Magic

Developer Guidelines

Table of Contents

[1 Magic Guidelines 3](#_Toc35327836)

[1.1 Table columns 3](#_Toc35327837)

[1.2 Document action icons 4](#_Toc35327838)

[1.3 Schema naming conventions 4](#_Toc35327839)

[1.4 Search forms 4](#_Toc35327840)

[1.5 Non-modal window 5](#_Toc35327841)

[1.6 Changing Labels 6](#_Toc35327842)

[1.7 Defining constants 6](#_Toc35327843)

[2 Coding Guidelines 6](#_Toc35327844)

[3 Java Best Practice 7](#_Toc35327845)

[3.1 Design 7](#_Toc35327846)

[3.2 General tips 8](#_Toc35327847)

[3.3 Comments 8](#_Toc35327848)

[3.4 Annotations 9](#_Toc35327849)

[3.5 Static checks 10](#_Toc35327850)

[3.6 Iterations 10](#_Toc35327851)

[4 Java Code Style 11](#_Toc35327852)

[4.1 Testing 11](#_Toc35327853)

[5 Java Collections 12](#_Toc35327854)

[5.1 Types 12](#_Toc35327855)

[5.2 Generics 13](#_Toc35327856)

[5.3 Sorting 14](#_Toc35327857)

[6 Java Exception Handling 14](#_Toc35327858)

[6.1 Catching Exceptions 14](#_Toc35327859)

[6.2 Throwing Exceptions 15](#_Toc35327860)

[7 Java Libraries 15](#_Toc35327861)

[7.1 Using Libraries 15](#_Toc35327862)

[7.2 Java Libraries List 16](#_Toc35327863)

[8 Java Logging 17](#_Toc35327864)

[8.1 Use a Logger 17](#_Toc35327865)

[9 Java Naming Convention 19](#_Toc35327866)

[9.1 Class, Interface, Enum and Annotation Names 19](#_Toc35327867)

[9.2 Variable Names 19](#_Toc35327868)

[9.3 Method Names 19](#_Toc35327869)

[9.4 Package Names 20](#_Toc35327870)

[10 JPO Guidelines 21](#_Toc35327871)

[10.1 Introduction 21](#_Toc35327872)

[10.2 Naming Convention 21](#_Toc35327873)

[10.3 Code 21](#_Toc35327874)

[10.4 Modifying existing OOTB JPO 22](#_Toc35327875)

[10.5 When to use and not to use JPO's 22](#_Toc35327876)

[11 JSP Guidelines 22](#_Toc35327877)

[11.1 JSP Coding Standards 22](#_Toc35327878)

[11.2 Using Tag Library 22](#_Toc35327879)

# Magic Guidelines

## Table columns

### Open in New Window

Objects can be opened either in then window already in use or in a new pop up window. For ease of use and consistency objects shall always open in the same window when opened via the “Name” column. For opening an object in a new window, a dedicated column should be used. Column with dedicated icon for opening an object in a new window should always be present before “Name” column as shown below.



Open same window

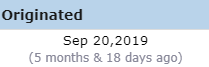
Always include

Below code snippet should be used for new window column.

1. **<Column>**
2. **<Href>**${COMMON\_DIR}/emxTree.jsp**</Href>**
3. **<Alt>**emxComponents.Common.OpenNew**</Alt>**
4. **<RegisteredSuite>**Components**</RegisteredSuite>**
5. **<ColumnType>**icon**</ColumnType>**
6. **<ColumnIcon>**images/iconActionNewWindow.gif**</ColumnIcon>**
7. **<Sortable>**false**</Sortable>**
8. **<TargetLocation>**popup**</TargetLocation>**
9. **<WindowWidth>**930**</WindowWidth>**
10. **<WindowHeight>**650**</WindowHeight>**
11. **<PopupModal>**false**</PopupModal>**
12. **</Column>**

### Date column

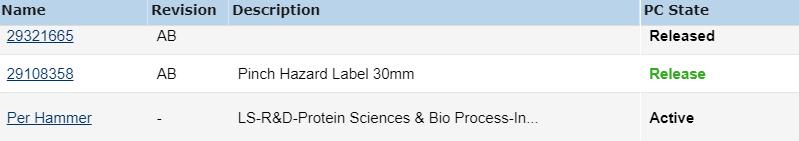
“Date” columns are used to show the date when the object was created, modified or released. For consistency, and ease of overview **only** the date should be shown. The elapsed time should be omitted. See examples below.

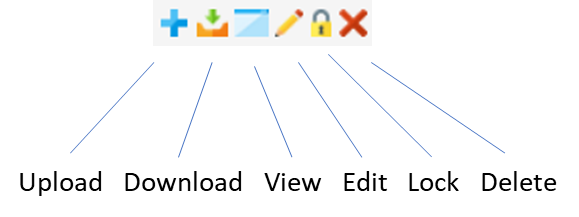
To add a "Date" column in the correct format use following code snippet:

### With of columns

Width of table columns should **not** be fixed. It shall so vary according to the text it holds when with is adjusted in the UI.



## Document action icons

* Action icons used anywhere should look like picture
* Preview icon should not be shown in any view
* New action columns should include below configuration

1. **<Setting** name="Always Show File Name" value="true"**/>**
2. **<Setting** name="Show Preview" value="false"**/>**

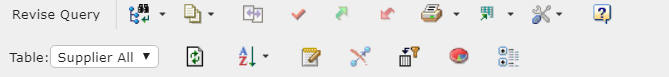
## Schema naming conventions

* Use symbolic name, not actual name for coding.
* Do not use GEHC for naming - use MAGIC as suffix instead.
* Do not change actual name of existing schema (attribute, relationship, type, ETC). The actual name might have many references in integration. Only change it if it is unavoidable, but check all references of that schema in the system carefully.

