What are Coding Conventions?

Coding conventions are a set of guidelines for a specific programming language that recommend programming style, practices and methods for each aspect of a piece program written in this language. These conventions usually cover file organization, indentation, comments, declarations, statements, white space, naming conventions, programming practices and etc. Software programmers are highly recommended to follow these guidelines to help improve the readability of their source code and make software maintenance easier. Coding conventions are only applicable to the human maintainers and peer reviewers of a software project. Conventions may be formalized in a documented set of rules that an entire team or company follows, or may be as informal as the habitual coding practices of an individual. Coding conventions are not enforced by compilers. As a result, not following some or all of the rules has no impact on the executable programs created from the source code.

General Coding Standards

1. Follow patterns that have been established (for instance naming, conventions of how to do certain functions, documentation, ...)
2. Be consistent with how things are coded
3. Favor object oriented solutions over linear programming
4. Strive to make the code easy to follow and understand
5. Avoid long chunks of code (for instance large methods, over 100 lines or so)
6. Follow SRP ('Single Responsibility Principle') and DRY ('Don't Repeat Yourself')
7. Use Generics where possible
8. Organize logical groupings:
    - Methods that are related, getter/setters towards bottom, following other class organization with similar methods
    - Code blocks that are related
    - Use blank lines to separate code blocks and make code more readable (after control statements, before return, a few lines of related code or variables)
    - Avoid hard coding strings (use instead constants or external resources) with the exception of log or exception strings
    - Avoid deeply nested control statements
9. Remove unused code (not comment out), if commenting out code temporally or for reference, place a TODO item before the block

Files

Here is a listing of commonly used file types in Kuali projects:

File Type	Example	Common Extension	Primary Use
Java source file	DocumentStatus.java	.java	Source code

Java bytecode file	DocumentStatus.class	.class	Compiled code
JSP files	ServiceRegistry.jsp	.jsp	Java Server Pages, dynamically compiled markup
Ant files	build.xml	.xml	High-level configuration of build process
Maven files	pom.xml	.xml	Configuration of build and dependency management process
Spring configuration file	KIMImplementationSpringBeans.xml	.xml	Configuration of dependency injection
SQL files	update_client_final_oracle.sql	.sql	Database scripts for ddl changes
XML Schema Documents	DocumentType.xsd	.xsd	Rules for XML document conformance
Web Services Description Language (WSDL) files	WorkflowDocumentActionsService.wsdl	*.wsdl	WSDL is an XML format for describing SOAP web services
Properties files	system-message.properties	.properties	Managing key/value properties for configuration, sometimes loaded at build-time
Kuali config files	rice-config.xml	.xml	Maintaining system configuration details, often loaded at deployment
Kuali data dictionary files	Country.xml	.xml	Configuring the data dictionary, can often be loaded at runtime

Directory organization

Generally in Kuali, files are organized into projects (e.g. Rice), and these projects are then divided into folders or directories. In the case of source code or other resource files, they are often further divided into Java packages or directory hierarchies.

It is common to break out the source folders into the following structure:

- api : interfaces and simple classes that will be shared
- impl : implementation classes
- web : uncompiled (at build time) files that need to be deployed as part of the webapp

The first two (api and impl) are generally compiled and deployed as jar files under the WEB-INF/lib directory of the third.

Indentation

Tabs

Kuali projects (at least Rice and KFS) seem to use the default Eclipse code formatter for Java, with the following settings for Indentation:

Name	Setting
Tab policy	Tabs only
Use spaces to indent wrapped lines	No
Indentation size	4
Tab size	4

Line Length

Lines of code should readable without horizontal scrolling, or ~160 characters

Line Wrapping

From the Java Programming Language Code Conventions document:

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 instead.

Comments

All source files need to include the appropriate Educational Community License statement, as well as reasonable Javadoc comments at the type, method, and field levels, and additional developer comments to promote clarity of code and support long-term maintenance.

Declarations

- Declare one variable per line
- Initialize variables in their declaration, unless the initial value depends on some calculation that makes this unfeasible
- In general, variables should be declared at the beginning of a block

Constants

- Use constants in favor of literals
- Business values (i.e. ones that may change based on institutional preferences) should not be constants, but should be (preferably) made into system parameters that can be modified in persistent store at runtime, or at worst configuration parameters that will be incorporated at build time
- Enums should be preferred over constants when dealing with multiple related values

Statements

- Enter one statement per line and in general avoid using semi-colons to produce lines with multiple statements.
    - One exception is the for loop, where something like "for(int i=0;i<10;i++)" is common and accepted practice in Java programming.
- Do not use parentheses to identify the argument of a return statement unless it is essential to the readability or function of that statement (for example, to enforce mathematical precedence).

