Dynamic Management of Any Values 9

An **any** can be passed to a program that doesn't have any static information for the type of the **any** (code generated for the type by an IDL compiler has not been compiled with the object implementation). As a result, the object receiving the **any** does not have a portable method of using it.

The facility presented here enables traversal of the data value associated with an **any** at runtime and extraction of the primitive constituents of the data value. This is especially helpful for writing powerful generic servers (bridges, event channels supporting filtering, etc.).

Similarly, this facility enables the construction of an **any** at runtime, without having static knowledge of its type. This is especially helpful for writing generic clients (bridges, browsers, debuggers, user interface tools, etc.).

Contents

This chapter contains the following sections.

9.1 Overview

Comment: **Editorial instructions: The old DynAny interface published with CORBA 2.2 is deprecated. All references to the old DynAny-related types and interfaces should be removed from the core (Chapter 4) and be replaced with a comment stating that the DynamicAny module takes their place. This affects the following types and definitions in the CORBA module:**

Comment: **DynAny, DynStruct, DynSequence, DynArray, DynUnion, DynEnum, DynFixed, DynValue, create_dyn_any, create_basic_dyn_any, create_dyn_struct, create_dyn_sequence, create_dyn_array, create_dyn_union, create_dyn_enum, create_dyn_fixed, InconsistentTypeCode, and the remark about the create_dyn_* operations on page 4-7.**

Comment: **Add DynAnyFactory to the list of reserved ObjectIds on page 4-19.**

Unless explicitly stated otherwise, all IDL presented in section 9.1 through section 9.3 is part of the **DynamicAny** module.

Any values can be dynamically interpreted (traversed) and constructed through **DynAny** objects. A **DynAny** object is associated with a data value which corresponds to a copy of the value inserted into an **any**.

A **DynAny** object may be viewed as an ordered collection of component **DynAny**s. For **DynAny**s representing a basic type, such as **long**, or a type without components, such as an empty exception, the ordered collection of components is empty. Each **DynAny** object maintains the notion of a current position into its collection of component **DynAny**s. The current position is identified by an index value that runs from 0 to n–1, where n is the number of components. The special index value –1 indicates a current position that points nowhere. For values that cannot have a current position (such as an empty exception), the index value is fixed at –1. If a **DynAny** is initialized with a value that has components, the index is initialized to 0. After creation of an uninitialized **DynAny** (that is, a **DynAny** that has no value but a **TypeCode** that permits components), the current position depends on the type of value represented by the **DynAny**. (The current position is set to 0 or –1, depending on whether the new **DynAny** gets default values for its components.)

The iteration operations **rewind**, **seek**, and **next** can be used to change the current position and the **current_component** operation returns the component at the current position. The **component_count** operation returns the number of components of a **DynAny**. Collectively, these operations enable iteration over the components of a **DynAny**, for example, to (recursively) examine its contents.

Comment: **Replaced notion of a "buffer" with the concept of an ordered collection (Issue 1117). Added the notion of a -1 index to solve the problem of what to do with values that do not have components (Issue 1670). Added description of component_count (Issue 1670).**

A constructed **DynAny** object is a **DynAny** object associated with a constructed type. There is a different interface, inheriting from the **DynAny** interface, associated with each kind of constructed type in IDL (fixed, enum, struct, sequence, union, array, exception, and valuetype).

Comment: **Added fixed, enum, exception, and valuetype to this list because they were missing.**

A constructed **DynAny** object exports operations that enable the creation of new **DynAny** objects, each of them associated with a component of the constructed data value.

As an example, a **DynStruct** is associated with a struct value. This means that the **DynStruct** may be seen as owning an ordered collection of components, one for each structure member. The **DynStruct** object exports operations that enable the creation of new **DynAny** objects, each of them associated with a member of the struct.

Comment: **Eliminated notion of "buffer" (Issue 1117).**

If a **DynAny** object has been obtained from another (constructed) **DynAny** object, such as a **DynAny** representing a structure member that was created from a **DynStruct**, the member **DynAny** is logically contained in the **DynStruct**.

