KNIME Noding Guidelines

V2.5 –November 2015

This document describes best practices for the development, documentation and deployment of KNIME nodes, plug-ins and features. Some of the items below are designed to ease usage of new KNIME modules whereas others target interoperability and maintenance. In the following, the term "KNIME modules" refers to nodes, plug-ins or plug-ins grouped into features that form a functional entity.

Table of Contents

CHANGELOG

- Version 2.5 (November 2015)
  - Updated adapter cell and auto-conversion section
- Version 2.4 (April 2015)
  - Added section about expected startup behavior
- Version 2.3 (October 2014)
  - New sections about expected behavior of reader and writer nodes

GENERAL GUIDELINES

INSTALLATION

- ✓ A new KNIME extension should be installable via the Eclipse update mechanism.
  - An installed extension may require an external setup of the system (e.g. setting LD_LIBRARY_PATH or the presence of an external executable/application). However, if these requirements are broken (e.g. path not set), the extension should report an error message (e.g. the node fails during execution or the renderer is disabled; it should also set an appropriate error message to the NodeLogger). In no case it should crash or freeze KNIME.
  - The installation of an extension must not launch other installers or make modifications to the system other than in the eclipse installation directory.
  - An extension must not require any changes to the KNIME launch mechanism, i.e. no changes to the `knime.ini` file or special start-up wrappers are required.
- ✓ An installed KNIME extension should continue to work properly when the underlying Eclipse or KNIME version is updated (minor updates).
- ✓ Minor updates to the KNIME extension should be update-able through the Eclipse update mechanism.
- ✓ KNIME extension should not attempt to write into the installation directory during runtime. (Users might not have write permissions, depending upon where KNIME itself is installed on the file system.)
- ✓ Installation of an extension should not alter the behavior of KNIME at start up (for instance by asking for a license!). If a license is missing, the corresponding node(s) can report this before execution or during/after configuration. (This avoids having to go through several of these dialogs before KNIME starts for the first time.)
- ✓ Distributions of KNIME with additional 3rd-party extensions should always be based on a clean KNIME product as distributed on the KNIME website. There is no need to build own products. It is, of course, preferable to point users to your own update site.
- ✓ Nodes from 3rd party extensions should all be grouped within their own (usually top-level) category in the Node Repository. Moving nodes into existing, internal KNIME categories makes it impossible for users to identify (and find) nodes contributed by a specific vendor (and is in fact not possible any more since KNIME 2.8).
- ✓ **Backwards Compatibility:** Nodes whose settings have changed from one version to another (more or different configuration possibilities) should be backward compatible. If possible the new version should provide default values for the new settings fields. These default values should cause the previous behavior of the node, and avoid exceptions if the new value cannot be found in the settings. If it is not possible to make the new version backward compatible with the old version in the way described above, the old version of the node should be moved into an extra plugin (deprecated) or put into a separate source folder (e.g. “src-deprecated”) and removed from the node repository. The new version should be registered to the node repository. Using this mechanism, old workflows will load, because the old node can be found in the deprecated plugin, but all new workflows will use the new version, since there are no means to use the old version of the node in the GUI.

- ✓ Nodes that should not be used in new workflows any more (e.g. because there is a better replacement or they computed wrong results) should be deprecated. This is achieved by
  1. adding the attribute `deprecated="true"` to the <knimeNode>-element of the XML description file, and
  2. by removing it from the org.knime.workbench.repository.nodes-extension point in plugin.xml. The node classes themselves must not be deleted so that existing workflows still load but the old node does not show up in the repository any more.

STARTUP

- ✓ Plug-ins or nodes should not contact external services (e.g license servers) while KNIME or a workflow is loaded. If they really need to do so, a missing internet connection or unavailability of the required service must not delay startup or workflow loading.
- ✓ Extension must not pop up windows during startup. Instead they should use the *org.knime.workbench.ui.StartupMessage* extension point.