## Search forms

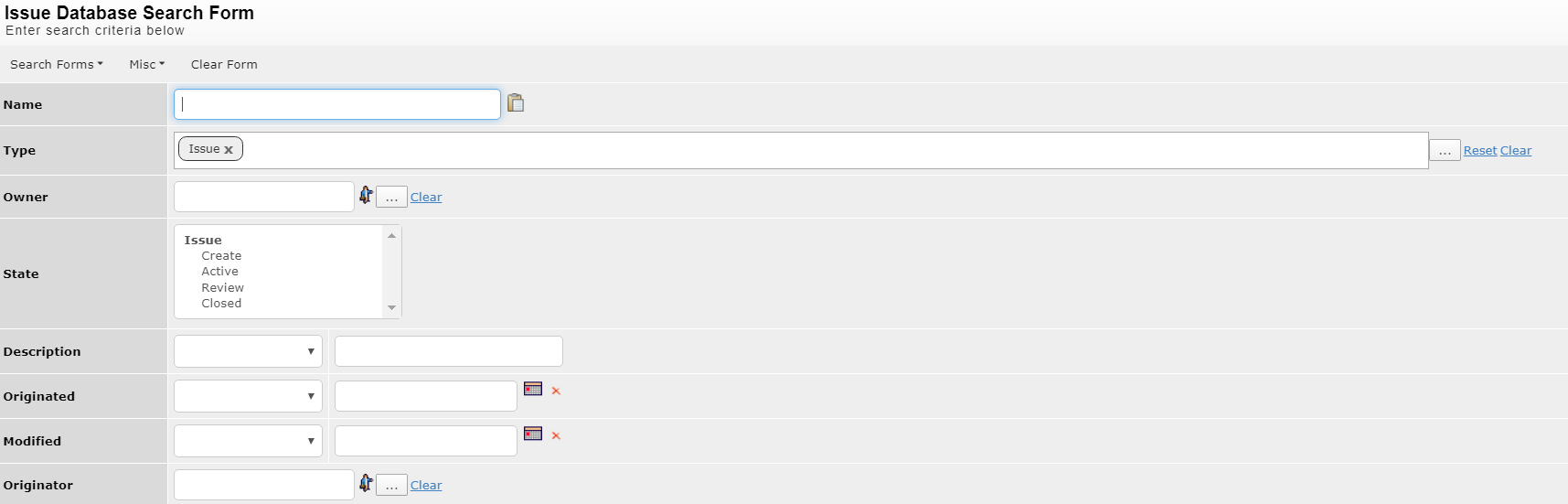
“Revise search” action should always be present in newly added forms. And it should be consistent everywhere in system. See example.





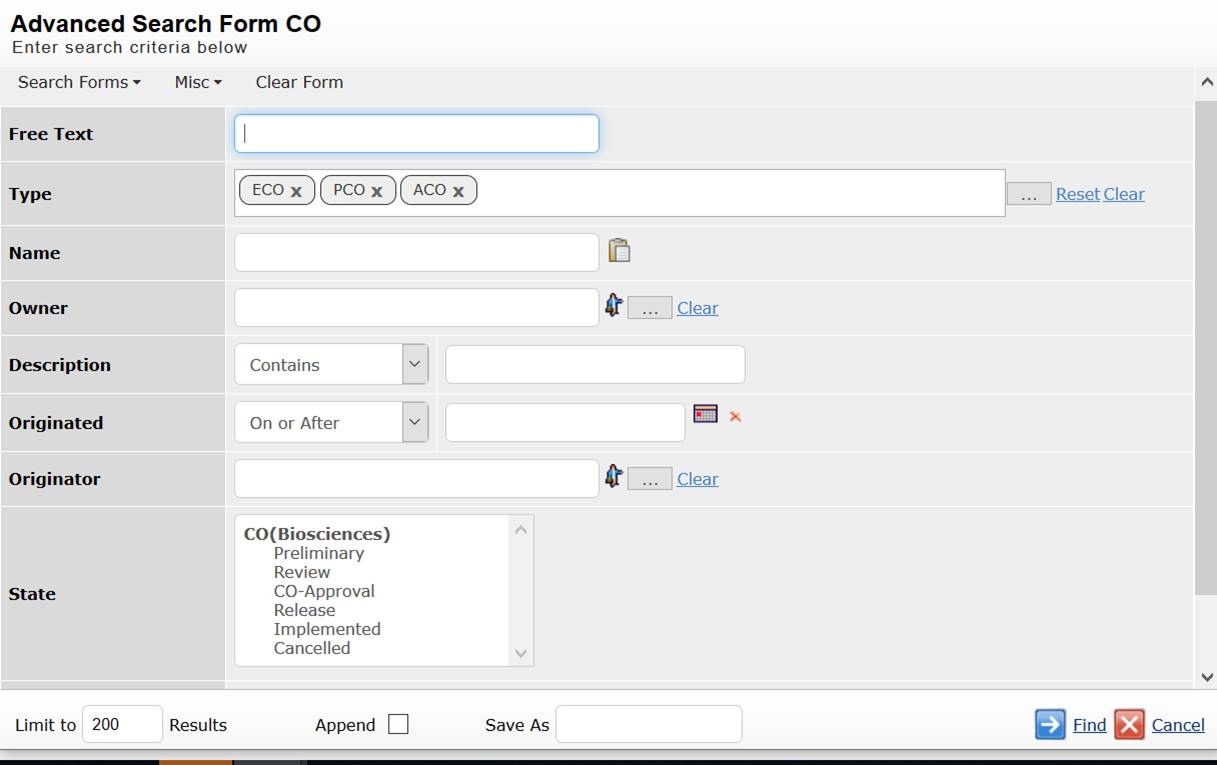
### Database Search

* Standard attributes for new DB search forms
  + Required
  + Same order



### Advanced Search

* Standard attributes for new advanced search forms
  + Required
  + Same order



## Non-modal window

All the new popups should be non-modal window so that users are not blocked to use other views at the same time.

