## ActionEvent

see also ActionListenerInterface

### Package

### Description

A semantic event which indicates that a component-defined action occured. This high-level event is generated by a component (such as a Button) when the component-specific action occurs (such as being pressed). The event is passed to every every ActionListener object that needs to be aware of it. The object that implements the ActionListener interface gets this ActionEvent when the event occurs. The listener is therefore spared the details of processing individual mouse movements and mouse clicks, and can instead process a "meaningful" (semantic) event like "button pressed".

### Methods

#### createInstance(command, sourceNode, value)

##### **Description**

Creates a new ActionEvent. Adobe does not allow object instances to be created outside of their packaging, so it must be done here.Parameters:

##### Parameters

command - a String that specifies a command associated with the event.

sourceNode - the Node that originated the event.

value- (optional) the value, if any, that is pertinent this event. Note that the data type is unspecified, since it is dependant on the context of the event.

Returns: a new ActionEvent instance

#### asString()

##### Description

Describes this object, it's properties and decendents.

##### Returns

the string description of this object.

## Action Listener Interface

Only one method that needs to be implemented for the ActionListener Interface: void actionPerformed.

actionPerformed takes one parameter, the type of event.

General interface implementation consists of

1. Create a script object for each event handler.
2. Define all legal actions for this event handler
3. In the actionPerformed event, compared the parameter with know actionEvents. Create additional functionality based on each action
4. Throw an exception for unknown actions, will be output to a log

### Example

var cActionEvent = script.core.event.ActionEvent;  
var cLog = script.core.util.Log;  
var cSystem = script.core.System;  
var cFormNodeManager = script.core.model.FormNodeManager; //definition to formNode manager  
  
//Define the ACTIONS for this handler  
var NAME\_CHANGE = "nameChanged";  
  
/\*\*  
\* Invoked when an action occurs.  
\* @param anEvent the ActionEvent instance that represents an action that  
\* has occurred in the User Interface.  
\*/  
function actionPerformed(anEvent)  
{  
switch(anEvent.getActionCommand())  
{  
case NAME\_CHANGE:  
myAddressChanged(anEvent.getSource());  
break;  
default:  
throw cException.createInstance("MyHandler"  
+ ".actionPerformed() - unrecognized event '"   
+ anEvent.getActionCommand() + "'.", "UnsupportedAction");  
}  
}

// functionality implemented for NAME\_CHANGE event  
function myAddressChanged(txtNode){  
cLog.trace("started");  
var nodeLabel = cFormNodeManager.getFormNodeForKey("labelHandlerEx");  
cLog.trace("started"+txtNode.rawValue);  
if(txtNode.rawValue && txtNode.rawValue==="show me"){  
nodeLabel.presence="visible";   
}else{  
nodeLabel.presence="hidden";  
}  
}//end fn

## ArrayUtil

A class that contains utility functions for manipulating Arrays.

### Methods

#### contains(anArray, aValue)

Description

Searches the passed Array to check if it contains the passed value. This is not very efficient, since it needs to do a scan of the array's contents looking for a match.

##### ****Parameters****

anArray - the Array to be checked.  
aValue - the value that may be in the Array. Note that if aValue

##### Returns

(boolean). true if the passed Array contains the passed value.

#### ****sortStringArray(aStringArray)****

##### **Description**

Sorts the passed String Array "in place" using a case insensitive sort.

##### Parameters

aStringArray- the string array to sort.

##### **Returns**

null

## BarcodeUtil

A class used to support forms that need to populate barcodes that contain all of their data.

### Methods

#### buildBarcodeData()

##### Description

Builds a String Array containing the data for a set of barcodes that will contain all of the data in this form. Each array entry represents the data for a single barcode. This barcode data conforms to version 2.0 of the NHPD barcode specification, where each barcode contains a header block followed by its content. When concatinated, the contents together represent base64 encoded zip data that contains the XML dataset(s) from this form.

*Typically called by populateBarcodes(), but exposed to support custom use.*

Header Format (all entries are padded as required):   
BARCODE\_VERSION - 2 digits  
NUMBER\_OF\_BARCODES - 2 digits  
CURRENT\_BARCODE\_INDEX - 2 digits (0 based)  
CURRENT\_BARCODE\_LENGTH - 4 digits (number of characters)  
  
e.g. version 2.0, the last of 15 barcodes, length 500: 201514000500

##### Returns

an array of strings of encoded data, suitable for creating individual barcodes.

#### clearBarcodes(barcodeInstMgr)

##### **Description**

Removes the barcodes from the form.

##### **Parameters**

barcodeInstMgr- the InstanceManager of the repeating subform that contains the individual barcode fields.

##### Throws

An UnexpectedDataType Exception if barcodeInstMgr isn't an Adobe instanceManager object.

#### populateBarcodes(barcodeInstMgr)

##### **Description**

raws a series of barcodes, containing base64 encoded zipped data on the form. The data comes from the form's default dataset. This assumes that the barcode fields are named "barcodeField".

##### **Parameters**

barcodeInstMgr - the InstanceManager of the repeating subform that contains the individual barcode fields.

##### **Throws**

UnexpectedState Exception if the "barcodeField" field isn't found in the subform.

## BasicFormatValidator

An object that supports validating the format of Adobe Field values against some basic formatting masks. This may be extended by custom implementations that support additional formats.

Implements FormatValidatorInterface.

### Constants

Supported format types are listed below:

EMAIL  
POSTAL\_CODE   
TELEPHONE  
ZIP\_CODE

### Methods

#### createInstance()

##### **Description**

Gets a new BasicFormatValidator. Adobe does not allow object instances to be created outside of their packaging, so it must be done here.

##### Returns

the new BasicFormatValidator instance

#### supportsFormatType(formatType)

Gets whether this implementation of format validator supports the passed format type.

##### Parameters

formatType a String that contains the name (key) of the format type in question.

##### **Returns**

a boolean - true if the passed format type is supported.

##### **Throws**

UnexpectedDataType Exception if formatType is undefined, null, or isn't a String.

#### validateFieldFormat (aField, formatType)

Validates the format of the value of the passed Adobe Field.

##### Parameters

aField- the Adobe Field to validate.

formatType- a String that contains the name (key) of the format validation to apply to the field.

##### **Returns**

a String containing the internationalized error message for any detected problem with the format of the value of the passed field. Null if there

were no problems.

##### Throws

UnexpectedDataType Exception if either of the parameters is undefined, null, or isn't the expected data type.

UnsupportedFormatType Exception if the formatType is unrecognized or unsupported by the implementation.

#### asString()

String representation of the object

##### **Returns**

a String that represents this object

## ChangeManager

### Description

An Object used to manager the attributes in a model Object

### Methods

#### createInstance()

##### **Description**

creates a new ChangeManager.

##### **Returns**

the new ChangeManager instance

#### isFieldChanged(aFieldName)

##### Description

Gets whether any changes have been registered for the field with the passed name.

##### Parameters

aFieldName- a String that contains the name of the field in question. These are written in camel case where the first letter is lower cased (eg. "firstName")

##### Returns

(boolean) -true if any changes are currently registered for the field with the passed name.

##### **Throws**

an UnexpectedDataType Exception if aFieldName is undefined, null, or isn't a String.

#### registerFieldChange(aFieldName)

##### **Description**

Registers a field as having changed.

##### Parameters

aFieldName a String that contains the name of the field in question. These are written in camel case where the first letter is lower cased (eg. "firstName");

##### **Returns**

null

##### Throws

an UnexpectedDataType Exception if aFieldName is undefined, null, or isn't a String.

#### reset()

##### Description

Clears all of the changes that have been registered with this manager.

##### Returns

Null

#### asString()

##### **Description**

String description of the object. Acts like a toString method.

##### **Returns**

string representation of the object

## CodeableInterface

This interface is implemented by model objects that need to be compared to Adobe node object. The methods defined here will allow the model objects that implement them to be uniquely identified. Adobe nodes that are bound to model objects that implement this interface should have an attribute named hdnCode. When binding an Adobe node with a model object make sure to set the value of getCode to the hdnCode attribute for the comparison to work.

### Methods

#### getItemCode()

##### **Description**

Gets the objects unique identifier code.

##### Returns

a String that contains this object's unique code.

## CodeSource

An object that provides unique codes to objects that require them. Typcally used by implementors of CodeableInterface or ListItemInterface when they don't already contain a unique code.

### Methods

#### getNextCode()

##### **Description**

Gets the next available unique code.

##### **Returns**

a String that contains the next available unique code.

#### resetLastcode(aNumber)

##### **Description**

Sets the last used unique code to the passed value if it is larger than the current one. This object uses a counter as a source of codes. The counter will always start at 0, so reopening a saved form may cause it to reissue existing codes. This can be called with the existing codes as part of the form initialization to increase the starting point to be above them.

##### Parameters

aNumber-the value of an existing code.

##### **Throws**

UnexpectedDataType Exception if aNumber is undefined, null, or doesn't contain a valid number.

## ConfigurationManager

A class that manages the form's configuration entries. It loads all JavaScript classes in "config" packages (i.e. treats them as configuration files). This creates a master configuration from all of these entries, so individual entries can be over-ridden by those in files that are subsequently loaded. Loading follows the ordering they are in the forms - proceding down the first branches to the first leaves before proceding to the next leaves and then the next branches.  
Configuration entry values can be of the following data types: Strings, Numbers, Booleans, Dates, or Arrays. Note that this will fail to load without warning if there is a problem with one of the script nodes (typically a JavaScript syntax problem that causes the code to not compile)! If this is suspected to be the case, turn on debugging in NodeUtil and check the Log for the last script processed - the next one is the likely problem.

### Package

script.core.util

### Methods

#### containsKey(key)

##### Description

Gets whether there is an entry for the specified key.

##### **Parameters**

Key- the key whose presence in this configuration is to be tested

##### **Returns**

(boolean) true if this configuration contains an entry for the specified key.

##### **Throws**

UnexpectedDataType Exception if key is undefined or null.

#### get(key)

##### **Description**

Gets the value for the specified key. Returns null if the configuration doesn't contain an entry for this key.

##### Parameters

key- the key whose associated value is to be returned.

##### **Returns**

the value for the specified key, or null if the configuration contains no entry for this key. Values can be of the following data types: Strings, Numbers, Booleans, Dates, or Arrays.

##### Throws

UnexpectedDataType Exception if key is undefined or null.

#### put(key, value)

##### Description

Over-rides the value of the specified key in the configuration.

##### Parameters

Key- the key whose associated value is to be added / updated. Note that this is coerced to a String by JavaScript when enforcing uniqueness. As a result, avoid using objects as keys since their String equivalents are the same ("Object").

##### Returns

the previous value associated with specified key, or null if there was no mapping for key. A null return can also indicate that the configuration previously associated null with the specified key.

##### **Throws**

An UnexpectedDataType Exception if key is undefined or null.

#### asString()

##### ****Description****

Lists the configuration entries

##### **Returns**

a string representation of all the configuration entities

## ControllerUtil

A class that contains utility functions for controllers.

### Methods

#### checkFormNode(value, controllerName, formNodeName)

##### ****Description****

Checks that the passed value is a FormNode. These are heavily used by controllers since they put a lot of these into constants. This is a simplified function similar to those in TypeEnforcer;

- This only checks that the FormNode isn't null (not its data type),

- it does not return anything, and

- it creates an Exception message more appropriate to the situation.

Parameters  
Value- the value to be checked.  
controllerName- the name of the calling controller class for exception reporting.  
formNodeName- the name of the FormNode in question for exception reporting.

Throws  
An UnexpectedDataType Exception if value is undefined or null.

## DataStore

A class used to store values that must persist. These are maintained in the data DOM of the form so that they continue to exist after the form has been saved and reopened. This behaves in a similar way to a Map. At this point, it only supports keys and values that are Strings.

### Methods

#### containsKey(key)

##### Parameters

key- a string that contains the key whose presence in this data store is to be tested.

##### **Returns**

(boolean) - true if this data store contains an entry for the specified key.

##### Throws

An UnexpectedDataType Exception if key is undefined or null.

#### get(key)

##### ****Description****

Gets the value for the specified key. Returns null if the data store doesn't contain an entry for this key.

##### Parameters

key - a string that contains the key whose associated value is to be returned.

##### Returns

a String -the value for the specified key, or null if the data store contains no entry for this key

##### **Throws**

An UnexpectedDataType Exception if key is undefined or null.

#### put(key, value)

##### **Description**

Sets the value of the specified key in the data store.

##### Parameters

key-a string that contains the key whose associated value is to be added / updated.

value-a String that contains the value to be associated with the specified key. Note that if this is undefined or null, an empty String will be stored.

##### ****Returns****

(String) -the previous value associated with specified key, or null if there was no mapping for key.

##### **Throws**

An UnexpectedDataType Exception if key is undefined or null.

#### remove(key)

##### **Description**

Removes the entry with the specified key from the data store

##### Parameters

**key-** a string that contains the key whose associated value is to be removed.

##### Returns

(String) -the previous value associated with specified key, or null if there was no mapping for key.

##### **Throws**

an UnexpectedDataType Exception if key is undefined or null.

#### asString()

##### **Description**

Lists the configuration entries.

##### **Returns**

the configuration entries.

## DateUtil

A class that contains utility functions for manipulating dates.

### Methods

#### parseIso8601Date(aString)

##### Description

Coerces the passed string to a Date. The string must follow the ISO 8601 standard. See "<http://www.w3.org/TR/NOTE-datetime>" for more details.

##### Parameters

aString-the string to be parsed.

Returns: a Date based on the passed String.

##### Throws

UnexpectedDataType Exception if aString is undefined, null, or not a String.

InvalidArgument Exception if aString does not follow the ISO 8601 standard.

#### toDateString(aDate, allowUndefined)

##### Parameters

aDate- the Date to be converted.

allowUndefined- (optional) true if this should allow an undefined or null value without throwing an exception. Defaults to false.

##### Returns

a String containing the formatted date portion of the passed Date. Null if aDate was null.

