



# InfoBus 1.2 Changes Summary

This document describes the changes planned for InfoBus for version 1.2. This release adds a new interface, `ReshapeableArrayAccess`, which extends `ArrayAccess` to allow various ways of changing the dimensions of an array. Some new interfaces, classes, and methods will be included. This release is source and binary compatible with applications written for InfoBus 1.1.1. Applications written for InfoBus 1.2 will not be backward compatible with InfoBus 1.1.1 if they use any of the new features.

The document presents only the changes and additions to the InfoBus spec, and assumes that the reader is familiar with the InfoBus 1.1.1 Specification. When the InfoBus 1.1.1 Specification is referenced by page number, it refers to the page number found in the PDF version of the spec dated August 11, 1998. All changes in this document will be applied to the InfoBus 1.2 Specification, which will be published separately. The changes will also appear in the InfoBus 2.0 Specification, which will be released at the same time or shortly after the InfoBus 1.2 Specification.

The purpose for this document is to allow InfoBus developers to review the planned changes, and offer feedback, suggestions, and requests for other changes. Such comments should be sent to [infobus-comments@java.sun.com](mailto:infobus-comments@java.sun.com).

© 1998 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, California 94303, U.S.A. All rights reserved.  
RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013(c)(1)(ii) and FAR 52.227-19. The release described in this document may be protected by one or more U.S. patents, foreign patents, or pending applications. Sun Microsystems, Inc.

(SUN) hereby grants to you a fully-paid, nonexclusive, nontransferable, perpetual, worldwide limited license (without the right to sublicense) under SUN's intellectual property rights that are essential to practice this specification. This license allows and is limited to the creation and distribution of clean room implementations of this specification that:

- (i) include a complete implementation of the current version of this specification without subsetting or supersetting;
- (ii) implement all the interfaces and functionality of the standard java.\* packages as defined by SUN, without subsetting or supersetting;
- (iii) do not add any additional packages, classes or methods to the java.\* packages;
- (iv) pass all test suites relating to the most recent published version of this specification that are available from SUN six (6) months prior to any beta release of the clean room implementation or upgrade thereto;
- (v) do not derive from SUN source code or binary materials; and
- (vi) do not include any SUN binary materials without an appropriate and separate license from SUN.

Sun, Sun Microsystems, Sun Microsystems Computer Corporation, the Sun logo, the Sun Microsystems Computer Corporation logo, Java, JavaSoft, JavaScript, and HotJava are trademarks or registered trademarks of Sun Microsystems, Inc. UNIX® is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. All other product names mentioned herein are the trademarks of their respective owners.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

---

## Changes in the InfoBus 1.2 Specification

---

In section 4.12, “The `ArrayAccess` interface”, the method `getItemByCoordinate()` now specifies that `null` should be returned to indicate an empty cell.

The following minor changes have been made in this version of the specification, mainly fixes to spec wording problems. These do not represent any changes to the functioning of InfoBus:

- In several places, references to InfoBus 1.2 have been changed to InfoBus 2.0. The release that uses JDK 1.2 security has been renamed to InfoBus 2.0. Additional releases of InfoBus 1.x for JDK 1.1.x will be numbered 1.2 and so on.
- In section 2.8, “The InfoBus class”, the incorrect reference to “InfoBusDefaultPolicies” has been corrected to “DefaultPolicy.”
- In section 3.2, “Event listeners,” the spec used to state that methods to add event listeners were called “in” the `start()` method. This has been corrected to “during the execution of” the `start()` method. The same correction was applied for removing event listeners during the execution of the `stop()` method. The changes reflects the fact that in the sample application, the event listeners are added and removed in the `propertyChange()` method of the `PropertyChangeListener` interface.
- In section 4.11, “The ImmediateAccess interface”, the API for `ImmediateAccess.setValue()` incorrectly specified `Object` as the return type. This has been corrected to `void`, to agree with the code and JavaDoc.
- In section 5.4, “The RowsetAccess interface”, a wording error was fixed.
- In section 6.4, “The `DataItemChangeEvent` class and event subclasses”, a wording error was fixed.
- In section 8.1, “The InfoBusPolicyHelper interface”, the text ran over the bottom of the page. The layout has been fixed.

A few major changes will be applied to the specification, as shown in the replacement text that follows.

