Code Style and Conventions

Generic Code Style And Convention

All working files (java, xml, others) should respect the following conventions:

- **License Header**: Always add the current ASF license header in all versioned files.
- **Trailing Whitespaces**: Remove all trailing whitespaces. If your are an Eclipse user, you could use the Anyedit Eclipse Plugin.

and the following style:

- **Indentation**: **Never** use tabs!
- **Encoding**: Always use UTF-8. If you are an Eclipse user go to: Window > Preferences > Workbench > Editors, then set: Text file encoding to UTF-8

Note: The specific styles and conventions, listed in the next sections, could override these generic rules.

Java

The uPortal style and conventions for Java are mainly:

- **Indentation**: Always use 4 space indents and **never** use tabs!
- **Naming**: Constants (i.e. static final members) values should always be in upper case. Using short, descriptive names for classes and methods.
- **Organization**: Avoid using a lot of public inner classes. Prefer interfaces instead of default implementation.
- **Modifier**: Prefer using final modifier on all member variables and arguments. Prefer using private or protected member instead of public member.
- **Exceptions**: Throw meaningful exceptions to makes debugging and testing more easy. **Always** chain exceptions, passing along the cause when throwing a new exception.
- **Documentation**: Document public interfaces well, i.e. all non-trivial public and protected functions should include Javadoc that indicates what it does.
- **Testing**: All non-trivial public classes should include corresponding unit or IT tests.

XML

The uPortal style and conventions for XML files are mainly:

- **Indentation**: Always use 2 space indents, unless you're wrapping a new XML tags line in which case you should indent 4 spaces.
- **Line Breaks**: Always use a new line with indentation for complex XML types and no line break for simple XML types. Always use a new line to separate XML sections or blocks, for instance:

```
<aTag>
  <simpleType>This is a simple type</simpleType>

  <complexType>
    <simpleType>This is a complex type</simpleType>
  </complexType>
</aTag>
```

In some cases, adding comments could improve the readability of blocks, for instance:

```
<!-- Simple XML documentation                                          -->
```

or

```
<!-- ============================================================================
 | Block documentation
 +-- ============================================================================
```

Reformatting Code

Do not beautify code just for the sake of beautifying code, it can make merging changes more difficult. Fix formatting problems by improving the formatting of the localized code that you're re-writing. Save everyone having to make decisions about the cost of change vs. the value of better formatting by having the best formatting you can have based on the recommendations here from when your code is first introduced.

Always Use Braces

Do not use the keywords **if**, **while**, **do**, **for**, or **switch** without braces indicating the block of code that goes along with the keyword. Non-braced statements are dangerous! The next developer modifying the code often adds another line to the conditional block, expecting it to be in the block. Having brackets makes it easier to read and safer to modify.

One De-reference Per Line

Do not use more dot opperators per line than needed.

Bad.java

```
lines.ofJavaCode(that.involve(numerousReferences).thatMightBeNull().suchThat().anNpe()
).isLessAmbigous();
```

Good.java

```
Involve i = that.involve(numerousReferences);
MightBeNull mbn = i.thatMightBeNull();
That t = mbn.suchThat();
Npe n = t.anNpe();
JavaCode jc = lines.ofJavaCode(n)
jc.isLessAmbigous();
```

The second example makes following a stack trace, finding a NullPointerException, ClassCastException or any other problem much easier as the line number will point you to the single statement that caused the problem. Putting one statement on each line also makes using an debugger much easier as you can inspect the value of each variable before it is used in the next statement.

Exceptions

Use checked exceptions for recoverable conditions and runtime exceptions for programming, configuration or non-recoverable state errors. Never declare a method as throwing Exception. If a checked exception must be declared on a method be sure the exception is very specific to the actual error condition.

Logging

Logging has enough content that it gets its own Logging Best Practices page, below are the important bits.

Exception Chaining

If a new exception is thrown after catching an old exception be sure to chain them like this:

```
catch (RuntimeException e) {
    //Creates a new exception, passing in the cause as a constructor argument
    throw new PortalException("Error while performing task", e);
}
```

If the new exception doesn't take a Throwable argument use the initCause method introduced in JDK1.4

```
catch (RuntimeException e) {
    //Creates a new exception, setting the cause using the new initCause method from
1.4
    PortalException pe = new PortalException("Error while performing task");
    pe.initCause(e);
    throw pe;
}
```

Do not ever do the following:

```
catch (RuntimeException e) {
    //Only includes the message from the cause exception in the new excepion
    //The cause stack trace and line numbers are lost.
    throw new PortalException("Error while performing task: " + e);
}
```

or worse

```
catch (RuntimeException e) {
    //No hint to the real cause or where the actual exception occurred is included in
the new exception
    throw new PortalException("Error while performing task");
}
```

Chaining throwables allows the full stack trace and all related error messages to be viewed by the person debugging the system. This is a very important practice, along with always logging throwables correctly, that greatly expedites debugging problems in the portal framework.