Coding standards

Coding Standards

This page describes the standards used for Apache Geronimo code (java, xml, whatever). Code is read by a human being more often than it is actually written by a human being, make the code a pleasure to read.

In this doc you will find:

- Indentation
  - Java
  - XML
- Interfaces
- Exceptions
- Package Naming
- Imports
  - IDE Auto-Formatting
- JavaDoc Tags
- Unit Test Cases
- Logging
  - Levels
  - Example
- Abbreviations and Acronyms in methods/classes/interfaces' names
- GBean attribute and reference naming convention

In addition, check the wiki for details on Developing Geronimo in Eclipse and Subversion Client Configuration.

Indentation

Java

Lets follow Sun's coding standard rules which are pretty common in Java.

http://java.sun.com/docs/codeconv/

http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

- 4 characters indentation
- **No tabs** please!

Correct brace style:

```
public class Foo {
    public void foo(boolean a, int x, int y, int z) {
        do {
            try {
                if (x > 0) {
                    int someVariable = a ?  x : y;
                } else if (x < 0) {
                    int someVariable = (y + z);
                    someVariable = x = x + y;
                } else {
                    for (int i = 0; i < 5; i++) {
                        doSomething(i);
                    }
                }

                switch (a) {
                    case 0:
                        doCase0();
                        break;
                    default:
                        doDefault();
                }
            } catch (Exception e) {
                processException(e.getMessage(), x + y, z, a);
            } finally {
                processFinally();
            }
        } while (true);

        if (2 < 3) {
            return;
        }

        if (3 < 4) {
            return;
        }

        do {
            x++
        } while (x < 10000);

        while (x < 50000) {
            x++;
        }

        for (int i = 0; i < 5; i++) {
            System.out.println(i);
        }
    }

    private class InnerClass implements I1, I2 {
        public void bar() throws E1, E2 {
        }
    }
}
```

XML

- Use 4 characters. This is to allow IDEs such as Eclipse to use a unified formatting convention
- **No tabs** please!

Interfaces

All methods of an interface are public abstract, therefore it is not necessary to specify public abstract modifiers. Similarly all fields are public static final.

However this behavior works best with most tools and IDEs and seems to be common practice so we see no reason to disallow this practice. e.g:

```
public interface MyInterface {
    public static final int MY_INTEGER = 0;

    public abstract void doSomething();
}
```

This has the added advantage that the interface can be converted into an abstract class (and copy and paste individual definitions) without changing anything.

Preferably add public/static/final to constants, and public/abstract to methods, but it's not mandatory. However, if it's there, don't take it out.

Exceptions

- A situation is only exceptional, if the program can not handle it with reasonably effort. Wrong input data should be an expected situation of the regular code, that could be handled gracefully.
- The intention of exception-handling is to separate real error-handling from the regular part of the code, so don't force the caller to mix it with unnecessary exceptions.
- Only if your code really has a problem to continue e.g., when a parameter is invalid, feel free to throw an exception!
- Do NOT throw an exception, if you only suppose the caller of your code could have a problem with a special result. Try to return a special result value instead e.g., null, and let the caller decide with a regular if-else-statement. If the caller really has a problem, HE WILL throw an exception on his own.
- But if your code throws an exception, even though it has no real problem and it could continue without an exception and return a special result value, you forestall the decision of the caller, whether the special result is really an error or not.
- If you throw an exception, where the caller would decide that it is no error in the context of the caller, you force the caller to write an exception handler in his regular part or to abort i.e., you force the caller to mix regular code with exception handling. That is the opposite of the intentention of exception handling.
- Bad example:
  `java.lang.Class.forName(String) throws ClassNotFoundException`
  In most programs/situations it is an error if this method does not find the class, therefore it throws an exception and forestalls the decision of the caller.
  But maybe there is a program that should check a list of class names, whether the classes are present or not. Such a program is forced to mix its regular code with error handling of an exception, that is no error at all in that context.
  The method should return a special result value instead: null. Many callers of that method have expected that situation and therefore are not in an unexpected situation/exceptional state. They could decide the situation on their own.
- Only throw checked exceptions (not derived from `RuntimeException`), if the caller has a chance to handle it.
- Exceptions that signal programming errors or system failures usually cannot be handled/repaired at runtime -> unchecked exception.
- If your code really has a problem to continue e.g., when a parameter is invalid, throw an unchecked exception (derived from `RuntimeException`) and do NOT throw a checked exception, because if not even your code can handle the problem, in the very most cases the caller has no chance to handle the problem, too. Instead there maybe somebody somewhere in the highest layers who catches all `RuntimeException`'s, logs them and continues the regular service.
- Only if it is not possible to return special result values cleanly, use checked exceptions to force the caller to decide the situation. The caller should deescalate the situation by catching and handling one or more checked exceptions, e.g. with special result values(?) or by escalating with an unchecked exception, because the situation is an error, that can not be handled.
- Checked exceptions are an official part of the interface, therefore do not propagate checked exceptions from one abstraction layer to another, because usually this would break the lower abstraction. E.g. do not propagate SQLException to another layer, because SQLExceptions are an implementation detail, that may change in the future and such changes should not affect the interfaces and their callers.
- Never throw `NullPointerException` or `RuntimeException`. Use either `IllegalArgumentException`, or `NullArgumentException` (which is a subclass of `IllegalArgumentException` anyway). If there isn't a suitable subclass available for representing an exception, create your own.