### 4.13 The ReshapeableArrayAccess interface

*Note to reviewers: This is a new section that will follow section 4.12, The `ArrayAccess` interface. Sections following this one will be renumbered in the new spec.*

Producers that allow a consumer to change the shape of an array, including changing the dimensions, the number of dimensions, or inserting or deleting all elements in a particular dimension (i.e. “delete column”), implement the `ReshapeableArrayAccess` interface to indicate their willingness to change the shape and provide methods used for this purpose. `ReshapeableArrayAccess` extends `ArrayAccess` to add these methods.

```
public void setDimensions(int[] newDimensions) throws IllegalArgumentException
```

This method can be called to change the dimensions of an existing array. Such changes involve appending or removing one or more new columns, rows, planes, etc from the end(s) of the array . Any new cells that are added in this operation must be empty cells (e.g., calling `getItemByCoordinate()` on a new cell returns `null`).

The number of integers in *newDimensions* indicate the requested number of dimensions in the object implementing the `ArrayAccess`, and the value of each integer indicates the requested number of elements in the `ArrayAccess` in that dimension.

All producers that implement this interface must be able to change existing dimensions. Producers may or may not support a change to the number of dimensions; those that do not support a change in number of dimensions throw the `UnsupportedOperationException` for such attempts.

`IllegalArgumentException` must be thrown if any of the specified *newDimensions* are less than one (zero for any dimension would result in an array with no cells).

For example, given a 2x2 array, passing *newDimensions* as [2,3] adds a new row. Calling this method with [2,2,3] either adds two new 2x2 planes to form a three-dimensional array of 2x2x3, or if the producer does not support changes to the number of dimensions, `UnsupportedOperationException` is thrown.

```
public void insert(int dimension, int position, int count)
    throws IllegalArgumentException
```

This method is called to insert additional cells, columns, rows, planes, etc, in a specified *dimension*. *dimension* specifies which entry will be affected in the array of dimensions for this array (as returned by **getDimensions()**). When adding cells in a given *dimension*, the extents of other *dimensions* are unchanged. For example, in a two-dimensional array, when inserting a new row of cells, the number of cells added is the extent of the column dimension.

*position* is a zero-based cell offset in the specified *dimension*. *position* can range from zero up to the maximum extent for the specified *dimension*. When *position* is zero up to the extent for the indicated *dimension*, cells are inserted before the indicated cell. When *position* is equal to the extent for the indicated *dimension*, cells are appended at the end of the array.

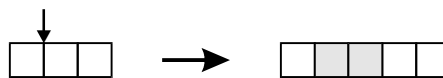
*count* specifies the number of cells to be inserted in the specified *dimension*; it must be greater than zero. The total number of cells inserted with the operation is *count* times the extents of each dimension other than the specified *dimension*. For example, when *count* is one for a 3x3 array, a new row or column of three cells is inserted. See the examples below for more information.

IllegalArgumentException must be thrown if *dimension* is greater than the count of entries from the dimensions returned by **getDimensions()**. It must be thrown if *position* is less than zero or greater than the count of cells in the specified *dimension*. It must be thrown if *count* is not greater than zero.

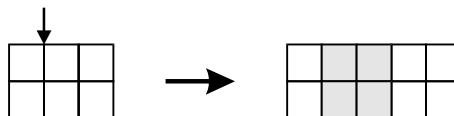
Finally, for the special case of an n-dimensional array, where n>1 and one or more of the dimensions other than the one specified by *dimension* has a zero extent (thus, the array has no cells), it is not possible to insert new cells, and IllegalArgumentException must be thrown. In this case the **setDimensions()** method can be used to change the dimensions. Note that an array with a zero extent can be the result of calling the **delete()** method when *position*=0 and *count* is set to the maximum extent for the given *dimension*.

The required behavior is easiest to understand when considering simple pictorial representations. In the following diagrams, the white squares represent cells of an ArrayAccess that already exist, and may be empty or not. The gray cells represent new, empty cells added as a result of the indicated operation. Note that the use of the terms columns, rows, and planes to refer to particular dimensions is arbitrary, and is only used for describing the diagrams for these examples.

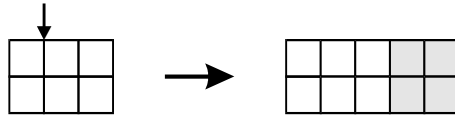
Example 1: Inserting cells in a single-dimension array. Before insertion, **getDimensions()** returns [3]. After calling **insert(0, 1, 2)** to insert two cells before cell 1, new cells are added as shown, and calling **getDimensions()** returns [5].



