The Context in Detail

HELP.BCWDABAPPROGMAN

Documentation

Preview

NetWeaver 7.1 (SPS13)

Copyright

Icons in Body Text

Icon	Meaning
	Caution
	Example
	Note
	Recommendation
	Syntax

Additional icons are used in SAP Library documentation to help you identify different types of information at a glance. For more information, see *Help on Help* → *General Information Classes and Information Classes for Business Information Warehouse* on the first page of any version of *SAP Library*.

Typographic Conventions

Type Style	Description
Example text	Words or characters quoted from the screen. These include field names, screen titles, pushbuttons labels, menu names, menu paths, and menu options. Cross-references to other documentation.
Example text	Emphasized words or phrases in body text, graphic titles, and table titles.
EXAMPLE TEXT	Technical names of system objects. These include report names, program names, transaction codes, table names, and key concepts of a programming language when they are surrounded by body text, for example, SELECT and INCLUDE.
`Example text`	Output on the screen. This includes file and directory names and their paths, messages, names of variables and parameters, source text, and names of installation, upgrade and database tools.
`Example text`	Exact user entry. These are words or characters that you enter in the system exactly as they appear in the documentation.
`<Example text>`	Variable user entry. Angle brackets indicate that you replace these words and characters with appropriate entries to make entries in the system.
`EXAMPLE TEXT`	Keys on the keyboard, for example, F2 or ENTER.

The Context in Detail

In the basics section of this documentation, the binding of UI element properties to context attributes has already been discussed. For a simple application, basically the property *value* is bound to an attribute to display a value from the context on the screen or to pass a user input to the context. Most of the other properties of a UI element can also be determined statically in the table by specifying a value, and can therefore be manipulated at design time without being bound to a context attribute. One example for this is the *enabled* property of a button. At design time, you can determine a button to be active all the time by setting the relevant check in the properties table of the button.

However, in a larger application it may make sense to vary the state of a number of properties depending on particular conditions. In such a case, you can no longer determine the value of the property in question at design time. The property is then bound to a context attribute that can accept different values at runtime. A button can then be active or inactive, depending on the program conditions. Another example is the *visible* property. A table can be visible in one context even though it should not be displayed in a different context.

The Web Dynpro context also offers a multitude of possibilities for more complex applications. However, a more thorough understanding of the functions of the context is required to make the most of these options.

Declaration of the Context

Declaration of the Context at Design Time

When working with the context, it is important to distinguish between the meaning of the word 'context' at design time and at runtime.

At design time, the context represents a description of the structure of the required value sets. Each attribute is typed and is only available once. At runtime, these structured shells are filled with values. A multitude of values can be contained in the context for a single attribute when, for example, depicting tables. You can use the cardinality of a context node to determine at design time whether or not you want to permit a multiple value set for a context attribute at runtime.

> In ABAP programming, the cardinalities 0..n and 1..n correspond to the declaration of an internal table. The cardinality 1..1 corresponds to the declaration of a structure.

Context nodes with a cardinality of 0..n or 1..n are designated as multiple nodes. This means that a single context attribute of a multiple node can adopt a multitude of values at runtime. To enable you to work sensibly in this kind of multidimensional context, the Web Dynpro ABAP Framework provides the *singleton* and *leadselection* properties. These two concepts are described in Chapter **Context Node: Properties**.

Filling the Context with Values at Runtime

There are several possibilities for filling a context with values at runtime. However, a method is called in each case. One of these options is the supply function.

Common Usage of Context Elements by Several Contexts

To enable the parallel use of context elements in different views and the transportation of values from one context to another, Web Dynpro provides the context mapping function. The architecture handbook for Web Dynpro contains a section called Data Binding and Mapping. It schematically depicts the principles of data transport from a UI element such as an input field to the context of the component controller (or vice versa).

Creating Contexts

The tools manual for Web Dynpro for ABAP contains a description of the tools that you use to declare a context and define a mapping. When you create a context node you can influence the result in many ways, such as by binding the structure of the node to be created to an existing structure in the ABAP dictionary. This can be beneficial in many ways, since it means that not only the structure of the context is generated automatically. In addition, all existing texts for, for example, column headers or labels, help texts, and value helps are proposed for use.

> For practical information on declaring and programming contexts, see Programming Notes for Web Dynpro ABAP Applications in Context.

Context Node: Properties

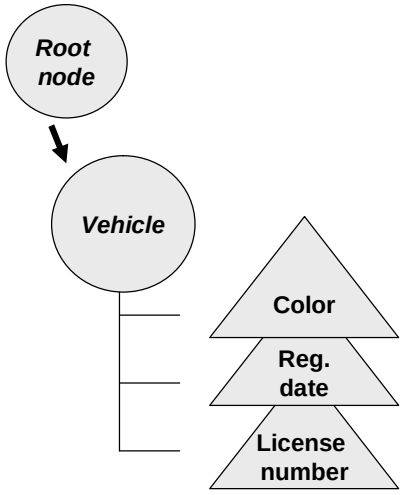
Cardinality of a Context Node

When a node is created in the context of a Web Dynpro component, the cardinality of the node is specified. The cardinality defines how often a node is to be instantiated at runtime – that is, how many elements of this node are available at runtime.

- **1…1** **Exactly one** element is instantiated.
- **0…1** At runtime, **no more than one** element is instantiated, but it is also possible that no element is instantiated.
- **1…n** n elements can be instantiated, but at **least one element** must be instantiated.
- **0…n** The number of instantiated elements of the context node can vary.