PLUG-IN AND FEATURE DEVELOPMENT

- ✓ Plug-ins in Eclipse can either be present as directories or Jar files in the installation. If your plug-in does not contain any external libraries (JARs or dynamic libraries) or resources, it should be installed as a single Jar file. For this, the plug-in's classpath (in the Runtime tab of the Manifest Editor), should only contain "." as single entry and in the Build tab there should only be an entry for ".", too. In the feature specify to not unpack the plug-in during installation. If you have external libraries, use a single Jar file for your classes in addition to the external libraries and let the plug-in be unpacked in the feature.
- ✓ If you are using external libraries, first make sure there is not already an Eclipse plug-in available that contains the library (Eclipse Orbit is a good source), which you can use. If you are using a library that may be used by others, then create a separate plug-in containing only the library and depend on it in your node plug-in. Try not to use any external libraries in node plug-ins if possible, because that may cause strange class loading problems and break other plug-ins using the same library.

KNIME USER EXPERIENCE

Using KNIME should be as easy as possible. Nodes which can guess their settings should do so if there is clear, unique preference for default settings. Dialogs that only have to be opened and closed before the node can be executed (because they guess the settings but force the user to open the dialog) should be avoided. The goal is to allow a fast creation and execution of workflows. Users should not be forced to manually set an option which can be automated. However, at the same time, nodes should not automatically set an option which could lead to confusion or significant waste of computing cycles.

When a node is first dragged onto the workbench, no node settings are set. There are three possible configuration scenarios (types of actions required inside a node – some of these may co-occur in a given node!):

1. **Auto-Configuration**

The node performs a defined operation on a column of a certain type. The configure method is the only place where you can determine if the input table has one or several columns of the compliant type:

a. **Auto-configure:** if there is only one column of the desired type available – choose it.
b. **Auto-guessing:** if there are several compliant columns, choose the first one and set a warning in order to inform the user about this selection.

Example: DecisionTreeLearner (auto-configure/auto-guessing of NominalValue column as class column), CDK Translators (auto-configure/auto-guessing of CDK column).

2. **Standard Settings**
Learner nodes often have many possible parameters and a do-nothing-configuration is impossible. But usually there are **standard settings**, which lead to reasonable results for most data and/or define a standard behavior for the underlying algorithm. If standard settings are possible, apply them in order to allow execution of the node with these standard settings without forcing the (possibly inexperienced) user to simply confirm them.

   Example: DecisionTreeLearner (use of standard settings like *gain ratio* as the quality measure and *no pruning*)

3. **User Action Required**

   For some nodes **no reasonable settings** can be guessed and a configuration where the node has no effect on the data is not possible. In this case, the node should stay red (unconfigured) in order to signal to the user that the settings have to be adjusted. When the dialog opens the default settings are such that they would not have any effect on the data. Thus, the user is forced to adjust the settings in the desired way in order to achieve any effect on the data.

Of course, the above-mentioned scenarios are not exclusive, i.e. a node can fall into several categories, e.g. apply standard settings and perform some auto-configuration/auto-guessing at the same time, as demonstrated in the example of the DecisionTreeLearner.

NODES

This section describes the expected behavior of a single node. This includes the results of the execution, the dialog and (if applicable) the view.

LOOK & FEEL

- ✓ Each node should have an icon (16px * 16px), which illustrates the functionality of the node. Reusing icons (from KNIME or own icons) should be avoided whenever possible.
- ✓ An icon should also be assigned to categories.
- ✓ A node should report its progress in the progress bar; provide an explanatory message whenever possible. This is accomplished by `ExecutionContext#setProgress(double progress, String message)`. If the task of the node consists of several subtasks, the `ExecutionContext` can be used to create sub-progress monitors with `ExecutionContext#createSubprogress(double d)`, where the argument in [0,1] defines the fraction of the whole task.
- ✓ It must be possible to cancel the node during execution without long delays. The call to `ExecutionContext#checkCanceled` will throw a `CanceledExecutionException` which is handled by KNIME.
- ✓ Every node should have comprehensible and extensive documentation for the *Node Description* view, which has to be located in the `*NodeFactory.xml` file, and should describe what the node does, which options are available in the dialog and – if the node has view(s) – what is displayed in the view(s).
- ✓ Port sorting: If a node has ports of different types, then they should be sorted in the following way: For out ports the data ports should be at the top and other port types towards the bottom. For in ports this is vice versa: data ports towards the bottom and other port types towards the top. (This avoids crossing connections between learner and predictor nodes.)