White space

- Use blank lines to improve readability of code by setting off logical sections
- Method declarations should always be preceded and followed by at least one blank line
- Binary operators should generally be separated from their operands by white spaces

Naming conventions

In general, the normal conventions for programming in Java should be followed. Here is a copy of the text from the Java Programming Language Coding Conventions mentioned above:

Identifier Type	Rules for Naming	Examples
Packages	The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981. Subsequent components of the package name vary according to an organization's own internal naming conventions. Such conventions might	com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese

	specify that certain directory name components be division, department, project, machine, or login names.	
Classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized. 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).	class Raster;class ImageSprite;
Interfaces	Interface names should be capitalized like class names.	interface RasterDelegate; interface Storing;
Methods	Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.	run(); runFast(); getBackground();
Variables	Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore _ or dollar sign $ characters, even though both are allowed. Variable names should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.	int i; char c; float myWidth;

Constants	The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_"). (ANSI constants should be avoided, for ease of debugging.)	static final int MIN_WIDTH = 4;
		static final int MAX_WIDTH = 999;
		static final int GET_THE_CPU = 1;

Kuali Rice Naming Standards

Package Naming Standards

Package names should reflect to the largest extent possible the module that they are in. Generally, this means they should follow a pattern like the following for the package prefix:

```
org.kuali.rice.<module>.<sub-module (optional)>.<domain>
```

Each of the portions of the package name are defined as follows:

- **module** - the *module* is the high-level module name for a particular component of Kuali Rice. The module itself does not have any code directly associated with it (in otherwords, no jar is produced), instead the module is divided into a series of sub-modules that contain code. Examples of Rice modules include: kew, kim, krad, ksb, etc.
- **sub-module** - a sub-module is a smaller unit of a larger module. It contains code and resources that are compiled and assembled into a jar file. In general, a sub-module uses one of a standard set of orientations which determines it's role within the larger Kuali Rice stack as well as how it is invoked. Examples there are currently 4 different sub-module orientations defined in Kuali Rice:
    1. api
    2. framework
    3. impl
    4. web
- **domain** - the domain represents the specific functional portion of the module which correspond to some logical domain. For example, in KIM this might include group, identity, role, etc. In certain cases classes may cross multiple domains in which case a "shared" package should be used. For framework modules like KRAD, they may skip the sub-module concept altogether and instead be packaged based on domain (i.e. document, uif, bo, dd, etc.)

As an example, the package prefixes for the various KIM modules would look like the following:

- org.kuali.rice.kim.api.*
- org.kuali.rice.kim.framework.*
- org.kuali.rice.kim.impl.*
- org.kuali.rice.kim.web.*

This portion of the package will generally correspond to the maven module. So in the above example, each of these package names would encompass the following maven modules:

- rice-kim-api
- rice-kim-framework
- rice-kim-impl
- rice-kim-web

As mentioned previously, within each of those modules it is expected that further non-Maven defined modularization of the module be handled via a further breakdown of package names based on domain.

For example, in KIM the following breakdown of domains makes sense:

- identity
- group
- role
- permission
- responsibility
- shared

In which case, KIM would be further broken down into packages as follows:

- org.kuali.rice.kim.api.identity.*
- org.kuali.rice.kim.api.group.*
- org.kuali.rice.kim.api.role.*
- org.kuali.rice.kim.api.permission.*
- org.kuali.rice.kim.api.responsibility.*
- org.kuali.rice.kim.api.shared.*

Finally, we should **discontinue** use of a "layer-based" approach to packaging our source code. Packaging according to layer instead of domain/feature is generally considered a bad practice [2].

Some of these that we have used previously which we should **discontinue use of** are as follows:

- service
- dto
- dao
- bo
- etc...

Package structures that are nested deeper than the domain level should be used judiciously. Such cases where it deemed that is necessary should be considered carefully. One specific case where this is necessary is when packaging object that are tied to a specific version of Rice (such as dtos and services used for remoting). In these cases, underneath the domain level should be a package that include a version id, as follows:

```
org.kuali.rice.<module>.<sub-module>.<domain>.<version id>
```

The version id should start with the letter "v" and contain the major and minor version numbers separated by underscores, as in the the following examples:

- v1_1
- v1_2
- v1_3
- v2_0

Note that we do not need to include the patch version number as part of this version id because **patch releases should not introduce any changes that affect version compatibility**.