##### Throws

an UnexpectedDataType Exception if aDate is undefined, null, or not a Date.

## Exception

A general Exception implementation. It provides the ability to specify the exception name to minimize the need for subclassing. Note that, since over-riding toString() is not supported and the console uses toString(), these Exceptions are actually using Adobe's Error class which has implemented toString()

### Methods

#### createInstance(message, type)

##### Description

Gets a new Exception. Adobe does not allow object instances to be created outside of their packaging, so it must be done here.

##### Parameters

message- a string that contains the message the description of the problem.

type (optional)- a string that contains the type of the exception.

##### Returns

the new Exception instance

##### Throws

UnexpectedDataType Exception if any of the parameters aren't as indicated.

#### getDetails()

##### Description

Gets the details of this exception.

##### ****Returns****

a String listing this exception's properties.

#### getMessage()

##### **Description**

gets the exception's message, which is typically the reason for the failure.

##### Returns

a String that contains the message that describes the problem.

#### getType()

##### Description

Gets the type of exception this is, if there is any specified.

##### Returns

a String that contains the type of exception. "Generic" if not specified.

#### asString()

##### **Description**

String description of the object. This is implemented to get the same result as its toString(), which is implemented by Error.

##### **Returns**

the string description of this exception.

#### getDetails(anException)

##### **Description**

Gets the details of the passed exception.

##### Parameters

anException-the exception, which can be anything. Usually this would be an Adobe error or custom Exception.

##### **Returns**

a String. If passed an object (e.g. an Adobe error or custom Exception), this lists its properties. Otherwise, it returns the item it was passed.

##### Examples

var cException = script.core.util.Exception;  
  
throw cException.createInstance("SoapResponse.getDateResult("+ propertyName + ") encountered a problem: " + exception, "UnexpectedDataType");

## FailureException

A FailureException represents an assertion failure (or a call to fail()) during the execution of a JsUnit Test Function. Note that, since over-riding toString() is not supported and the console uses toString(), these are actually using Adobe's Error class which has implemented toString().

### Methods

#### createInstance()

##### **Description**

Gets a new FailureException. Adobe does not allow object instances to be created outside of their packaging, so it must be done here.

##### Parameters

Source (optional)- a string that contains the source of the failure.

Reason- a string that contains the reason for the failure.

##### **Returns**

the new FailureException instance.

#### getMessage()

##### **Description**

Gets the exception's message, which is typically the reason for the failure.

##### **Returns**

A String that contains the message that describes the problem.

#### getReason()

##### **Description**

Gets the reason for the failure

##### **Returns**

a string that contains the reason for the failure.

#### getSource()

##### **Description**

Gets the source of the failure.

##### **Returns**

a string that contains the source of the failure. Returns an empty string if source is not specified.

#### asString()

##### **Description**

Describes this object. This is implemented to get the same result as its toString(), which is implemented by Error.

##### **Returns**

a string description of the object

## FailureReporter

A class used to summarize and format Validation Failures for display purposes

### Methods

#### getFailureSummary(aValidationFailure)

#### ****Description****

Gets a summary of the error messages in the passed ValidationFailure. This will automatically traverse all nested structures to report on all errors. All instances of the same message are collapsed into one and the specifics about the originator of the messages is lost.

##### Parameters aValidationFailure the ValidationFailure that contains the error messages to be reported.

##### Returns

a String that contains a summary of error messages

##### Throws

UnexpectedDataType Exception if any of aValidationFailure is null, undefined, or not a ValidationFailure.

### getFailureDetails(aValidationFailure)

#### **Description**

Gets a detailed listing of the error messages in the passed ValidationFailure. This will automatically traverse all nested structures to report on all errors. All messages are reported separately in an indented structure under the sections that they happened in.

##### Parameters

aValidationFailure the ValidationFailure that contains the error messages to be reported.

##### Returns

a String that contains the details of the error messages.

##### Throws

UnexpectedDataType Exception if any of aValidationFailure is null, undefined, or not a ValidationFailure.

## FormatValidatorInterface

This defines an Interface that is implemented by Objects that are responsible for validating the format of Adobe Field values. Note that these must be implemented by custom Objects in the standard way;  
-The Class must contain a "createInstance" method.  
-The instance must contain a public "className" variable set to its name.  
-The instance must implement "asString()" to get descriptive information.

### Variables

var interfaceName = "FormatValidatorInterface"; //Required for all interfaces to identify the interface

### Methods

#### supportsFormatType(formatType)

##### **Description**

Gets whether the implementation of format validator supports the passed format type.

##### Parameters

formatType- a String that contains the name (key) of the format type in question.

##### **Returns**

(boolean) True if the passed format type is supported.

##### Throws

an UnexpectedDataType Exception if formatType is undefined, null, or isn't a String.

#### validateFieldFormat(aField, formatType)

##### ****Description****

Validates the format of the value of the passed Adobe Field.

##### Parameters

aField- the Adobe Field to validate.

formatType- a String that contains the name (key) of the format validation to apply to the field.

##### Returns

a String containing the internationalized error message for any detected problem with the format of the value of the passed field. Null if there were no problems.

##### Throws

UnexpectedDataType Exception if either of the parameters is undefined, null, or isn't the expected data type.

UnsupportedFormatType Exception if the formatType is unrecognized or unsupported by the implementation.

## FormNode

### Description

This is the object that represents nodes in the XFA form. These nodes can be retrieved using the FormNode Manager.

### Package

core.model.FormNode

### Methods

#### addChildNode()

##### Description

Adds a FormNode in the array of direct desendents

##### Parameters

aFormNode- a node of type FormNode to add to the collection

##### Returns

null

##### Throws

UnexpectedDataType if aFormNode is undefined, null, or not of type FormNode

#### getChildren()

##### Description

Gets all the immediate descendants for a Node

##### Parameters

none

##### Returns

An array of FormNodes with any direct descendants. An empty array is returned if there are no descendants.

#### hasChildren()

##### Description

Returns if the node has children

##### Parameters

None

##### Returns

(boolean) 'true' if there are children, false otherwise.

#### getDefaultValue()

##### Description

returns a node's internationalized value.

##### Parameters

None

##### Returns

(string) the default value for the node, undefined if not applicable

#### setDefaultValue()

##### Description

sets a nodes default value. Used to retrieve a value from the ResourceManager or to select the correct value from a picklist. This is used by the FormNodeFactory when initializing the Object

##### Parameters

a String that is the key to a node's default value

#### getDelegate(aFormNodeDelegate)

##### Description

Gets the delegate that is used to process requests about the characteristics of this node, if any.

##### Parameters

aFormNodeDelegate- the Form Node object that may contain a FormNode Delegate

##### Returns

the formNode Delegate object, null if there is none

#### hasDelegate()

##### Description

returns a boolean value for whether a FormNode has a FormNode or not.

##### Parameters

none

##### Returns

true if the formNode has a Delegate, false otherwise

#### setDelegate(aFormDelegate)

##### Description

Sets the delegate used to process requests

##### Parameters

aFormDelegate- the delegate object used to associate with a FormNode

#### getFieldNumber()

##### Description

the UI field Number that is displayed on the label

##### Parameters

none

##### Returns

a string that contains the number of this field node in the UI. Null if not applicable

#### setFieldNumber(aString)

##### Description

sets the field number for the node. This sets the number in the label of the Node

##### Parameters

aString- the string containing the numerical value

##### Returns

Null

#### getFormatType(aNode)

##### Description

gets the type of format validation to be applied to the data

##### Parameters

aNode- the Adobe node instance

##### Returns

a String that contains the format validation to be applied to the data entered into this node. Must be the built in types that are supported in the FormatValidator. Null if not applicable

#### setFormatType(aString)

##### Description

sets the type of format Validation to be applied to a node. These must be the types that are used in formatValidator

##### Parameters

:aString- a string containing the format validation to be applied to the node. Null if not applicable

##### Returns

null

#### getFullSomExpression()

##### Description

gets the full SOM expression (path) for a node

##### Parameters

none

##### Returns

a string representation of the full SOM expression

#### getKey()

##### Description

gets the unique key given to a FormNode. This will be undefined for the base FormNode

##### Parameters

None

##### Returns

a string value representing the key

#### getLabel(aNode)

##### Description

gets the UI label for the FormNode

##### Parameters

the Adobe Node instance to obtain the label from

##### Returns

a String that contains the UI label for the passed Node. Null if not applicable

#### setLabelKey(aString)

##### Description

sets the key for the label used for the node. This is used to retrieve the internationalized value from the ResourceManager

##### Parameters

aString- the string representing the unique key value

##### Returns

Null

#### isMandatory(aNode)

##### Description

returns whether an Adobe Node has been set as mandatory or not

##### Parameters

aNode- the Adobe Node instance

##### Returns

a boolean, true if the field is set as mandatory

#### setMandatory(aBoolean)

##### Description

Sets the mandatory value for the Adobe Node. The node will either be set to Mandatory or Optional

##### Parameters

aBoolean- a boolean value, true if the Node is to be set as required

##### Returns

null

#### getName(aNode)

##### Description

returns the node name that is being used for information purposes. Will either be the label or the key

##### Parameters

aNode- the adobe node instance to return the Node for

##### Returns

a string that contains the name of this node, either the label or key (if there is no label)

#### getParentNode()

##### Description

gets the parent formNode object

##### Parameters

None

##### Returns

The parent FormNode of this object. Null if it is the base node

#### getParentNodeKey()

##### Description

gets the unique key value for the parent of this node

##### Parameters

none

##### Returns

Returns a string representing the key value for the parent. Null if the parent is the root node

#### setParentNodeKey(aString)

##### Description

sets the unique key value of this FormNode's parent

##### Parameters

aString: the string value representing the key of the node

##### Returns

Null

#### hasParent()

##### Description

returns whether a node has a parent FormNode

##### Parameters

None

##### Returns

a boolean, true if the FormNode has a parent

#### getPickListValues (aNode)

##### Description

the picklist to be used in the UI for this node. If the node has a picklist key, gets the picklist from the resource manager. If it does not have a key checks to see if the delegate has pick list values.

##### Parameter

aNode- the Adobe Node instance who's pick list is required

##### Returns

the object containing the list of values. This will be either:

1) a String containing the comma delimited list of value

2) an array of objects that implements the ListItem Interface

3) an array of strings that are the value

Null if this is not applicable

#### setPickListKey()

##### Description

sets the key for the list of picklist values. These will be set in a resource file

##### Parameters

aString- the string representation of the key name for the picklist

##### Returns

Null

#### getSomExpression()

##### Description

returns the SOM expression for this node relative to it's parent

##### Parameters

None

##### Returns

a string representation of the relative SOM expression

#### setSomExpression(aString)

##### Description

sets the SOM expression of the Form Node relative to its parent

##### Parameters

the string object representing the relative SOM expression

##### Returns

Null

#### getToolTip(aNode)

##### Description

for an Adobe Node instance returns the string tooltip Value

##### Parameters

aNode- the adobe Node instance

##### **Returns**

a string representing the tooltip value of the node, null if there is no tooltip.

#### setToolTipKey(aString)

##### Description

sets the unique key value for the lookup to the tooltip value. Tooltip will be internationalized depending on the key value  
  
Parameters

aString- the string object representing the tooltip  
  
Returns

Null

#### isValidatedSeparately(aNode)

##### Description

when set to true, this node's parent will not call this nodes validateNode as part of the validation  
  
Parameters

aNode- the formNode object to check  
  
Returns

true if validatedSeparately is set to true

#### setValidatedSeparately(aBoolean)

##### Description

sets whether a FormNode object should be validated separately or not. When set to true, parentNode will not call validateNode()  
  
Parameters:

aBoolean: boolean value   
  
Returns

null

#### getInstanceManager (ancestorNodeInstance)

##### Description

gets the InstanceManager for an Adobe Node  
  
Parameters

ancestorNodeInstance- an optional parameter, the Adobe Node instance of the parent node.  
  
Returns

the InstanceManager object

#### getNodeInstance(ancestorNodeInstance)

##### Description

gets the Adobe node instance for this FormNode  
  
Parameters

ancestorNodeInstance- an optional parameter that is the parent Adobe Node Instance  
  
Returns

the instance, null if it is not found

##### Throws

UnexpectedDataType exception if the ancestorNode is not an Adobe Node

UnexpectedStateException of more than one instance of the Adobe Node is found

#### getNodeInstances(ancestorNodeInstance,fromAllAncestors)

##### Description

gets all instances of the Adobe Node that represents this formNode  
  
Parameters

ancestorNodeInstance- optional Adobe node of the parent of these instances

fromAllAncestors- a boolean if all instances under all ancestors should be returned

##### Returns

a nodeList of all the instances, null if none are found  
Throws

an UnexpectedDataType exception if the ancestorNode is not an Adobe Node

#### lock(aNodeInstance)

##### Description

locks this instance of the Adobe node and all children of the Adobe Node  
  
Parameters

aNodeInstances- optional if not repeating, represents the Adobe Node Instance to lock  
  
Returns

null

##### Throws

UnexpectedDataType exception if the ancestorNode is not an Adobe Node

UnexpectedStateException of more than one instance of the Adobe Node is found

#### unLock(aNodeInstance)

##### Description

makes this node instance and all the children editable  
  
Parameters

aNodeInstances- optional if not repeating, represents the Adobe Node Instance to unlock  
  
Returns

null

##### Throws

UnexpectedDataType exception if the ancestorNode is not an Adobe Node

UnexpectedStateException of more than one instance of the Adobe Node is found

#### updateUI(aNodeInstance)

##### Description

Updates the visual compoents in the form that are used to render the form Instances. Done mainly to update localization but can also be used for validation,setting required fields  
  
Parameters

aNodeInstance: the Adobe node instance that should be updated  
  
Returns

null

##### Throws

UnexpectedDataType exception if the ancestorNode is not an Adobe Node

UnexpectedStateException of more than one instance of the Adobe Node is found

#### validateNode(aNodeInstance, updateDisplay, aValidationFailure)

##### Description

valdiates this node and all the child FormNodes. Base version checks for mandatory fields, but functionality may be overridden by delegates   
  
Parameters

aNodeInstance- the instance of the Adobe node

updateDisplay- a boolean to indicate whether this should automatically update the colour of the node to reflect whether it passed validation

aValidationFailure- (optional) the ValidationFailure object to append failures to  
  
Returns

a ValidationFailure object to append failures to

##### Throws

UnexpectedDataType- exception if not the NodeIsntance is not an Adobe node)

UnexpectedState Exception- if FormNode is the base and has no children

#### asString()

##### Description

describes the object for this method, outputs detailed information about the node

##### Outputs

className, SOM, parentNodeKey, tooltipKey, formatType, picklistValues, fieldNumber, defaultValueKey, labelKey

##### Returns

a string description of the object

## FormNodeFactory

### Description

This class creates instances of FormNode. Used by the FormManager class

### Methods

#### createFormNode(key, formNodeValues)

##### Description

Creates a FormNode based on a map of passed values. Used directly by the FormManager to create instances of a requested node.