EXECUTION BEHAVIOUR

DATA HANDLING

- ✓ A node should be able to handle empty input tables (with no rows and/or no columns). Usually an empty output table is created, in some special cases the node may raise an exception and tell the user that it could not execute due to missing input data.
- ✓ A node should be able to handle tables containing missing values. Either the missing values can be handled in some way or the node refuses to execute and tells the user that he/she should remove the missing values (e.g. by using the Missing Value node).
- ✓ By default the columns of the input table should be retained in the output table. If a column is generated from other column(s), a node can offer an option to either replace or retain the used columns.
- ✓ Columns whose types are not handled by the node should be ignored, but passed through the node and not removed.
- ✓ A node should generally avoid the use of temporary files – but if unavoidable it must take care to close and delete them. Temporary files shall be created in the directory represented by `System.getProperty("java.io.tmpdir")`. (This variable can be changed in the KNIME preferences.) The methods in

`org.knime.core.util.FileUtil` take care of removing temporary files and directories and use the anticipated locations.

- ✓ Proper handling of KNIME data tables is supported by the use of `BufferedDataTable`, `BufferedDataContainer`, and `ColumnRearranger`. These tools take advantage of intelligent memory caching functionality inherent in KNIME and are optimally designed for the efficient manipulation and creation of data tables.
- ✓ When columns are added, changed, or removed, do not copy the table, but use the `ColumnRearranger` instead.
- ✓ Do not store any tables as member variables. If a view displays the incoming data, copy reasonable portions of it into a `BufferedDataContainer`, let the user configure how many rows should be displayed but use a reasonable small default value (2500 is recommended) and warn the user when rows are skipped. (Displaying millions of data points results in visual clutter and does not support understanding of the data.)
- ✓ A node implementation must not cast the elements of a data row to specific cell implementations. Instead it should always refer to the corresponding DataValue interface (as there may be more than only one data cell implementation). The following code is error-prone:

```
int x = .. // index of column containing DoubleValues
for (DataRow r : inData[0]) {
  DataCell cell = r.getCell(x);
  if (!cell.isMissing()) {
    // WILL FAIL FOR OTHER DOUBLE VALUE IMPLEMENTATIONS
    double d = ((DoubleCell)r.getCell(x).getDoubleValue());
    // do something with d
  }
}
```

A correct implementation would be:

```
double d = ((DoubleValue)r.getCell(x).getDoubleValue());
```

ROW ID

- ✓ If the incoming table is updated the row IDs should be preserved in the output table whenever possible.
- ✓ New row IDs should only be generated if new tables are generated. It is recommended to use the pattern “RowXX” for the row IDs (counts start at “0”!). Use the static method `RowKey.createRowKey(int)` to ensure compliance.

CONFIGURE

- ✓ The configure method should check for columns the node can work with. If there is no such column, throw an `InvalidSettingsException` with a detailed message.
- ✓ In the configure method it should be checked – if necessary – if the incoming table spec fits the user settings or if it contains columns the node can work with.

- ✓ Provide output table spec as complete as possible, i.e. also try to provide domain information such as possible values for nominal columns and lower and upper bound for numerical columns.
- ✓ See also the section on KNIME user experience, above.

EXECUTE

- ✓ All time-consuming tasks and calculations should happen in the execute method.
- ✓ Poll the cancel status regularly, by calling

  `ExecutionContext#checkCanceled()`.
- ✓ Incoming data tables can be arbitrarily large therefore do not keep them in memory and avoid unnecessary iterations.
- ✓ Before accessing the value of a DataCell, check if it is a missing value.
- ✓ Report progress by using ExecutionContext#setProgress.

LOGGING

- ✓ Do not use `System.out` or `System.err`, but rather the KNIME built-in logging mechanism via `org.knime.core.node.NodeLogger`.

