import java.io.Serializable;
import java.net.URL;
import java.net.MalformedURLException;


/**
 *Hold the URI reference and reference the base URI
 *or the namespace in which the URI is defined. It
 *allows URI references to be passed between application
 *components as well as encoded into XML documents.
 *
 *@author <a href=e-mail address> author name </a>
 *@version $Revision$
 *@see URNNamespace
 */

public class URIReference implements Serializable{

    /**
     * Holds the identifier for this URI
     */
    private String _identifier;
```

```
/**
 * the URI reference to which this reference is relative
 */
private URIReference _baseURI;


/**
 * The URNNamespace for this URI
 *
 *@see URNNamespace
 */
private URNNamespace _namespace;

/**
 * Constructs a new URI reference from the given URL
 *
 * @param url The URL
 */
public URIReference(String url) {
    _identifier = url;
}


/**
 * Returns the resource identifier as a string. The return
 * Is either the URL (absolute or relative) or the URN
 * identifier (namespace specific string).
 *
 * @return The resource identifier as a string
 */
public String getIdentifier() {
    return _identifier;
}

/**
 * Returns true if both URI references are equal.
 * Two URI references are equals if they have the same
 * identifier, the base URI is null or equal, and the
 * URN namespace is null or equal
 */
public boolean equals(Object other) {
    ...
```

```
    while (condition) {
        ...
    }

    do {
        ...
    } while (condition);

} //equals
```

```
public int hashCode() {
    ...
    switch (condition) {

         case ABC:
             if (condition) {
                 ...
             } else {
                 ...
             }
             break;

         case DEF:
             for (initialization; condition; update) {
                 statements;
             }
             break;

         case XYZ:
             {
                 ...
             }
             break;

         default:
             statements;
             break;
    } //switch
```

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

    }//hashCode

}//-- Example
```

5.2 package.html

```
<html>
 <body>
  <p><b>The Java Data Objects API</b></p>

  <dl>
   <dt><b>Version: </b></dt><dd>$Revision: 1.7 $ $Date:
       2000/05/13 00:45:01 $</dd>

   <dt><b>Author:</b></dt>
     <dd><ahref="mailto:arkin@exoffice.com">
       Assaf Arkin</a></dd>
  </dl>

<p>The class {@link org.exolab.castor.jdo.JDO}
  provides the Castor JDO engine used for obtaining
  database connection. A JDO object is constructed with the
  name of a database and other properties, and <tt>
  getDatabase}</tt> is used to obtain a new database connection.</p>

<p>The class {@link org.exolab.castor.jdo.Database}
  represents an open connection to the database that
  can be used to perform transactional operations on
  the database.</p>

<p>Database operations can only be performed in the
  context of a transaction. Client  applications
  should begin and commit a transaction  using the
  <tt>begin</tt> and <tt>commit</tt> methods. Server
  applications should use implicit transaction demarcation
  by the container or explicit transaction demarcation
  using <tt>javax.transaction.UserTransaction</tt>.</p>

 <p>All objects queried and created during a transaction
   are persistent. Changes to persistent
   objects will be stored in the database when the
   transaction commits. Changes will not be
   stored if the transaction is rolled back or fails to
   commit.</p>

 <p>The class {@link org.exolab.castor.jdo.OQLQuery} is
```

```
  obtained from the database and used to
  construct and execute a query on that database. All
  query operations are bound to the database
  transaction.</p>

 <p>The following example opens a connection to the
  database 'mydb' configured from the
   configuration file '<tt>database.xml</tt>', retrieves
   all the products in the specified
   groups and marks them down with a 25% discount and on-
   sale status.</p>

 <pre>
          JDO       jdo;
          Database    db;
          Query      oql;
          QueryResults results;

<font color="red">// Define a new database source</font>
jdo = new JDO( "mydb" );
jdo.setConfiguration( "database.xml" );

<font color="red">// Open a new database, begin a transaction</font>
db = jdo.getDatabase();
db.begin();

<font color="red">// Select all the products in a given group</font>
oql = db.getQuery( "SELECT p FROM Product p WHERE group=$" );
oql.bind( groupId );
results = oql.execute();
while ( results.hasMore() ) {
<font color="red">// A 25% mark down for each    product and mark as sale</font>
 prod = (Product) results.next();
 prod.markDown( 0.25 );
 prod.setOnSale( true );
   }
<font color="red">// Commit all changes, close the database</font>
db.commit();
db.close();
 </pre></p>

 </body>
</html>
```

6 Related Documents

For some others standards and style guides, see:

- The JavaDoc Conventions:
  http://java.sun.com/products/jdk/javadoc/writingdoccomments/index.html
- Javasoft Coding Standard (the present document uses some sections of it)
  http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html
- joodcs standards, with links to a Coding Standards Repository for various languages.
  http://www.meurrens.org/ip-Links/java/joodcs/
- Doug Lea's Java Coding Standard (the base of this document)
  http://gee.cs.oswego.edu/dl/html/javaCodingStd.html
- The AmbySoft Inc. Coding Standard for Java
  http://www.AmbySoft.com/javaCodingStandards.pdf
- Jonathan Hardwick's Java Optimization pages
  http://www.cs.cmu.edu/~jch/java/optimization.html
- Ten Java maxims From Bruce Eckel
  http://www.java-zone.com/free/articles/top10/

Appendix A : Document History

10/18/2000	Added the ‘note to the reader’section
	Modified the section 2.4 (variables order)
	Added the wrapping lines section (section 2.9)
	Modified the section 6 (new references)
9/14/2000	Added a recommendation (section 4.4)
	Modified the constant naming convention
9/8/2000	Added document history
9/1/2000	Added new examples and change section 2.8
8/29/2000	Bones of contention ‘solved’ by a vote
	New section : 2.8 blank spaces
8/14/2000	Creation of the document