Comment: **Eliminated notion of "buffer" (Issue 1117).**

Destroying a top-level **DynAny** object (one that was not obtained as a component of another **DynAny**) also destroys any component **DynAny** objects obtained from it. Destroying a non-top level **DynAny** object does nothing. Invoking operations on a destroyed top-level **DynAny** or any of its descendants raises **OBJECT_NOT_EXIST**. Note that simply releasing all references to a **DynAny** object does not delete the **DynAny** or components; each **DynAny** created with one of the create operations or with the **copy** operation must be explicitly destroyed to avoid memory leaks.

Comment: **Eliminated notion of "buffer" (Issue 1117). Specified behavior for destruction of components (Issue 1644).**

If the programmer wants to destroy a **DynAny** object but still wants to manipulate some component of the data value associated with it, then he or she should first create a **DynAny** for the component and, after that, make a copy of the created **DynAny** object.

The behavior of **DynAny** objects has been defined in order to enable efficient implementations in terms of allocated memory space and speed of access. **DynAny** objects are intended to be used for traversing values extracted from **any**s or constructing values of **any**s at runtime. Their use for other purposes is not recommended.

9.2 DynAny API

The **DynAny** API comprises the following IDL definitions, located in the **DynamicAny** module:

```
// IDL
// File: DynamicAny.idl
#ifndef _DYNAMIC_ANY_IDL_
#define _DYNAMIC_ANY_IDL_
#pragma prefix “omg.org”
#include <orb.idl>

module DynamicAny {

    interface DynAny {
```

Comment: **Deleted the Invalid exception here. It was used only by assign(), from_any(), and to_any(). For these, it did not distinguish between type mismatch and not initialized error conditions, effectively mangling both error conditions into a single exception. For consistency with the resolution of Issue 654, assign(), and from_any() now raise TypeMismatch and InvalidValue, and to_any() raises InvalidValue.**

```
        exception InvalidValue {};
        exception TypeMismatch {};
```

Comment: **Deleted definition of OctetSeq here (Issue 1652). It was defined twice, once in the CORBA scope and once in the CORBA::DynAny scope. Because DynAny itself does not use OctetSeq, and because supporting type definitions for other DynAny types, such as DynStruct, also appear in the CORBA scope, deleting the definition here was the cleanest fix.**