COMPREHENSIVE LAYOUT

- ✓ The dialog should be comprehensive and should not change layout when different input tables are available. Grey out options which are not applicable instead. (Do not surprise the user with unexpected changes.)
- ✓ Each input field should provide a label which (very) briefly explains the input.
- ✓ Resizing of the component should not destroy the layout of the dialog.

DOCUMENTATION

- ✓ Each option in the dialog should be explained in the node description using the `<option>`-tag in the `*NodeFactory.xml` file.

MODAL SUB-DIALOGS

- ✓ Sub-dialogs should be used as rarely as possible. Input fields can either be grouped inside one panel or by using additional tabs (especially useful for expert settings). If sub-dialogs are unavoidable they must be modal, i.e. they open and remain in front of the parent dialog and disable the parent dialog. For example:

```
// figure out the parent to be able to make the dialog modal
Frame f = null;
Container c = getPanel().getParent();
while (c != null) {
   if (c instanceof Frame) {
        f = (Frame)c;
        break;
   }
   c = c.getParent();
}
// pop open the advanced settings dialog with our current
// settings
JDialog dialog = new DerivedJDialog(f);
dialog.setModal(true);
dialog.setVisible(true);
```

LOADING AND SAVING

- ✓ Loading and saving must work correctly, i.e. the settings are stored and restored on re-opening the dialog and if the workflow was saved and re-opened.
- ✓ Do not rely on the settings creation of your node model (e.g. it is possible to store node settings in the dialog, change them in a text editor and load them again).
- ✓ Throw a `NotConfigurableException` after loading all settings if the user cannot make a valid choice, e.g. if no column of correct type is in input data table spec. Provide a comprehensible message for the user. See also the section on KNIME User Experience.

VIEWS

- ✓ The view must resize properly.
- ✓ The KNIME built-in view properties *color*, *size* and *shape* should be supported where possible.

OPEN VIEW & EXECUTE

- ✓ It should be possible to open a view before execution of the underlying node without any exception. (KNIME makes sure nothing is displayed.)
- ✓ It should be possible to execute the node while the view is opened. (After execution is done, the view will be notified.)
- ✓ All time-consuming aspects of a view should be created during the execution of the node to ensure that the view opens instantaneously.
- ✓ Views are updated asynchronously to node execution. This means that in general the model must not provide information to the view before it has been fully created. Common errors are to build the view data during execute in several steps using an instance variable that the view can access. During a view update the view may now see incomplete data and even fail with e.g. ConcurrentModificationExceptions. Instead use a local variable in execute and assign it to the instance variable just before execute finishes. Exceptions are interactive views that display data while the node executes. Also in such cases make sure that the data provided to the view is consistent and does not change while the view is reading it.

OPEN VIEW & RESET

- ✓ It should be possible to reset the node while the view is opened.
- ✓ The view must then clean up all its displayed content.

SEVERAL VIEW INSTANCES PER NODE

- ✓ Several view instances for one node should be possible and independent of each other.
- ✓ Selection only happens locally in one view instance.

LOADING AND SAVING

- ✓ View content must be restored when an executed workflow is re-opened.
- ✓ *Hilite* status must be reset for a newly opened workflow.

HILITING

LISTENING

- ✓ Views should support *hiliting* as a listener when appropriate.
- ✓ On opening a view, the current *hilite* state should be displayed correctly.
- ✓ If a *hilite* event is received for a row, it should immediately be visible in the view. The standard KNIME visual variables for *hilite* should be used (`ColorAttr.HILITE`).
- ✓ If an *unhilite* event is received for that row, it should be displayed like the other *unhilited* data points (e.g. normal, faded or hidden).
- ✓ A clear *hilite* event should result in all displayed rows being displayed as *unhilited*.
- ✓ In order to detect the hilite status of a data point in the paint method, do not ask the HiliteHandler each time, but store the hilite status in the model.
- ✓ Aggregating nodes should use hilite translators to forward hilite events for their output data to the input data table(s).
- ✓ If a view listens to `HiLiteHandler`(s) from input ports, the listener has to be deregistered from the old `HiLiteHandler` and registered with the new `HiLiteHandler` in the `modelChanged` method. Be aware, that the HiliteHandler can be `null`.