So, in KIM this might look like:

- org.kuali.rice.kim.api.group.v1_1.*
- org.kuali.rice.kim.api.identity.v1_1.*
- etc.

Database Table and Column Name Standards

Max Identifier Lengths on Common RDBMS:

RDMBS	Table Name Max Length	Column Name Max Length
Oracle	30	30
MySQL	64	64
Derby	128	128
PostgreSQL	31	31

SQL Server	128	128
DB2	128	128
Sybase	30	30
Sap DB	32	32

As can be seen there are a couple of databases (most notably Oracle) which restrict Table and Column name length to 30 characters. So our target should be 30 characters or less.

Tables, Views and Sequences

Because of the 30 character restriction, we need to design our database object name prefixes so that they take up the least amount of that space while still being descriptive enough.

We will use the following standard prefix:

<Application Acronym><2-letter Module Acronym>_

For Kuali Rice, this will be:

Rice Module	Prefix	Pre-Refactoring Notes
KSB	**KRSB_**	• Quartz table names should begin with **KRSB_** (in Rice standalone) as well • Many of the current tables begin with **EN_** • EN_SERVICE_DEF_DEUX_T should be renamed to something like KRSB_SVC_DEF_T
KNS	**KRNS_**	Majority of tables currently prefixed with one of: • **FS_** • **SH_** • **FP_**
KEW	**KREW_**	Tables are currently prefixed with **EN_**
KIM	**KRIM_**	
KEN	**KREN_**	• Most tables currently prefixed with **NOTIFICATION_** • KCB tables should be included under KREN_ as well

Table Naming Standards

1. Table names should start with **<Application Acronym><2-letter Module Acronym>_**
2. Table names should end in **_T**
3. Full table name should be no longer than **30** characters.
4. Table names should consist only of **capital letters and underscores**

5. Reasonable abbreviations should be used where possible
6. Separate words should be separated by underscores

View Naming Standards

Standards are the same as for Tables Names with the exception of:

- View names should end in **_V**

Sequence Naming Standards

Standards are the same as for Tables Names with the exception of:

- Sequence names should end in **_S**

Columns

Column names on tables and views should not contain prefixes or suffixes. Also, there are a few cases in our current column names where we are duplicating parts of the table names. This is a bit redundant and should be eliminated. For example, on EN_ACTN_RQST_T there are columns named ACTN_RQST_RECP_TYP_CD, ACTN_RQST_PRIO_NBR, etc. Where they could be named just RECP_TYP_CD, PRIO_NBR, etc. This would serve to make the column names as compact as possible

Besides what mentioned above, the rules for column are similar to those for table names.

Column Naming Standards

1. Column names should contain no standard prefix
2. Column names should contain no standard suffix
3. Full column name should be no longer than 30 characters
4. Column names should consist only of capital letters and underscores
5. Reasonable abbreviations should be used where possible
6. Separate words should be separated by underscores
7. Column names should not be prefixed with portions of the table name unless necessary. Examples from the KEW Action Request table:
    - use ID for a primary key identifier rather than ACTN_RQST_ID
    - use STAT_CD instead of ACTN_RQST_STAT_CD
    - use CRTE_DT instead of ACTN_RQST_CRTE_DT

Primary Keys, Foreign Keys, Indexes and Unique Constraints

The naming standards are very similar for PKs, FKs, indexes and unique constraints:

Key / Index / Constraint Naming Standards

- for primary keys:
    - <table_name w/out the trailing T>PK
- for foreign keys:
    - <table_name w/out the trailing T>FK<#>

- For indexes and constraints:
    - <table_name><type><#> where type is I for indexes and C for unique constraints.

numbering starts at 1, and increments for each additional element of the same type.

Examples using the table name KRMS_CNTXT_T:

the primary key:
KRMS_CNTXT_PK

indexes (indices?):
KRMS_CNTXT_TI1
KRMS_CNTXT_TI2
KRMS_CNTXT_TI3

foreign keys:
KRMS_CNTXT_FK1
KRMS_CNTXT_FK2

unique constraints:
KRMS_CNTXT_TC1
KRMS_CNTXT_TC2

Exceptions to Standards

In some cases it is not possible to follow these naming standards. This is a particular problem if using vended libraries which have their own pre-defined table names.