##### Parameters

key- the string value that uniquely identifies the node

formNodeValues: a map of the node's property values

## FormNodeManager

See also: nodeDefinitions, FormNode

### Description

Form node manager loads form nodes and allows lookup via SOM or node definition alias. The Form Node Manager loads any packages labelled as 'nodeDefinitions'. Node definitions are loaded in hierarchical order, so if a node alias is duplicated it will be overwritten by the last entry

### Public Methods

#### getBaseNode()

##### Description

Method that returns the root node i.e. in the NodeDefinitions namespace the node that does not have a parent.

##### Returns

the root node for the project

#### getFormNodeForKey(key)

##### Description

Returns a specific node object based on the key value that passed as a parameter

##### Params

key- a string value representing the alias given to a node

##### Returns

the form node if the key is found, otherwise null

##### throws

UnexpectedDataType exception if key is null or undefined

#### getFormNodeForSom(som)

##### Description

Returns a specific node based one the som expression provided for the node

##### Parameters

som- the SOM string for the node

##### Returns

the FormNode object for the SOM. Returns nulli if not found

##### Throws

UnexpectedDataType exception if som is null or undefined

### Examples

var cFormNodeManager = script.core.model.FormNodeManager;

var nodeFirstName = cFormNodeManager.getFormNodeForKey("firstName");

var nodeLastName = cFormNodeManager.getFormNodeForSom("xfa.form.MyTest.sfTombstone.txtLastName");

## InstanceManagerUtil

This class provides functionality to augment Adobe's instanceManager object. InstanceManagers are used to manage (add, remove, etc.) the instances of a given subform. Note that most of the provided methods require the subform's instanceManager be passed as a parameter. These are obtained by prepending an underscore to the name of the subform. For example, the instanceManager for "Pla.subPla" is "Pla.\_subPla".

### Package

core.util.InstanceManagerUtil

### Methods

#### addInstance(anInstanceManager)

##### **Description**

Adds a new instance of the subform being managed by the passed instanceManager to the bottom of the collection.

##### Parameters

anInstanceManager- an Adobe subform's instanceManager object.

##### **Returns**

the subform instance that was added.

##### Throws

an UnexpectedDataType Exception if anInstanceManager isn't an Adobe instanceManager object.

#### clearInstances(anInstanceManager)

##### **Description**

Removes all instances of the subform being managed by the passed instanceManager.

##### Parameters

anInstanceManager- an Adobe subform's instanceManager object.

##### **Returns**

null

##### Throws

UnexpectedDataType Exception if anInstanceManager isn't an Adobe instanceManager object.

#### getInstanceCount(anInstanceManager)

##### **Description**

Gets the number of instances there are of the subform being managed by the passed instanceManager.

##### Parameters

anInstanceManager - an Adobe subform's instanceManager object.

##### **Returns**

the number of subform instances that it is managing.

##### Throws

UnexpectedDataType Exception if anInstanceManager isn't an Adobe instanceManager object.

#### getInstanceManagerForSom(aSomExpression)

DescriptionGets the instanceManager for the given somExpression, it reformats the somExpression to the relative somExpression of its instanceManager.

##### Parameters

aSomExpression- a somExpression for an given Adobe node.

##### **Returns**

the instanceManager of the given somExpression.

##### Throws

InstanceManagerNotResolved Exception if aSomExpression can't be resolved to an InstanceManager (it isn't a valid node, or the node isn't repeating so doesn't have an InstanceManager).

#### hasInstance(anInstanceManager)

##### **Description**

Gets whether the subform being managed by the passed instanceManager has any instances.

##### Parameters

anInstanceManager- an Adobe subform's instanceManager object.

##### **Returns**

true if there are any instances of the passed instanceManager's subform.

##### Throws

UnexpectedDataType Exception if anInstanceManager isn't an Adobe instanceManager object.

#### initializeInstances(anInstanceManager, amount)

##### **Description**

Makes sure that there is at least the indicated minimum number of instances of the subform being managed by the passed instanceManager. New instances are added to the bottom as required. Any existing instances are left untouched.

##### Parameters

anInstanceManager - an Adobe subform's instanceManager object.  
amount (optional) - an integer representing the number of subform instances there should be as a minimum. Defaults to 1.

##### Throws

UnexpectedDataType Exception if the amount isn't a number or if anInstanceManager isn't an Adobe instanceManager object.

#### insertInstance(anInstanceManager, index)

##### **Description**

Inserts a new instance of the subform being managed by the passed instanceManager to its collection at the passed index.

##### Parameters

anInstanceManager- an Adobe subform's instanceManager object.  
index- an integer representing the 0 based index position of the subform to insert.

##### **Returns**

the subform instance that was inserted.

##### **Throws**

an UnexpectedDataType Exception if the index isn't a number, is out of range, or if anInstanceManager isn't an Adobe instanceManager object.

#### removeInstance(anInstanceManager, index)

##### **Description**

Removes an instance of the subform being managed by the passed instanceManager from its collection at the passed index.

##### Parameters

anInstanceManager- an Adobe subform's instanceManager object.  
index- an integer representing the 0 based index position of the subform to remove.

##### **Returns**

null

##### Throws

UnexpectedDataType Exception if the index isn't a number, is out of range, or if anInstanceManager isn't an Adobe instanceManager object.

#### removeSpecificInstance(aSubform, index)

##### **Description**

Removes the passed subform or, optionally, the one with the specified index.

##### **Parameters**

Subform- an Adobe subform object.  
index (optional)- an integer representing the 0 based index position of the subform to remove.

##### **Returns**

null

##### Throws

UnexpectedDataType Exception if the index isn't a number, is out of range, or if aSubform isn't an Adobe subform object.

#### resetInstances(anInstanceManager, amount)

##### **Description**

Clears any existing instances of the subform being managed by the passed instanceManager then adds the requested number of new instances.

##### **Parameters**

anInstanceManager - an Adobe subform's instanceManager object.  
amount - an integer representing the number of new instances of the subform there should be.

##### **Returns**

null

##### **Throws**

UnexpectedDataType Exception if the amount isn't a number or if anInstanceManager isn't an Adobe instanceManager object.

#### resetInstances(anInstanceManager, amount)

##### **Description**

Clears any existing instances of the subform being managed by the passed instanceManager then adds the requested number of new instances.

##### Parameters

anInstanceManager- an Adobe subform's instanceManager object.

amount an integer- representing the number of new instances of the subform there should be.

##### **Returns**

null

##### Throws

UnexpectedDataType Exception if the amount isn't a number or if anInstanceManager isn't an Adobe instanceManager object.

#### setNumberOfInstances(anInstanceManager, amount)

##### **Description**

Sets the number of instances there are of the subform being managed by the passed instanceManager. New instances are added to, or excess instances are removed from, the bottom of the collection as required. Otherwise, existing instances are left untouched.

##### Parameters

anInstanceManager- an Adobe subform's instanceManager object.  
amount- an integer representing the number of instances of the subform there should be.

##### **Returns**

null

##### Throws

UnexpectedDataType Exception if the amount isn't a number or if anInstanceManager isn't an Adobe instanceManager object.

## JsUnit

### Description

A class that contains the implementation of the core JsUnit functionality assertions. Used for unit testing.  
Based on work by Edward Hieatt, edward@jsunit.net (http://www.jsunit.net)

### Methods

#### assertArrayEquals()

##### **Description**

Checks that an array is equal to another by checking that both are arrays and then comparing their elements using assertObjectEquals

##### Parameters

comment -optional, displayed in the case of failure. Typically the name of the function being tested.

expected- value of the expected array

actual- value of the actual array

##### Throws

FailureException- if the actual value does not equal the expected value

AssertionArgument Exception- if an incorrect number of arguments is passed

assertArrayEqualsIgnoringOrder()

##### **Description**

Checks that two arrays have the same contents, ignoring the order of the contents.

##### **Parameters**

Comment (optional) **-** displayed in the case of failure. Typically the name of the function being tested.

array1- first array

array2 -second array

##### **Throws**

FailureException if the two arrays have different contents

AssertionArgument Exception if an incorrect number of arguments are passed

#### assertContains()

##### Description

Checks that a collection contains a value by checking that collection.indexOf(value) is not -1.

##### Parameters

Comment (optional):-displayed in the case of failure. Typically the name of the function being tested.

Collection - the collection to be checked

Value- the values to search for

##### Throws

FailureException if the collection does not contain the value

AssertionArgument Exception if an incorrect number of arguments are

assertEquals()

##### **Description**

Checks that two values are equal (using ===)

##### Parameters

Comment (optional)- displayed in the case of failure. Typically the name of the function being tested.

Expected- the expected value

Actual- the actual value

##### **Throws**

FailureException if the values are not equal

AssertionArgument Exception if an incorrect number of arguments is passed

assertEvaluatesToFalse()

##### Description

Checks that a value evaluates to false in the sense that value == false. It is safer to use assertFalse(), since many values coerce to an unexpected boolean equivalent.

##### Parameters

comment (optiona)l- displayed in the case of failure. Typically the name of the function being tested.

value- the value to test against false

##### Throws

FailureException if the actual value does not evaluate to false

AssertionArgument Exception if an incorrect number of arguments is passed

#### assertEvaluatesToTrue()

##### **Description**

Checks that a value evaluates to true in the sense that value == true. It is safer to use assertTrue(), since many values coerce to an unexpected boolean equivalent.

##### Parameters

Comment (optional)- displayed in the case of failure. Typically the name of the function being tested.

Value- the value to be tested

##### Throws

FailureException if the actual value does not evaluate to true

AssertionArgument Exception if an incorrect number of arguments is passed

#### assertFalse()

##### **Description**

Checks that the passed value is a boolean data type or object with a value of false.

##### Parameters

Comment (optional)- displayed in the case of failure. Typically the name of the function being tested.

value - value that is being tested and expected to be false

##### Throws

FailureException if the given value is not a boolean data type or or object, or if its value isn't false

AssertionArgument Exception if an incorrect number of arguments is passed

#### assertHashEquals()

##### **Description**

Checks that a hash is has the same contents as another by iterating over the expected hash and checking that each key's value is present in the actual hash and calling assertEquals on the two values, and then checking that there is no key in the actual hash that isn't present in the expected hash.

##### Parameters

Value- the expected hash

Value- the actual hash

##### Throws

FailureException if the actual hash does not evaluate to true

AssertionArgument Exception if an incorrect number of arguments is passed

assertNaN()

##### **Description**

Checks that a value is NaN (Not a Number)

##### Parameters

Value- the value to be tested

##### Throws

FailureException if the value is a number

AssertionArgument Exception if an incorrect number of arguments is passed

#### assertNotEquals()

##### **Description**

comment optional, displayed in the case of failure. Typically the name of the function being tested.

##### **Parameters**

value1- the first value to compare

value2- the second value to compare

##### **Throws**

FailureException if the values are equal

AssertionArgument Exception if an incorrect number of arguments is passed

#### assertNotNaN(value,comment)

##### **Description**

Checks that a value is not NaN (i.e. is a number)

##### Parameters

Comment- optional, displayed in the case of failure. Typically the name of the function being tested.

Value- the value to check

##### Throws

FailureException if the value is not a number

AssertionArgument Exception if an incorrect number of arguments is passed

#### assertNotNull(value)

##### **Description**

Checks that a vaiue is not null

##### Parameter

Value- the value to test

##### Throws

FailureException if the value is null

AssertionArgument Exception if an incorrect number of arguments is

#### ****assertNotUndefined()****

##### **Description**

Checks that the value is not defined

##### **Parameters**

value- the value to test against

##### **Throws**

FailureException if the value is undefined

AssertionArgument Exception if an incorrect number of arguments is

#### assertNull(value)

##### **Description**

Checks that a value is null

##### Parameters

Value- the value to test against

##### Throws

FailureException- if the value is not null

AssertionArgument- Exception if an incorrect number of arguments is

#### assertObjectEquals()

##### **Description**

Checks that an object is equal to another using === for primitives and their object counterparts but also descending into collections and calling

##### Parameters

Comment- optional, displayed in the case of failure. Typically the name of the function being tested.

value1- the expected value

value2- the actual value

##### Throws

FailureException if the actual value does not equal the expected value

AssertionArgument Exception if an incorrect number of arguments is passed

#### assertRoughlyEquals()

##### **Description**

hecks that two numeric values are within a tolerance of one another

##### Parameters

Comment -optional, displayed in the case of failure. Typically the name of the function being tested.

Value1- a value used for comparision

Value2- the other value for comparision

Tolerance- the range the two values need to fall between

##### Throws

FailureException if the two values are not within tolerance of each other

AssertionArgument Exception if an incorrect number of arguments is passed

#### assertTrue()

##### Description

hecks that the passed value is a boolean data type or object with a value of true.

##### **Parameters**

Comment -optional, displayed in the case of failure. Typically the name of the function being tested.