Comment: **Deleted InvalidSeq exception because it was redundant. InvalidValue can be used just as well because there were no operations that could raise both InvalidValue and InvalidSeq.**

```
        CORBA::TypeCode type();

        void assign(in DynAny dyn_any) raises(TypeMismatch);
        void from_any(in any value) raises(TypeMismatch, InvalidValue);
```

```
any to_any();

boolean equal(in DynAny dyn_any);

void destroy();
DynAny copy();

void insert_boolean(in boolean value)
    raises(TypeMismatch, InvalidValue);
void insert_octet(in octet value)
    raises(TypeMismatch, InvalidValue);
void insert_char(in char value)
    raises(TypeMismatch, InvalidValue);
void insert_short(in short value)
    raises(TypeMismatch, InvalidValue);
void insert_ushort(in unsigned short value)
    raises(TypeMismatch, InvalidValue);
void insert_long(in long value)
    raises(TypeMismatch, InvalidValue);
void insert_ulong(in unsigned long value)
    raises(TypeMismatch, InvalidValue);
void insert_float(in float value)
    raises(TypeMismatch, InvalidValue);
void insert_double(in double value)
    raises(TypeMismatch, InvalidValue);
void insert_string(in string value)
    raises(TypeMismatch, InvalidValue);
void insert_reference(in Object value)
    raises(TypeMismatch, InvalidValue);
void insert_typecode(in CORBA::TypeCode value)
    raises(TypeMismatch, InvalidValue);
void insert_longlong(in long long value)
    raises(TypeMismatch, InvalidValue);
void insert_ulonglong(in unsigned long long value)
    raises(TypeMismatch, InvalidValue);
void insert_longdouble(in long double value)
    raises(TypeMismatch, InvalidValue);
void insert_wchar(in wchar value)
    raises(TypeMismatch, InvalidValue);
void insert_wstring(in wstring value)
    raises(TypeMismatch, InvalidValue);
void insert_any(in any value)
    raises(TypeMismatch, InvalidValue);
void insert_dyn_any(in DynAny value)
    raises(TypeMismatch, InvalidValue);
void insert_val(in ValueBase value)
    raises(TypeMismatch, InvalidValue);

boolean get_boolean()
    raises(TypeMismatch, InvalidValue);
octet get_octet()
```

```
        raises(TypeMismatch, InvalidValue);
    char get_char()
        raises(TypeMismatch, InvalidValue);
    short get_short()
        raises(TypeMismatch, InvalidValue);
    unsigned short get_ushort()
        raises(TypeMismatch, InvalidValue);
    long get_long()
        raises(TypeMismatch, InvalidValue);
    unsigned long get_ulong()
        raises(TypeMismatch, InvalidValue);
    float get_float()
        raises(TypeMismatch, InvalidValue);
    double get_double()
        raises(TypeMismatch, InvalidValue);
    string get_string()
        raises(TypeMismatch, InvalidValue);
    Object get_reference()
        raises(TypeMismatch, InvalidValue);
    CORBA::TypeCode get_typecode()
        raises(TypeMismatch, InvalidValue);
    long long get_longlong()
        raises(TypeMismatch, InvalidValue);
    unsigned long long get_ulonglong()
        raises(TypeMismatch, InvalidValue);
    long double get_longdouble()
        raises(TypeMismatch, InvalidValue);
    wchar get_wchar()
        raises(TypeMismatch, InvalidValue);
    wstring get_wstring()
        raises(TypeMismatch, InvalidValue);
    any get_any()
        raises(TypeMismatch, InvalidValue);
    DynAny get_dyn_any()
        raises(TypeMismatch, InvalidValue);
    ValueBase get_val()
        raises(TypeMismatch, InvalidValue);
```

Comment: **Added the InvalidValue exception to all get operations (Issue 654). This exception is raised if a get operation is invoked on a DynAny that has the correct type code but a current position of –1.**

```
    boolean seek(in long index);
    void rewind();
    boolean next();
    unsigned long component_count() raises(TypeMismatch);
    DynAny current_component() raises(TypeMismatch);
};
```

```
interface DynFixed : DynAny {
    string get_value();
    boolean set_value(in string val) raises(TypeMismatch, InvalidValue);
};
```

Comment: **Replaced octet sequence with string because there is no way to generically pass a fixed type in IDL (Issue 1668, 1653). Changed return type of set_value to boolean to permit detection of loss of precision.**

```
interface DynEnum : DynAny {
    string get_as_string();
    void set_as_string(in string value) raises(InvalidValue);
    unsigned long get_as_ulong();
    void set_as_ulong() raises(InvalidValue);
};
```

Comment: **Changed attributes to operations (Issue 1119, 1675).**

```
typedef string FieldName;

struct NameValuePair {
    FieldName id;
    any value;
};
typedef sequence<NameValuePair> NameValuePairSeq;

struct NameDynAnyPair {
    FieldName id;
    DynAny value;
};
typedef sequence<NameDynAnyPair> NameDynAnyPairSeq;

interface DynStruct : DynAny {
    FieldName current_member_name()
        raises(TypeMismatch, InvalidValue);
    CORBA::TCKind current_member_kind()
        raises(TypeMismatch, InvalidValue);
    NameValuePairSeq get_members();
    void set_members(in NameValuePairSeq value)
        raises(TypeMismatch, InvalidValue);
    NameDynAnyPairSeq get_members_as_dyn_any();
    void set_members_as_dyn_any(in NameDynAnyPairSeq value)
        raises(TypeMismatch, InvalidValue);
};
```