Quartz is an example of this. However, Quartz does provide the ability to specify custom table prefixes. Such features should be taken advantage of when they are available.

See the table at the top of this document which indicates how the Quartz tables should be prefixed in the case of Kuali Rice.

Abbreviations

Before using any abbreviation in any of the database identifiers, an attempt should be made to establish if an abbreviation has already been used in other tables for that same word. Some examples, of common abbreviations seen throughout Rice are below:

Word	Abbreviation
action	ACTN
code	CD
date	DT

description	DESC_TXT
document	DOC
header	HDR
identifier	ID
indicator	IND
namespace	NMSPC
parameter	PARM
request	RQST
title	TTL
type	TYP
version number	VER_NBR
...	...

This is by no means exhaustive but demonstrates the general idea when choosing abbreviations.

Programming practices

Autoboxing and Auto-Unboxing of Primitive Types

Beginning with Java 5, primitive types are autoboxed or auto-unboxed as necessary by the Java compiler. This can helpful, since it reduces the need for extensive casting of primitives into their corresponding Object, for example: to place a primitive int into a Collection.

However, it is important to remember that auto-unboxing (converting an Object into its primitive type at assignment) can throw a NullPointerException if that Object is null at the point of assignment. This is true generally, but auto-unboxing can make it more difficult to recognize when glancing over some code, since statements like this one don't generally make developers think they need to check for nulls:

```
int x = y;
```

Only if y is an Integer, then it might be null. Using a static code checker like FindBugs will address this, and Eclipse has a configuration setting under Preferences > Java > Errors/Warnings > Potential programming problems > Boxing and unboxing conversions that will make these jump out.

Generics

Use Java generics with Collections to indicate type, especially on method return values, to reduce the need for unchecked casting and to improve readability of code by developers coming after you.

Generics may also be particularly useful when developing wrapper classes, or interfaces that may be implemented to work with a specific type or some small set of distinct types.

Inversion of Control

TODO: Service locator versus dependency injection...

Programming for Accessible Web Applications

Web Content Accessibility Guidelines

In order that Kuali applications can be made accessible to the broadest possible audience, it is recommended that developers follow the guidelines laid out in the W3C Recommendation on Web Content Accessibility Guidelines (WCAG) 2.0, specifically that user interface code or output that may eventually be surfaced in a user interface developed for the project be:

1. Perceivable
    a. Provide text alternatives for any non-text content so that it can be changed into other forms people need, such as large print, braille, speech, symbols or simpler language.
    b. Provide alternatives for time-based media.
    c. Create content that can be presented in different ways (for example simpler layout) without losing information or structure.
    d. Make it easier for users to see and hear content including separating foreground from background.
2. Operable
    a. Make all functionality available from a keyboard.
    b. Provide users enough time to read and use content.
    c. Do not design content in a way that is known to cause seizures.
    d. Provide ways to help users navigate, find content, and determine where they are.
3. Understandable
    a. Make text content readable and understandable.
    b. Make Web pages appear and operate in predictable ways.
    c. Help users avoid and correct mistakes.
4. Robust
    a. Maximize compatibility with current and future user agents, including assistive technologies.

General

1. All exposed API elements should have meaningful javadoc which documents the contract for the API.
2. Internal classes and methods (with the exception of very simple getters/setters like service injection) should have a meaningful javadoc
3. All method parameters and return types (except for void methods) should be documented

4. Any unchecked exceptions that the method can throws should be documented using the Javadoc `@throws` tag
    - examples: `IllegalArgumentException`, `IllegalStateException`, `NullPointerException`, etc.
    - do **not** use the `throws` keyword to include unchecked exceptions in the method declaration except in the case of JAX-WS annotated services where this is required in order for meaningful soap faults to be produced
5. Add line comments to explain code intent and outline the process, but use these only when needed. Code should be as self-documenting as possible.