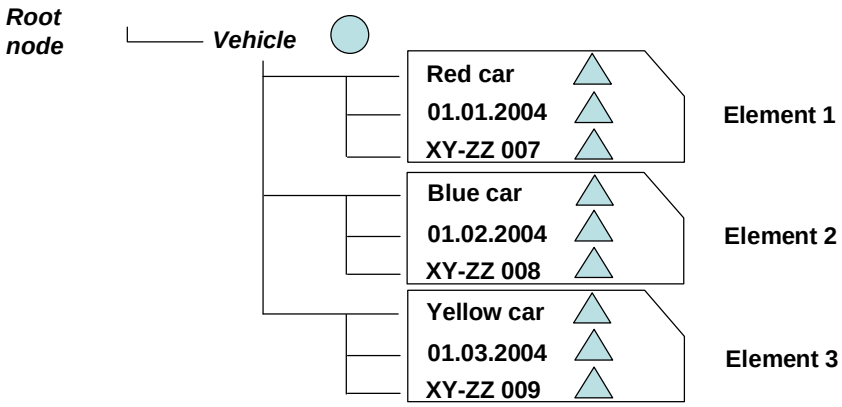
Example:

The *Vehicle* context node is used to describe the fleet of a car rental company. It has the cardinality 1…n and is filled from a database table. A number of attributes of this node have a specific value for each vehicle.

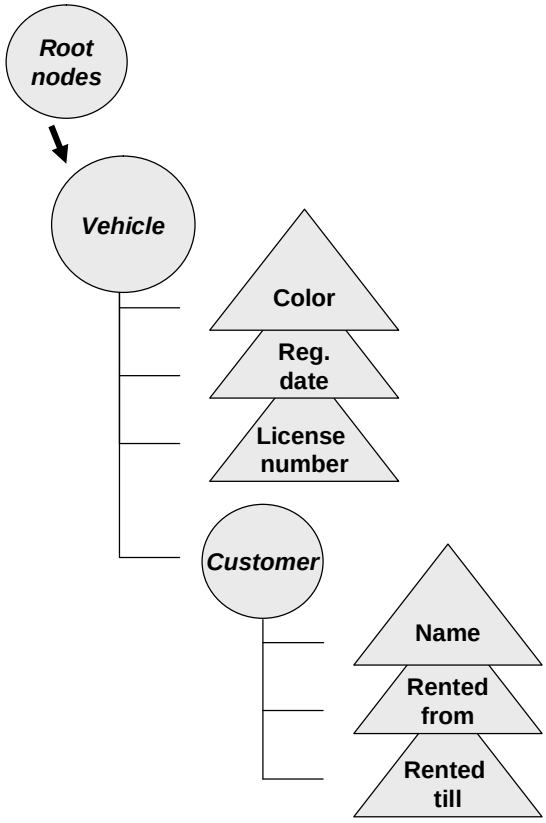


The database table indicates that the company owns three vehicles, each with unique registration dates and unique license plate numbers. Thus, to display this table in the Web Dynpro application, 3 elements of the context node *Vehicle* must be instantiated; the cardinality of the node must therefore be 0…n or 1…n. (If the *Vehicle* node is to be filled with values in the context of another function – for example, from a table of all currently available cars – the cardinality 0…n should be used since this table could be empty – that is, if all cars are rented out.)

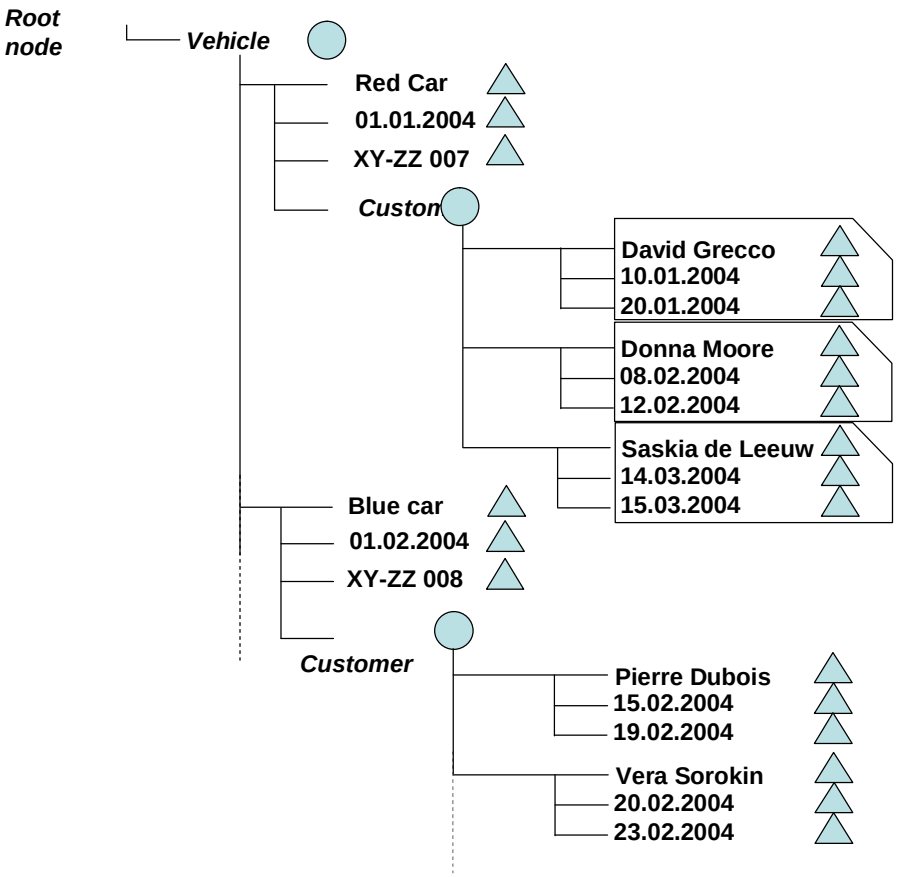
At runtime, the *Vehicle* context node should contain three elements and appear as follows:



