Johar Interface Description File (IDF)
Format Specification

This document describes the specification of the Johar Interface Description File (IDF), version 1.0.

1 Syntax

1.1 Syntax of Attribute Declarations

An IDF consists of *attribute declarations*, possibly separated by whitespace. Whitespace has no particular meaning except inside string or longtext values (see below). Each attribute declaration is of one of the following forms:

Attribute
Attribute name
Attribute = value
Attribute name = value

Each *Attribute* is an upper-case identifier, i.e. a sequence of letters, numbers and underscores starting with an upper-case letter. Attributes must be those named in the specification below. Each *name* is a lower-case identifier, i.e. a sequence of letters, numbers and underscores starting with a lower-case letter. Generally, each *name* is chosen by the writer of the IDF.

Values are of four different forms: identifiers, strings, longtext, and structures. An *identifier* is a sequence of letters, numbers and underscores. A *string* is a sequence of characters in-between paired double quote characters. Inside a string, the backslash is the escape character: the sequence `\"` indicates a literal double quote, the sequence `\\` indicates a literal backslash, and the backslash followed by a newline indicates a literal newline.

Although multi-line strings can be constructed by using the backslash followed by a newline, this is not a very convenient way of entering them. A *longtext* is a value format designed for multi-line text. A longtext value consists of a sequence of two open curly braces ("`{{`") at the end of a line, followed by any lines of arbitrary text, followed by a sequence of two close curly braces ("`}}`") on a separate line.

A *structure* consists of a single open curly brace, followed by any number of attribute declarations, followed by a single close curly brace. Each attribute declaration has the format described at the top of this section. If an attribute declaration contains a structure, then the attributes in the structure are referred to as *child* attributes, and the enclosing attribute as the *parent* attribute; we also say that the child attributes are *sub-attributes* of the parent.

1.2 Example

As an example of all of the above, consider the attribute declarations shown in Figure 1. In that example, there happen to be no repetitions of attribute names, although in some cases attributes can be repeated. The `Table` attribute has a name but no value; the `Command` and `Parameter` attributes each have both a name and a value (both values happen to be structures); and all other attributes have a value but no name. `Type` has an identifier value, `BriefHelp` has a string value, and `MultiLineHelp` has a longtext value. `BriefHelp`, `MultiLineHelp`, and `Parameter` are sub-attributes of `Command`, and `Type` is a sub-attribute of `Parameter`.

```
Table clientData
Command open = {
  BriefHelp = "Open or create client data file."
  MultiLineHelp = {{
    Open the file containing data about clients.
    If the file does not exist, it is created.
  }}
  Parameter preferredExtension = {
    Type = text
  }
}
```

Figure 1: Example of IDF syntax.

1.3 Processing of Identifiers as Values

Wherever a double-quoted string literal can appear as a value for an attribute, an upper- or lower-case identifier can also appear if such an identifier would be in the right format. In this case the value is treated as a string consisting of just the characters in the identifier. For instance, the string `"tableEntry"` and the lower-case identifier `tableEntry` are treated exactly the same when they appear as the value of a parameter.

2 Allowed Attributes

In what follows below, if no mention is made of the *name* corresponding to an attribute, then the attribute must have no associated name. Each attribute is followed by a description of the *multiplicity* of the attribute, i.e. the number of times that the attribute can appear at the top level and/or as a sub-attribute of another attribute. If an attribute is optional (has multiplicity "0 or 1"), it may have a *default value*, which is also described.

2.1 Top-level Attributes

This section describes the attributes that can appear at the top level of the IDF.

- `Application`: The unique identifier of the application. (Required)
  - *value*: An upper-case identifier (the name of the application).

  Multiplicity: Exactly 1.

- `ApplicationEngine`: The class which contains the application-specific logic of the application.
  - *value*: An upper-case identifier (the name of the application engine class).

  Multiplicity: 0 or 1.
  Default value: The value of the `Application` attribute.

- `Command`: A structure representing the basic top-level unit of user-application interaction.

- *name*: A lower-case identifier (the name of the command).
- *value*: A structure. See Sections 2.2, Sub-attributes of `Command`, and 2.3, Sub-attributes of `Stage` and Single-Stage `Command`s.

Multiplicity: 1 or more.

- `CommandGroup`: A structure representing a group of related commands.
  - *name*: A lower-case identifier (the name of the command group).
  - *value*: A structure. See Section 2.6, Sub-attributes of *CommandGroup*.