Value- the value that is expected to be true

##### **Throws**

FailureException if the given value is not a boolean data type or or object, or if its value isn't true

AssertionArgument Exception if an incorrect number of arguments is passed

assertUndefined()

##### **Description**

checks if a value is undefined

##### Parameters

Comment (optional)- displayed text in the case of failure. Typically the name of the failed value/method

Value- the value to check

##### Throws

FailureException if the value is not undefined

AssertionArgument Exception if an incorrect number of arguments is passed

#### logException(exception)

##### **Description**

A convenience method that creates a log entry for the passed exception. Note that this only properly deals with AssertionArgument Exceptions and FailureExceptions.

##### Parameters

Exception- the exception to log. If its unrecognized, this will log a message to that effect.

#### logNonException(comment)

##### Parameters

Comment- comment the name of the offending function.

## ListItemInterface

This defines an Interface that is implemented by Objects that can be displayed in the User Interface picklists. These must have a unique code that is hidden and a description that is displayed.

Extends CodeableInterface - not technically, but we make it look that way by giving ListItemInterface every method and constant available to CodeableInterface

### Variables

var interfaceName = "ListItemInterface"; //Required for all interfaces

### Methods

#### getItemCode()

##### **Description**

Gets this list item's unique code. This is not displayed in the list.

##### **Returns**

A String that contains this list item's unique code.

#### getItemDescription()

##### **Description**

Gets this list item's description. This is displayed in the list.

##### **Returns**

A String that contains this list item's description.

## LocalizedException

### Description

A LocalizedException is an Exception whose message is looked up using ResourceManager. These are used when the exception message is likely to be  
displayed to a user. Note that, since over-riding toString() is not supported and the console uses toString(), these are actually using Adobe's Error class which has implemented toString().

### Methods

#### asString()

##### **Description**

Provides a string description this object.

##### **Returns**

the string description of this object

#### createInstance(key)

##### **Description**

creates a new LocalizedException.

##### Parameters

Key- the String whose associated value is the message

additionalValues (optional)- these additional parameters are used to support the dynamic replacement of placeholders in a retrieved String message. Placeholders follow the syntax "{#}", where # is the 1-based index of the additional arguments. Any number of additional arguments can be passed. Eg. get("test", "this", "out") will retrieve the message with the key "test", then replace all "{1}"s with "this", and replace all "{2}"s with "out".

##### **Returns**

the new LocalizedException instance

#### getKey()

##### **Description**

Gets the key used to look up the message in the ResourceManager.

##### **Returns**

a string that contains the key used to look up the message.

#### getMessage()

##### **Description**

Gets the exception's localized message, which is typically the reason for the failure.

##### **Returns**

a String that contains the message that describes the problem.

## Log

This class is used to log messages, and behaves in a similar way to Log4j. It will look for a "logLevel" entry in configuration for its settings.

### Package

Core.Resources.Util

### Constants

LEVEL\_CRITICAL   
LEVEL\_ERROR   
LEVEL\_WARN   
LEVEL\_INFO   
LEVEL\_TRACE

### Methods

#### setAcrobatDocument(aDocument)

##### Description

Sets the Adobe Document object to be used when writing log entries into a file. The easiest way to retrieve that object is by calling "event.target" from an event. As a result, it is recommended that this be called as the first thing in your form initialization event processing.

##### Parameters

aDocument- an Adobe Document object to be used when writing log entries to file.

##### **Returns**

Null

##### Throws

UnexpectedDataType Exception if aDocument is undefined, null, or isn't an Adobe Document object.

#### getLogLevel()

##### **Description**

Gets the level of messages that are to be logged. Refer to the class constants to understand the meaning of the values.

##### **Returns**

a number that indicates the level of messages that are to be logged.

#### setLogLevel(aNumber)

##### **Description**

Sets the level of messages that are to be logged. This must be set to one of class constants.

##### Parameters

aNumber - the message to be logged.

##### **Returns**

null

##### **Throws**

an InvalidArgument Exception if aNumber is an invalid level.

#### isShowExceptionDetail()

##### **Description**

Gets whether an optional Exception, when passed, is displayed using its toString() or shows its full detail (i.e. includes the stack trace).

##### **Returns**

a boolean that is true if Exceptions should have their full detail displayed.

#### setShowExceptionDetail(aBoolean)

##### **Description**

Sets whether an optional Exception, when passed, is displayed using its toString() or shows its full detail (i.e. includes the stack trace).

##### Parameters

aBoolean -( true) if Exceptions should have their full detail displayed.

##### Throws

UnexpectedDataType Exception if aBoolean is undefined, null, or isn't a boolean.

#### iswriteToFile()

##### **Description**

Gets whether this should also write log entries into a file. When set to true, his will create a file attachment in the form with the log entries. These entries are cached and written to the file when \_MAX\_CONSOLE\_SIZE is exceeded or when flushFileCache() is called.

Note: when using this feature, setAcrobatDocument() must be called from an initialization event and flushFileCache() must be called from a preSave event.

##### Returns

(boolean) true if this should also write log entries into a file.

#### setWriteToFile(aBoolean)

##### **Description**

Sets whether this should also write log entries into a file. When set to true, this will create a file attachment in the form with the log entries. These entries are cached and written to the file when \_MESSAGE\_MAX\_SIZE is exceeded or when flushFileCache() is called.

##### Parameters

aBoolean true if this should also write log entries into a file.

##### Throws

UnexpectedDataType Exception if aBoolean is undefined, null, or isn't a boolean.

#### critical(message, anException)

##### **Description**

Logs a critical message. Typically used when an event has happened that will cause the application to behave erratically or cease to function at all.

##### Parameters

message- the message to be logged.  
anException (optional)- an Exception that was thrown, This is appended to the message when it is displayed.

##### **Returns**

null

#### error(message, anException)

##### **Description**

Logs an error message. Typically used when an event has happened that is unexpected / should never happen, but that the application can recover from.

##### Parameters

message - the message to be logged.

anException (optional)0 an Exception that was thrown, This is appended to the message when it is displayed.

##### **Returns**

Null

#### info(message, anException)

##### **Description**

Logs an information message. Used to indicate basic processing flow / values for high-level debugging / code tracing.

##### Parameters

message- the message to be logged.

anException (optional) - an Exception that was thrown, This is appended to the message when it is display

##### **Returns**

null

test(message, anException)

##### **Description**

Logs a test result message. Used to record unit test results.

##### Parameters

message - the message to be logged.  
anException (optional) - an Exception that was thrown, This is appended to the message when it is displayed.

##### **Returns**

Null

trace(message, anException)

##### **Description**

Logs a debug/trace message. Used for detailed debugging / code tracing. This is used for the highest, most verbose level of activity reporting.

##### Parameters

message the message to be logged.  
anException (optional) an Exception that was thrown, This is appendedo the message when it is displayed.

##### **Returns**

null

#### warn(message, anException)

##### **Description**

Logs a warning message. Typically used when an event has happened that may indicate a potential problem. A situation that is permissible but unlikely.

##### **Parameters**

message- the message to be logged.  
anException (optional)- an Exception that was thrown, This is appended to the message when it is displayed.

##### **Returns**

null

#### flushFileCache()

##### **Description**

When the Log is set to write log entries into a file, they are temporarily cached in memory. This is called to append them to a file attachment in the form and clear the cache. Note that the form must be subsequently saved for the attachment to be preserved. This should be called by a preSave event to make sure that all entries are written to the file.

##### **Returns**

Null

## Map

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value. Note that, although both the keys and values can be any desired value, using Objects as keys should be avoided since JavaScript coerces them to their String equivalents (which is alway "Object") when enforcing uniqueness. This implements most of the methods in Java's Map interface.

### Package

Core.Util.Map

### Methods

#### createInstance()

##### Description

Adobe does not allow object instances to be created outside of their packaging, so it must be done here.

##### Returns

the new Map instance

#### asSimpleMap()

##### Description

Returns a "simple" map with the values contained in this Map object. These are simple JavaScript objects created as follows: {key1: value1} Some of the Adobe custom objects expect these structures as parameters.

##### Returns

a simple map of the values contained in this Map.

#### containsKey (key)

##### Description

Returns whether this map contains a mapping for the specified key. Note that this is accurate even if the key maps to null or undefined.

##### Parameters

key-the key whose presence in this map is to be tested.

##### Returns

a boolean - true if this map contains a mapping for the specified key.

##### Throws

UnexpectedDataType Exception if key is undefined or null.

#### containsValue (value)

##### **Description**

Returns whether this map maps one or more keys to the specified value.

##### Parameters

value-the value whose presence in this map is to be tested.

##### **Returns**

(boolean) - true if this map maps one or more keys to the specified value.

#### get(key)

##### **Description**

Returns the value to which this map maps the specified key. Returnsnull if the map contains no mapping for this key. A return value of null does not necessarily indicate that the map contains no mapping for the key; it's also possible that the map explicitly maps the key to null or undefined.

##### Parameters

key - the key whose associated value is to be returned.

##### Returns

the value to which this map maps the specified key Null if the map contains no mapping for this key or if the key maps to null or undefined.

##### Throws

an UnexpectedDataType Exception if key is undefined or null.

#### isEmpty()

##### **Description**

Returns whether this map contains no key-value mappings.

##### **Returns**

(boolean) - true if this map contains no key-value mappings.

#### keyset()

##### **Description**

Returns an array of the keys contained in this map. The array is independant of this map, so changes to the map are not reflected in the array and vice-versa.

##### **Returns**

an Array containing the keys contained in this map.

#### put (key, value)

##### **Description**

Associates the specified value with the specified key in this map. If the map previously contained a mapping for this key, the old value is replaced by the specified value.

##### ****Parameters****

key- the key whose associated value is to be added / updated

\* Note that this is coerced to a String by JavaScript when enforcing uniqueness. As a result, avoid using objects as keys since their String equivalents are the same ("Object").

value- the value to be associated with the specified key.

##### **Returns**

the previous value associated with specified key, or null if there was no mapping for key. A null return can also indicate that the map previously associated undefined or null with the specified key.

##### **Throws**

an UnexpectedDataType Exception if key is undefined or null.

#### putAll (aMap)

##### **Description**

Copies all of the mappings from the specified map to this map. The effect of this call is equivalent to that of calling put(k, v) on this map once for each mapping from key k to value v in the specified map.

##### Parameters

aMap- the Map whose mappings are to be stored in this map.

##### **Returns**

Null

##### Throws

UnexpectedDataType Exception if key is undefined or null.

#### remove(key)

##### **Description**

Removes the mapping for this key from this map if it is present.

##### Parameters

key -key whose mapping is to be removed from the map.

##### **Returns**

the previous value associated with specified key, or null if here was no mapping for key. A null return can also indicate that the map previously associated undefined or null with the specified key.

##### Throws

UnexpectedDataType Exception if key is undefined or null.

#### size()

##### **Description**

Returns the number of key-value mappings in this map.

##### **Returns**

number of key-value mappings in this map.

#### values()

##### **Description**

Returns an array of the values contained in this map. The array is independant of this map, so changes to the map are not reflected in the array and vice-versa.

##### **Returns**

An Array of the values contained in this map. This will not include values of undefined or null.

#### asString()

##### **Description**

Describes this object. This is implemented in all objects since their toString() cannot be overridden!

##### **Returns**

the string description of this object.

## NodeUtil

This class provides functionality to augment Adobe's node object.Note that some of these routines are used to load the configuration, so Log cannot be used for debugging since that will create an endless loop. Write out directly to the console instead.

### Package

Core.Util.Node

### Methods

#### getChildNode(aNode, childFormNodeKey)

##### **Description**

Gets the child node of the passed node with the passed FormNode key.

##### Parameters

aNode- the node whose child is of interest.

childFormNodeKey- a String that contains the FormNode key of the child node to be returned.

##### **Returns**

the child node with the specified key, or null if there is none found for this key. This will issue a warning if the FormNode for the child is not found. If there is nore than one child with this key, the first will be returned.

##### Throws

An UnexpectedDataType Exception if either of the parameters is undefined, null, or the wrong data type.

#### getNodeListAsString(aNodeList)

##### **Description**

Convert the passed NodeList into String format.  
  
Parameters

aNodeList -the Node List to be converted into String.

##### **Returns**

a String that contains a comma delimited list of node names. E.g. "nodeList[node1, node2, node3]".

#### getPeerNode(aNode, peerFormNodeKey)

##### **Description**

Gets the peer node to the passed node with the passed FormNode key.  
  
Parameters

aNode- the node whose peer is of interest.  
peerFormNodeKey- a String that contains the FormNode key of the peer node to be returned.

##### **Returns**

Null  
  
Throws

UnexpectedDataType Exception if either of the parameters is undefined, null, or the wrong data type.

getRepeatingAncestorNode(aNode, startAtParent)

##### **Description**

gets the first ancestor that is repeating (i.e. has 2 or more potential instances).

##### Parameters

aNode- The node to start the upward traversal

startAtParent(optional) - a boolean that indicates whether the search

##### **Returns**

The Adobe Node if successful, or null if the traversal reaches the form's root node.

#### getScripts(scriptPackage)

##### **Description**

Gets the JavaScript script nodes that are in the passed package (container node). This only returns immediate children.  
  
Parameters

scriptPackage- the container node that is the JavaScript package (subfolder) to be checked.

##### ****Returns****

array of script nodes in the same order as they were in the package. An empty array if there weren't any scripts nodes in the package.  
  
Throws

an UnexpectedDataType Exception if scriptPackage is not a container node.

#### getScriptPackages(packageName)

##### **Description**

Gets all nodes with the indicated name that represent JavaScript packages.  
  
Parameters

packageName a string that contains the name of the script packages of interest.

##### **Returns**

an array of nodes in the same order as they were found. An empty array if there weren't any nodes with the indicated name.

#### getTemplate(aNode)

##### **Description**

Gets the "peer" of the passed node in the template structure. The returned node contains the settings that it had when it was originally defined in the IDE.  
  
Parameters  
aNode -an Adobe node (field) on the form.

##### **Returns**

the equivalent Adobe node in the template structure. Null if the passed node had no equivalent in the template (i.e. must have been created programmatically).  
  