As well as these attributes, a context node can also contain additional child nodes:



For each customer of a car rental company, the two attributes “rented from” and “rented till” are listed in the context. At runtime, the context then contains the additional relevant values:



At runtime, each element of the *Vehicle* node contains a subnode called *Customer.* These subnodes can contain further elements in their turn: Thus, the *Customer* node of the first element of the *Vehicle* node (the node with the descriptive data for the red car) contains three elements in its turn, one element for each of the three stored rental procedures.

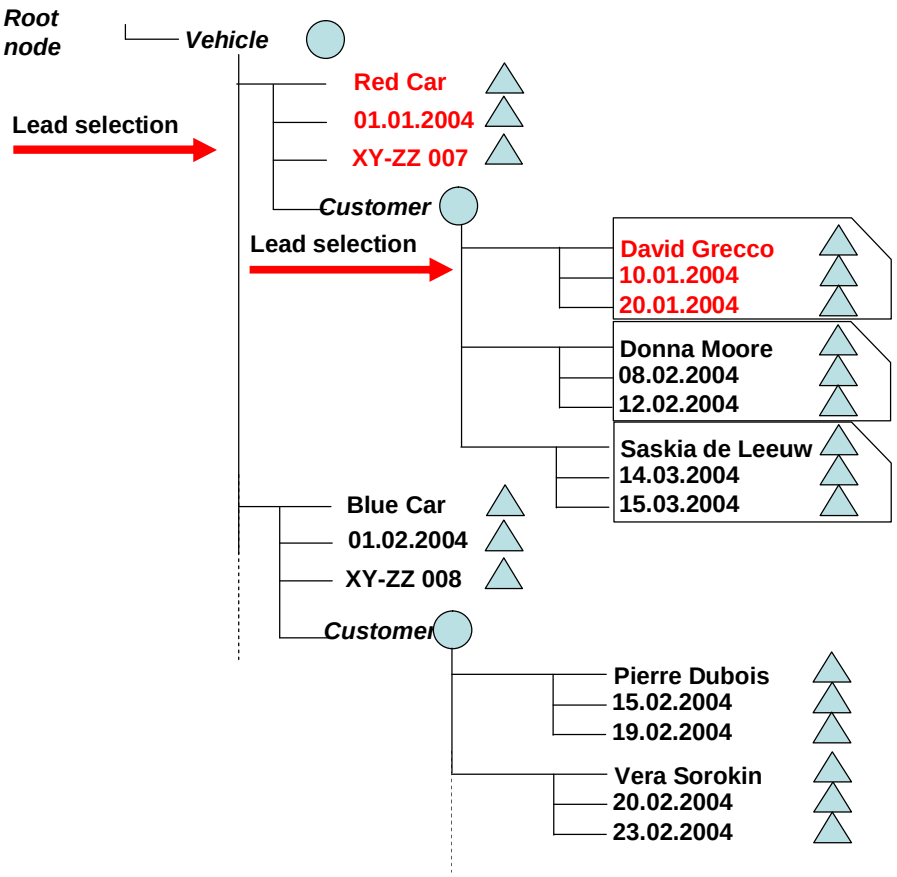
Lead Selection

The lead selection is extremely important within the nested context structure. The lead selection defines which element of the context is ultimately accessed at runtime. This behavior is easiest to explain using an example:

The UI of the example application contains a TextView element that should display the name of the customer at runtime. Without a clear regulation of the selection path for the context structure, all values of the *Name* attribute would be equal and it would not be possibly to correctly display the name in the TextView element. For this reason, one element of the set of possible elements for the *Customer* node must be clearly designated. This is achieved by initializing lead selection. The automatic initialization of the lead selection always designates **the first element of a node.** For the above example, this means the following:

The lead selection is always automatically on the first element of the *Vehicle* node (on the node with the data for the red car). Analog to this, the lead selection is on the first element of the *Customer* subnode (on the element that contains the data for David Grecco).

This special way of designating elements can be depicted as follows in the runtime context.



Only if the lead selection falls on another element of the *Vehicle* node (as a result of user action or a program step in a method) is the corresponding element of this *Customer* subnode displayed in the UI element. If the lead selection is set on the second element of the *Vehicle* node, the customer Pierre Dubois is displayed. The lead selection of the *Customer* node can be varied at runtime in this way. This enables the names of all available customers to be displayed one after the other in the UI.

Automatic/Manual Initialization

Lead selection is initialized automatically for each newly created context node.

- Using the preset automatic initialization of lead selection:

  In this case, the **first element of a node** is always assigned the lead selection property.

- Manual initialization of lead selection:

  If automatic initialization was deactivated, the lead selection can be programmed manually. In this case it is possible to assign this property to an element other than the first element of a node (for example, using an index).