Comment: **Added TypeMismatch exceptions to deal with empty exceptions. (Issue 1679)**

```
interface DynUnion : DynAny {
    DynAny get_discriminator();
    void set_discriminator(in DynAny d) raises(TypeMismatch);
    void set_to_default_member() raises(TypeMismatch);
    void set_to_no_active_member() raises(TypeMismatch);
    boolean has_no_active_member();
    CORBA::TCKind discriminator_kind();
    DynAny member() raises(InvalidValue);
    FieldName member_name() raises(InvalidValue);
    CORBA::TCKind member_kind() raises(InvalidValue);
};

typedef sequence<any> AnySeq;
typedef sequence<DynAny> DynAnySeq;

interface DynSequence : DynAny {
    unsigned long get_length();
    void set_length(in unsigned long len) raises(InvalidValue);
    AnySeq get_elements();
    void set_elements(in AnySeq value)
        raises(TypeMismatch, InvalidValue);
    DynAnySeq get_elements_as_dyn_any();
    void set_elements_as_dyn_any(in DynAnySeq value)
        raises(TypeMismatch, InvalidValue);
};
```

Comment: **Replaced attribute with operations (Issue 1119, 1675).**

```
interface DynArray : DynAny {
    AnySeq get_elements();
    void set_elements(in AnySeq value)
        raises(TypeMismatch, InvalidValue);
    DynAnySeq get_elements_as_dyn_any();
    void set_elements_as_dyn_any(in DynAnySeq value)
        raises(TypeMismatch, InvalidValue);
};

interface DynValue : DynAny {
    FieldName current_member_name()
        raises(TypeMismatch, InvalidValue);
    CORBA::TCKind current_member_kind()
        raises(TypeMismatch, InvalidValue);
    NameValuePairSeq get_members();
    void set_members(in NameValuePairSeq value)
        raises(TypeMismatch, InvalidValue);
    NameDynAnyPairSeq get_members_as_dyn_any();
    void set_members_as_dyn_any(in NameDynAnyPairSeq value)
        raises(TypeMismatch, InvalidValue);
};
```

```
interface DynAnyFactory {
    exception InconsistentTypeCode {};
    DynAny create_dyn_any(in any value)
        raises(InconsistentTypeCode);
    DynAny
        create_dyn_any_from_type_code(in CORBA::TypeCode type)
        raises(InconsistentTypeCode);
};

}; // module DynamicAny

#endif // _DYNAMIC_ANY_IDL_
```

Comment: **Added DynValue interface here because it was missing.**

9.2.1 Locality and usage constraints

DynAny and **DynAnyFactory** objects are intended to be local to the process in which they are created and used. This means that references to **DynAny** and **DynAnyFactory** objects cannot be exported to other processes, or externalized with **ORB::object_to_string**. If any attempt is made to do so, the offending operation will raise a **MARSHAL** system exception.

Since their interfaces are specified in IDL, **DynAny** objects export operations defined in the standard **CORBA::Object** interface. However, any attempt to invoke operations exported through the **Object** interface may raise the standard **NO_IMPLEMENT** exception.

An attempt to use a **DynAny** object with the DII may raise the **NO_IMPLEMENT** exception.

9.2.2 Creating a DynAny object

A **DynAny** object can be created as a result of:

- invoking an operation on an existing **DynAny** object
- invoking an operation on a **DynFactory** object.

A constructed **DynAny** object supports operations that enable the creation of new **DynAny** objects encapsulating access to the value of some constituent. **DynAny** objects also support the **copy** operation for creating new **DynAny** objects.

In addition, **DynAny** objects can be created by invoking operations on the **DynFactory** object. A reference to the **DynFactory** object is obtained by calling **CORBA::ORB::resolve_initial_references("DynFactory")**.

```
interface DynAnyFactory {
    exception InconsistentTypeCode {};
    DynAny create_dyn_any(in any value)
        raises(InconsistentTypeCode);
```

```
        DynAny create_dyn_any_from_type_code(in CORBA::TypeCode type)
            raises(InconsistentTypeCode);
};
```

The **create_dyn_any** operation creates a new **DynAny** object from an **any** value. A copy of the **TypeCode** associated with the **any** value is assigned to the resulting **DynAny** object. The value associated with the **DynAny** object is a copy of the value in the original any. The **create_dyn_any** operation sets the current position of the created **DynAny** to zero if the passed value has components; otherwise, the current position is set to −1. The operation raises **InconsistentTypeCode** if **value** has a **TypeCode** with a **TCKind** of **tk_Principal**, **tk_native**, or **tk_abstract_interface**.