Package Naming

- Package names are lowercase.
- Package names should only contain alpha-numberic characters.
- Package names should be suffixed with the name of the module in which they are defined

For example, if the module name is *common*, then the base package name should be `org.apache.geronimo.common`.

Note: This is more of a *guideline* than a *rule*, as some modules simply can not follow this convention, but where applicable they should.

Imports

- Should be fully qualified e.g. `import java.util.Vector` and not `java.util.*`
- Should be sorted alphabetically, with *java*, then *javax* packages listed first, and then other packages sorted by package name.

IDE Auto-Formatting

- Eclipse users can
- use `Source -> Organise Imports` to organize imports

- use `Source -> Format` to format code (please make sure you are using spaces instead of tabs. See Developing Geronimo in Eclipse for details)
- IntelliJ users can
- use `Tools -> Organise Imports` to organize imports
- use `Tools -> Reformat code` to format code (uses the code style setting in IDE options)

JavaDoc Tags

@version Should be: `@version $Revision$ $Date$`

@author Should not be used in source code at all.

Unit Test Cases

- Use the naming scheme `*Test.java` for unit tests.
- Do not define `public static Test suite()` or constructor methods, the build system will automatically do the right thing without them.

Logging

- Log as much as necessary for someone to figure out what broke 🙂
- Use SLF4J `org.slf4j.Logger` rather than raw *Log4j* or *JCL*
- Do not log throwables that you throw - leave it to the caller
- Use flags to avoid string concatenation for *debug* and *trace*
- Use SLF4J varargs {} expansion
- Cache flags (especially for *trace*) to avoid excessive `isTraceEnabled()` calls

Levels

- Use *trace* level for detailed/diagnostic logging
- Use *debug* level for things an application developer would need to know
- Use *info* level for things an administrator would need to know
- Use *warn* level for things indicating an application or transient problem
- Use *error* level for things indicating a problem with the server itself

Example

```
private final Logger log = LoggerFactory.getLogger(getClass());

public void doSomeStuff(Stuff stuff) throws StuffException {
    try {
        log.trace("About to do stuff: {}", stuff);
        stuff.doSomething();
        log.trace("Did some stuff");
    } catch (BadException e) {
        // don't log - leave it to caller
        throw new StuffException("Something bad happened", e);
    } catch (IgnorableException e) {
        // didn't cache this as we don't expect to come here a lot
        if (log.isDebugEnabled()) {
            log.debug("Ignoring problem doing stuff " + stuff, e);
        }
    }
}
```

Abbreviations and Acronyms in methods/classes/interfaces' names

Abbreviations/acronyms are all written in capitals in methods/classes/interfaces' names.

```
public final class J2EELoader {

    public static EJBRef[] loadEJBRefs(Element parent) {
        //...
    }

}
```

GBean attribute and reference naming convention

The current convention is for attributes to follow Java variable naming conventions (i.e., lower case first character camel cased after) and for references to follow Java class naming conventions (i.e., capital first character camel cased after).