HANDLING

- ✓ A view should support – where possible – the triggering of *hilite* for selected rows. Standard KNIME visual variables should be used for displaying selected, *hilited* and selected&*hilited*.
- ✓ A *hilite* menu should be provided with at least the following entries:
  - *hilite* selected
  - *unhilited* selected
  - clear *hilite*
- ✓ The use of predefined names defined by `HiLiteHandler.HILITE_SELECTED` etc. is highly recommended.
- ✓ It is also recommended to provide a menu in the menu bar and a context menu.
- ✓ Hilite events triggered in a view should only be sent to the handler and not be executed in the view until the event comes back through the listener interface (that means the view (if it is a listener) will also receive hilite events it triggered itself).

READER NODES

For nodes that read data from external sources, some additional rules apply:

- ✓ If possible, a reader node that reads from a single source should be able to read also from URLs (such as *file*, *http*, *knime*, or *ftp*) in addition to local file system paths.
- ✓ Reader nodes must not check the existence of the input sources while loading settings. They may check if the source location has a valid syntax but not test its existence. This should be done during `configure()`. (Reason being that settings may be loaded into a workflow, whereby the file isn't present on the hard disk – in this case the node should successfully load all settings but fail during configure()).
- ✓ The settings in the dialog of a reader must be savable even if the source location does not exist. The dialog should show a warning label, though.

WRITER NODES

For nodes that write data to external resources, some additional rules apply:

- ✓ Every writer should support writing to arbitrary URLs (if applicable) in addition to the local file system. It should support especially the *file* and the *knime* protocols but also be able to handle file system paths. The node should store local file system paths as entered by the user and not automatically convert it into file-URLs.
- ✓ Every writer node should have an option whether existing destination files should be overwritten or if the node should fail in this case.
- ✓ The dialog should check for file existence and display a warning label if the file exists and overwriting is disabled (consider using the class `org.knime.core.node.util.FilesHistoryPanel`). The settings in the dialog of a writer must be savable even if the destination file exists and overwrite is disabled.
- ✓ Writer nodes should check for existence of the destination file in `configure()`. Depending on the overwrite policy they should either fail with an error message telling the use that the destination file exists and cannot be overwritten due to the user setting, or a warning that the file is going to be overwritten (consider using `org.knime.core.node.util.CheckUtils.checkDestinationFile`).
- ✓ During execute the existence of the destination file (if on the file system) must be checked and the option from the dialog must be respected: if the file exists and can be overwritten the node should not display a warning (as it's a completely normal operation). If the file exists and must not be overwritten the node must fail with an error message, saying that the destination file exists and cannot be overwritten due to the user setting. The error message should show the file's path (consider using `org.knime.core.node.util.CheckUtils.checkDestinationFile`).
- ✓ Non-existing directories should not be created automatically, the node should fail instead.
- ✓ If the user cancels the node after a file has been created it should be deleted. If the writer creates multiple files at least the last recently created file should be deleted (usually it's not feasible to keep track all created files).
- ✓ Remote files (denoted e.g. by *ftp* or *knime* on a server, etc.) are always overwritten without any warning because it's not possible to determine if they exist by just using the URL.

- ✓ If new types are introduced, they should have an icon.
- ✓ New types should provide a comparator. If no comparator is provided, the standard comparator based on the string representation will be used. Therefore sorting will be possible even without a custom comparator.
- ✓ In addition a specialized serializer is recommended to ensure good performance. If possible do not rely on Java built-in serialization, since it is very slow and may break if the serialized classes change. Even for simple cells (holding an integer or an enumeration), this is recommended.
- ✓ Creation of a renderer is recommended for new data types. Note that if the renderer paints an image, then it is highly recommended to extend the `AbstractPainterDataValueRenderer`. If it displays some kind of textual representation then the `DefaultDataValueRenderer` should be used.
- ✓ Renderers should be registered at the corresponding extension point (org.knime.core. DataValueRenderer).
- ✓ If the new type claims to be compatible to other types a lossless conversion between those types must be possible (for example every integer can be converted to a double without any loss – but not vice versa). Although almost every value can be *displayed* as a String, it is not a String, thus it should not implement the StringValue interface.