The 'Singleton' Property

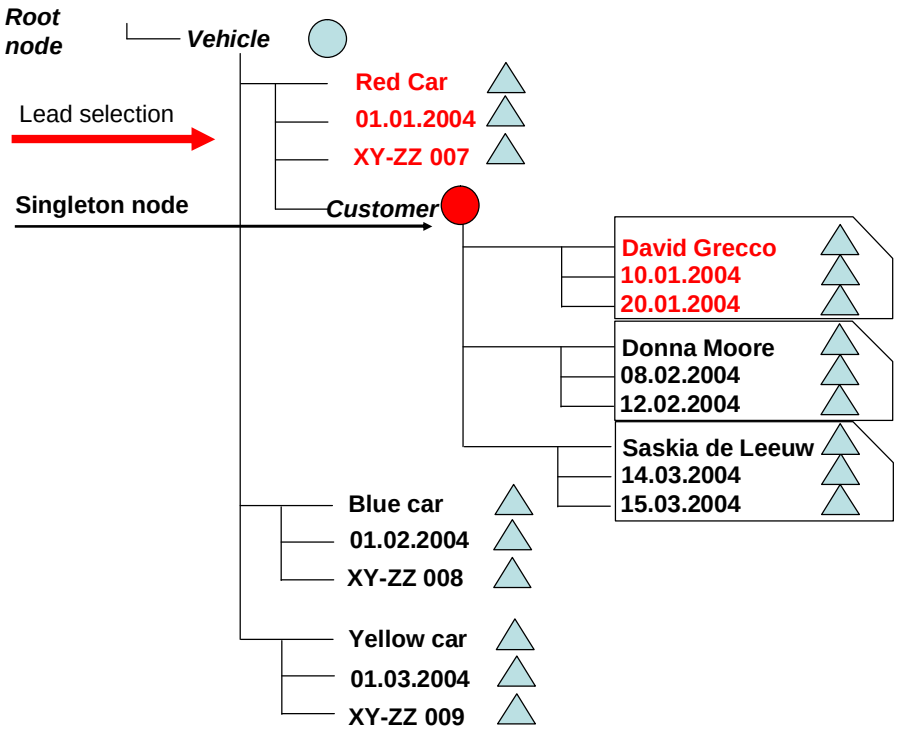
The singleton property of the *Customer* context node specifies that the elements of this node are only instantiated for one of the three vehicles at runtime – that is, the element that bears

the lead selection. If lead selection was initialized automatically, this is the first element of the *Vehicle* node. In this case, this is element 1, which contains the data on the red car.

If each customer has multiple addresses, it may be necessary to include a child node for the addresses below the *Customer* child node. In this manner, the data content of a root node can rapidly become very large if, at runtime, all customers are displayed with all their addresses for each vehicle of the car rental company. To limit the content of a context node at runtime, the context node can be assigned the 'singleton' property. As a result, **the elements of the relevant node are instantiated for only one element of the parent node**. In other words:

Unlike the cardinality of a node, which describes the number of possible elements within the node, the 'singleton' property determines whether or not these elements are set for all elements of the parent node (non-singleton) or for exactly one element of the parent node (singleton).

If the *Customer* node in our example is set as a 'singleton', the context will be as follows at runtime:



The elements of the *Customer* child node are only available to one element of the *Vehicle* parent node and not to all other elements. However, if you want to instantiate the elements of the *Customer* node for all vehicles, you must set the singleton property for the *Customer* node to 'false'.

> Since the root node of a context is only ever instantiated once, every node directly below the root node (in our example, *Vehicle)* is always automatically a singleton node.

Programming of the Context

By declaring a context you essentially defined its structure. However, to fill a context with values and work with these values at runtime, you need to use the program to control the runtime behavior of the context. To do this, you use the controller methods for the context in question. The most simple example is the WDDOINIT hook method. However, in principle you can use all controller methods whose implementation you can influence to edit the context. As well as hook methods, these are event handlers, free methods, and supply functions.

The sections below contain information on the individual program steps required in relation to context programming.

Direct Reference to a Context Node

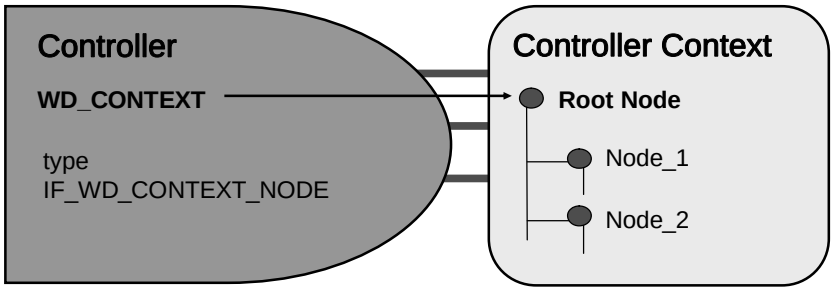
To enable programming in a method on the context, the context of the method in question must be known. The first step therefore always consists of passing the reference to the context node.

Exception: A supply function is always called on a context node. The reference to this node is therefore automatically passed by the framework and the application developer does not pass the reference in the source code for the method. The following parameters are automatically recognized and are listed in the method signature:

NODE of type IF_WD_CONTEXT_NODE

PARENT_ELEMENT of type IF_WD_CONTEXT_ELEMENT

In all other controller methods, the determination of the reference must be programmed explicitly. This takes place on the basis of the context root node of a controller.