Comment: **Added InconsistentTypeCode to create_dyn_any to deal with initializers that do not make sense for DynAny.**

The **create_dyn_any_from_type_code** operation creates a **DynAny** from a **TypeCode**. Depending on the **TypeCode**, the created object may be of type **DynAny**, or one of its derived types, such as **DynStruct**. The returned reference can be narrowed to the derived type.

In all cases, a **DynAny** constructed from a **TypeCode** has an initial default value. The default values of basic types are:

- **FALSE** for **Boolean**
- zero for numeric types
- zero for types **octet**, **char**, and **wchar**
- the empty string for **string** and **wstring**
- nil for object references
- a type code with a **TCKind** value of **tk_null** for type codes
- for **any** values, an **any** containing a type code with a **TCKind** value of **tk_null** type and no value

For complex types, creation of the corresponding **DynAny** assigns a default value as follows:

- For **DynSequence**, the operation sets the current position to −1 and creates an empty sequence.
- For **DynEnum**, the operation sets the current position to −1 and sets the value of the enumerator to the first enumerator value indicated by the **TypeCode**.
- For **DynFixed**, operations set the current position to −1 and sets the value zero.
- For **DynStruct**, the operation sets the current position to −1 for empty exceptions and to zero for all other **TypeCode**s. The members (if any) are (recursively) initialized to their default values.
- For **DynArray**, the operation sets the current position to zero and (recursively) initializes elements to their default value.

- For **DynUnion**, the operation sets the current position to zero. The discriminator value is set to a value consistent with the first named member of the union. That member is activated and (recursively) initialized to its default value.
- For **DynValue**, the members are initialized as for **DynStruct**.

Comment: **Specified semantics of create operations with respect to current position and initial values (Issue 1144, 1648, 1649).**

Dynamic interpretation of an **any** usually involves creating a **DynAny** object using **DynAnyFactory::create_dyn_any** as the first step. Depending on the type of the **any**, the resulting **DynAny** object reference can be narrowed to a **DynFixed**, **DynStruct**, **DynSequence**, **DynArray**, **DynUnion**, **DynEnum**, or **DynValue** object reference.

Comment: **Added DynFixed and DynValue to this list because they were missing.**

Dynamic creation of an **any** involves creating a **DynAny** object using **DynAnyFactory::create_dyn_any_from_type_code**, passing the **TypeCode** associated with the value to be created. The returned reference is narrowed to one of the complex types, such as **DynStruct**, if appropriate. Then, the value can be initialized by means of invoking operations on the resulting object. Finally, the **to_any** operation can be invoked to create an **any** value from the constructed **DynAny**.

9.2.3 The DynAny interface

The following operations can be applied to a **DynAny** object:

- Obtaining the **TypeCode** associated with the **DynAny** object
- Generating an **any** value from the **DynAny** object
- Comparing two **DynAny** objects for equality
- Destroying the **DynAny** object
- Creating a **DynAny** object as a copy of the **DynAny** object
- Inserting/getting a value of some basic type into/from the **DynAny** object
- Iterating through the components of a **DynAny**
- Initializing a **DynAny** object from another **DynAny** object
- Initializing a **DynAny** object from an **any** value

9.2.3.1 Obtaining the TypeCode associated with a DynAny object

CORBA::TypeCode type();

A **DynAny** object is created with a **TypeCode** value assigned to it. This **TypeCode** value determines the type of the value handled through the **DynAny** object. The **type** operation returns the **TypeCode** associated with a **DynAny** object.

Note that the **TypeCode** associated with a **DynAny** object is initialized at the time the **DynAny** is created and cannot be changed during lifetime of the **DynAny** object.

9.2.3.2 Initializing a DynAny object from another DynAny object

void assign(in DynAny dyn_any) raises(TypeMismatch);

The **assign** operation initializes the value associated with a **DynAny** object with the value associated with another **DynAny** object.