1. **<PopupModal>**false**</PopupModal>**

## Changing Labels

The table columns labels and form field labels should always use property from properties files based on different component present for different modules using Registered Suite as shown below:

Ex:

1. column
2. label      emxFramework.Basic.Owner
3. businessobject owner
4. autoheight **false**
5. autowidth  **false**
6. editable   **false**
7. hidden     **false**
8. name       Owner
9. sorttype   none
10. setting    No Wrap value TRUE
11. setting    Registered Suite value Framework

## Defining constants

If user requires to use some static values or schema names such as Type name, Relationship Name or Attribute name, then user should use predefined values in existing files or define a new property instead of hardcoding the values

Some constant Files are as below:

1. com.matrixone.apps.domain.DomainConstants
2. com.gehcmagicvalu.constants.Attributes
3. com.gehcmagicvalu.constants.Relationships
4. com.gehcmagicvalu.constants.Types
5. com.gehcmagicvalu.constants.Vaults
6. com.gehcmagicvalu.constants.States
7. com.gehcmagicvalu.constants.Policies
8. com.gehcmagicvalu.constants.Roles

## Notification Utility

If developer wants to develop a functionality to send and email to the Magic users, either of the below utilities should be used for consistent results:

emxNotificationUtil\_mxJPO.sendJavaMail

com.gehcmagicvalu.common.Utility.sendMail

# Coding Guidelines

* Use existing utility- Check if there is already a Utility Method available for desired functionality,

e.g.

1. com.technia.tvc.core.util.StringUtils should be used **for** string related utility APIs.
2. com.technia.tvc.core.db.BusinessObjectUtils should be used **for** operations related to object
3. com.technia.tvc.core.db.RelationshipUtils should be used **for** operations related to Relationship ids
4. com.technia.tvc.core.db.RelToRelUtils should be used **for** rel to rel operations

* New method- If possible, write a new method, instead of making the correction at the same location.
* Before pushing, always debug the new code to make sure it is working as expected, this practice also provides some hint about scenario where the current correction might fail.
* Return type- Carefully decide on the return type, Especially action triggers and check triggers.
* Copy-Pasting Code- Code should be written with a sense of ownership, simply copy-pasting a code is bad practice.

# Java Best Practice

The following Java best practices will when followed improve code readability and quality.

## Design

### Program to an Interface, Not an Implementation

It is best practice that method return types, method arguments and variables of any kind are as general as possible. In many cases this translates to typing by an interface instead of an implementation class, but it is also true for class hierarchies; prefer a super type since it is more general than a sub type. The main goal with this practice is to avoid coupling code to types and behavior that is more specific than the context requires.

1. // Bad    -- Coupled to a specific implementation
2. DefaultStructureBean structure = **new** DefaultStructureBean();
4. // Good   -- The StructureBean interface defines basic behavior for structures;
5. //           We can use this variable to work with the structure that it
6. //           represents without coupling the code to a specific implementation
7. StructureBean structure = **new** DefaultStructureBean();
9. // Better -- The TableBean interface is more general than StructureBean and
10. //           defines basic behavior for structures and flat tables alike; We
11. //           can use this variable to work with the data in the table without
12. //           coupling the code to a specific implementation or (depending on
13. //           the context) unnecessarily specific interface
14. TableBean<?> table = **new** DefaultStructureBean();
16. // Bad    -- Coupled to a specific implementation
17. ArrayList<String> strings = **new** ArrayList<String>();
19. // Good   -- The List interface defines basic operations for lists; we're free
20. //           to chose a different List implementation later without it requiring
21. //           a rewrite all code that use it
22. List<String> strings = **new** ArrayList<String>();
24. // Better -- The Collection interface defines basic operations for collections;
25. //           since it it more general than List we're free to chose a different
26. //           kind of collection later without having to rewrite all code that
27. //           use it. Of course, if the code requires access to List specific
28. //           behavior typing it to Collection is not ideal
29. Collection<String> strings = **new** ArrayList<String>();
31. // Better Yet -- The Iterable interface represents anything that code can
32. //           iterate over; using this interface is preferred if the code is
33. //           agnostic about Collection specific operations to add, remove, etc
34. //           and only need to iterate over the elements.
35. Iterable<String> strings = **new** ArrayList<String>();
36. // Good -- Variable typed to interface
37. Set<String> items = **new** HashSet<String>();
38. List<Integer> indexes = **new** LinkedList<Integer>();
40. // Good -- Method arguments and return value typed to interface
41. **public** **static** <K, V> Map<K, V> intoMap(Map<K, V> map,
42. Iterable<? **extends** K> keys,
43. Transformer<? **super** K, ? **extends** V> keyToValue) {
44. **for** (K key : keys) {
45. map.put(key, keyToValue.transform(key));
46. }
47. **return** map;
48. }
50. // Bad -- Variable type coupled to implementation class
51. ArrayList<BigDecimal> costs = **new** ArrayList<BigDecimal>();

## General tips

* String variable should be initialized to null.
* getRelatedObjects or any other database call should not use wildcard characters to avoid performance problems in huge data.
* Variables should be checked for null before usage to avoid unexpected behaviour.
* While using collections, if the size is already known, it should be initialized with fixed size so that unnecessary memory is not allocated.

## Comments

Code that is properly documented is easier to understand and reuse.