Each controller method automatically recognizes the reference to the root node of its associated context. This reference is called WD_CONTEXT and is always of the type IF_WD_CONTEXT_NODE. The GET_CHILD_NODE method now enables the generation of a reference to the required subnodes.

```
method EXAMPLE .

data: l_node type ref to IF_WD_CONTEXT_NODE.

l_node = wd_context->get_child_node( 'NODE_1' ).

. . . . . .

endmethod.
```

The name of the child node must always be specified in uppercase.

Avoiding Syntax Errors: Constants for Context Nodes

For every node you create in the context of a controller, a constant with the name WDCTX_<node name> is automatically created in the corresponding interface IG_<Controller_Name> and IF_<Controller_Name>. In the program source code for the controller, this constant can then be used instead of a string literal for the node name by using the reference to the local controller interface (WD_THIS attribute). Example:

```
wd_context->get_child_node( wd_this->wdctx_node_1 ).
```

instead of

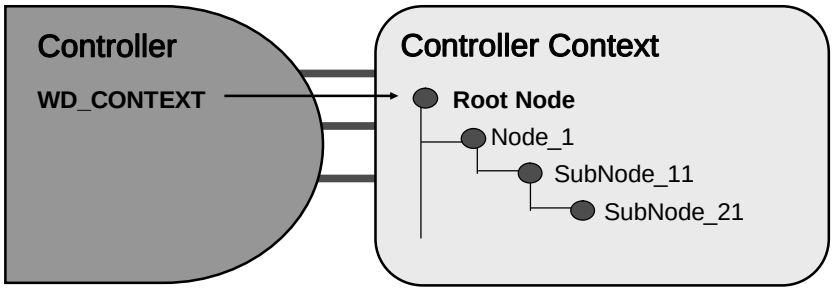
```
wd_context->get_child_node( 'NODE_1' ).
```

The advantage of using constants is that the compiler knows the constant and, therefore, syntax errors are reported if the name of the context node contains typing errors. However, it is also possible to pass a string literal.

Using this kind of direct reference gives you access to a subnode of the context root node. For more information on referencing deeper context nodes, see chapter **Reference Path to Deeper Context Nodes**.

Reference Path to Deeper Context Nodes

If you require access to a node that is a few levels below the root node in the context tree structure, there are two possibilities. The graphic below depicts the structure situation of the context again. The context node to process with the program is SubNode_21.



1st Variant

You can repeat the process outlined in chapter **Direct Reference to a Context Node** step by step until you 'reach' the required node.

```
method EXAMPLE .

data: l_node           type ref to IF_WD_CONTEXT_NODE,
      l_snode_11    type ref to IF_WD_CONTEXT_NODE,
      L_snode_21    type ref to IF_WD_CONTEXT_NODE.


l_node     = wd_context->get_child_node( wd_this->wdctx_node_1 ).
l_snode_11 = l_node->get_child_node( wd_this->wdctx_subnode_11 ).
l_snode_21 = l_snode_11->get_child_node( wd_this->wdctx_subnode_21 ).
. . . . . .

endmethod.
```

2nd Variant

You can use the PATH_GET_NODE method to reference the required subnode.

```
method EXAMPLE .

data:    l_snode_21    type ref to IF_WD_CONTEXT_NODE.

l_snode_21 = wd_context->path_get_node( 'NODE_1.SUBNODE_11.SUBNODE_21' ).
. . . . . .

endmethod.
```

This formulation can be varied depending on which attribute in the method is already known at the time of the call of PATH_GET_NODE.

```
method EXAMPLE .

data: l_node            type ref to IF_WD_CONTEXT_NODE,
            l_snode_21    type ref to IF_WD_CONTEXT_NODE.

. . . . . .

l_snode_21 = l_node->path_get_node( 'SUBNODE_11.SUBNODE_21' ).
. . . . . .

endmethod.
```

 You **cannot** formulate the use of the PATH_GET_NODE **using constants**. You must make sure that you get the node name exactly right, since errors are **not picked up by the syntax check.**

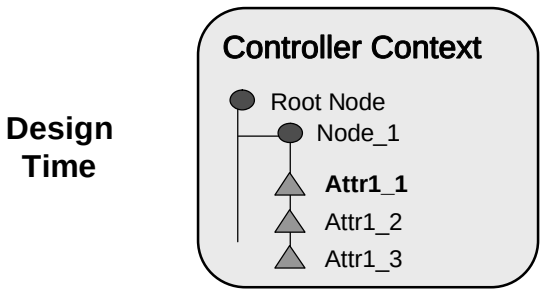
Reading the Attribute Value of a Context Element

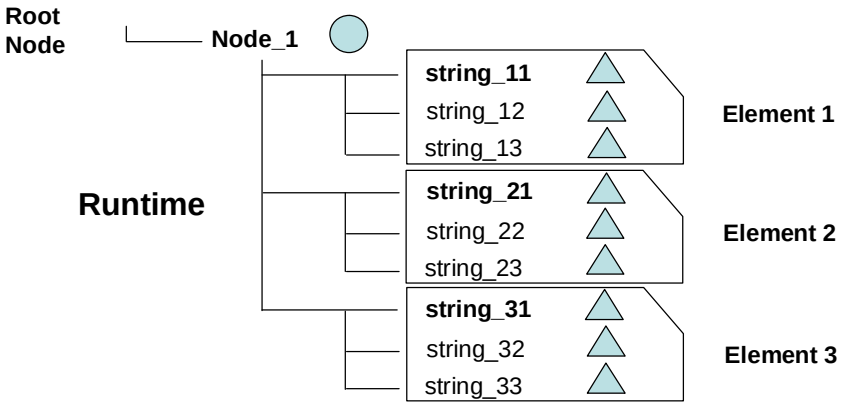
Context nodes retain the Web Dynpro application data at runtime. The IF_WD_CONTEXT_NODE and IF_WD_CONTEXT_ELEMENT interfaces provide appropriate methods for working with this data. In particular, both interfaces contain the GET_ATTRIBUTE method.