If the type of the passed **DynAny** is not equivalent to the type of target **DynAny**, the operation raises **TypeMismatch**. The current position of the target **DynAny** is set to zero for values that have components and to −1 for values that do not have components.

Comment: **Issue 654, 1144.**

9.2.3.3 Initializing a DynAny object from an any value

void from_any(in any value) raises(TypeMismatch, InvalidValue);

The **from_any** operation initializes the value associated with a **DynAny** object with the value contained in an **any**.

If the type of the passed **Any** is not equivalent to the type of target **DynAny**, the operation raises **TypeMismatch**. If the passed **Any** does not contain a legal value (such as a null string), the operation raises **InvalidValue**. The current position of the target **DynAny** is set to zero for values that have components and to −1 for values that do not have components.

Comment: **Issue 654, 1144.**

9.2.3.4 Generating an any value from a DynAny object

any to_any();

The **to_any** operation creates an **any** value from a **DynAny** object. A copy of the **TypeCode** associated with the **DynAny** object is assigned to the resulting **any**. The value associated with the **DynAny** object is copied into the **any**.

9.2.3.5 *Comparing DynAny values*

boolean equal(in DynAny dyn_any);

The **equal** operation compares two **DynAny** values for equality and returns true of the **DynAny**s are equal, false otherwise. Two **DynAny** values are equal if their **TypeCodes** are equivalent and, recursively, all component **DynAnys** have equal values. The current position of the two **DynAny**s being compared has no effect on the result of **equal**.

Comment: **Issue 1972.**

9.2.3.6 *Destroying a DynAny object*

void destroy();

The **destroy** operation destroys a **DynAny** object. This operation frees any resources used to represent the data value associated with a **DynAny** object. **destroy** must be invoked on references obtained from one of the creation operations on the **ORB** interface or on a reference returned by **DynAny::copy** to avoid resource leaks. Invoking **destroy** on component **DynAny** objects (for example, on objects returned by the **current_component** operation) does nothing.

Comment: **Specified that destroy need be invoked only on top-level objects and is a no-op on component DynAnys to deal with the problem of what should happen if I call destroy on a DynAny in the middle of a containment hierarchy. (Issue 1644).**

Destruction of a **DynAny** object implies destruction of all **DynAny** objects obtained from it. That is, references to components of a destroyed **DynAny** become invalid; invocations on such references raise **OBJECT_NOT_EXIST**.

Comment: **Added clarification to make it clear that component references dangle once the owning (parent) DynAny is destroyed.**

Comment: **Deleted one para that said something vague and non-normative about taking care with the representation of values.**

It is possible to manipulate a component of a **DynAny** beyond the life time of the **DynAny** from which the component was obtained by making a copy of the component with the **copy** operation before destroying the **DynAny** from which the component was obtained.

Comment: **Reworded this para to get rid of awkward wording.**

9.2.3.7 Creating a copy of a DynAny object

```
DynAny copy();
```

The **copy** operation creates a new **DynAny** object whose value is a deep copy of the **DynAny** on which it is invoked. The operation is polymorphic, that is, invoking it on one of the types derived from **DynAny**, such as **DynStruct**, creates the derived type but returns its reference as the **DynAny** base type.

Comment: **Made it clear that copy is a polymorphic clone operation.**

9.2.3.8 Accessing a value of some basic type in a DynAny object

The insert and get operations enable insertion/extraction of basic data type values into/from a **DynAny** object.

Both bounded and unbounded strings are inserted using **insert_string** and **insert_wstring**. These operations raise the **InvalidValue** exception if the string inserted is longer than the bound of a bounded string.

Comment: **Issue 1639.**

Calling an insert or get operation on a **DynAny** that has components but has a current position of −1 raises **InvalidValue**.

Comment: **Issue 1147.**

Get operations raise **TypeMismatch** if the accessed component in the **DynAny** is of a type that is not equivalent to the requested type. (Note that **get_string** and **get_wstring** are used for both unbounded and bounded strings.)