In short:

* Person who is modifying the code should enter Jira case number and team name.
* Write and design code that is easy to understand; write inline comments only to describe non-obvious details
* Write documentation when you know it's needed; good documentation is an investment that should be made selectively
* If you update the code, you're also responsible to update the documentation

### Inline code comments

Well-designed code that in itself is easy to read and understand require less inline comments to explain intent; the focus should always be to write and design code well and only comment the non-obvious aspects of the code, such as why a piece of code does something rather than what or how (since the latter should be obvious in code that's well written and designed). It is every developer’s responsibility to update comments as necessary when the code is updated; outdated comments can obfuscate the intent of the code and thus impact quality.

1. **private** **void** createIndexFiles(List<PackageHandlerFileObject> indexFileList) throws TVCException, IOException {
3. // Passing null as we are passing the temp directory also ,
4. //so it will not go in the logic for using ctx value
6. addFileIndex(indexFileList, tempDirectory.toString());
8. }

### Class level comments

Each class should have a class level comment which signifies the usage of class.

1. /\*\*
2. \* When user wants to Create a Top Level Part(End Product, Quote Item) on
3. \* selecting a Product COnfiguration in Related Info tab of a Project This class
4. \* does one click create of Top Level Part according to the attributes available
5. \* in corresponding Product Model(Hardware Product) Also it connects the
6. \* required objects as below: 1)Connect Existing Product Documents from PC
7. \* 2)Connecting RDO and Requirement from Product 3)Set class attributes from PC
8. \* 4)Cloning and Connect Document Templates 5)Connect Existing Customer 6)Create
9. \* and connect DS/PS Product
10. \*
11. \* @author Shweta Gupta
12. \*
13. \*/
14. **public** **class** CreateTopLevelPart extends TableDataAction {}

### Method level comments

1. /\*\*
2. \* Populating the download column for allowing user to download EBL package
3. \* by Technia for ZWAWP-1464
4. \*/
5. @Override
6. **public** **void** populateCell(Cell cell, EvaluatedData data) {
7. DownLoadIconCell downloadCell = (DownLoadIconCell) cell;
9. String affectedEBL = data.getObjectData().getSelectValue(Attributes.AFFECTED\_EBL\_SELECT);
10. List<String> eblData = StringUtils.splitToList(affectedEBL, Symbols.SPACE);
11. **if**(null != eblData && eblData.size() == 3) {
12. String eblName = eblData.get(1);
13. String eblRevision = eblData.get(2);
14. }
15. }

## Annotations

Java 5 and above supports meta data (information about the code) in the form of annotations.

### Suppress Warnings

Warnings in code should be taken seriously and should only be suppressed for a valid reason.

The following steps should be taken to deal with warnings in code:

* Fix the error:
  + Remove unused code.
  + Update code to not use deprecated APIs.
  + Make sure that all generic types have type parameters.
  + Make sure that the code isn't performing invalid cast operations.
  + Add serial version UID to serializable classes.
* If you cannot fix the code to make the warning, go away:
  + If the code deals with a legacy API that doesn't support generics, unchecked cast warnings may be suppressed assuming the type parameters are known.
  + Otherwise, don't suppress the warning – it is there for a reason

1. // OK -- A legacy API returns a non-generic List implementation that we can cast to a parameterized List to use in enhanced for-loops, etc
2. @SuppressWarnings("unchecked")
3. List<BusinessObject> objects = Query.tempQuery(SchemaConstants.type\_SeasonalPlan.get(), "\*", "\*", where.getExpression());
4. **for** (BusinessObject object : objects) {
6. }
8. // Not OK -- Unused code should be removed!
9. @SuppressWarnings("unused")
10. **private** **void** someFunction() {
12. // Not OK -- Code using deprecated APIs should be updated!
13. @SuppressWarnings("deprecation")
14. **public** **class** SketchVersionDataHandler **extends** com.technia.tvc.core.db.table.evaluator.HtmlHandler {
16. // Not OK -- Generic types should be parameterized!
17. @SuppressWarnings("rawtypes")
18. Iterator cellValueItr = cell.getValues();

### Override

The override annotation should be added to all methods that override a super class method.

For codebases targeting Java 6 or above the annotation should also be added to all methods that implement a method from an interface.

1. @Override
2. **public** ArrayList<String> perform(Form form) **throws** Exception {

## Static checks

Every developer should use some static tool, e.g. SonarQube for finding and resolving all static errors in the code.

## Iterations

### Enhanced for-loop

Using the enhanced for-loop in Java is preferred when possible to iterate over items in an arrays or classes that implement Iterable (e.g., unless you need manual access to a loop counter or Iterator).

1. List<String> items = **null**;
2. // code that initializes 'items' to a non-null list of strings...
3. **for** (String item : items) {
4. // code that does something with 'item'...
5. }

### Invariant Expressions/Database Calls

Expressions (or sub expressions) that are loop-invariant should occur before the loop to avoid unnecessary computations.

Note: This will prevent the performance problems, e.g. if some database calls are happening inside loops which are repetitive should be moved before the loop.