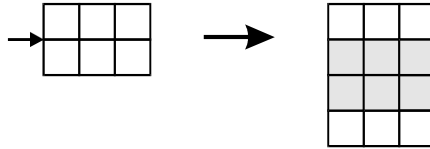
Example 2: Inserting columns in a two-dimension array. Before insertion, **getDimensions()** returns [3,2]. After calling **insert(0, 1, 2)** to insert two columns before column 1, new cells are added as shown, and calling **getDimensions()** returns [5,2].



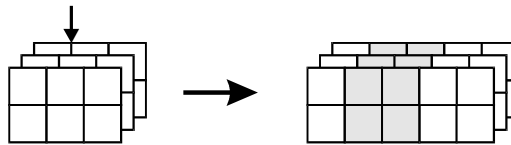
Example 3: Appending columns in a two-dimension array. Before insertion, **getDimensions()** returns [3,2]. After calling **insert(0, 3, 2)** to append two columns (note that the *position* here is equal to the count of cells in dimension 0, meaning “append”), new cells are added as shown, and calling **getDimensions()** returns [5,2].



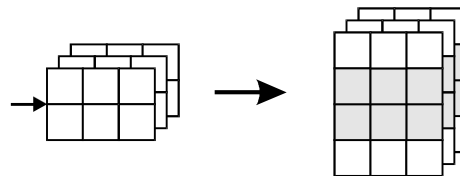
Example 4: Inserting rows in a two-dimension array. Before insertion, `getDimensions()` returns `[3,2]`. After calling `insert(1, 1, 2)` to insert two rows before row 1, new cells are added as shown, and calling `getDimensions()` returns `[3,4]`.



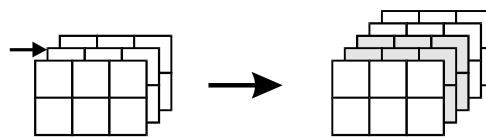
Example 5: Inserting columns into three-dimension array. Before insertion, `getDimensions()` returns `[3,2,3]`. After calling `insert(0, 1, 2)` to insert two columns before column 1, new cells are added as shown in all three planes, and calling `getDimensions()` returns `[5,2,3]`.



Example 6: Inserting rows into three-dimension array. Before insertion, `getDimensions()` returns `[3,2,3]`. After calling `insert(1, 1, 2)` to insert two rows before row 1, new cells are added as shown in all three planes, and calling `getDimensions()` returns `[3,4,3]`.



Example 7: Inserting planes into three-dimension array. Before insertion, `getDimensions()` returns `[3,2,3]`. After calling `insert(2, 1, 2)` to insert two planes before plane 1, new cells are added as shown in two new planes, and calling `getDimensions()` returns `[3,2,5]`.



```
public void delete(int dimension, int position, int count)
    throws IllegalArgumentException
```

This method is called to delete cells, columns, rows, planes, etc, in a specified *dimension*. *dimension* specifies which entry will be affected in the array of dimensions for this array (as returned by `getDimensions()`).

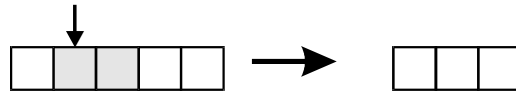
*position* is a zero-based cell offset in the specified *dimension*. When *position* is zero up to the extent for the indicated dimension, deletions start at the indicated cell.

*count* specifies the number of deletions in the specified *dimension*; it must be greater than zero, and less than the remaining cells of the indicated *dimension* starting with the indicated *position*.

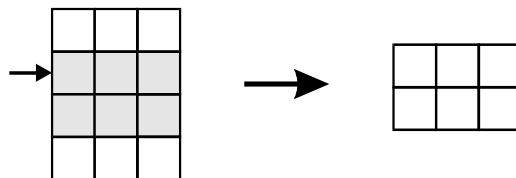
IllegalArgumentException must be thrown if *dimension* is greater than the count of entries from the dimensions returned by `getDimensions()`. It must be thrown if *position* is less than zero or greater than or equal to the count of cells in the specified *dimension* starting with the indicated *position*. It is thrown if *count* is not greater than zero, or if *position* plus *count* is greater than the current extent in the indicated *dimension*.