Format

1. First line of javadoc should be a short summary (one or two sentences) of the method. No blank line before summary
2. Additional detailed information should be enclosed in a paragraph (<p>) tag with a blank line before each paragraph
3. One blank line between the method descriptions and the list of tags
4. for tag phrases (e.g. parameters) use phrases (do not start with capital letter or end with a period)
5. For inline comments use // followed by a space
6. For inline comments do not capitalize first letter or end with period
7. For field Javadocs, give description on getter where exist, else the field. Setter can have generic statement or if needed special information about the setter

Method/Class Javadocs

```
Good:

/**
* Determines the primary key for the data object
*
* <p>
* The primary key is found by ....
* ....
# </p>
*
* @param dataObject - data object instance to retrieve primary key for
* @return String primary key found, or null
*/

Bad:

/**
*
* Determines the primary key for the data object. The primary key ...
# ....
* @param dataObject data object instance to retrieve primary key for.
* @return String primary key found, or null
*/

Note: Blank line before summary, should have paragraph for more detailed
information,
no blank line before param tag, no dash after param tag and period after
param tag description
```

Inline Comments

Good:

```
// attempt to find the key
```

Bad:

```
//Attempt to find the key.
```

Style

1. Method comments should begin with a verb (avoid 'This method ...')
2. Class/interface/field comments should state the object (avoid 'This interface/class ...')
3. Set the @author tag in the Class javadoc to 'Kuali Rice Team (rice.collab@kuali.org)'
4. Use the Javadoc {`@code ...`} tag for keywords and names
5. Use 3rd person (descriptive) not 2nd person (prescriptive) ('Gets the label' not 'Get the label')
6. Use "this" instead of "the" when referring to an object created from the current class ('Gets the toolkit for this component' not 'Gets the toolkit for the component')
7. Avoid use of end-line comments

Content

1. Add description beyond the API name
2. As much as possible, write doc comments as an implementation-independent API specification
3. Make it complete enough for conforming implementers
4. Describes why the code is doing what it does
5. Include examples
6. Specify implementation specific behavior in a separate paragraph
7. For property getters, specify what the returned field is used for (field javadoc)
8. Use @see for interface implementations, overridden methods, setters for properties (point to getter). Add additional comments if necessary

See http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html for more details

Documenting Thread Safety

To enable safe concurrent use of exported API elements, a class must clearly document it's thread safety policy. The presence of the `synchronized` modifier does not satisfy proper documentation of thread safety, this is because the `synchronized` keyword is an implementation detail and not part of the exported API.

There are multiple levels of thread safety, the following are from the book *Effective Java*:

- **immutable** - Instances of this class appear constant. No external synchronization is necessary. Examples include `String`, `Long`, `BigInteger`, etc., in addition to the various immutable model object classes that can be found throughout the Rice APIs.
- **unconditionally thread-safe** - Instances of this class are mutable, but the class has sufficient internal synchronization that its instances can be used concurrently without the need for any external synchronization. Examples include `Random` and `ConcurrentHashMap`.

- **conditionally thread-safe** - Like unconditionally thread-safe, except that some methods require external synchronization for safe concurrent use. Examples include the collections returned by Collections.synchronized wrappers, whose iterators require external synchronization.
- **not thread-safe** - Instances of this class are mutable. To use them concurrently, clients must surround each method invocation (or invocation sequence) with external synchronization of the clients' choosing. Examples include the general-purpose collection implementations such as`ArrayList` and `HashMap`.
- **thread-hostile** - This class is not safe for concurrent use even if all the method invocations are surrounded by external synchronization. Thread hostility usually results from modifying static data without synchronization. No one writes a thread-hostile class on purpose; such classes result from the failure to consider concurrency. Luckily, there are very few thread-hostile classes or methods in the Java libraries. The `System.runFinalizersOnExit`method is thread-hostile and has been deprecated.

Naming Conventions

1. Use names that have meaning (not i, j, .. with the exception possibly of loop counters)
2. Follow naming patterns that have been established for similar types of classes, variables, and methods
3. In general avoid abbreviations, with the exception of very well known abbreviations or to prevent excessively long names
4. Names use camel casing with first letter upper-cased for class names and lower cased for method and variable names
5. Use nouns when naming classes and fields, verbs when naming methods
6. Pluralize the names of collection references

Code Cleanup (on new/changed code)

1. Perform formatting (before check-in)
2. Perform code cleanup
3. Organize imports
4. Fix warnings where possible

Deprecating Code

1. Mark the method/class/package/etc. with @Deprecated annotation. This compiles the object's deprecated status in the .class file (and can even be accessed with reflection). When a client compiles against the class file, the compiler will issue a warning regardless of whether source/javadoc is present.
2. Mark the method/class/package/etc. with @deprecated javadoc tag. Unlike, the annotation this tag allows us to put a description what why the thing is deprecated and what to do.

UIF Specific

1. Group together lifecycle methods in order (initialize, applyModel, finalize)
2. Make methods public/protected to allow for overriding any behavior
3. Strive to provide declarative options (via XML) where possible