1. // Bad -- An invariant sub expression is evaluated each iteration
2. **for** (PositionApproval posApproval : field.getValues()) {
3. HttpServletRequest request = ctx.getRequest();
4. String url = URLUtils.addParam(
5. ActionUtils.getActionURL(request, COLOR\_AC\_ACTION),
6. OBJECT\_ID\_PARAM,
7. posApproval.getColor().getObjectId());
8. out.startElement("input").addAttribute("autocompleteurl", url);
9. // etc
10. }
12. // Good -- The invariant sub expression has been moved outside of loop
13. HttpServletRequest request = ctx.getRequest();
14. String baseUrl = ActionUtils.getActionURL(request, COLOR\_AC\_ACTION);
15. **for** (PositionApproval posApproval : field.getValues()) {
16. String objectId = posApproval.getColor().getObjectId();
17. String url = URLUtils.addParam(baseUrl, OBJECT\_ID\_PARAM, objectId);
18. out.startElement("input").addAttribute("autocompleteurl", url);
19. // etc
20. }

### Extract Local Variable with Eclipse

Eclipse offers a rather easy way to refactor invariant expressions. Simply select the expression that is invariant and press Alt + Shift + L (or select Refactor > Extract Local Variable from the menu). When you do this Eclipse will try to find all occurrences of the same (sub) expression in the current method and replace them with the local variable that was just extracted. In most cases refactoring the code to extract a local variable instead of computing the expression several times have no side effects, but this of course depends on what the expression does (for example if it calls methods that have side effects that must be accounted for).

### Extract Methods to Smaller Methods

For reducing complexity and increase readability Eclipse offers a rather easy way to extract methods to smaller methods. Simply select the piece of code that must be extracted and press Alt + Shift + M (or select Refactor > Extract method from the menu). When you do this Eclipse will try to find all occurrences of the same code in the class and replace them with the new method that was extracted.

# Java Code Style

## Testing

The level of testing to be performed in a project and who is responsible should be agreed upon in the contract.

### Unit Testing by the Developer

A developer is responsible for unit testing of the code before checking it in to the code versioning system. Below are some questions that could work as a starting point for writing test cases:

* What happens if you look for something that doesn't exist in the db?
* Is user input validated?
  + Test if invalid characters allowed.
  + Test upper and lower limit on values.
* Is there a circumstance in which the method could return null, or an empty collection, or an empty array?
* Test with many objects to ensure good performance, see Best Practices for details.
* If it’s possible to integrate to other modules this is preferable that these tests are performed.
* Test with actual user IDs; don't test with a super user like "Test Everything".

### Testing done by Testers

The following test may not be covered by the developer’s unit testing.

* Stress test with heavy user load
* Integration with other modules

### Automated tests - JUnit

JUnit is a framework for writing and running automated tests. see <http://sourceforge.net/projects/junit/> for more info.

# Java Collections

The following section describe guidelines applicable to usage of collections in Java, in particular the Java Collections API.

## Types

Key interfaces in the Java Collections API are:

* Collection
* List
* Queue
* Set
* SortedSet
* Map
* SortedMap

Note: In *most* cases ArrayList is probably the preferred choice. Iteration is equally fast, and I would assume that the memory footprint is lower.

LinkedList makes sense when your algorithm calls for inserting or removing objects in the middle/beginning of the list, which is not a very common use case. If you only add to the end of List and iterate the result ArrayList makes most sense.

### Program to an Interface, Not an Implementation

It is best practice that method arguments, return types and variables are typed to an interface (not the implementation type) to ensure that the code is as loosely coupled as possible. Note that this practice is preferred for all types, not just collection types.

1. // Good -- Variable typed to interface
2. Set<String> items = **new** HashSet<String>();
3. List<Integer> indexes = **new** LinkedList<Integer>();
5. // Good -- Method arguments and return value typed to interface
6. **public** **static** <K, V> Map<K, V> intoMap(Map<K, V> map,
7. Iterable<? **extends** K> keys,
8. Transformer<? **super** K, ? **extends** V> keyToValue) {
9. **for** (K key : keys) {
10. map.put(key, keyToValue.transform(key));
11. }
12. **return** map;
13. }
15. // Bad -- Variable type coupled to implementation class
16. ArrayList<BigDecimal> costs = **new** ArrayList<BigDecimal>();

See also section: [Java Best Practice](#_Java_Best_Practice).

### Java Convention for Methods that Return a Collection (or Array)

Methods declaring that they return some kind of collection or array should by convention never return null. Instead an empty collection or array should be returned.

1. // Good -- This method will always return a non empty collection.
2. //         In this case the javadoc should declare that null is
3. //         an accepted value for `s`.
4. **public** List<String> splitToList(String s, String regex) {
5. **if** (s == **null**) {
6. **return** Collections.emptyList();
7. } **else** {
8. **return** Arrays.asList(s.split(regex));
9. }
10. }
12. // Good -- This method will always return a non empty collection.
13. //         In this case javadoc should declare that an exception
14. //         will be thrown when `s` is null.
15. **public** List<String> splitToList(String s, String regex) {
16. **if** (s == **null**) {
17. **throw** **new** IllegalArgumentException();
18. } **else** {
19. **return** Arrays.asList(s.split(regex));
20. }
21. }
23. // Bad  -- This method may return null, which is contrary to
24. //         standard Java coding conventions.
25. **public** List<String> splitToList(String s, String regex) {
26. **if** (s == **null**) {
27. **return** **null**;
28. } **else** {
29. **return** Arrays.asList(s.split(regex));
30. }
31. }

## Generics

Type parameters allow for compile time type safety of value types in collections, as well as eliminating the need to explicitly cast values retrieved from a collection.

It is best practice to always declare type parameters for types that expose them. Raw types – generic types without type parameters – should not be used.

1. // Good -- Type parameters provided
2. List<String> names = **new** ArrayList<String>();
3. Iterator<? **extends** TableData> itr = table.getTableData();
5. // Bad -- No type parameter provided
6. List stuff = **new** LinkedList();

## Sorting

The Java Collections API exposes methods to sort lists and arrays that uses an optimized sorting algorithm. Please don't roll your own bubble sort (or other equally bad or worse algorithm).

If the collection isn't required to maintain duplicate elements it is often recommended to use a sorted set (such as TreeSet) instead of using a list or array that must be explicitly sorted.

# Java Exception Handling

## Catching Exceptions

If you catch an exception, it's your responsibility to log that the exception occurred (if you handle it) or throw a new exception (or in some cases re-throw the original exception).