Throws

UnexpectedDataType Exception if the passed node is undefined, null, or isn't an Adobe node.

#### hasProperty(aNode, propertyName)

##### **Description**

Gets whether the passed node has the passed property. As of Adobe 9, an exception is thrown if code refers to a node property that doesn't exist. This is non-standard behaviour - other objects simply return undefined. Nodes now have an isPropertySpecified() method to test for the property's existance, however that method always returns false in the current Adobe implementation.  
  
Parameters

aNode- an Adobe node (field) on the form.

propertyName- a String that contains the name of the property in question.

##### **Returns**

(boolean) - true if the passed node has the passed property.

##### Throws

UnexpectedDataType Exception if either of the parameters is undefined, null, or isn't the indicated data type.

#### sortNodeArray(aNodeArray)

##### **Description**

Sorts the passed array of Nodes "in place" using a case insensitive sort. Node array is not being type checked because   
  
Parameters

aNodeArray the Node List to be sorted.  
  
Returns

Null

#### updateNode(aNode, value)

**//TODO Deprecated**

##### **Description**

Sets the value of the passed node to the passed value. Note that this does not alter the node if its current value is the same as the one passed. This is done to avoid unnecessary UI updates.  
  
Parameters

aNode- an Adobe node (field) on the form.

value -the new value for the field. This must be a String, boolean, number, or null.

##### **Returns**

(boolean) - true if the node was updated.  
  
Throws

UnexpectedDataType Exception if the aNode is undefined, null, or isn't an Adobe node or if the value isn't one of the required types.

## NodeValidator

A class used to validate Adobe Nodes in this XFA form.

### Methods

#### validateField(aField, formatType)

##### Description

Validates the value of the passed Adobe Field. This includes a check of mandatory fields as well as performing a format validation if one has been defined for the field.

##### Parameters

aField- the Adobe Field to validate.

formatType (optional)- a String that contains the name (key) of a format validation to apply to the field.

##### Returns

a String containing the internationalized error message for any detected problem with the value of the passed field. Null if there were no problems.

##### Throws

An UnexpectedDataType Exception if aField is undefined, null, or isn't an Adobe Field, or if formatType isn't a String.

#### validateGroup(aGroup)

##### **Description**

Validates the value of the passed Adobe ExclusionGroup (group of Radio Buttons). This includes a check of its value if it is defined as mandatory.

##### Parameters

aGroup- the Adobe ExclusionGroup to validate.

##### Returns

a String containing the internationalized error message for any detected problem with the value of the passed field. Null if there were no problems.

##### Throws

An UnexpectedDataType Exception if aGroup is undefined, null, or isn't an Adobe ExclusionGroup.

## NumberFormatter

A class that formats numbers.

### Package

Core.Util.NumberFormatter

### Methods

#### formatNumber(aNumber, decimalPlaces)

##### Description

Formats a number with commas as thousands separators.

Parameters  
aNumber - the number to be formatted.

decimalPlaces- (optional) the number of decimal places to round to. If

##### Returns

string containing the formatted number. "0" is substituted if aNumber was undefined or null.

Throws  
UnexpectedDataType Exception if aNumber or decimalPlaces (when used) don't contain valid numbers.

#### function formatAsCurrency(aNumber)

##### Description

Formats a number as "$1,234.56".

##### Parameters

aNumber - the number to be formatted

##### Returns

a string containing the formatted number. "$0.00" is substituted if aNumber was undefined or null.  
  
Throws

UnexpectedDataType Exception if aNumber doesn't contain a valid number.

#### function pad(aNumber, width)

##### ****Description****

Pads a number with leading 0's. This is only intended to be used with positive numbers (negative signs will be within the padding).

Parameters

aNumber- the number to be padded. If null, this will be treated as 0.

width- the number of characters to pad to.

##### ****Returns****

a string containing the 0 padded number.  
  
Throws

InvalidArgument Exception- if aNumber has more characters in it than width allows.

UnexpectedDataType Exception -if aNumber or width don't contain valid numbers.

round(aNumber, decimalPlaces)

##### **Description**

Rounds a number to a specific number of decimal places. It also pads it with trailing 0's.  
  
Parameters

aNumber -the number to be rounded.  
decimalPlaces- the number of decimal places to round to. If 0, the decimal is suppressed. If a negative number, it is rounded to the left of the decimal by that amount (e.g. 1234 with -1 results in 1230).

##### **Returns**

the rounded number as a String. 0 is substituted if aNumber was undefined or null.  
  
Throws

UnexpectedDataType Exception if aNumber or decimalPlaces don't contain valid numbers.

## NumberUtil

A class that contains utility functions for manipulating numbers.

### Package

Core.Util.NumberUtil

### Methods

#### asBoolean(aNumber)

##### Description

Coerces the passed number to a boolean. This will only accept 1 as true and 0 as false.  
  
Parameters

aNumber the number to be converted.

##### Returns

a boolean equivalent to the number.

##### Throws

UnexpectedDataType Exception if aNumber is undefined, null, not a number, or isn't one of the expected true or false values.

#### round(aNumber, decimalPlaces)

##### Description

Rounds a number to a specific number of decimal places without padding it with trailing 0's (this actually returns a number result rather than a String).  
  
Parameters

aNumber - the number to be rounded.

decimalPlaces -the number of decimal places to round to. If a negative number, it is rounded to the left of the decimal by that amount (e.g.1234 with -1 results in 1230).  
  
Returns

the rounded number as a number. 0 is substituted if aNumber was undefined or null.  
  
Throws

UnexpectedDataType Exception if aNumber or decimalPlaces don't contain valid numbers.

## ObjectUtil

A class that contains utility functions for dealing with Objects.

### Package

Core.Util.ObjectUtil

### Methods

#### asString(anObject, level, indentProperties)

##### Description

Lists the details of the passed Object. Note that, in the case of complex Objects, functions are not listed. If this is one of our custom objects, this will return its asString().  
  
Parameters

anObject - the Object of interest. This is a "loose" sense of Object it can actually be anything.

level (optional)- an integer that represents the number of levels of indentation this should use. Defaults to 0. Used for listing nested objects.

indentProperties (optional) - a boolean that indicates whether only the properties should be indented. Defaults to false (indent the entire entry). Used for listing nested objects.  
  
Returns

the string that lists the details of the passed Object.

#### getDescription(anObject)

##### **Description**

Gets a simple description of the passed Object. This will be the type of the Object, if that can be determined. Otherwise, it will be the object itself (which will result in its toString() being used when printed).  
Parameters

anObject - the Object of interest

##### **Returns**

a string that contains a simple description of the passed Object, or the object itself.

#### getFunctions(anObject)

Description: Gets an Array of the passed Object's functions' names. Note that this filters out Adobe functions (all classes are wrapped by Adobe script objects that add their own functions).  
  
Parameters

anObject - the Object of interest.

##### **Returns**

a String Array that contains the names of the passed Object's functions. Empty if this was not an Object.

#### getProperties(anObject)

##### **Description**

Gets a Map of the passed Object's properties (variables), where the key is the property name and the value is the property's value.  
  
Parameters

anObject - the Object of interest.  
  
Returns

a Map of the passed Object's properties not including its functions. Empty if this was not an Object.

#### getType(anObject)

##### **Description**

Gets the type (class) of the passed Object.  
  
Parameters

anObject - the Object of interest.  
  
Returns

a string that contains the passed Object's class. Null if it is not an object, or if its type cannot be determined.

#### isComplexObject(anObject)

##### **Description**

Gets whether the passed Object is "complex", which is to say that it is an object that contains properties that won't display properly using its toString().  
  
Parameters

anObject - the Object of interest.

##### **Returns**

a boolean that indicates whether the passed Object is "complex". False if it is not an Object.

#### isXfaObject(anObject)

##### **Description**

Gets whether the passed Object is an XFA object, which is provided by Adobe. Not only will it not display properly using its toString(), its properties cannot be dynamically dicovered.  
  
Parameters

anObject - the Object of interest.

##### **Returns**

a boolean that indicates whether the passed Object is an XFA object. False if it is not an Object.

## ResourceManager

### Description

A class that manages the form's resource files which contains labels and messages in the language specified. It loads all JavaScript classes in "resources" packages (i.e. treats them as resrouce files). This creates a master resource file from all of these entries, so individual entries can be over-ridden by those in files that are subsequently loaded. Loading follows the ordering they are in the forms - proceding down the first branches to the first leaves before proceding to the next leaves and then the next branches. Resource entry values are usually Strings.

### Package

core.lang.ResourceManager

### Constants

LANG\_ENGLISH- indicates that the form language is English

LANG\_FRENCH- indicates that the form language is French

### Methods

#### containsKey(key)

##### **Description**

Gets whether there is an entry for the specified key.

##### Parameters

key- the key whose presence in this configuration is to be tested.

##### **Returns**

a boolean - true if this configuration contains an entry for the specified key.

##### Throws

an UnexpectedDataType Exception if key is undefined or null.

#### get(key)

##### **Description**

Gets the value for the passed key. Returns string $key$ if the resource file doesn't contain an entry for this key.

##### **Parameters**

key- the key whose associated value is to be returned.

additionalValues (optional)- these additional parameters are used to support the dynamic replacement of placeholders in a retrieved String message. Placeholders follow the syntax "{#}", where # is the 1-based index of the additional arguments. This is only appropriate if the return value is a String. Any number of additional arguments can be passed. Eg. get("test", "this", "out") will retrieve the message with the key "test", replace all "{1}"s with "this", and replace all "{2}"s with "out".

##### **Returns**

the value for the specified key, or "$key$" if the resource files contain no entry for this key. Values are usually, but not limited to, Strings.

##### **Throws**

an UnexpectedDataType Exception if key is undefined or null.

#### getCurrentLanguage()

##### **Description**

Gets the current language setting. If null, it is automatically set to the default language in the config file and then loads the resources.

##### **Returns**

the current language setting.

#### getPickList(key)

##### D**escription**

Gets the pick list for the passed key. Returns null if the resource files don't contain an entry for this key. This is a String of the format "displayValue1, code1, displayValue2, code2, displayValue3, key3...".

##### **Parameters**

Key- the key whose associated pick list is to be returned.

##### **Returns**

pick list for the specified key, or null if the resource files contain no entry for this key.

##### ****Throws****

an UnexpectedDataType Exception if key is undefined or null, or if the pick list is not properly defined in the resource entry.

#### setCurrentLanguage(lang)

##### **Description**

Sets the language which will be in use for the UI presentation

##### **Parameters**

lang- the language to be set, required to be one of the class constants

##### **Returns**

(boolean) true if the language actually changed from its previous value.

##### **Throws**

an UnexpectedLanguage Exception if lang not one of the constants.

## Resources

see also ResourceManager

### Description

Resource classes contain all the key and value entries required for localization of object labels, tooltips, and picklists.

Resources are recognized by creating a 'resources' subform. Any script objects under this subform will be recognized as resource files. There can be multiple script objects and multiple script objects. English and French values are defined in separate script objects and identified using a "\_en" or "\_fr" suffix for the script object name

Resource files must be structured as:

var content={

key1:"value1",

key2:"value2",

key3:"value3" };

For picklists, they must be structured as:

var content={

picklist\_Key:[

["saved value1",display value1"]

,["saved value2",display value2"]

,["saved value3",display value3"]]

}

The 'core' framework has some built in translations provided labels. These include country lists and some basic error messaging.

## SecurityUtil

Utility methods to support security.

Includes the following:  
A JavaScript implementation of the RSA Data Security, Inc. MD5 Message Digest Algorithm, as defined in RFC 1321. Version 2.1 Copyright (C) Paul Johnston 1999 - 2002. Other contributors: Greg Holt, Andrew Kepert, Ydnar, Lostinet Distributed under the BSD License

See <http://pajhome.org.uk/crypt/md5> for more info.

### Package

Core.Util.SecurityUtil

### Methods

#### function getBase64MD5Hash(aString)

##### **Description**

Gets a base 64 representation of an MD5 hash of the passed String.  
  
Parameters

aString the String to be hashed.

##### **Returns**

a string containing the resulting hexadecimal MD5 hash.

#### getMD5Hash(aString)

##### **Description**

Gets an MD5 hash of the passed String.  
  
Parameters

aString - the String to be hashed.

##### **Returns**

a string containing the resulting MD5 hash

## Set

A collection that contains no duplicate elements. Items in this structure are indexed the same way as in an Array, so "for" loops will work. Only pop() and push() (each taking a single argument) are implemented. Note that custom objects stored in this set that implement the EqualityComparableInterface use their getEqualityKey() function to determine uniqueness. Those that don't implement that interface use their asString() function. Simple objects stored in this set use their toString() function to determine uniqueness.

### Package

Core.Util.SecurityUtil

### Methods

#### createInstance()

##### **Description**

Gets a new Set. Adobe does not allow object instances to be created outside of their packaging, so it must be done here.  
  
Returns

the new Set instance

#### asArray()

##### **Description**

Returns an Array with the values contained in this Set object.  
  
Returns

an Array that contains this Set's entries.

#### contains(anObject)

##### Description

gets whether the passed object is already in the set.

Parameters  
anObject - the object of interest.

##### **Returns**

true if the passed object is in the set.

#### pop()

##### **Description**

Removes the last element of the Set (this will also change the length of the Set.  
  
Returns

the removed element. Undefined if the Set was already empty

#### push (anObject)

##### **Description**

Adds a new element to the end of the Set and returns the new length. The object is not added if there is already one in the Set.  
Parameters

anObject - the Object to add to the end of the Set

##### **Returns**

a number that contains the new length of the set.

#### pushAll(anArray)

##### **Description**

Copies all of the entries from the specified Array to this Set. The effect of this call is equivalent to that of calling push() on this Set once for each entry in the specified Array. As a result, order is preserved but duplicates are removed.  
  
Parameters

anArray - the Array whose entries are to be stored in this Set.

##### **Returns**

Null  
  
Throws

UnexpectedDataType Exception if anArray is undefined, null, or not an Array.