  Multiplicity: 0 or more.
  A `CommandGroup` is a group of conceptually related commands. The commands themselves are defined in `Command` attributes; `CommandGroup`s group them by name, and contain no additional information about the `Command`s.

  An interface interpreter may use command group information in order to structure the interface. For instance, a classic GUI might present each command group in a separate dropdown menu on a menu bar, and any interface interpreter might use command group information in order to structure help information.

  A command cannot appear as a member of more than one command group. All commands that are not explicitly put into a command group in an IDF are put into an implicitly-defined command group, whose name is `commands`. In particular, if no `CommandGroup`s are defined, all commands become members of `commands`.

- `IdfVersion`: The version of the IDF format used in the creation of the file. (Required)
  - *value*: A double-quoted string literal, containing a valid major and minor version number.

  Multiplicity: Exactly 1.
  The version of the first public release of Johar will be 1.0. Therefore, initially all Johar IDFs should contain the declaration `IdfVersion = "1.0"`.

- `InitializationMethod`: The name of the application engine method used to initialize the engine.
  - *value*: A lower-case identifier (the name of the application engine method).

  Multiplicity: 0 or 1.
  Default value: `applicationEngineInitialize`.

- `Table`: The structure representing a list of similar entities presented to the user by the application engine.
  - *name*: A lower-case identifier (the name of the table).
  - *value*: A structure. See Section 2.7, Sub-attributes of `Table`.

  Multiplicity: 0 or more.

2.2 Sub-attributes of `Command`

This section describes the attributes that are acceptable sub-attributes of any `Command` attribute. In addition, single-stage `Command`s (commands that have no `Stage` attribute) can contain any of the attributes described below in Section 2.3, Sub-attributes of `Stage` and Single-Stage `Command`s.

- `ActiveIfMethod`: The name of an application engine method that can be called to determine if the user should be able to access the command.
  - *value*: A lower-case identifier. This is the method that will be called to determine if the command is active, i.e. if the user should be allowed to issue the command.

  Multiplicity: 0 or 1.
  If no `ActiveIfMethod` attribute is given, then the command is always active.

  It is recommended that the `ActiveIfMethod` is as efficient as possible (e.g. returning only the value of a field or data member), because it may be called frequently by the interface interpreter.

- `BriefHelp`: A brief help message describing the purpose of the command.
  - *value*: A string of 30 characters or fewer, containing no carriage return.

  Multiplicity: 0 or 1.
  Default value: The `Label` of the command, truncated to 30 characters, if needed.
  This attribute, if given, gives a very brief help message describing the purpose of the command, suitable for such things as ToolTips.

- `CommandMethod`: The application engine method that is called to actually carry out the command.
  - *value*: A lower-case identifier. This is the method that will be called to carry out the actual command.

  Multiplicity: 0 or 1.
  Default value: The command name.

- `Label`: The text describing the `Command` in the interface.
  - *value*: Any string.

  Multiplicity: 0 or 1.
  Default value: Derived from the name of the command using standard camel-case translation (see Section 4).
  The `Label`, if given, is the text that will appear (possibly with an appended ellipsis, "...") on the menu item, button, or other interface element corresponding to the command.

- `MultiLineHelp`: A thorough help message describing the purpose of the command.
  - *value*: A string or multi-line text of any length.

  Multiplicity: 0 or 1.
  Default value: The value of the `OneLineHelp` attribute.
  This attribute, if given, gives a thorough help message. IDF writers wanting to give thorough help for each *parameter* of a command should use the `MultiLineHelp` attributes of the *parameters*, as this will promote encapsulation and may (depending on the interface interpreter) give the user more control of the volume of help given.

- `OneLineHelp`: A one-line help message describing the purpose of the command.
  - *value*: A string of 80 characters or fewer, containing no carriage return.

  Multiplicity: 0 or 1.
  Default value: The value of the `BriefHelp` attribute.
  This attribute, if given, gives a help message describing the purpose of the command, somewhat longer than the `BriefHelp`. The `OneLineHelp` messages may be useful for the interface interpreter to display in a list.