### If you handle the exception

If your code can handle the exception in a graceful way, e.g., by falling back to an alternative code path, retrying the same operation or by presenting an error message to the user, you should **also** log that the exception occurred with a proper logger. By also logging that the exception occurred we facilitate future debugging of erroneous logic; perhaps the exception that the code caught was not the error that it was intended to catch?

1. **try** {
2. // do something that can raise an exception ...
3. } **catch** (TVCException e) {
4. logger.error("<Error message describing the operation that failed>", e);
5. // handle the exception in a graceful way
6. }

See also: [Java Logging](#_Java_Logging)

### If you don't handle the exception

If your code cannot gracefully handle the exception in a specific location it should preferably **not catch it**. If you do catch the exception, you're responsible to rethrow the exception, throw a different exception or otherwise return an error condition so that the caller gets a chance to handle it instead. If you throw a different exception, you're responsible to set the initial exception as the root cause of the new exception, to avoid truncating the stack trace and complicate debugging. A common reason to catch an exception that you cannot handle is to be able to re-throw the error as a different kind of exception (see below).

1. **try** {
2. // do something that can raise an exception ...
3. } **catch** (MatrixException e) {
4. **throw** **new** TVCException("<Error message describing the operation that failed>", e);
5. }

## Throwing Exceptions

Exceptions should be used to report to the caller that an **exceptional** condition occurred, i.e., an error that the code couldn't recover from in a graceful way. Exceptions should not be used for normal flow control. When throwing an exception, always make sure that you provide a good (human readable) exception message that helps users and fellow programmers understand why the error happened (or at least what the code was trying to do when the error happened). If you throw an exception because the code caught another exception it is your responsibility to set the original exception as the root cause of the new exception to not truncate the stack trace.

### Exception types

Avoid declaring methods that throw the generic java.lang.Exception (or worse java.lang.Throwable). Try to be as specific as possible. If necessary, because you want to give the caller the possibility to distinguish between different kinds of exceptional conditions that can occur (so it can decide to handle them in different and appropriate ways), you should consider creating your own exception type (if no appropriate exception type already exist). Your custom exception types should directly or indirectly extend java.lang.Exception. Also, to maximize reuse of your custom exception type, it should have at least one public constructor that accepts a String message and a root cause of type java.lang.Throwable.

# Java Libraries

## Using Libraries

Don't reinvent the wheel. Chances are the problem you're trying to solve is not unique, it has probably been solved before (at least once). The good news is that – when working with ubiquitous technologies and programming languages such as Java and JavaScript – there's a pretty good chance that there's a library that solves the problem you're up against. Of course, there won't always be a library for you to use; perhaps its use is restricted or its quality bad. However, the fact is that there are a lot of open source libraries available that we can use if we needed to, and the goal should be to use one of these when it makes sense. The latter part of that statement is important, because we shouldn't use a library if the cost of using it is higher than the cost of doing it ourselves (assuming no other library is available that's better).

### Do I need it?

There's a lot of libraries already available in TVC, optimizers, ENOVIA, etc.

* Make sure that the same library or a similar library isn't already included in the ecosystem of your application
* Check with colleagues if they have any recommendations

### What license does it have?

The license is an important aspect of choosing a library.

* Does it have a "business-friendly" open source license? E.g., does it restrict use or distribution of the library?
* If no open source license (that meets requirements) is available:
* Can I get a commercial license and what are it's restrictions?
* If yes, due to financial impact, make sure you escalate the decision to the proper levels (Project Lead and above)

### What's the quality?

If the quality is bad you might be better off with a different library, or no library.

* Is it documented?
* Do they have: a wiki, code comments, book, etc?
* Who is using the library?
* If a lot of (well known) companies are using the library that's usually a good sign (just make sure you pick a stable version).
* Does it have an active community?
* Is it being actively developed?
* By individuals only or are companies sponsoring development?
* Do they have a mailing list or forum where users can ask questions?
* Are questions about the library being asked (and answered) on generic programming community sites, such as [StackOverflow](http://stackoverflow.com/)?
* Has any stable releases of the library been made?
* When was the last release?
* When is the next release?
* Do they have a road map?

## Java Libraries List

Below are the help libraries listed for different areas.

Note: Many of the libraries listed below are included with TVC, but likely not the latest version due to compatibilty issues (TVC has many dependencies). Since TVC repackages libraries (renames the packages to be below **com.technia.tvc.\***) you might be able to use a later version of a library if needed.

|  |  |  |  |
| --- | --- | --- | --- |
| **Area** | **Library** | **Package Name** | **TVC Package Name** |
| Email | [Commons Email](http://jakarta.apache.org/commons/email) | org.apache.commons.mail | N/A |
| Generic utilities | [Commons Lang](http://jakarta.apache.org/commons/lang/) | org.apache.commons.lang | com.technia.tvc.commons.lang |
| Java Bean | [Commons Beanutils](http://commons.apache.org/beanutils/) | org.apache.commons.beanutils | com.technia.tvc.commons.beanutils |
| Collections | [Commons Collections](http://commons.apache.org/collections/) | org.apache.commons.collections | com.technia.tvc.commons.collections com.technia.tvc.commons.collections15 (Use the latter for generics support in Java 5+) |
| XML | [Commons Digester](http://commons.apache.org/digester/) | org.apache.commons.digester | com.technia.tvc.commons.digester |
| XML | [JXPath](http://commons.apache.org/jxpath/) | org.apache.commons.jxpath | com.technia.tvc.commons.jxpath |
| HTTP | [Commons FileUpload](http://commons.apache.org/fileupload/) | org.apache.commons.fileupload | com.technia.tvc.commons.fileupload |
| Network | [Commons Net](http://commons.apache.org/net/) | org.apache.commons.net | com.technia.tvc.commons.net |
| Validation | [Commons Validator](http://commons.apache.org/validator/) | org.apache.commons.validator | com.technia.tvc.commons.validator |
| Logging | [Commons Logging](http://commons.apache.org/logging/) | org.apache.commons.logging | com.technia.tvc.commons.logging |
| Logging | [Log4j](http://logging.apache.org/log4j/) | org.apache.log4j | com.technia.tvc.log4j |
| Documents (Office) | [POI](http://poi.apache.org/) | org.apache.poi | com.technia.tvc.poi |
| Documents (PDF) | [iText](http://itextpdf.com/) (Since version 5.0.0 distributed under [*AGPL*](http://en.wikipedia.org/wiki/Affero_General_Public_License)) | com.lowagie | com.technia.tvc.lowagie |