Comment: **Made it clear that bounded strings use insert_(w)string and get_(w)string. (Issue 1639)**

A type is consistent for inserting or extracting a value if its **TypeCode** is equivalent to the **TypeCode** contained in the **DynAny** or, if the **DynAny** has components, is equivalent to the **TypeCode** of the **DynAny** at the current position.

The **get_dyn_any** and **insert_dyn_any** operations are provided to deal with **any** values that contain another **any**.

Calling an insert or get operation leaves the current position unchanged.

These operations are necessary to handle basic **DynAny** objects but are also helpful to handle constructed **DynAny** objects. Inserting a basic data type value into a constructed **DynAny** object implies initializing the current component of the constructed data value associated with the **DynAny** object. For example, invoking **insert_boolean** on a

DynStruct implies inserting a boolean data value at the current position of the associated struct data value. If `dyn_construct` points to a constructed **DynAny** object, then:

```
result = dyn_construct->get_boolean();
```

has the same effect as:

```
DynamicAny::DynAny_var temp =
   dyn_construct->current_component();
result = temp->get_boolean();
```

Calling an insert or get operation on a **DynAny** whose current component itself has components raises **TypeMismatch**.

Comment: **Deleted call to next() here because calling a get operation does not advance the current position.**

In addition, availability of these operations enable the traversal of **any**s associated with sequences of basic data types without the need to generate a **DynAny** object for each element in the sequence.

9.2.3.9 *Iterating through components of a DynAny*

The **DynAny** interface allows a client to iterate through the components of the values pointed to by **DynStruct**, **DynSequence**, **DynArray**, **DynUnion**, **DynAny**, and **DynValue** objects.

As mentioned previously, a **DynAny** object may be seen as an ordered collection of components, together with a current position.

boolean seek(in long index);

The **seek** operation sets the current position to **index**. The current position is indexed 0 to n–1, that is, index zero corresponds to the first component. The operation returns true if the resulting current position indicates a component of the **DynAny** and false if **index** indicates a position that does not correspond to a component.

Calling **seek** with a negative index is legal. It sets the current position to –1 to indicate no component and returns false. Passing a non-negative index value for a **DynAny** that does not have a component at the corresponding position sets the current position to –1 and returns false.

Comment: **Cleaned up semantics to fully define the behavior under various boundary conditions. (Issue 1649).**

void rewind();

The **rewind** operation is equivalent to calling **seek(0);**

boolean next();

The **next** operation advances the current position to the next component. The operation returns true while the resulting current position indicates a component, false otherwise. A false return value leaves the current position at –1. Invoking **next** on a **DynAny** without components leaves the current position at –1 and returns false.

Comment: **Defined current position after next returns false. (Issue 1648).**

unsigned long component_count() raises(TypeMismatch);

The **component_count** operation returns the number of components of a **DynAny**. For a **DynAny** without components, it returns zero. The operation only counts the components at the top level. For example, if **component_count** is invoked on a **DynStruct** with a single member, the return value is 1, irrespective of the type of the member.

For sequences, the operation returns the current number of elements. For structures, exceptions, and valuetypes, the operation returns the number of members. For arrays, the operation returns the number of elements. For unions, the operation returns 2 if the discriminator indicates that a named member is active; otherwise, it returns 1. For **DynFixed** and **DynEnum**, the operation returns zero.

Comment: **Added component_count() because otherwise, there is no way to find out how many components are in a DynAny, other than to iterate through to the end or randomly calling seek() until I've found the place at which it switches from true to false. (Issue 1670).**

DynAny current_component() raises(TypeMismatch);

The **current_component** operation returns the **DynAny** for the component at the current position. It does not advance the current position, so repeated calls to **current_component** without an intervening call to **rewind**, **next**, or **seek** return the same component.

The returned **DynAny** object reference can be used to get/set the value of the current component. If the current component represents a complex type, the returned reference can be narrowed based on the **TypeCode** to get the interface corresponding to the to the complex type.

Calling **current_component** on a **DynAny** that cannot have components, such as a **DynEnum** or an empty exception, raises **TypeMismatch**. Calling **current_component** on a **DynAny** whose current position is –1 returns a nil reference.