Some examples of the required behavior:

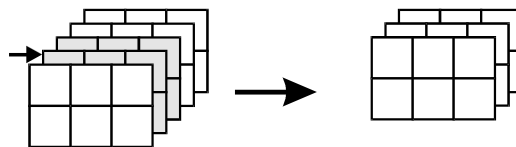
Example 1: Deleting cells in a one-dimension array. Initially, `getDimensions()` returns [5]. After calling `delete(0,1,2)` to delete two cells starting with cell 1, cells are removed as shown, and calling `getDimensions()` returns [3].



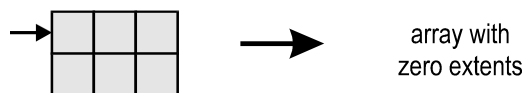
Example 2: Deleting rows in a two-dimension array. Initially, `getDimensions()` returns [3,4]. After calling `delete(1,1,2)` to delete two rows starting with row 1, cells are removed as shown, and calling `getDimensions()` returns [3,2].



Example 3: Deleting planes in a three-dimension array. Initially, `getDimensions()` returns [3,4,5]. After calling `delete(2,1,2)` to delete two planes starting with plane 1, cells are removed as shown, and calling `getDimensions()` returns [3,4,3].



Example 4: Deleting entire contents of a two-dimension array. Initially, `getDimensions()` returns [3,2]. After calling `delete(1,0,2)` to delete two rows starting with row 0 (i.e., all rows), cells are removed as shown, and calling `getDimensions()` returns [0,0]. Calling `delete(0,0,3)` to delete three columns starting with column 0 (i.e., all columns) has the same result. In both cases, empty cells can be added only by calling `setDimensions()` with an array where both extents are non-zero.



## 6.2 The `DataItemChangeManagerSupport` class

*Note to Reviewers: This section has two changes:*

- The name of this class has changed from `DataItemChangeSupport` to `DataItemChangeManagerSupport`. The class with the old name will remain unchanged and marked deprecated.
- In the subsection titled “Event-firing methods”, for which the replacement text appears below, one new method, `fireShapeChangeEvent`, has been added to this support class. For all methods that fire

*events, a description of the `propertyMap` parameter has been added.*

## Event-firing methods

Each method below creates an appropriate change event and sends it to all the listeners at that level only. Events should also be distributed to other levels according to the rules specified in “Distribution and handling of change events,” later in this chapter.

```
public void fireItemValueChanged(Object changedItem,  
    InfoBusPropertyMap propertyMap)
```

This method should be called when an item, usually an `ImmediateAccess`, changes value. The caller indicates the *changedItem* as the one whose value changed. Producers that wish to supply additional information about the change may do so by supplying a *propertyMap*; producers that do not offer this information should supply `null` for this parameter.

```
public void fireItemAdded(Object changedItem, Object changedCollection,  
    InfoBusPropertyMap propertyMap)
```

This method should be called when one or more new items are being added to a collection. The caller indicates the *changedItem* as the one being added, and *changedCollection* as the collection that gained an item. *changedItem* can be `null` when more than one item is added in the same operation. Producers that wish to supply additional information about the change may do so by supplying a *propertyMap*; producers that do not offer this information should supply `null` for this parameter.

```
public void fireItemDeleted(Object changedItem, Object changedCollection,  
    InfoBusPropertyMap propertyMap)
```

This method should be called when one or more items are being removed from a collection. The caller indicates the *changedItem* as the one being removed, and *changedCollection* as the collection that lost an item. *changedItem* can be `null` when more than one item is removed in the same operation. Producers that wish to supply additional information about the change may do so by supplying a *propertyMap*; producers that do not offer this information should supply `null` for this parameter.

```
public void fireItemRevoked(Object changedItem, InfoBusPropertyMap propertyMap)
```

This method should be called when an item or collection is no longer available, such as when the data source is going offline. The caller indicates the *changedItem* as the item or collection that is being revoked. Unlike the other events, this event is sent to the data item passed during `rendezvous`, and to all sub-items in a collection hierarchy. Producers that wish to supply additional information about the change may do so by supplying a *propertyMap*; producers that do not offer this information should supply `null` for this parameter.

```
public void fireRowsetCursorMoved(Object changedItem,  
    InfoBusPropertyMap propertyMap)
```

This method should be called when a rowset’s cursor has moved to a different row. The caller indicates the *rowset* whose cursor changed. Producers that wish to supply additional information about the change may do so by supplying a *propertyMap*; producers that do not offer this information should supply `null` for this parameter.

```
public void fireItemShapeChanged(Object changedItem,  
    InfoBusPropertyMap propertyMap)
```

This method should be called when a data item changes shape. For example, a `DataItemShapeChangeEvent` should be fired when any of the extents of a `ReshapeableArrayAccess`, as reported by `getDimensions()`, change, for example as a result of calling `setDimensions()`, `insert()`, or `delete()`.