# Java Logging

Logging at critical points in the code can be very useful when debugging and testing code.

When the code is run in a production system it should not generate any debug statements. It is often good to leave debug statements that can be turned on in a production (or other) system to trace problems. However, avoid trivial logging in code that's called repeatedly (e.g., hundreds or thousands of times during a single operation) if possible to not fill log files or terminal windows with noise (this excludes logging critical errors, they should always be logged).

## Use a Logger

Generally, a proper logging API should be used that allow the programmer to distinguish between critical and non-critical log messages, so that the output can be filtered, e.g., by turning off non-critical logging in production systems. The API should also allow programmers and system administrators to configure how and where log messages will be recorded (e.g., standard out, rolling log files, email, etc). Specifically, when developing a TVC plugin all logging should be performed with an instance of the Logger class included in TVC core (which is an adaptation of the famous Log4j framework). Each class that require logging should declare and use its own Logger instance as a static final field; the name of the field must be "logger" in lowercase and the enclosing class should be used as the logger category.

1. **import** com.technia.tvc.log4j.Logger;
2. // Other imports...
4. **public** **class** UpdateStatusAction {
5. **private** **static** **final** Logger logger = Logger.getLogger(UpdateStatusAction.**class**);
6. // The rest of the class...
7. }

As a consequence of above statements, explicit logging to standard out or err is not allowed. This includes:

* Writing to System.out or System.err
* Dumping stack traces by calling:
  + Thread.dumpStack()
  + printStackTrace() on a Throwable – unless the overloaded version of this method is used and the output captured in a stream that isn't connected to System.out nor System.err

### Message Criticality

A Log4j Logger requires that you indicate the importance of a message. The following levels are supported by default (ordered by criticality from lowest to highest):

|  |  |
| --- | --- |
| **Level** | **Description** |
| Trace | The lowest level. Used for tracing method calls, etc |
| Debug | For verbose (high frequency) messages |
| Info | For low frequency information messages |
| Warn | For (assumed) non-critical errors |
| Error | For critical errors |
| Fatal | The highest level. For errors that severely affect the well-being of the system as a whole |

### Logging Debug Messages

All non-critical log messages, that would mainly be used to debug the application when you know there's an error, should be wrapped in a condition to save CPU cycles when logging at the target level is disabled. The condition can be omitted when no processing is required to compute the log message, such as when the log message is a hard coded string literal (e.g., no string concatenation or other processing is required).

1. **if** (logger.isDebugEnabled()) {
2. logger.debug(...);
3. }

Critical log messages should not be wrapped in a similar condition.

### Logging Exceptions

As mentioned above, do not use printStackTrace() to write stack traces to standard out or err. Instead you should send the exception to the logger so that it can record the stack trace in the desired location when logging is enabled (as a general rule, logging of critical messages should always be enabled even in a production system to facilitate debugging).

1. logger.error("<Error message describing the operation that failed>", e);

### Exception handling

If you catch an exception it is your responsibility to log it or (re) throw.

* If you log and throw it's likely that the error will be logged more than once, which might be confusing or litter the log file with unnecessary noise
* If you throw a new exception, always add the original exception as root cause to not truncate the stack trace

See also section: [Java Exception Handling](#_Java_Exception_Handling)

# Java Naming Convention

The naming convention follows the **[standard Java naming convention](http://www.oracle.com/technetwork/java/codeconv-138413.html)**. Below are the most basic rules. In general, use meaningful names to maximize code readability.

## Class, Interface, Enum and Annotation Names

Starts with a capital letter, and each following word starts with a capital letter.

1. // Examples:
2. **public** **class** TaskDurationFieldDef **extends** FormFieldDef { }
3. **public** **class** ArticleNumberDataHandler **implements** DataHandler { }
4. **public** **class** DepartmentRangeHandler **implements** RangeHandler { }
5. **public** **class** UpdateSupplierCostAction **extends** NewTxAction { }

## Variable Names

Starts with a lower case letter, and each following word starts with a capital letter.

1. // Examples:
2. **private** String objectId;
3. **private** String name;
4. **private** String description;
5. **private** **int** position;

Note: Hungarian notation should NOT be used

The [**Hungarian Notation**](http://en.wikipedia.org/wiki/Hungarian_notation) naming style should not be used. Information about the type of a variable is superfluous in strongly typed languages like Java.

1. // bad
2. **int** iQuantity;
3. String sTypeName;
4. List<String> typeNameList;
6. // good
7. **int** quantity;
8. String typeName;
9. List<String> typeNames;

## Method Names

Starts with a lower case letter, and each following word starts with a capital letter. Follow variable naming convention for method parameter names.