PREDICTIVE ANALYTICS CONVENTIONS

LEARNERS

For so-called "learner" nodes that create a predictive model (decision tree, regression, clustering) an output table with parameters of the learned model should be provided. The contents are specific to the used algorithms. Here is an example for regression algorithms:

Parameter	**Value**
x	1
x^2	2
Squared error	0.34
…	…

PREDICTORS

For the corresponding predictors the following rules should be obeyed:

- ✓ All prediction columns should be named identical "Prediction (class)", where "class" is replaced by the corresponding column from the model.
- ✓ The user should be able change the name of the prediction column, e.g. in order to add the algorithm name ("Prediction (class) - Naive Bayes").
- ✓ Wherever applicable class probabilities/confidences should be output on demand. The columns should be names "P (column=value)", e.g. "P (sex=male)" or "P (sex=female)".
- ✓ The user should be able to add an optional suffix to each of these column, e.g. in order to specify the method that has been used (decision tree, neural network, …)

CHEMISTRY CONVENTIONS

This section summarizes the conventions regarding chemical types to ensure interoperability of different KNIME software partners.

SUPPORTED TYPES

- ✓ The standard molecular formats are:
  - ✓ SMILES
  - ✓ SDF
    - Structure block (Mol) can be extracted using standard KNIME node
    - SDF cells must be terminated by $$$$ characters
- ✓ The standard reaction format is
  - ✓ RXN
- ✓ Biological formats are available in org.knime.bio.types, currently only PDB. No standards were discussed as of now – please contact KNIME if that is required.
- ✓ Types in KNIME are simple textual wrappers (no interpretation!) A simple test should be able to read any of the above formats, write it out, and the result should be an exact textual match!
- ✓ Every node working on molecular representations should
- ✓ Accept and create if possible the standard formats (e.g. not SMILES if 3D structures are created/expected)
- ✓ If necessary accept and create (preferably) no more than one partner-specific format (Maestro, …) – these will not be part of the base chemistry classes (except for cases where they are of general interest; see below).
- ✓ Types of possibly general interest should be part of KNIME.
- ✓ If an extension is able to render any of the standard types (e.g. SD), it should register a renderer at the DataValueRenderer extension point.
- ✓ Each renderer implementation should make any attempt to not block or crash KNIME if it is unable to render an entry (because of a missing license, e.g.). Instead it should indicate the problem in the drawing area and issue an error to the logging facilities (once!)
- ✓ If you are supporting other types, make sure to use the ones available in org.knime.chem.types or inform KNIME to include it if there is a remote chance that others may also use this type.

  As of KNIME 2.9 the following types are part of KNIME:

  - Sdf
  - Mol
  - Ctab
  - Smiles
  - CML
  - Mol2
  - Sln
  - Rxn
  - Smarts
  - Inchi

  Please check org.knime.chem.types for an up-to-date list of currently supported types and contact the KNIME team before adding own, new types.
- ✓ Type conversion nodes should be provided if new types are introduced
  - A “Molecule to X” converter, taking ALL standard types as an input,

- A "X to Molecule" converter, producing ALL standard types on the output.

✓ All nodes should provide clear guidance about the expected and generated type:
- Tooltips on output, e.g. "table with additional X type column"
- Clear error if compatible type is missing, e.g. "expecting SDF, Maestro, or SLN column"
- Respect the following convention
  - "Molecule" stands for any one of the standard molecule types.
  - Individual types are: SDF, Smiles, Mol2, ChemML, or "X" (where X is descriptive and sufficiently unique).