- `Prominence`: An integer describing how prominently the `Command` should be shown to the user.
  - *value*: An integer greater than or equal to 0. Value ranges:
    - 3000 or more: High prominence. For instance, in a classic GUI, commands with prominence 3000 or more might be placed on the screen as buttons so that they are as quickly accessible as possible.
    - 2000-2999: Normal prominence.
    - 1000-1999: Reduced prominence.
    - 0-999: Low prominence. For instance, in a classic GUI, commands with prominence 0-999 might be placed on a "More commands" popup.

  Multiplicity: 0 or 1.
  Default value: 2000.

- `Question`: A question which may be asked of the user, given the values of the `Parameter`s and previous `Question`s in the command.
  - *name*: A lower-case identifier (the name of the question).
  - *value*: A structure. See Section 2.5, Sub-attributes of `Question`.

  Multiplicity: 0 or more.

- `QuitAfter`: An indication of whether the interface interpreter running the application should always terminate after the command terminates.
  - *value*: A Johar boolean (e.g., `yes` or `no`). See Section 3.

  Multiplicity: 0 or 1.
  Default value: `no`.
  A value of `yes` means that the interface interpreter should always quit after execution of the `CommandMethod`. A value of `no` means that it should expect more commands, unless there is a `QuitAfterIfMethod` which returns `true` (see below).

- `QuitAfterIfMethod`: An application engine method that is called to determine whether the interface interpreter running the application engine should terminate after the command terminates.
  - *value*: A lower-case identifier (the method to be called to determine quit status). The method should return a boolean (true if the application should terminate, false otherwise).

  Multiplicity: 0 or 1.
  If a `QuitAfterIfMethod` attribute is present, the indicated method is called after the `CommandMethod` is called. If the `QuitAfterIfMethod` returns true, then the interface interpreter takes this to be an indication that the application should terminate.

- Stage: One stage in the processing of the Command. Each stage may take separate Parameters.
  - *name*: A lower-case identifier (the name of the Stage).
  - *value*: A structure. See Section 2.3, Sub-attributes of Stage and Single-Stage Commands.

  Multiplicity: 0 or more.
  A Command can be broken up into several Stages. There can be zero or more explicit Stage sub-attributes. If there are zero explicit Stage attributes, then one stage is implicitly defined. All the attributes mentioned in Section 2.3 which appear as sub-attributes of the Command are then placed into the one implicitly-defined stage. The names of the Parameters in all of the Stages in a command must be disjoint.

Each stage in a multi-stage command may be handled at a separate time by an interface interpreter. A classic GUI interface interpreter, for instance, may present a multi-stage command using a "wizard"-style dialog box, in which the user can move forward or backward through the stages by clicking a "next" button. This may facilitate the elicitation of parameters for infrequently-given commands or commands that have many parameters.

If a Command has more than one Stage, then the interface interpreter may call the ParameterCheckMethods of each stage separately, and the DefaultValueMethods of each parameter separately (see below for more thorough information). This gives a mechanism by which the user can supply values for parameters in one stage, which are then used to compute the default values of other parameters. Thus, if the application programmer needs to collect user-input values of parameter A (e.g. input file name) before computing the default value of parameter B (e.g. output file name), parameter A can be in an earlier stage and parameter B in a later stage.

2.3 Sub-attributes of Stage and Single-Stage Commands

These sub-attributes can appear inside a Stage in a Command. If the command has no explicit Stages, they can also appear directly inside the Command, in which case they define the sub-attributes of the one implicit stage of the command.

- Parameter: A piece of data which comes from the user and is relevant to the Command, such as an integer, floating-point number or string.
  - *name*: A lower-case identifier (the name of the parameter).
  - *value*: A structure. See Section 2.4, Sub-attributes of Parameter.

  Multiplicity: 0 or more.

- ParameterCheckMethod: A method in the application engine that can be called to check the validity of the stage's parameters.
  - *value*: A lower-case identifier. This is the method that will be called to check the validity of the parameter values. It should return a string value (null or the empty string if all the parameters are valid, an error message if one or more parameters are invalid).

  Multiplicity: 0 or 1.
  If a ParameterCheckMethod is given, then if the user has input invalid values, the interface

interpreter can display the error message and allow the user to edit the erroneous values they originally gave. The `ParameterCheckMethod` should not do any processing related to the main function of the command, since the user may later decide to change the parameters, or even to cancel the command.

If no `ParameterCheckMethod` is given, then the parameter values will be checked by any implicit rules given for the parameter (e.g. the `MinValue` and `MaxValue` attributes for an `int` parameter).