1. // Examples:
2. **public** **void** load(Form form) { }
3. **public** **void** prepareEvaluation(EvaluationInput input, Column column) { }
4. **public** <T> T doAs(TxAction<T> action, String username, String password) { }
5. **public** **static** <I, O> List<O> map(Iterable<I> itr, Transformer<? **super** I, ? **extends** O> transformer) { }

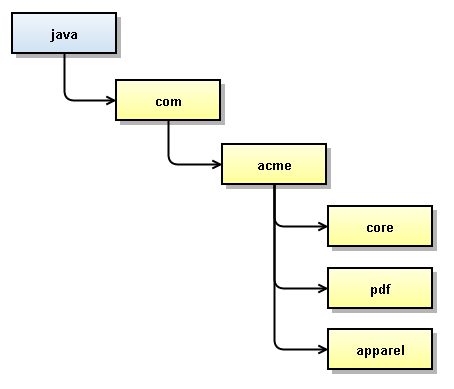
## Package Names

The purpose of a package naming convention is to standardize how code is structured across projects, with the intention to facilitate code reuse and reduce ramp up time for new developers (the time required to get up to speed with a project's code base). All classes must go into a package; use of the default package is discouraged. Package names should be in all lowercase letters and adhere to the following standard.

### Package Structure

1. com.<customer>.<module>.<type>

This example depicts a project named **acme** that has three modules: **core, pdf** and **apparel**. The names of these modules are used for dividing **Java, JSPs** and **TVC XML configurations** into separate directories (domains, modules).



### Standard type package names

This table lists standard type package names:

|  |  |
| --- | --- |
| **Type** | **Description** |
| action | For actions (classes that extend the Action class or a subclass thereof, such as NewTxAction). Classes in this package should end with Action and must be registered within the containing plugin's action mapping file (XML). |
| cellrenderer | For cell renderers (Classes that extend the TableCellRenderer class, or a subclass thereof). Classes in this package should end with Renderer. |
| datahandler | For data handlers (classes that implement the DataHandler interface). Classes in this package should end with Handler. |
| entryprocessor | For entry processors (classes that implement the EntryProcessor interface). Classes in this package should end with EntryProcessor. |
| formfielddef | For form field implementations (classes that extend the FieldDef class, or a subclass thereof). Classes in this package should end with FieldDef. |
| formprocessor | For form processors (classes that extend FormProcessor, or a subclass thereof). Classes in this package should end with Processor. |
| loader | For table loaders (classes that implement TableBeanLoader). Classes in this package should end with Loader. |
| model | For classes that represent data and contain business logic (see [Architecture](https://technia.jira.com/wiki/spaces/TECHNIA/pages/132841903/Architecture)). |

1. // Examples:
2. **package** com.technia.tva.apparel.bom.cellrenderer;
3. **package** com.technia.tva.apparel.common.datahandler;
4. **package** com.technia.tva.engineering.part.bom.action;
5. **package** com.technia.tva.program.task.wbs.updater;
6. **package** com.newbalance.spec.formfielddef;

# JPO Guidelines

## Introduction

This is a **JPO (Java Program Object)** in short:

* Defined in Enovia Business Modeler
* Written in Java
* Executed during trigger execution or for various data retrieval purposes

Note: Every **new** JPO that is created at Technia **must** extend a Java class. This approach enables standard debugging and logging. The need for de-mangled JPO's is also greatly reduced.

## Naming Convention

When creating a new JPO the name must be prefixed. The prefix should be the name of the project or a shorter version thereof. Examples:

* **ACMECommon**
  + The project is ACME
  + The JPO contains code not particular to any type or feature in the project
  + In short: a utility JPO. Please note that this type of common placeholders should be avoided since most code belongs to some specific type or feature
* **ACMEPart**
  + The project is ACME
  + The JPO contains **all** JPO-calls made within the project related to the type **Part**

## Code

The JPO should **not** contain any other code than below:

1. **public** **class** ${CLASSNAME} **extends** com.acme.program.PartProgram {}

The JPO extends a regular **Java** class which implements its methods. All subsequent calls on the JPO from Enovia will by Java inheritance be executed by the extended class. This approach leads to developers being able to use a standard logger and debugger appropriately.

Example **extended** class:

1. **public** **class** PartProgram **extends** EnoviaProgram {
2. **public** **void** setMyChange(Context ctx, String[] args) **throws** Exception {
3. }
4. }

**Use package naming convention** com.<customername>.<module>.program **as base for these classes.**

## Modifying existing OOTB JPO

If you have to modify a method in an existing JPO, it's generally easier to modify the Base JPO since changing the JPO the inherits it, usually forces you to change more than you want to change (or need to change). So, we should always add the super JPO method to lowest level class rather than modifying Base JPO

## When to use and not to use JPO's

JPO's are powerful but could also be a cause for bad performance.

**Use JPO:**

* When complicated logic is needed in a trigger

**Don't use JPO:**

* To generate a table cell value, the JPO will be executed for each cell and will be a performance problem. TVC datahandlers are a high performing alternative when TVC is available.

# JSP Guidelines

## JSP Coding Standards

Never put business logic in JSP-files as java code. If writing a new interface that requires new JSP pages, consider using tag libraries.

* If a change is needed in an existing JSP and the amount of Java-code is more than a few lines, create a Java-class in the source directory. Then call a method in that class, from the JSP, instead. This approach is superior in many ways; you can use Eclipse for your Java needs, use Jalopy for javadoc- and formatting needs and the changes in the JSP will be easier to keep track of.

## Using Tag Library

Tag libraries is a way of adding own html tags that can perform actions.

JSP custom actions are expressed using XML syntax. They have a start tag and end tag, and possibly a body. See example below:

1. **<technia:iterateBusinessObject** anAttribute="myInput"**>**
2. Object Name: **<technia:name/>**, **<technia:revision/>**
3. **</technia:iterateBusinessObject>**