There are various different ways to read data from an element of a context node. Some of these options are outlined below.

Reading an Attribute Value of the Lead Selection Element

The easiest way to proceed is to read the attribute value of the lead selection element of a node. The two graphics below schematically depict the context at design time and runtime in order to clarify this.





At runtime, there is a sequence of elements, each with a corresponding data set for the Attr1_1 to Attr1_3 attributes for the Node_1 context node. In the UI, each element might be represented by a table row, for example. Normally, one of these elements (1, 2, or 3) is designated using lead selection. The value of the element designated using lead selection is easy to read from the runtime context.

A local variable of the type IF_WD_CONTEXT_ELEMENT is created in the method currently being processed. This variable represents the relevant element of the node - in this case, the lead selection element. In addition, a variable of the appropriate type must be created. It represents the attribute value in the method.

```
method EXAMPLE .

data: l_node                    type ref to IF_WD_CONTEXT_NODE,
      l_element                 type ref to IF_WD_CONTEXT_ELEMENT,
      l_my_string               type string.

l_node     = wd_context->get_child_node( wd_this->wdctx_node_1 ).

l_element  = l_node->get_element( ).
l_element->get_attribute
            ( exporting name = 'ATTR1_1' importing value = l_my_string ).
. . . . . .

endmethod.
```

A reference to the required node element is generated using the GET_ELEMENT method of the IF_WD_CONTEXT_NODE interface. In the next step, the value of the Attr1_1 attribute of this element is passed to the `l_my_string` local variable.

Because the names of attributes are unique within a context node, you can achieve the same result without the reference to the node element in question.

```
method EXAMPLE .

data: l_node              type ref to IF_WD_CONTEXT_NODE,
      l_my_string             type string.

l_node = wd_context->get_child_node( wd_this->wdctx_node_1 ).

l_node->get_attribute
              ( exporting name = 'ATTR1_1' importing value = l_my_string ).

. . . . . .

endmethod.
```

In relation to the context schema above, this means the following:

Depending on the current position of the lead selection, the `l_my_string` variable has the value *string_11*, *string_21,* or *string_31.*

Reading an Attribute Value from Any Element

In addition to accessing the attribute value of the lead selection element, you can also access any specific element of the runtime context. For this reason, there is an index for the list of elements (the elements of the context node have a defined order). This sequence is adopted/created when the runtime context is built. For example, if a context node is filled from an internal table, the sequence of the elements in the context matches the sequence in the internal table.

To read from an element other than the lead selection element, the GET_ELEMENT method must be passed a value for the INDEX parameter.

```
method EXAMPLE .

data: l_node               type ref to IF_WD_CONTEXT_NODE,
      l_element            type ref to IF_WD_CONTEXT_ELEMENT,
      l_my_string          type string.

l_node     = wd_context->get_child_node( wd_this->wdctx_node_1 ).
l_element  = l_node->get_element( 2 ).
l_element->get_attribute
             ( exporting name = 'ATTR1_1' importing value = l_my_string ).

. . . . . .

endmethod.
```

In this case, a reference to the second element of the context node is generated. The value of the read attribute is therefore ***string_21***.

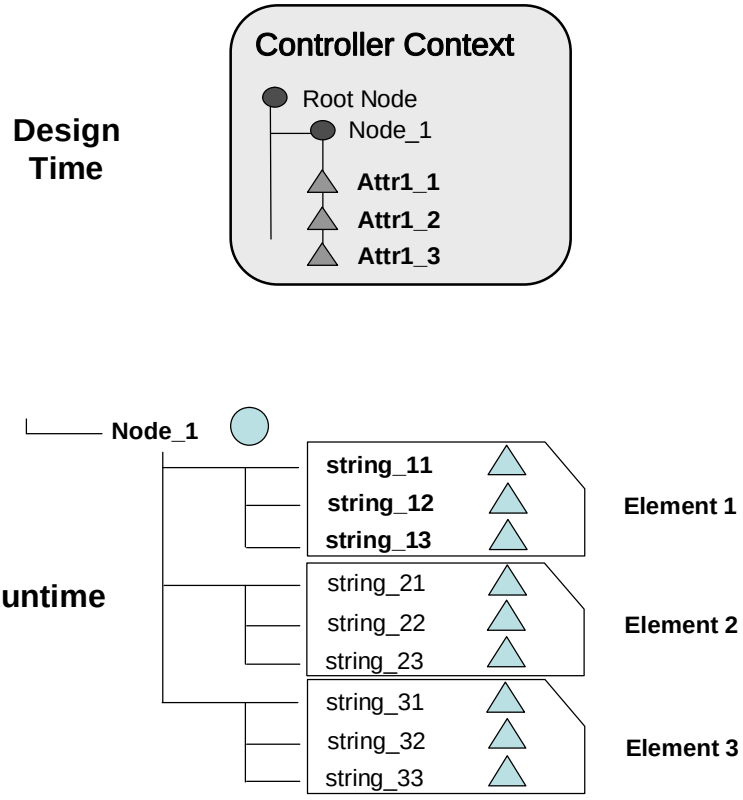
Even in the case of the shorter variants that do not use the element reference, the GET_ATTRIBUTE method can be passed a value for the INDEX parameter.

```
method EXAMPLE .

data: l_node               type ref to IF_WD_CONTEXT_NODE,
      l_my_string          type string.

l_node     = wd_context->get_child_node( wd_this->wdctx_node_1 ).

l_node->get_attribute
           ( exporting index = 2 exporting name = 'ATTR1_1' importing
value = l_my_string ).

. . . . . .

endmethod.
```