#### asString()

##### **Description**

Describes this object. This is implemented in all objects since their toString() cannot be overridden!  
  
Returns

the string description of this object.

## StringUtil

### Package

core.util.StringUtil

### Methods

asBoolean(aString)

Description: coerces the passed string to a boolean. This will trim the string and accept the following values (case insensitive) as true: "t", "true", "yes", "on", \* "1". Their opposites will be false.

**Parameters:**  
aString - the string to be converted.

**Returns:** a boolean equivalent to the string.  
  
**Throws:**  
UnexpectedDataType Exception if aString is undefined, null, not a string, or isn't one of the expected true or false values.

asRichText(aString)

**Description:** Converts the passed string to rich text by wrapping it in the required tags for Adobe to recognize it as such. Note that this does not attempt to validate or adjust the passed string to assure that it is valid rich text.  
  
**Parameters:**

aString the string to be converted.  
  
**Returns**: a String that will be recognized by Adobe as rich text.

**Throws:**

UnexpectedDataType Exception if aString is undefined, null, or not a string.

cleanPickListEntry(aString)

**Description:** Pick list strings use commas to delimeter entries as well as the codes from their descriptions. As a result, Adobe will get confused if either contain embedded commas. This escapes any commas in the passed String with a backslash.  
  
**Parameters:**

aString- the string used as a pick list code or description.  
  
**Returns:** a string with any embedded commas properly escaped.  
  
**Throws:**

An UnexpectedDataType Exception if aString is undefined, null, or not a string.

encodeXml(s)

**Description:** Encodes characters that are incompatible with XML to XML entities

**Parameters:**  
s- a string to encode

**Returns:** an encoded string

formatSomExpression(aSomExpression)

**Description:** Formats the passed SOM expression for display. This removes the "xfa[0].form[0]" prefix, as well as all instances of "[0]" and "#variables[0]" if they exist.  
  
**Parameters:**

aSomExpression - the string containing a SOM expression to be formatted.

**Returns:** a string containing the formatted SOM expression.  
  
**Throws:**

UnexpectedDataType Exception if aSomExpression is undefined, null, or not a string.

generalizeSomExpression(aSomExpression)

**Description:** Generalizes the passed SOM expression by removing instance information. This removes all instances of "[#]" where "#" is any number.  
  
**Parameters:**

aSomExpression - the string containing a SOM expression to be generalized.

**Returns:** a string containing the generalized SOM expression.  
  
**Throws:**

UnexpectedDataType Exception if aSomExpression is undefined, null, or not a string.

=

spliceSomExpressions(aBaseSomExpression, aChildSomExpression)

**Description:** Splices a child SOM expression onto a base SOM expression, which is to say that the lowest nodes of the child are replaced with those of the base. This is commonly used to prepend a SOM expression with instance information to one that has been generalized so that the result can be resolved to the child of that instance. For example, a base of "xfa[0].form[0].Pla[0].sfPart1[2]" and a child of "xfa.form.Pla.sfPart1.sfSectionA1.sfRow1.sfHeading" would produce "xfa[0].form[0].Pla[0].sfPart1[2].sfSectionA1.sfRow1.sfHeading", which resolves to the sfHeading under the 3rd sfPart1.  
  
**Parameters:**

aBaseSomExpression the string containing the base SOM expression that includes instance information.

aChildSomExpression the string containing a (usually generalized) SOM expression to a child node under the base.

**Returns:** a string containing the SOM expression that results from merging the two.

**Throws:**

UnexpectedDataType Exception if aBaseSomExpression or aChildSomExpression is undefined, null, or not a string.

InvalidArgument Exception if aChildSomExpression doesn't have more node levels than aBaseSomExpression.

toLowerCase(aString)

**Description:** Converts the passed string to lower case.  
  
**Parameters:**

aString - the string to be converted to lower case.  
  
**Returns:** the string converted to lower case. If passed an undefined or null, an empty String will be returned.

**Throws:**

UnexpectedDataType Exception if aString is not a string.

toUpperCase(aString)

**Description:** Converts the passed string to upper case.  
  
**Parameters:**

aString - the string to be converted to upper case.  
  
**Returns**: the string converted to upper case. If passed an undefined or null, an empty String will be returned.  
  
**Throws:**

UnexpectedDataType Exception if aString is not a string.

trim(aString)

**Description:** Strips whitespace from either end of the passed string.  
  
**Parameters:**

aString - the string to be trimmed  
  
**Returns:** the string with leading and trailing whitespace removed. An empty string if nothing remains. If passed an undefined or null, the same value will be returned.  
  
**Throws:** an UnexpectedDataType Exception if aString is not a string.

XFA Forms Framework: <DRAFT> TestSuiteInterface

This defines an Interface that is implemented by JsUnit test suites, which are collections of JsUnit tests.

Methods

getNumberOfFailures()

**Description:**Gets the number of tests that failed on the last run.

**Returns**: number the number of test failures that occurred.

getNumberOfSuccesses()

**Description:** Gets the number of tests that succeeded on the last run.

**Returns:** (number) the number of tests that were successful.

runTests()

**Description:** Runs all the tests in the suite. Logs all failures.

**Returns:** (boolean). True if all of the tests passed.

XFA Forms Framework: <DRAFT> TypeChecker

Package

core.util.TypeChecker

Methods

implementsInterface(anObject, anInterface)

**Description:** Checks that the passed value is a class or object that implements a specific interface. To pass this test, the class or object being checked must implement of all of the functions present in the interface. Unfortunately, the function parameters and return type are not checked in this implementation.  
  
**Parameters:**

anObject- the class or object to be checked.

anInterface - a custom class that defines the interface.   
  
**Returns:** rue if the class or object implements the passed interface, otherwise false. Undefined and null return false.  
  
**Throws:**

UnexpectedDataType Exception if anInterface is undefined, null, or isn't an interface definition.

-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

isArray(value)

**Description:** Checks that the passed value is an Array object.  
  
**Parameters:**

value- the value to be checked.  
  
**Returns:** true if it is an Array, otherwise false. Undefined and null return false.

isBoolean(value)

**Description:** Checks that the passed value is a boolean or a Boolean object.  
  
**Parameters:**

value - the value to be checked.  
  
**Returns**: true if it is a boolean, otherwise false. Undefined and null return false.

isCheckBox(value)

**Description:** Checks that the passed value is a check box object.  
  
**Parameters:**

value - the value to be checked.

**Returns:** true if it is a check box, otherwise false. Undefined and null return false.

isCollection(value)

**Description:** Checks that the passed value is a collection (an Array or Set object). These contains items that are indexed and can be processed with a "for" loop.  
  
**Parameters:**

value - the value to be checked  
  
**Returns:** true if it is a collection, otherwise false. Undefined and null return false

isComboBox(value)

**Description:** hecks that the passed value is a comboBox field (a pickList that allows the user to enter custom text).  
  
**Parameters:**

value - the value to be checked.  
  
**Returns:** true if it is a comboBox, otherwise false. Undefined and null returns false.

isCustomObject(value)

**Description:** Checks that the passed value is a custom object instance. To pass this test, the object must populate a className variable and implement an asString() function. This is only applicable to objects defined in this code base.  
  
**Parameters:**

value - the value to be checked.  
  
**Returns**: true if it is a custom object instance, otherwise false. Undefined and null return false.

isCustomObjectFactory(value)

**Description:** checks that the passed value is a custom class that can be used to create object instances. To pass this test, the class must implement a createInstance() function to retrieve object instances. This is only applicable to objects defined in this code base.  
  
**Parameters:**

value - the value to be checked.

**Returns**: rue if it is a custom class that can be used to create object instances, otherwise false. Undefined and null return false.

isDate(value)

**Description**: Checks that the passed value is a Date.  
  
**Parameters:**

value - the value to be checked.  
  
**Returns:** true if it is a Date, otherwise false. Undefined and null return false.

isFunction(value)

**Description:** checks that the passed value is a function object.  
  
**Parameters:**

value - the value to be checked.  
  
**Returns:** true if it is a function, otherwise false. Undefined and null return false.

isInterface(value)

**Description:** checks that the passed value is a custom Interface definition. To pass this test, the object must populate an interfaceName variable. This is only applicable to interfaces defined in this code base.  
  
**Parameters:**

value - the value to be checked.  
  
**Returns:** true if it is a custom Interface definition, otherwise false. Undefined and null return false.

isMap(value)

**Description:** checks that the passed value is a Map (dictionary).  
  
**Parameters:**

value- the value to be checked.  
  
**Returns:** true if it is a Map, otherwise false. Undefined and null return false.

isNode(value)

**Description:** Checks that the passed value is an instance of an Adobe Node. This isn't failsafe, since it does so by testing for the presence of a somExpression property (which other objects may also implement). This is necessary since there are multiple Node subclasses.  
  
**Returns:** true if it is an Adobe Node, otherwise false. Undefined and null return false.

isNumber(value)

**Description:** Checks that the passed value is, or can be successfully converted to, a number.  
  
**Parameters:**

value- the value to be checked.  
  
**Returns**: true if it is a number, otherwise false. Undefined and null return false.

isPickList(value)

**Description:** Checks that the passed value is a pickList field.  
  
**Parameters:**

value- the value to be checked.  
  
**Returns:** true if it is a pickList, otherwise false. Undefined and null return false.

isRadioButton(value)

**Description:** Checks that the passed value is a radio button field.  
  
**Parameters:**

value - the value to be checked.

**Returns:** true if it is a radio button, otherwise false. Undefined and null return false.

isSet(value)

**Description:** checks that the passed value is a Set (type of Array).  
  
**Parameters:**

value - the value to be checked.  
  
**Returns:** true if it is a Set, otherwise false. Undefined and null return false.

isSimpleMap(value)

**Description**: Checks that the passed value is a simple map / dictionary. These are simple JavaScript objects created as follows: {key1: value1, key2: value2} This is a very basic test, since it can only tell that it is passed an Object. Note that it is preferable to use a true Map object (see isMap()).  
  
**Parameters:**

value - the value to be checked.  
  
**Returns**: true if it is a simple map, otherwise false. Undefined and null return false.

isSoapResponseNode(value)

**Description:** checks that the passed value is a node in a Web Services (SOAP) response. These objects contain a soapName, soapQName, and soapValue. This does not rely on soapValue being populated.  
  
**Parameters**:

value- the value to be checked.  
  
**Returns**: true if it is a SOAP response node, otherwise false. Undefined and null return false.

isSpecificCustomObject(value, className)

**Description:** Checks that the passed value is an instance of a specific custom Object. This supports objects that an "instanceof" test won't work for but, as a result, have included a "className" variable to use for this purpose. This includes Adobe objects as well as those defined in this code base.  
  
**Parameters:**

value - the value to be checked.

className - a string containing the name of the class. This value must be set in a "className" variable in the object being tested for the test to pass.  
  
**Returns**: true if it is an object of the indicated type, otherwise false. Undefined and null return false.  
  
**Throws:**

UnexpectedDataType Exception if className is undefined, null, or isn't a String.

isString(value)

**Description:** Checks that the passed value is a string or a String object.  
  
**Parameters:**

value - the value to be checked.  
  
**Returns:** true if it is a string, otherwise false. Undefined and null return false.

isStringArray(value, allowUndefinedContent)

**Description:** Checks that the passed value is an array that only contains strings and/or String objects.  
  
**Parameters:**

value - the value to be checked.

allowUndefinedContent (optional) - true if this should allow the array to contain undefined or null values. Defaults to false.

**Returns:** true if it is an array that is empty or only contains strings, otherwise false. Undefined and null return false.

**Throws:**

UnexpectedDataType Exception if allowUndefined isn't a boolean, undefined, or null.

XFA Forms Framework: <DRAFT> TypeEnforcer

This class provides functionality to enforce type checking. Its methods throw exceptions when type checks fail and, in some cases, support type coercion.

Package

core.util.TypeEnforcer

Constants

EXCLUSION\_GROUP\_CLASS

FIELD\_CLASS

INSTANCE\_MANAGER\_CLASS

SOAP\_RESPONSE\_CLASS

SUBFORM\_CLASS

Methods

checkArray(value, callingClassName, callingMethodName, parameterName, allowUndefined)

**Description:** Checks that the passed value is an array object.

**Parameters:**

value- the value to be checked.

callingClassName - the name of the calling class for exception reporting.

callingMethodName- the name of the calling method for exception reporting.

parameterName - the name of the calling method's parameter for exception reporting.

allowUndefined (optional) true if this should allow an undefined or null value without throwing an exception. Defaults to false.

**Returns:** the passed array. When allowUndefined is true, this returns an empty array when value is undefined or null.  
  
**Throws:** an UnexpectedDataType Exception if value is undefined, null, or isn't an array.

checkBoolean(value, callingClassName, callingMethodName, parameterName, allowUndefined, allowCoercion)

**Description:** hecks that the passed value is a boolean or a Boolean object, or (optionally) can be safely be coerced to one.  
  
**Parameters:**

value - the value to be checked.

callingClassName - the name of the calling class for exception reporting.

callingMethodName - the name of the calling method for exception reporting.

parameterName - the name of the calling method's parameter for exception reporting.

allowUndefined (optional) - true if this should allow an undefined or null value without throwing an exception. Defaults to false.

allowCoercion (optional) - true if this should allow another data type be coerced to a boolean. Defaults to false.

**Returns:** the passed boolean. When allowUndefined is true, this returns false when value is undefined or null. When allowCoercion is true, this will attempt to coerce numbers and strings to a boolean using NumberUtil and StringUtil.

**Throws:** UnexpectedDataType Exception if value is undefined, null, or isn't a valid boolean.

checkCheckBox(value, callingClassName, callingMethodName, parameterName)

**Description:** Checks that the passed value is a check box object, To pass this test, the object must populate a className variable and implement an asString() function. This is only applicable to objects defined in this code base.

**Parameters:**

value - the value to be checked.

callingClassName - the name of the calling class for exception reporting.

callingMethodName - the name of the calling method for exceptionreporting.

parameterName - the name of the calling method's parameter for exception reporting.