The caller indicates in *changedItem* the `ReshapeableArrayAccess` object that is changing dimensions. Producers that wish to supply additional information about the change may do so by supplying a *propertyMap*; producers that do not offer this information should supply `null` for this parameter.

## 6.4 The `DataItemChangeListener2` interface

*Note to Reviewers:* This interface extends `DataItemChangeListener` to add one new method, the `DataItemShapeChangeEvent`. See also the new `DataItemChangeListenerSupport` class described in Section 6.5 (below). Section 6.4 of the *InfoBus 1.1.1 Specification*, and all sections following it, will be renumbered to accommodate these new sections.

About the name: we are not allowed to add methods to existing interfaces for new releases of the spec, because it breaks binary and source compatibility with components written for an earlier version of the spec. Hence the name has the number 2 to indicate a new version of the interface. I'm open to other ideas on naming this. Keep in mind that if we add other change event types in the future, we'll need to use the same technique for adding event delivery methods; this is as a result of having a separate event delivery method for each event type, rather than a common one for all events, as in the original design.

This interface extends the `DataItemChangeListener` interface to add a new change event delivery method.

```
public void dataItemShapeChanged(DataItemShapeChangedEvent event)
```

Indicates a change in the shape for a data item. For example, a `ReshapeableArrayAccess` has been changed such that one or more of its dimensions has changed. Provided by `get`. A reference to the data item that changed can be obtained from the *event*.

## 6.5 The `DataItemChangeListenerSupport` class

*Note to Reviewers:* This is a new support class to make it easier to implement change listeners. This new section will follow the new section 6.4, the `DataItemChangeListener2` interface, above. Section 6.4 of the *InfoBus 1.1.1 Specification*, and all sections following it, will be renumbered to accommodate these new sections.

This class implements `DataItemChangeListener2`, and can be used by a consumer as a base class for implementing a change listener class. All methods in this class have empty-body implementations. Thus, to use this class, the consumer should subclass it and override the methods for handling events of interest.

```
public void dataItemValueChanged(DataItemValueChangedEvent event)
```

Default handler for the `DataItemValueChangedEvent`, which simply ignores the *event*. If the *event* is of interest, the consumer should override this method with one that handles the event.

```
public void dataItemAdded(DataItemAddedEvent event)
```

Default handler for the `DataItemValueAddedEvent`, which simply ignores the *event*. If the *event* is of interest, the consumer should override this method with one that handles the event.

```
public void dataItemDeleted(DataItemDeletedEvent event)
```

Default handler for the `DataItemValueDeletedEvent`, which simply ignores the *event*. If the *event* is of interest, the consumer should override this method with one that handles the event.

```
public void dataItemRevoked(DataItemRevokedEvent event)
```

Default handler for the `DataItemRevokedEvent`, which simply ignores the *event*. If the *event* is of interest, the consumer should override this method with one that handles the event.

```
public void rowsetCursorMoved(RowsetCursorMovedEvent event)
```

Default handler for the `RowsetCursorMovedEvent`, which simply ignores the *event*. If the event is of interest, the consumer should override this method with one that handles the event.

```
public void dataItemShapeChanged(DataItemShapeChangedEvent event)
```

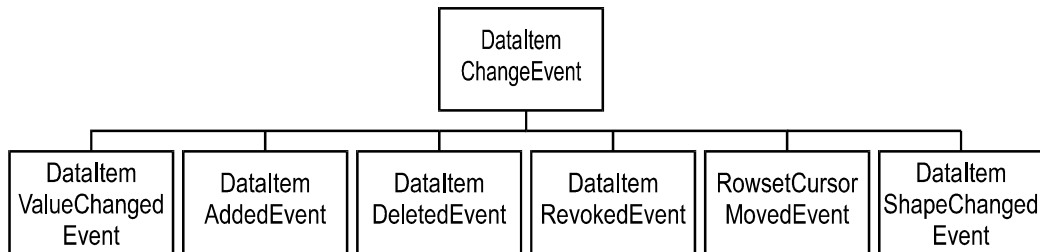
Default handler for the `DataItemShapeChangeEvent`, which simply ignores the *event*. If the event is of interest, the consumer should override this method with one that handles the event.