2.4 Sub-attributes of `Parameter`

This section describes the attributes that are acceptable sub-attributes of any `Parameter` attribute.

- `BriefHelp`: A brief help message describing the meaning of the parameter.
  - *value*: A string of 30 characters or fewer, containing no carriage return.

  Multiplicity: 0 or 1.
  Default value: The `Label` of the parameter, truncated to 30 characters, if needed.
  This attribute, if given, gives a very brief help message describing the purpose of the parameter. For instance, a classic GUI could use this as the text of a ToolTip.

- `Choices`: A string representing the possible choices of values of a parameter of type `choice` (see below, attribute `Type`).
  - *value*: A double-quoted string literal.

  Multiplicity: For parameters of type `choice`, exactly 1. For other parameters, 0.
  The string contains the possible choices of values, separated by bar ("`|`") characters; e.g. `"portrait|landscape"`, `"clubs|diamonds|hearts|spades"`. To include a literal bar character in a choice, the bar should be preceded by a backslash ("\") character. To include a literal backslash character in a choice, use two backslashes ("\\").

- `DefaultValue`: The default value for the parameter.
  - *value*: A boolean, integer, real number or double-quoted string literal, as appropriate.

  Multiplicity: 0 or 1.
  If the user gives no explicit value for a parameter, then the interface interpreter will act as if they have given the `DefaultValue` as the value. For a parameter of type `choice`, the value must be one of the choices in the `Choices` attribute string.

- `DefaultValueMethod`: A method in the application engine to be called to give the default value for the parameter.
  - *value*: A lower-case identifier (the name of the method to be called to return the default value).

  Multiplicity: 0 or 1. A parameter cannot have both a `DefaultValue` attribute and a `DefaultValueMethod` attribute.
  In a Java application engine, the `DefaultValueMethod` must return a value of type `boolean`, `long`, `double` or `String`, as appropriate. For a parameter of type `choice`, the value returned must be one of the choices in the `Choices` attribute string.

- FileConstraint: A constraint on the status of a parameter of type `file` (see below, attribute `Type`).
  - *value*: A lower-case identifier. Accepted values:
    - `mustExist`: The file must exist at the time the command is issued.
    - `mustBeReadable`: The file must exist and be readable by the application at the time the command is issued.
    - `mustNotExistYet`: The file must not exist at the time the command is issued.
    - `none`: No constraint.

  Multiplicity: For parameters of type `file`, 0 or 1. For other parameters, 0.
  Default value: `none`.

- Label: The string that will be used to describe the parameter if and when the user is asked to enter a value for the parameter.
  - *value*: A double-quoted string literal.

  Multiplicity: 0 or 1.
  Default value: Derived from the name of the command using standard camel-case translation (see Section 4).
  The `Label`, if given, is the text that will appear to the user to indicate what parameter they are to enter. For instance, in a classic GUI, this would be text displayed beside the user-controllable widget used to set the value of the parameter.

- MaxNumberOfChars: The maximum number of characters that can be entered by the user as the value of a `text` parameter (see below, attribute `Type`).
  - *value*: An integer literal greater than or equal to 1; or the lower-case identifier `unlim`.

  Multiplicity: For parameters of type `text`, 0 or 1. For other parameters, 0.
  Default value: `unlim`.

- MaxNumberOfLines: The maximum number of lines that can be entered by the user as the value of a `text` parameter (see below, attribute `Type`).
  - *value*: An integer literal greater than or equal to 1; or the lower-case identifier `unlim`.

  Multiplicity: For parameters of type `text`, 0 or 1. For other parameters, 0.
  Default value: 1.

- MaxNumberOfReps: The maximum number of repetitions of the parameter that the user can give.
  - *value*: An integer literal greater than or equal to 1, giving the maximum number of repetitions allowed for this parameter; or the lower-case identifier `unlim`.

  Multiplicity: 0 or 1.
  Default value: 1.
  The user is not allowed to give more than `MaxNumberOfReps` repetitions of the parameter. See `MinNumberOfReps` for more detail.

- MaxValue: The maximum possible value of a parameter of type `int` or `float` (see below, attribute `Type`).

– *value*: An integer or real number literal.

Multiplicity: For parameters of type `int` or `float`, 0 or 1. For other parameters, 0.
Default value: The maximum value representable in a signed 64-bit integer (resp. 64-bit floating-point) number.