**Returns:** the passed boolean. When allowUndefined is true, this returns false when value is undefined or null. When allowCoercion is true, this will attempt to coerce numbers and strings to a boolean using NumberUtil and StringUtil.  
  
**Throws:** an UnexpectedDataType Exception if value is undefined, null, or isn't a valid boolean.  
.

**checkCollection(value, callingClassName, callingMethodName, parameterName, allowUndefined)**

**Description:** Checks that the passed value is a collection (an Array or Set).  
  
**Parameters:**

value- the value to be checked.

callingClassName - the name of the calling class for exception reporting.

callingMethodName - the name of the calling method for exceptionreporting.

parameterName - the name of the calling method's parameter for exception reporting.

allowUndefined (optional)- true if this should allow an undefined or null value without throwing an exception. Defaults to false.

**Returns:** the passed collection. When allowUndefined is true, this returns an empty Array when value is undefined or null.  
  
**Throws:** an UnexpectedDataType Exception if value is undefined, null, or isn't an Array or Set.

checkCustomObject(value, callingClassName, callingMethodName, parameterName)

**Description:** Checks that the passed value is a custom object instance. To pass this test, the object must populate a className variable and implementan asString() function. This is only applicable to objects defined in this code base.  
  
**Parameters:**

value the value to be checked.

callingClassName the name of the calling class for exception reporting.

callingMethodName the name of the calling method for exception reporting.

parameterName the name of the calling method's parameter for exception reporting.

**Returns**: the passed custom object instance.  
  
**Throws**: UnexpectedDataType Exception if value is undefined, null, or isn't a custom object instance.

checkCustomObject(value, callingClassName, callingMethodName, parameterName)

**Description:** Checks that the passed value is a custom object instance. To pass this test, the object must populate a className variable and implement an asString() function. This is only applicable to objects defined in this code base.

**Parameters**:

value - the value to be checked.

callingClassName - the name of the calling class for exception reporting.

callingMethodName - the name of the calling method for exception reporting.

parameterName - the name of the calling method's parameter for exception reporting.

**Returns:** the passed custom object instance.  
  
**Throws:** an UnexpectedDataType Exception if value is undefined, null, or isn't a custom object instance.

checkCustomObjectFactory(value, callingClassName, callingMethodName, parameterName)

**Description:** Checks that the passed value is a custom class that can be used to create object instances. To pass this test, the class must implement a createInstance() function to retrieve object instances. This is only applicable to objects defined in this code base.  
  
**Parameters:**

value - the value to be checked.

callingClassName - the name of the calling class for exception reporting.

callingMethodName - the name of the calling method for exception reporting.

parameterName - the name of the calling method's parameter for exception reporting.  
  
**Returns:** the passed Field.  
  
**Throws:** an UnexpectedDataType Exception if value is undefined, null, or isn't a custom class that can be used to create object instances.  
  
**checkDate(value, callingClassName, callingMethodName, parameterName, allowUndefined)**

**Description:** Checks that the passed value is a Date.  
  
**Parameters:**

value the value to be checked.

callingClassName the name of the calling class for exception reporting.

callingMethodName the name of the calling method for exception reporting.

parameterName the name of the calling method's parameter for exception reporting.

allowUndefined (optional) true if this should allow an undefined or null value without throwing an exception. Defaults to false.  
  
**Returns**: he passed map. When allowUndefined is true, this returns a new Date when value is undefined or null.  
  
**Throws:** an UnexpectedDataType Exception if value is undefined, null, or isn't a map.

checkExclusionGroup(value, callingClassName, callingMethodName, parameterName)

**Description:** Checks that the passed value is an Adobe Exclusion Group (group of radio buttons).  
  
**Parameters:**

value- the value to be checked.

callingClassName - the name of the calling class for exception reporting.

callingMethodName - the name of the calling method for exception reporting.

parameterName - the name of the calling method's parameter for exception reporting.  
  
**Returns:** the passed Field.  
  
**Throws:** throws an UnexpectedDataType Exception if value is undefined, null, or isn't an Adobe Exclusion Group.

checkField(value, callingClassName, callingMethodName, parameterName)

**Description:** Checks that the passed value is an Adobe Field.  
  
**Parameters:**

value-the value to be checked.

callingClassName - the name of the calling class for exception reporting.

callingMethodName- the name of the calling method for exception reporting.

parameterName - the name of the calling method's parameter for exception reporting.  
  
**Returns**:- the passed Field.  
  
**Throws:** an UnexpectedDataType Exception if value is undefined, null, or isn't an Adobe Field.

checkInstanceManager(value, callingClassName, callingMethodName, parameterName)

**Description:** Checks that the passed value is an Adobe InstanceManager.  
  
**Parameters:**

value - the value to be checked.

callingClassName - the name of the calling class for exception reporting.

callingMethodName - the name of the calling method for exception reporting.

parameterName - the name of the calling method's parameter for exception reporting.  
  
**Returns:** the passed InstanceManager.  
  
**Throws:** an UnexpectedDataType Exception if value is undefined, null, or isn't an Adobe InstanceManager.

checkInterface(value, callingClassName, callingMethodName, parameterName, anInterface)

**Description:** Checks that the passed value is a class or object that implements a specific interface.  
  
**Parameters:**

value- the class or object to be checked.

callingClassName - the name of the calling class for exception reporting.

callingMethodName - the name of the calling method for exception reporting.

parameterName- the name of the calling method's parameter for exception reporting.

anInterface - a custom class that defines the interface. The class or object being checked must implement of all of the functions present in the interface to pass. Unfortunately, the parameters and return types are not checked in this implementation.

**Returns:** the passed class or object.  
  
**Throws**: an UnexpectedDataType Exception if value is undefined, null, or does not implement all of the required functions.

checkMap(value, callingClassName, callingMethodName, parameterName, allowUndefined)

**Description:** Checks that the passed value is a Map (dictionary).  
  
**Parameters:**

value - the value to be checked.

callingClassName - the name of the calling class for exception reporting.

callingMethodName - the name of the calling method for exception reporting.

parameterName- the name of the calling method's parameter for exception reporting.

allowUndefined (optional) - true if this should allow an undefined or null value without throwing an exception. Defaults to false.  
  
**Returns:** the passed Map. When allowUndefined is true, this returns an empty Map when value is undefined or null.  
  
**Throws:** an UnexpectedDataType Exception if value is undefined, null, or isn't a Map.

checkNode(value, callingClassName, callingMethodName, parameterName, allowUndefined)

**Description:** Checks that the passed value is an Adobe Node. This isn't failsafe, since it does so by testing for the presence of a somExpression property (which other objects may also implement). This is necessary since there are multiple Node subclasses.  
  
**Parameters:**

value the value to be checked.

callingClassName - the name of the calling class for exception reporting.

callingMethodName - the name of the calling method for exception reporting.

parameterName - the name of the calling method's parameter for exception reporting.

allowUndefined (optional)- true if this should allow an undefined or null value without throwing an exception. Defaults to false.  
  
**Returns:** the passed value.  
  
**Throws:** an UnexpectedDataType Exception if value is undefined, null, or isn't a Node.

checkNodeArray(value, callingClassName, callingMethodName, parameterName, allowUndefined)

**Description:** Checks that the passed value is an array of strings and/or String objects.  
  
**Parameters:**

value - the value to be checked.

callingClassName - the name of the calling class for exception reporting.

callingMethodName - the name of the calling method for exception reporting.

parameterName - the name of the calling method's parameter for exception reporting.

allowUndefined (optional) - true if this should allow an undefined or null value without throwing an exception. Defaults to false.  
  
**Returns:** he passed value, even if its an empty array. When allowUndefined is true, this returns an empty array when value is undefined or null.  
  
**Throws:** UnexpectedDataType Exception if value is undefined, null, or isn't a valid string.

=

checkNumber(value, callingClassName, callingMethodName, parameterName, allowUndefined)

**Description:** Checks that the passed value is, or can be successfully converted to, a number

**Parameters:**

value - the value to be checked.

callingClassName - the name of the calling class for exception reporting.

callingMethodName - the name of the calling method for exception reporting.

parameterName- the name of the calling method's parameter for exception reporting.

allowUndefined (optional) - true if this should allow an undefined or null value without throwing an exception. Defaults to false.

**Returns:** the passed value as a number primitive. When allowUndefined is true, this returns 0 when value is undefined or null. Note that this only uses basic JavaScript coersion to create the number.  
  
**Throws:** an UnexpectedDataType Exception if value is undefined, null, or doesn't contain a valid number.

checkRadioButton(value, callingClassName, callingMethodName, parameterName)

**Description:** Checks that the passed value is a radio button field.  
  
**Parameters:**

value - the value to be checked.

callingClassName - the name of the calling class for exception reporting.

callingMethodName - the name of the calling method for exception reporting.

parameterName - the name of the calling method's parameter for exception reporting.  
  
**Returns:** the passed radio button field.  
  
**Throws:** an UnexpectedDataType Exception if value is undefined, null, or isn't a valid boolean.

checkSet(value, callingClassName, callingMethodName, parameterName, allowUndefined)

**Description:** Checks that the passed value is a Set.  
  
**Parameters:**

value the value to be checked.

callingClassName the name of the calling class for exception reporting.

callingMethodName the name of the calling method for exception reporting.

parameterName the name of the calling method's parameter for exception reporting.

allowUndefined (optional) - true if this should allow an undefined or null value without throwing an exception. Defaults to false.

**Returns:** the passed collection. When allowUndefined is true, this returns an empty Set when value is undefined or null.  
  
**Throws:** UnexpectedDataType Exception if value is undefined, null, or isn't a Set.

checkSimpleMap(value, callingClassName, callingMethodName, parameterName, allowUndefined)

**Description:** checks that the passed value is a simple map / dictionary. These are simple JavaScript objects created as follows: {key1: value1, key2: value2}  
Note that it is prefereable to use a true Map object (see checkMap()).

**Parameters:**

value - the value to be checked.

callingClassName - the name of the calling class for exception reporting.

callingMethodName - the name of the calling method for exception reporting.

parameterName - the name of the calling method's parameter for exception reporting.

allowUndefined (optional) - true if this should allow an undefined or null value without throwing an exception. Defaults to false.

**Returns:** the passed map. When allowUndefined is true, this returns an empty map when value is undefined or null.  
  
**Throws:** an UnexpectedDataType Exception if value is undefined, null, or isn't a map.

=

checkSoapResponse(value, callingClassName, callingMethodName, parameterName)

**Description**: Checks that the passed value is a SoapResponse.

**Parameters:**

value - the value to be checked.

callingClassName - the name of the calling class for exception reporting.

callingMethodName - the name of the calling method for exception reporting.

parameterName - the name of the calling method's parameter for exception reporting.  
  
**Returns:** the passed SoapResponse.  
  
**Throws:** an UnexpectedDataType Exception if value is undefined, null, or isn't a SoapResponse.

checkSpecificCustomObject(value, callingClassName, callingMethodName, parameterName, className)

**Description:** Checks that the passed value is a specific custom class. This supports objects that an "instanceof" test won't work for but, as a result, have included a "className" variable to use for this purpose. This includes Adobe objects as well as those defined in this code base.

**Parameters**:

value - the value to be checked.

callingClassName - the name of the calling class for exception reporting.

callingMethodName - the name of the calling method for exception reporting.

parameterName - the name of the calling method's parameter for exception eporting.

className -a string containing the name of the class. This value must be set in a "className" variable in the object being tested for the test to pass.

**Returns:** the passed value.  
  
**Throws:**

UnexpectedDataType Exception if value is undefined, null, or not a custom object of the expected class.

checkString(value, callingClassName, callingMethodName, parameterName, allowUndefined)

**Description:** Checks that the passed value is a string or a String object.  
  
**Parameters:**

value - the value to be checked.

callingClassName - the name of the calling class for exception reporting.

callingMethodName - the name of the calling method for exception reporting.

parameterName - the name of the calling method's parameter for exception reporting.

allowUndefined (optional) - true if this should allow an undefined or null value without throwing an exception. Defaults to false.

**Returns:** the passed value as a string primitive. When allowUndefined is true, this returns "" when value is undefined or null.  
  
**Throws:**

UnexpectedDataType Exception if value is undefined, null, or isn't a valid string.

checkStringArray(value, callingClassName, callingMethodName, parameterName, allowUndefined, allowUndefinedContent)

**Description:** Checks that the passed value is an array of strings and/or String o

bjects.  
  
**Parameters:**

value - the value to be checked.

callingClassName - the name of the calling class for exception reporting.

callingMethodName - the name of the calling method for exception reporting.

parameterName - the name of the calling method's parameter for exception reporting.

allowUndefined (optional) - true if this should allow an undefined or null value without throwing an exception. Defaults to false.

allowUndefinedContent (optional)- true if this should allow the array to contain undefined or null values without throwing an exception. Defaults to false.

**Returns:** he passed value, even if its an empty array. When allowUndefined is true, this returns an empty array when value is undefined or null.  
  
**Throws:** UnexpectedDataType Exception if value is undefined, null, or isn't a valid string.

checkSubform(value, callingClassName, callingMethodName, parameterName)

**Description:** Checks that the passed value is an Adobe Subform.  
  
**Parameters:**

value - the value to be checked.

callingClassName - the name of the calling class for exception reporting.

callingMethodName - the name of the calling method for exception reporting.

parameterName -the name of the calling method's parameter for exception reporting.  
  
**Returns:** the passed Subform.  
  
**Throws:** an UnexpectedDataType Exception if value is undefined, null, or isn't an Adobe Subform.

XFA Forms Framework: <DRAFT> UiUtil

A class that contains utility functions for dealing with the user interface.

Package

core.util.UiUtil

Methods

allowCustomTextEntry(aPickList, aBoolean)

**Description:** Adjusts the passed pickList to allow the entry of custom text (behave like a combo box) or not.  
  
**Parameters:**

aPickList- an Adobe field (pickList) on the form.

aBoolean - true if the pickList should allow the entry of custom text.  
  