HANDLING OF MULTIPLE REPRESENTATIONS – SETS/LISTS OF CELLS

✓ In case of multiple representations of a molecule (e.g. node computed conformations)
- If at all possible create one row for each conformation and add information about original row ID (similar to a pivoting node)
- Alternatively use "List of DataCells" (available in KNIME 2.0) to add several representations of the same type.
- KNIME allows composing/decomposing such sets/lists

READING & WRITING

✓ Readers
- Validate format
- Offer the ability to extract additional information
- Leave textual information intact!

✓ Writers
- Offer to combine additional columns into the type (e.g. SD cell alone or Mol-Cell with add'l cols)
- Otherwise leave textual information intact!

✓ Nodes computing additional information for molecular types (3D conformation…)
- Create a new column holding cells of the same type with this additional information "inserted".

FINGERPRINTS

✓ Use Fingerprint cell implementations available in KNIME
- DenseBitVector, SparseBitVector for {0, 1}
- DenseByteVector, SparseByteVector for counts (0-255)

✓ Different operations available on fingerprints (see package description), e.g.
- concatenate, and, or, xor, mask

TESTING & CERTIFICATION

✓ Developers should whenever possible submit test workflows for their own nodes
- KNIME may not have suitable test structures (format, specific properties)
- Some testing will be done by non-chemists.

✓ Developers should submit test cases
- Allows to test interoperability of 3rd party tools

Since molecules can be represented in different formats and especially vendor-specific formats, building complex workflows may require lots of converter nodes. Therefore an auto-conversion mechanism can be used. Auto-conversion is supported by so-called adapter cells that allow storing the converted molecules in an existing column next to other representations of the same (!) molecule.

For the complete API of adapter cells and type converters see the current Javadoc. Here are some rules that every vendor should adhere when using adapter cells.

- Check the incoming data types for *isCompatible* **and** *isAdaptable* when handling adapter cells.
- In execute either cast data cells to desired value type when dealing with "plain" cells or use the adapter access methods (*getAdapter*) when dealing with adapter cells.
- Never add any of the standard representations into an existing adapter, only vendor specific formats are supposed to be in adapter cells. Otherwise downstream nodes may not use the original representation but instead use a standard representation converted from a vendor specific format.
- Your own renderers should be aware of adapter cells.

In case you have your own molecular data format and you need to convert the standard formats (SDF & Smiles) into your format, you should provide your own adapter cell implementation that implements the same data value interfaces as the "normal" cell. The preferred value of the adapter cell should indicate the original source of the molecule. I.e. if you see a SmilesAdapterCell you know the original representation was Smiles. Likewise if you see a CDKAdapterCell this means that the molecule was created from scratch by CDK and all other representations in the adapter were derived from it.

When creating the automatic type converter, consider the following cases:

1. The input is already adaptable to your desired type. Then no conversion is necessary and you can use the input as-is (see example below)

```
CDKTypeConverter createConverter(DataTableSpec tableSpec, int columnIndex) {
   DataType type = tableSpec.getColumnSpec(columnIndex).getType();
   if (type.isAdaptable(CDKValue.class)) {
      return new CDKTypeConverter(type) {
         public DataCell convert(final DataCell source) throws Exception {
            return source;
         }
      };
   } ...
```

2. The input is compatible to your desired type. Then you should create an adapter cell that wraps the normal cell.

```
   else if (type.isCompatible(CDKValue.class)) {
      return new CDKTypeConverter(CDKAdapterCell.TYPE) {
         public DataCell convert(final DataCell source) throws Exception {
            return new CDKAdapterCell(source);
         }
      };
   } ...
```

3. The input is a writable adapter that contains types you can convert from. Then add the converted representation to the existing adapter.

```
else if (type.isCompatible(RWAdapterValue.class) &&
   (type.isAdaptable(SmilesValue.class) || type.isAdaptable(SdfValue.class))) {
   DataType resultType = type.createNewWithAdapter(CDKValue.class);
   return new CDKTypeConverter(resultType) {
      public DataCell convert(final DataCell source) throws Exception {
         RWAdapterValue adapter = (RWAdapterValue) source.
         return adapter.cloneAndAddAdapter(
               parseMolecule(adapter.getAdapter(SdfValue.class|SmilesValue.class),
               CDKValue.class);
      }
   };
} …
```

4. The input is a read-only adapter that contains types you can convert from. Then add the converted representation to the existing adapter. This case should not happen in practice because all subclasses of AdapterCell are writeable and implement RWAdapterValue.