- `MinNumberOfReps`: The minimum number of repetitions of the parameter that the user can give (see below).

  – *value*: An integer literal greater than or equal to 0, giving the minimum number of repetitions allowed for this parameter.

  Multiplicity: 0 or 1.
  Default value: 1.
  `MinNumberOfReps` must be less than or equal to `MaxNumberOfReps`.

  The minimum and maximum number of repetitions indicate how many times the parameter can be repeated. If a `Parameter` has a `DefaultValue` or `DefaultValueMethod`, then any repetitions up to the `MinNumberOfReps` that are not explicitly changed by the end user are filled in by that value. If the `Parameter` has neither a `DefaultValue` nor a `DefaultValueMethod`, and `MinNumberOfReps` is greater than 0, then the end user is required to fill in at least `MinNumberOfReps` repetitions.

  For example, a command which takes a person's name as a parameter might have a `Parameter` of type `text` with no default value and a minimum and maximum number of repetitions equal to 1, obliging the user to enter a name. As another example, a command to show the differences between two files might take exactly two file parameters. To enforce this restriction, the programmer might create a `Parameter` with type `file` and with `MinNumberOfReps` and `MaxNumberOfReps` both equal to 2.

- `MinValue`: The minimum possible value of a parameter of type `int` or `float` (see below, attribute `Type`).

  – *value*: An integer or real number literal.

  Multiplicity: For parameters of type `int` or `float`, 0 or 1. For other parameters, 0.
  Default value: The minimum value representable in a signed 64-bit integer (resp. 64-bit floating-point) number.
  `MinValue` must be less than or equal to `MaxValue`.

- `MultiLineHelp`: A thorough help message describing the meaning of the parameter.

  – *value*: A string or multi-line text of any length.

  Multiplicity: 0 or 1.
  Default value: The `OneLineHelp` of the parameter.
  This attribute, if given, gives a thorough help message regarding the parameter.

- `OneLineHelp`: A one-line help message describing the meaning of the parameter.

  – *value*: A string of 80 characters or fewer, containing no carriage return.

Multiplicity: 0 or 1.
Default value: The `BriefHelp` of the parameter.
This attribute, if given, gives a help message describing the purpose of the parameter, somewhat longer than the `BriefHelp`. The `OneLineHelp` messages may be useful for the interface interpreter to display in a list.

- `ParentParameter`: The name of another parameter that controls the existence of the current parameter.

  – *value*: A lower-case identifier (the name of the parent parameter).

  Multiplicity: 0 or 1.
  This attribute and `ParentValue` are used for parameters that only make sense when some other parameter has a certain value. In this case the other parameter is referred to as the parent parameter.

  As an example, when printing a document, the user may choose to "print to a file", in which case the user must enter a file name parameter. However, if the user does not choose to print to a file, then there is no reason to expect the user to enter a file name. This situation can be set up by giving the `print` command two parameters: `printToFile`, a `boolean` parameter, and `outputFileName`, a `file` parameter whose parent parameter is `printToFile` and whose `ParentValue` is `true`. If the parent parameter does not have the indicated `ParentValue`, then the user is not required to enter any value for the parameter, regardless of any information about the minimum number of repetitions required.

  Circular `ParentParameter` references are not allowed. That is, a chain of `ParentParameter` references must end in a parameter with no `ParentParameter`.

- `ParentValue`: The value of the parent parameter that triggers the existence of this parameter.

  – *value*: An integer, floating-point or string literal, as appropriate.

  Multiplicity: For parameters with a `ParentParameter` value, exactly 1. For other parameters, 0. See `ParentParameter` for more detail.

- `Prominence`: An integer describing how prominently the `Parameter` should be shown to the user.

  – *value*: An integer greater than or equal to 0. Value ranges:
    * 3000 or more: High prominence. For instance, in a classic GUI, parameters with prominence 3000 or more may be placed in the most prominent location in the dialog box.
    * 2000-2999: Normal prominence.
    * 1000-1999: Reduced prominence.
    * 0-999: Low prominence. For instance, in a classic GUI, parameters with prominence 0-999 might be accessed only through an "Advanced" button in the parameter dialog box.

  Multiplicity: 0 or 1.
  Default value: 2000.

- `RepsModel`: A description of the intended model for the repetitions of the parameter.