**Returns:** null  
  
**Throws:**

throws an UnexpectedDataType Exception if the either of the passed parameters are undefined, null, or aren't the expected data types.

convertListItemsToString(aListItemCollection)

**Description:** Converts a collection of Objects that implement the ListItemInterface to a comma-delimited String suitable for updating a picklist in the UI. This may be a relatively expensive process.

**Parameters:**

aListItemCollection - an Array or Set of the Objects that implement the ListItemInterface that are to be displayed in the list.  
  
**Returns:** a comma delimited String containing the valid values to be displayed in a pick list. An empty String if aListItemCollection is empty.  
  
**Throws:** an InvalidArgument Exception if any of the items in aListItemCollection don't implement the interface.

hide(aNode)

**Description:** Makes the passed node hidden. Sets presence to hidden  
  
**Parameters:**

aNode an Adobe node (field) on the form.  
  
**Returns:** null  
  
**Throws:**

throws an UnexpectedDataType Exception if the passed node is undefined, null, or isn't an Adobe node.

isCheckBoxChecked(aCheckBoxNode)

**Description:** Gets the value of the passed checkBox.  
  
**Parameters:**

aRadioButtonNode the Adobe checkBox field node to check.  
  
**Returns:** a boolean - true if the checkBox is checked.  
  
**Throws:**

UnexpectedDataType Exception if aCheckBoxNode is undefined, null, or isn't an Adobe checkBox node.

isMandatory(aNode)

**Description:** Gets whether the passed node is required (mandatory).  
  
**Parameters:**

aNode an Adobe node (field) on the form.  
  
**Returns:** a boolean - true if the passed node is mandatory. This will also return false if the node doesn't have a mandatory property.  
  
**Throws:**

UnexpectedDataType Exception if the passed node is undefined, null, or isn't an Adobe node.

isRadioButtonSelected(aRadioButtonNode)

**Description:** Gets whether the passed checkButton (radio button) field node is selected for the passed exclGroup node. This is done to avoid unnecessary UI updates.  
  
**Parameters:**

aRadioButtonNode - a checkButton field Adobe node (i.e. radio button) that is a child of the given exclGroup node  
  
**Returns:** a boolean - true if the node was updated.  
  
**Throws:** an UnexpectedDataType Exception if the aNode is undefined, null, or isn't an Adobe node or if the value isn't one of the required types.

isReadOnly(aNode)

**Description:** Gets whether the passed node is read-only.  
  
**Parameters:**

aNode an Adobe node (field) on the form.  
  
**Returns:** a boolean - true if the passed node is read-only.  
  
**Throws:**

UnexpectedDataType Exception if the passed node is undefined, null, or isn't an Adobe node.

function isVisible(aNode)

**Description:** Gets whether the passed node is visible.  
  
**Parameters:**

aNode- an Adobe node (field) on the form.  
  
**Returns:** a boolean - true if the passed node is visible.  
 **Throws:**

an UnexpectedDataType Exception if the passed node is undefined, null, or isn't an Adobe node.

moveFocus()

**Description:** Moves the focus to the target field that was set in setFocus(). Has no effect if the target field has not been set. This should be called from a function bound to a ready:layout event in the form. The target is automatically cleared so it only happens once. See setFocus() for more details.  
  
**Returns**: Null

resetColour(aField, force)

**Description:** Sets the colour of the passed field to its "normal" state. If it is currently read-only, it is set to that color. If not, it is set to how it was defined in the template (i.e. as it was originally coded in the IDE).  
  
**Parameters:**

aField- an Adobe field on the form.  
force (optional) -a Boolean. When true, this will set the field's color to the value of its template even if it is read only.

**Returns:** null  
  
**Throws:**

UnexpectedDataType Exception if the passed field is undefined, null, or isn't an Adobe field or if force isn't a booelan.

MissingDefinition Exception if the passed field was not defined in the template.

selectRadioButton(aRadioButtonNode)

**Description:** Sets the value of the exclGroup node to the value of the passed checkButton field node. Note that this does not alter the node if its current value is the same as the one passed. This is done to avoid unnecessary UI updates. Relies on updateNode to perform the update. This method extracts the value from a checkButton and sends it to updateNode to do the job.  
  
**Parameters:**

aRadioButtonNode a checkButton field Adobe node (i.e. the radio button) that is a child of the given exclGroup node  
  
**Returns:** a boolean - true if the node was updated.  
  
**Throws:** an UnexpectedDataType Exception if the aNode is undefined, null, or isn't an Adobe node or if the value isn't one of the required types.

setCaption(aNode, text)

**Description:** Sets the caption (label) of a field. This handles both plain text and rich text (XML). It will update the plain text if that is present on the node. When dealing with plain text, it will automatically append an asterix to the end of the label if the field is mandatory.  
  
**Parameters:**

aNode - an Adobe node (field) on the form.

text - a String that contains the caption to be set on the field.

**Returns:** Null

**Throws:**

UnexpectedDataType Exception if the either of the passed parameters are undefined, null, or aren't the expected data types.

setColour(aField, colour)

**Description:** Sets the colour of the passed field.  
  
**Parameters:**

aNode - an Adobe field on the form.

colour - a String that contains the colour as comma delimited RGB  
  
**Returns:** Null  
  
**Throws:** an UnexpectedDataType Exception if the either of the passed parameters are undefined, null, or aren't the expected data types.

setFocus(aField)

**Description:** This is used to move focus to a specific field. Focus cannot be moved directly using the Adobe functions, since JavaScript is processed before the form layout is ready to be rendered. It will have no effect. Instead, this method is used to set the target field so that it can be moved to when the layout is ready. For this to work, a call to moveFocus() should be made from a function bound to a ready:layout event in the form. That will move the focus to the target field and clear it so it only happens once.  
  
**Parameters:**

aField - an Adobe field node where focus should be passed to.

**Returns:** Null  
  
**Throws:** an UnexpectedDataType Exception if the passed field is undefined, null, or isn't an Adobe field.

setMandatory(aNode, aBoolean)

**Description:** Sets whether the passed node is required (mandatory). This is usually part of its UI definition, but it occasionally needs to be changed based on business rules.  
  
**Parameters:**

aNode - an Adobe node (field) on the form.  
aBoolean - a boolean - true if the passed node should be mandatory.  
  
**Returns**: Null  
  
**Throws:**

UnexpectedDataType Exception if the either of the passed parameters are undefined, null, or aren't the expected data types.

setPickList(aNode, anObject)

**Description:** Sets a pick list for a given field, based on the passed object that contains the values. This will automatically add an empty item to the top of the list if the field isn't mandatory. It will also automatically select the item from the list if there is only one.  
  
**Parameters:**

aNode - an Adobe node (field) on the form.

anObject - the object that contains the list values. This must be one of:  
1)A String containing a comma delimited list of values to be displayed in the list. This assumes 2 columns (see numberOfColumns in setPickListFromString() for interpretation details).  
2)An Array or Set of Objects that implement the ListItem Interface.  
3)An Array or Set of the Strings that are the values. Note that this assumes these are not coded (i.e. 1 column).  
4)Null, which will clear the list.  
  
**Returns:** Null  
  
**Throws:**

UnexpectedDataType Exception if the aNode is undefined, null, or isn't an Adobe Node.

InvalidArgument Exception if anObject isn't one of the required types.

setPickListFromString(aNode, commaDelimitedString, numberOfColumns, forceClear)

**Description:** Sets a pick list for a given field, based on a comma-delimited string. This will automatically add an empty item to the top of the list if the field isn't mandatory. It will also automatically select the item from the list if there is only one.  
  
**Parameters:**

aNode - an Adobe node (field) on the form.

commaDelimitedString - a String containing the valid values to be displayed in the list. Any embedded commas in the values must be escaped with an additional comma. See numberOfColumns for interpretation details.If passed null or an empty String, the list will be cleared.

numberOfColumns (optional) a number that indicates the number of columns of data there are. When used, this must be a 1 or 2. When it is 1, the commaDelimitedString is interpreted as "value1,value2,value3...". When it is 2, the commaDelimitedString is interpreted as "displayValue1,code1,displayValue2,code2,...". Defaults to 2.

forceClear (optional) a boolean that indicates whether this should always clear the selected value. If false, this only clears it if it isn't in the new pick list. Defaults to false.

**Returns:** Null  
  
**Throws:**

UnexpectedDataType Exception if the passed parameters are undefined, null, or aren't the expected data types.

InvalidArgument Exception if numberOfColumns isn't 1 or 2.

setReadOnly(aNode, aBoolean)

**Description:** Sets whether the passed node is read-only. Note that this also adjusts the colour.  
  
**Parameters:**

aNode - an Adobe node (field) on the form.

aBoolean - true if the passed node is read-only.  
  
**Returns**: Null

**Throws:**

UnexpectedDataType Exception if the either of the passed parameters are undefined, null, or aren't the expected data types.

setToolTip(aNode, plainText)

**Description:** Sets the plain-text tool tip of a field.  
  
**Parameters:**

aNode - an Adobe node (field) on the form.

plainText - a String that contains the tooltip to be set on the field. Undefined or null are treated the same as an empty string - they clear any existing tool tip

**Returns:** Null

**Throws:**

UnexpectedDataType Exception if the passed node is undefined, null, or isn't an Adobe node, or if the plainText isn't a String.

show(aNode)

**Description:** Makes the passed node visible.  
  
**Parameters:**

aNode an Adobe node (field) on the form.  
  
**Returns:** Null  
  
**Throws:**

UnexpectedDataType Exception if the passed node is undefined, null, or isn't an Adobe node.

updateCheckBox(aCheckBoxNode, aBoolean)

**Description:** Sets the value of the passed checkBox to the passed boolean.  
  
**Parameters:**

aRadioButtonNode - the Adobe checkBox field node to update.

Boolean -true if the checkBox should be checked.  
  
**Returns:** Null  
  
**Throws:**

UnexpectedDataType Exception if aCheckBoxNode is undefined, null, or isn't an Adobe checkBox node.

UnexpectedDataType Exception if aBoolean is undefined, null, or isn't a boolean.

updateCheckBox(aCheckBoxNode, aBoolean)

**Description:** Sets the value of the passed checkBox to the passed boolean.  
  
**Parameters:**

aRadioButtonNode- the Adobe checkBox field node to update.

aBoolean- true if the checkBox should be checked.  
  
**Returns:** Null

**Throws:**

UnexpectedDataType Exception if aBoolean is undefined, null, or isn't a boolean.

updateNode(aNode, value)

**Description:** Sets the value of the passed node to the passed value. Note that this does not alter the node if its current value is the same as the one passed. This is done to avoid unnecessary UI updates.  
  
**Parameters:**

aNode - an Adobe node (field or exclusion group) on the form.

value - the new value for the field. This must be a String, boolean, number, or null.  
  
**Returns:** boolean - true if the node was updated.  
  
**Throws:**

UnexpectedDataType Exception if the aNode is undefined, null, or isn't an Adobe node or if the value isn't one of the required types.

XFA Forms Framework: <DRAFT>ValidationFailure

This is a "value" object that contains the information about an object's validation failures. These can be nested to support an object reporting its child object's failures within its own (in a tree structure). In one scenario, these are generated by FormNodes. A field will generate a single instance with a separate message for each way it is invalid (usually only one). The subform that contains the field will generate a single instance with all of its fields' failures in its nested failures collection. If that subform has a parent subform, the parent will generate a single instance with the child subform's instance as part of its nested failures collection. All entries are intended to be localized strings so they can be displayed to the user.

Methods

createInstance(sourceName, failureMessage)

**Description:** Gets a new ValidationFailure. Adobe does not allow object instances to be created outside of their packaging, so it must be done here.

**Parameters:**

sourceName- a String that contains the name of the object that had the validation failure.

failureMessage (optional) a String that contains the localized validation failure message (reason this validation failed).

**Returns:** the new ValidationFailure instance

**Throws:**

UnexpectedDataType Exception if any of the parameters aren't as indicated.

addFailureMessage(aString)

**Parameters:**

aString a String containing a localized validation failure message. A null or undefined value is ignored.

**Throws:**

an UnexpectedDataType Exception if aString isn't a String.

hasFailureMessages()

**Description:** Gets whether there are any localized validation failure messages (reasons this validation failed) for the named source object.

**Returns:** (boolean) true if there are any localized validation failure messages for the named source object

addNestedValidationFailure(aValidationFailure)

**Description:** Adds a ValidationFailure for a nested object to the Array. Note that only one nested ValidationFailure with the same name and index is supported. Attempting to add another with the same name and index as an existing one will cause its error messages to be merged into those of the existing one.

**Parameters:**

aValidationFailure- a ValidationFailure representing the failures that a particular nested object had. A null or undefined value is ignored.

**Throws:**

throws an UnexpectedDataType Exception if aValidationFailure isn't a ValidationFailure.

getNestedValidationFailures()

**Description:** Gets an Array of ValidationFailures for nested objects.

**Returns:** an Array of ValidationFailures for nested objects

hasNestedValidationFailures()

**Description:** gets whether there are any ValidationFailures for nested objects.

**Returns**: (boolean) true if there are any ValidationFailures for nested objects.

getSourceIndex()

**Description:** Gets the index of the object that had the validation failure. Only used when that object is part of a collection.

**Returns:** a Number containing the index of the object that had the validation failure. -1 if this is not applicable. Defaults to -1.

setSourceIndex(aNumber)

**Description:** Sets the index of the object that had the validation failure. Only used when that object is part of a collection.

**Parameters:**

aNumber- a Number containing the index of the object that had the validation failure. Defaults to -1.

**Throws:**

UnexpectedDataType Exception if aNumber isn't a Number.

getSourceName()

**Description:** Gets the name of the object that had the validation failure.

**Returns:** a String containing the name of the object that had the validation failure.

=

hasAnyValidationFailures()

**Description:** gets whether there were any failures (failure messages or nested validation failures.

**Returns:** (boolean) - true if there were any validation failures by the source object, its fields, or its nested objects.

asString()

**Description:** Describes the object

**Returns:** a string description of the object