5. The input is not an adapter cell but can be converted into your representation. Then create an adapter cell corresponding to the input type (e.g. an SdfAdapterCell or a SmilesAdapterCell) and add your representation to the adapter. Do not create your own adapter cell and add the original representation to it!

```
else if (type.isCompatible(SdfValue.class) {
   return new CDKTypeConverter(SdfAdapterCell.TYPE) {
      public DataCell convert(final DataCell source) throws Exception {
         SdfValue sdf = (SdfValue) source;
         AdapterCell ac = SdfCellFactory.createAdapterCell(sdf.getSdfValue());
         return ac.cloneAndAddAdapter(parseMolecule((SdfValue) source));
      }
   };
}
```

If you create new molecules or modify existing molecules, you must create a completely new column (or override the input column) using your own adapter cell implementations.

NODE CHECKLIST

for _____

CODE

Yes	No	N/A	
☐	☐	☐	load/save internals implemented if necessary
☐	☐		Proper progress reporting (percentage or at least messages)
☐	☐		Checks cancelation regularly during execute
☐	☐		Deletes temporary objects before execute finishes
☐	☐	☐	If node has non-data ports, it handles null PortObjects
☐	☐		Proper handling of missing values
☐	☐	☐	Uses `ColumnRearranger` whenever possible

DOCUMENTATION

Yes	No	N/A	
☐	☐		Node description matches node functionality
☐	☐		Node description is understandable
☐	☐		Port description(s) and port name(s) exist
☐	☐		Node has its own icon
☐	☐		Node's type (source, learner, etc.) is specified correctly
☐	☐	☐	View names look nice in context menu
☐	☐		Port names look nice in context menu

DIALOG

Yes	No	N/A	
			If node is not fully connected
☐	☐		allows configuration with useful settings or does not appear at all
			If node is fully connected
☐	☐		has default settings
☐	☐		after saving, closing, and re-opening the workflow the dialog shows the same settings
☐	☐		Labels and/or descriptions are correct and understandable
☐	☐		Layout is nice
☐	☐		Resizing the dialog doesn't mess up the dialog
☐	☐	☐	If dialog has sub-dialogs they are modal and don't disappear in background
			If node applies auto-guessing and is immediately executable
☐	☐	☐	opening and closing the dialog while the node is executed does not change settings, i.e. node stays executed

VIEW

Yes	No	N/A	
			If node is **not** executed
☐	☐		view shows correct content or message that no content is available
☐	☐		no exceptions are thrown (in event dispatch thread)
			If node is executed
☐	☐		View has proper window title
☐	☐	☐	Hiliting works as expected
			Standard colors/decorations are used for
☐	☐	☐	Normal data point
☐	☐	☐	Selected data points
☐	☐	☐	Hilited data points
☐	☐	☐	Colored data points (according to Color Manager)
☐	☐	☐	If the view has sub-dialogs they are modal and don't disappear in background
			After the node is reset,
☐	☐		view show correct content
☐	☐		no exceptions are thrown (in event dispatch thread)

EXECUTION

Yes	No	N/A	
☐	☐	☐	Node handles empty tables gracefully
☐	☐	☐	Node handles table with no columns gracefully
☐	☐	☐	Node handles table with no rows gracefully
			After re-opening a saved workflow with the node in **executed** state
☐	☐		node's state is restored
☐	☐		no error messages are shown on the node
☐	☐	☐	view(s) show correct (same) content
			After re-opening a saved workflow with the node in **configured** state
☐	☐		node's state is restored
☐	☐		no error messages are shown on the node
☐	☐	☐	view(s) show correct (same) content
☐	☐		node can be executed right away

TESTFLOWS

Yes	No	N/A	
☐	☐		Testflow(s) with Disturber node exist