  – *value*: A lower-case identifier. Accepted values:

    - ∗ `set`: The repetitions of the parameter are considered to be a set of disjoint values. The interface interpreter does not have to keep track of the order in which the user has input the values, and does not have to load them in the Gem in the order that the user has given them. The interface interpreter does not have to provide a way for the user to input multiple repetitions that have the same value.
    - ∗ `multiset`: The repetitions of the parameter are considered to be a set of values, with one repetition possibly having the same value as another. The interface interpreter does not have to keep track of the order in which the user has input the values, and does not have to load them in the Gem in the order that the user has given them. However, it does have to provide the user with the ability to give multiple repetitions with the same value.
    - ∗ `sequence`: The repetitions of the parameter are considered to be a sequence of values, with one repetition possibly having the same value as another. The interface interpreter must load them in the Gem in the order that the user has given them.

  Multiplicity: 0 or 1.
  Default value: `set`.

  The `RepsModel` may be used by an interface interpreter in order to structure the interface. For instance, in a classic GUI, a `choice` parameter with `MaxNumberOfReps = unlim` and `RepsModel = set` may be presented as a set of radio buttons, any of which can be turned on. In contrast, a `file` parameter with `MaxNumberOfReps = unlim` and `RepsModel = sequence` must be presented so that the user can specify a sequence of files, for instance so that the application engine method is guaranteed to process them in that order.

- `SourceTable`: The `Table` associated with a `tableEntry` parameter (see below, attribute `Type`).
    - *value*: A lower-case identifier (the name of the table from which the user selects values for the parameter).

  Multiplicity: For a parameter of type `TableEntry`, exactly 1. For other parameters, 0.

- `Type`: One of a few values describing what kind of parameter it is.
    - *value*: A lower-case identifier, signifying the type of the parameter. Possible values are:
        - ∗ `boolean`: Either true or false.
        - ∗ `choice`: One of a fixed number of choices. Every parameter with a `Type` of `choice` must have a `Choices` attribute.
        - ∗ `date`: A calendar date.
        - ∗ `file`: A file name.
        - ∗ `float`: A floating-point number.
        - ∗ `int`: An integer.
        - ∗ `text`: A text string.
        - ∗ `tableEntry`: The parameter is an entry from one of the tables declared in the IDF. Every parameter with a `Type` of `tableEntry` must have a `SourceTable` attribute.
        - ∗ `timeOfDay`: A time of day.

  Multiplicity: exactly 1.

2.5 Sub-attributes of `Question`

This section describes the attributes that are acceptable sub-attributes of any `Question` attribute.

`Question`s are similar to `Parameter`s. However, the intention is that the system expects a value for a `Question` only when an application engine method judges that a value is required, based on the values of the `Parameter`s. An interface interpreter may also provide a "cancel" option when asking a question, to allow the user to cancel the command in response to the question.

- `AskIfMethod`: The application engine method to call in order to see if the question should be asked. (Required)
  - *value*: A lower-case identifier, the name of the method to call to determine whether to require a value for the `Question`.

  Multiplicity: exactly 1.
  The application engine programmer can assume that the values for the `Parameter`s of the command, and all previous `Question`s, are accessible from the Gem when the ask-if method is called.

  For example, say that the programmer of an editor application wants the question "File has been modified. Save changes?" to be asked at appropriate points (for instance, for the `close`, `new` and `quit` commands). They could do this by adding the following `Question` to all appropriate commands:

```
Question saveIfModified = {
  Type = boolean
  Label = "File has been modified.  Save changes?"
  AskIfMethod = fileModified
}
```

- The following attributes of Parameter are also acceptable sub-attributes of any `Question`:
  - `BriefHelp`
  - `Choices`
  - `DefaultValue`
  - `DefaultValueMethod`
  - `FileConstraint`
  - `Label`
  - `MaxNumberOfChars`
  - `MaxNumberOfLines`
  - `MaxValue`
  - `MinValue`
  - `MultiLineHelp`
  - `OneLineHelp`
  - `Prominence`
  - `SourceTable`
  - `Type`

- The following attributes of Parameter are *not* acceptable sub-attributes of any `Question`:
  - `MaxNumberOfReps`
  - `MinNumberOfReps`
  - `ParentParameter`
  - `ParentValue`
  - `RepsModel`

  In addition, the `SourceTable` of any question of type `tableEntry` must refer to a non-browsable table.