In this case, too, the local variable will have the value of the Attr1_1 attribute of the second element (***string_21***).

Reading the Structure from a Context Element

The attribute set for a context node can be structured in different ways. It is common to use an ABAP Dictionary structure. A list of attributes compiled manually at design time also constitutes a structure. Attributes assigned at design time are called **static attributes** (in contrast to attributes that are dynamically added to a context node at runtime). In our simple example, the structure consists of the three static attributes Attr1_1, Attr1_2, and Attr1_3.



Reading the Structure of the Lead Selection Element

A variable of the type of an element of the node is required to read this structure in its entirety. This local variable (`l_my_struc` in the source code example below), is easiest to declare using the context type definition that is defined for each context node in the IF_WD_MY_CONTROLLER interface.

You can view the IF_WD_<controller_name> interface of the controller that you are currently processing in the ABAP Workbench. To do so, click the following icon:

Display Controller Interface

```
method EXAMPLE .

data: l_node              type ref to IF_WD_CONTEXT_NODE,
      l_my_struc          type ref to IF_MY_CONTROLLER=>element_node_1.


l_node = wd_context->get_child_node( wd_this->wdctx_node_1 ).
l_node->get_static_attributes( importing static_attributes = l_my_struc ).
. . . . . .

endmethod.
```

The GET_STATIC_ATTRIBUTES method has the `static_attributes` formal parameter. Its value is passed to the `l_my_struc` local variable of the EXAMPLE method.

In principle, it is also possible to declare the type of the local variable in a different way than directly using the type definition of the node being read from. In this case, a MOVE_CORRESPONDING statement is executed automatically at runtime (see ABAP Language Reference).

Reading from the Structure of Any Element

Analog to reading an attribute value from any node element, it is also possible to read a structure of any element. The GET_STATIC_ATTRIBUTES method also has the INDEX parameter.

```
method EXAMPLE .

data: l_node              type ref to IF_WD_CONTEXT_NODE,
      l_my_struc          type ref to IF_MY_CONTROLLER=>element_node_1.


l_node = wd_context->get_child_node( wd_this->wdctx_node_1 ).

l_node->get_static_attributes( exporting index = 2 importing
static_attributes = l_my_struc ).
. . . . . .

endmethod.
```

Analog to reading an attribute value, it is possible to navigate to a specific element of the node before reading the structure.

```
method EXAMPLE .

data: l_node              type ref to IF_WD_CONTEXT_NODE,
      l_element           type ref to IF_WD_CONTEXT_ELEMENT,
      l_my_struc          type ref to IF_MY_CONTROLLER=>element_node_1.

l_node     = wd_context->get_child_node( wd_this->wdctx_node_1 ).
l_element  = l_node->get_element( 2 ).

l_element->get_static_attributes( importing static_attributes = l_my_struc
).
. . . . . .

endmethod.
```

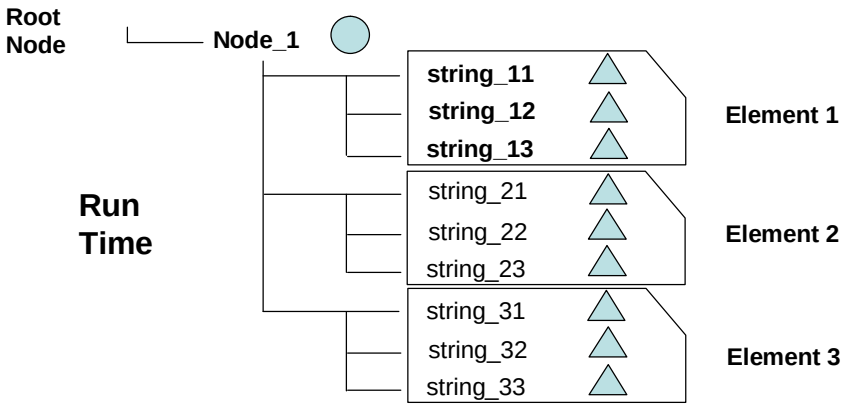
Writing Data to Existing Context Elements

The two previous sections described the procedure for reading the current values of context elements to a controller method. These values can now be processed within the method. The following information describes the opposite procedure - passing values from the method to a context element. There are two different situations, as described below.