## 6.6 The `DataItemChangeEvent` class and event subclasses

*Note to Reviewers:* This section has three minor changes compared to the *InfoBus 1.1.1 Specification* section

#### 6.4 The `DataItemChangeEvent` class and event subclasses:

- The section number is incremented because of the two new sections above.
- The subsection formerly titled “`DataItemValueChangedEvent`, `DataItemRevokedEvent` and `RowsetCursorMovedEvent` classes” is renamed to “`DataItemValueChangedEvent`, `DataItemRevokedEvent`, `DataItemShapeChangedEvent` and `RowsetCursorMovedEvent` classes”
- Figure 6-2 has been revised to include the new event in the class hierarchy, as shown below:



**Figure 6-2: Class Hierarchy for `DataItemChangeEvent`s**

### 6.7 Distribution and handling of `DataItemChangeEvent`s

*Note to Reviewers:* This section was numbered 6.5 in the *InfoBus 1.1.1 Specification*. The section is the same as in that spec, with a few changes.

In the subsection titled `DataItemDeletedEvent`, the following paragraph will be inserted after the first and before the second paragraph of the section:

Operations on a `ReshapeableArrayAccess` data item that reduce the number of cells in the array also result in firing `DataItemDeletedEvent` for those cells that are not empty. For example, calling `setDimensions()` when one of the new dimensions is smaller than the corresponding existing dimension will cause the removal of cells; calling the `delete()` method will generally result in the removal of cells as well. For cells being removed that are not empty (i.e. calling `getItemByCoordinate()` would return a non-null reference to a data item), the `DataItemChangeEvent` should be fired for each, indicating the reference to the data item that is being deleted; alternatively, one `DataItemChangeEvent` can be fired with null as the item that is being deleted, meaning that more than one non-empty cell is being removed. These events must be fired before firing the `DataItemShapeChangeEvent`.

*Note to Reviewers:* The following subsection is new; it will appear just before the existing subsection describing `RowsetCursorMovedEvent`:

#### **`DataItemShapeChangeEvent`**

Indicates that the one or more of the dimensions of a `ReshapeableArrayAccess` data item has changed. The event must be sent to listeners of the `ReshapeableArrayAccess` item whose shape has changed, with `changedItem` referring to the `ReshapeableArrayAccess`.

Note that when changes to the shape of this data item results in the removal of one or more non-empty cells, the `DataItemDeletedEvent` must be fired first, indicating the removal of the cells, before sending `DataItemShapeChangeEvent`. See the subsection “`DataItemDeletedEvent`” above for more details.

### 6.8 Examples of event propagation

*Note to Reviewers:* This section was numbered 6.6 in the *InfoBus 1.1.1 Specification*. The section is the same as in that spec, with the addition of three new bullets, which will be placed before the existing bullet that describes the `RowsetCursorChangedEvent`:

- Suppose ‘All’ is a `ReshapeableArrayAccess`. If the `delete()` or `setDimensions()` methods are called such that existing, non-empty cells will be removed, one `DataItemDeletedEvent` must be fired for each non-empty cell, indicating a reference to the cell that was deleted and the

ReshapeableArrayAccess item that contained it, followed by a DataItemShapeChangeEvent sent to listeners of the ReshapeableArrayAccess. Alternatively, for convenience, it is permissible to fire one DataItemDeletedEvent with `null` as the reference to the cell being deleted (meaning more than one non-empty cell was deleted), followed by a DataItemShapeChangeEvent sent to listeners of the ReshapeableArrayAccess. In both cases, the rules for propagating the DataItemDeletedEvent are the same as those described for that event, above. The ReshapeableArrayAccess event must be propagated upward to the parent of the array that changed, and its parent, up to and including the top-level node of the data item.

- For a ReshapeableArrayAccess whose dimensions change such that empty cells are being added, the DataItemShapeChangedEvent should be sent to listeners of the array, to its parent, up to and including the top-level node of the data item.
- For a ReshapeableArrayAccess where more than one dimension is changing as a result of **setDimensions ()**, it is possible that cells are being removed from one dimension while being added in another. In this case, only one ReshapeableArrayAccess is sent per call to **setDimensions ()** to each level of the data item hierarchy from the array up to the top-level node of the data item.

*\*\*\* end of InfoBus 1.2 Changes Summary \*\*\**