2.6 Sub-attributes of `CommandGroup`

This section describes the attributes that are acceptable sub-attributes of any `CommandGroup` attribute.

- `Label`: The text describing the `CommandGroup` in the interface.
  - *value*: Any string.

  Multiplicity: 0 or 1.
  Default value: Derived from the name of the command group using standard camel-case translation (see Section 4).
  The `Label`, if given, is the text that will appear on the menu heading, button, or other interface element corresponding to the command group.

- `Member`: One of the commands which is a member of this `CommandGroup`.
  - *value*: a lower-case identifier specifying one command that is a member of this command group.

  Multiplicity: 1 or more.

2.7 Sub-attributes of `Table`

This section describes the attributes that are acceptable sub-attributes of any `Table` attribute.

- `Browsable`: A value indicating whether the user should be able to browse the table.
  - *value*: A Johar boolean (e.g., `yes` or `no`). See Section 3.

  Default value: `yes`.

  Some tables are intended to be presented to the user for them to browse and select rows in, separately from the processing of a given command. Others are simply intended to be tables of candidate values for parameters. The former kind of table is referred to as “browsable”, the latter “not browsable”. The `Browsable` attribute controls this behaviour.

- `DefaultHeading`: The default heading of the table.
  - *value*: A double-quoted string or long text.

Default value: The `Label` of the table.
This attribute gives the heading that will be associated with the table if no heading is set by the application engine. If no `DefaultHeading` attribute is given, then the heading will be derived from the name of the table using standard camel-case translation (see Section 4).

- `Label`: The text describing the `Table` in the interface.
  - *value*: Any string.

  Multiplicity: 0 or 1.
  Default value: Derived from the name of the table using standard camel-case translation (see Section 4).
  The `Label`, if given, is the text that will appear on the menu item, button, tab, or other interface element corresponding to the table.

2.8 Generated Attribute Values

When an IDF is read, it is processed into an internal form that can be used by an interface interpreter. For convenience, default values are generated automatically for some sub-attributes if they are not given explicitly in the IDF. The descriptions of these default values are given above, in the sections pertinent to the individual parameters.

The label and help attributes, if not given, are generated in a specific sequence. The sequence, along with the default values generated, are as follows.

- `Label`: The de-camel-cased version of the name of the command, command group, parameter or question.
- `BriefHelp`: The `Label` for the command, parameter, or question, truncated to 30 characters if necessary.
- `OneLineHelp`: The value of `BriefHelp`.
- `MultiLineHelp`: The value of `OneLineHelp`.

3 Johar Booleans

In some places in Johar IDFs, it is possible to give a boolean value. In all of these places, the following values are acceptable:

- `yes`, `Yes`, `YES`, `true`, `True`, or `TRUE`, all of which mean the same thing.
- `no`, `No`, `NO`, `false`, `False`, or `FALSE`, all of which mean the same thing.

This flexibility is intended to mirror the flexibility which interface interpreters are encouraged to have in accepting boolean values from end users of the applications.

Camel-Case Identifier	Translation
`add`	Add
`saveAs`	Save as
`findInThisPage`	Find in this page
`inputTextFile`	Input text file
`inputXMLFile`	Input x m l file
`inputXmlFile`	Input xml file

Figure 2: Examples of camel-case translation.

4 Camel Case Translation

To simplify interface description files, some values of attributes of commands and parameters are translated into strings shown to the user by following an algorithm for translating camel-case identifiers. This section describes this algorithm.

"Camel case" is the phrase used to refer to the practice of writing identifiers with mixed upper- and lower-case letters but no underscores. Often, when camel case is used, each upper-case letter is intended to start a word. The algorithm used for camel-case translation is therefore as follows.

1. Separate the words in the identifer by single spaces, assuming that each upper-case letter starts a new word.
2. Capitalize the first letter of the resulting string if it is not capitalized.
3. Translate all the rest of the letters in the resulting string to lower-case.

Figure 2 shows some examples of the effect of the camel-case translation algorithm. The first four examples are likely to be what the developer wants; the last two are unlikely to be what the developer wants. If the effect of the translation algorithm is not what the developer wants, then they can use the `Label` attribute to achieve what they want – for instance, by using a parameter name `inputXMLFile` with the attribute `Label = "Input XML file"`.