Passing Values to an Existing Element

In this case, the element whose values are to be set already exists. The individual context attributes are initialized for this element and can contain a value. The SET_ATTRIBUTE and SET_STATIC_ATTRIBUTES methods of the IF_WD_CONTEXT_NODE and IF_WD_CONTEXT_ELEMENT interfaces can be used to set or overwrite the values of the element. In this case, the following still applies: If the INDEX parameter is not used, the lead selection element is automatically addressed (more information: **Reading the Attribute Value of a Context Element**).

In our example, several values of element 1 in the EXAMPLE method are to be set. Element 1 already exists at runtime.



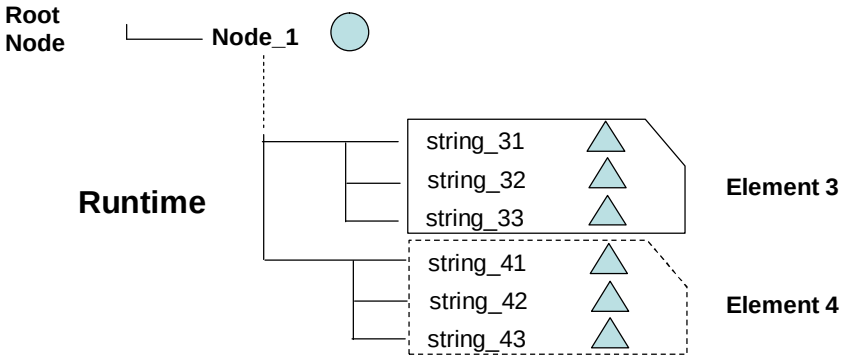

```
method EXAMPLE .

data: l_node               type ref to IF_WD_CONTEXT_NODE,
      l_my_struc               type ref to IF_MY_CONTROLLER=>element_node_1.

. . . . . .
l_node = wd_context->get_child_node( wd_this->wdctx_node_1 ).
l_node->set_static_attributes( static_attributes = l_my_struc ).
. . . . . .

endmethod.
```

The method call is analog to that for the read procedure. Since the pass direction for the value of the local variable is reversed in this case (from the EXAMPLE method to the context), the formal parameter of the GET_STATIC_ATTRIBUTES method is an importing parameter here. The SET_ATTRIBUTE method is used to set a single attribute value.

Writing Data to New Context Elements

In many cases, an additional node element is needed to store further data. This element must be generated before the attributes can be given values.

The graphic below shows an example schema that requires a fourth context element and a corresponding data record.



In principle, three steps are required to do this:

1. Creating the new element
2. Setting the values of the corresponding attributes
3. Binding the new element into the node hierarchy

```
method EXAMPLE .

data: l_node            type ref to IF_WD_CONTEXT_NODE,
      l_new_element               type ref to IF_WD_CONTEXT_ELEMENT,
      l_my_struc                  type ref to
IF_MY_CONTROLLER=>element_node_1.

. . . . . .
l_node = wd_context->get_child_node( wd_this->wdctx_node_1 ).

l_new_element     = l_node->create_element( ).
l_new_element->set_static_attributes( static_attributes = l_my_struc ).
l_node->bind_element( l_new_element ).
. . . . . .

endmethod.
```

The BIND_ELEMENT method has an INDEX importing parameter. This means that there is an opportunity here to define the position of the new element in the runtime context. If no value is passed in this position, the new element is automatically sorted into a position following the last element prior to the addition.

BIND_STRUCTURE and BIND_TABLE Methods

The BIND_STRUCTURE method can be used to merge this way of generating a new context element into one step.

```
method EXAMPLE .

data: l_node              type ref to IF_WD_CONTEXT_NODE,
      l_my_struc          type ref to IF_MY_CONTROLLER=>element_node_1.

. . . . . .
l_node = wd_context->get_child_node( wd_this->wdctx_node_1 ).

l_node->bind_structre( new_item = l_my_struc ).
. . . . . .

endmethod.
```

The BIND_STRUCTURE method generates a new context element, sets the values of the attributes with the values of the `l_my_struc` local variable, and binds the new element into the runtime context.

If several new elements are to be bound into a runtime context at the same time, and you would therefore need to call the BIND_STRUCTURE method several times in a row, you can bind a set of elements with a single call instead. A set of context elements corresponds to an internal table in ABAP Objects. The method for binding in several new context elements at the same time is therefore called BIND_TABLE.

```
method EXAMPLE .

data: l_node            type ref to IF_WD_CONTEXT_NODE,
      l_my_table                type ref to
IF_MY_CONTROLLER=>elements_node_1.

. . . . . .
l_node = wd_context->get_child_node( wd_this->wdctx_node_1 ).

l_node->bind_table( new_items = l_my_table ).
. . . . . .

endmethod.
```

 In this case, the local variable has the type `IF_MY_CONTROLLER=>elements_node_1` and the formal parameter is called NEW_ITEM**S**.

All BIND methods have the **SET_INITIAL_ELEMENTS** parameter. Its value is preset to ***true***. As a result, all elements of the context node that are available when the BIND method is called are deleted. After the call, the context only has those elements that were generated and inserted by the BIND method. However, if the context elements to be generated are to be added to the existing elements, the SET_INITIAL_ELEMENTS must be passed with the value *false.*

```
. . . . . .

l_node->bind_table( set_initial_elements = false new_items = l_my_table ).
. . . . . .
```

In this case, the elements are placed in one of the following positions:

- At the beginning (if the index value is not passed)
- At the position defined by the index value

 The use of the two BIND methods results in an optimization of element generation at runtime in comparison with multiple use in accordance with the detailed programming example at the beginning of this section.