Comment: **Deleted one para here that mentioned IDL identifiers "member_type" and "component_type", none of which exist in the IDL.**

The iteration operations, together with **current_component**, can be used to dynamically compose an **any** value. After creating a dynamic any, such as a **DynStruct**, **current_component** and **next** can be used to initialize all the components of the value. Once the dynamic value is completely initialized, **to_any** creates the corresponding **any** value.

Comment: **Tightened semantics to indicate what happens for values that don't have components (Issue 1144, 1648, 1649). Added TypeMismatch to solve the problem of what to do if the operation is invoked on a DynAny without components. (Issue 1670). Specified nil return value if no component exists for the current position to make iteration easier. Tightened up last para to use clearer language.**

9.2.4 The DynFixed interface

DynFixed objects are associated with values of the IDL **fixed** type.

```
interface DynFixed : DynAny {
    string get_value();
    boolean set_value(in string val)
        raises (TypeMismatch, InvalidValue);
};
```

Because IDL does not have a generic type that can represent fixed types with arbitrary number of digits and arbitrary scale, the operations use the IDL **string** type.

The **get_value** operation returns the value of a **DynFixed**.

The **set_value** operation sets the value of the **DynFixed**. The **val** string must contain a **fixed** string constant in the same format as used for IDL fixed-point literals. However, the trailing **d** or **D** is optional. If **val** contains a value whose scale exceeds that of the **DynFixed** or is not initialized, the operation raises **InvalidValue**. The return value is true if **val** can be represented as the **DynFixed** without loss of precision. If **val** has more fractional digits than can be represented in the **DynFixed**, fractional digits are truncated and the return value is false. If **val** does not contain a valid fixed-point literal or contains extraneous characters other than leading or trailing white space, the operation raises **TypeMismatch**.

Comment: **Changed representation of fixed values from octet sequence to string. I cannot see another way of dealing with IDL's inability to represent different fixed-point types generically. (The previous solution using the CDR encoding for fixed was pragmatically useless to application programmers -- Issue 1653). Added TypeMismatch and InvalidValue exceptions to deal with error conditions. Changed return type of set_value to boolean to permit detection of loss of precision (Issue 1668).**

9.2.5 The DynEnum interface

DynEnum objects are associated with enumerated values.

```
interface DynEnum : DynAny {
    string get_as_string();
    void set_as_string(in string value)
        raises(InvalidValue);
    unsigned long get_as_ulong();
    void set_as_ulong()
        raises(InvalidValue);
};
```

The **get_as_string** operation returns the value of the **DynEnum** as an IDL identifier.

The **set_as_string** operation sets the value of the **DynEnum** to the enumerated value whose IDL identifier is passed in the **value** parameter. If **value** contains a string that is not a valid IDL identifier for the corresponding enumerated type, the operation raises **InvalidValue**.

The **get_as_ulong** operation returns the value of the **DynEnum** as the enumerated value's ordinal value. Enumerators have ordinal values 0 to n–1, as they appear from left to right in the corresponding IDL definition.

The **set_as_ulong** operation sets the value of the **DynEnum** as the enumerated value's ordinal value. If **value** contains a value that is outside the range of ordinal values for the corresponding enumerated type, the operation raises **InvalidValue**.

The current position of a **DynEnum** is always –1.

Comment: **Changed attributes to operations because otherwise, there is no way to report error conditions, such as "not initialized" and "impossible value" (1675). Tightened semantics to specify the ordinal values of enumerators (Issue 1119). Specified the current position (Issue 1144).**

9.2.6 The DynStruct interface

DynStruct objects are associated with struct values and exception values.

```
typedef string FieldName;

struct NameValuePair {
    FieldName id;
    any value;
};
typedef sequence<NameValuePair> NameValuePairSeq;

struct NameDynAnyPair {
    FieldName id;
    DynAny value;
};
typedef sequence<NameDynAnyPair> NameDynAnyPairSeq;
```

Comment: **Added types for DynAny. (Issue 1679)**

```
typedef sequence<NameValuePair> NameValuePairSeq;

interface DynStruct : DynAny {
    FieldName current_member_name()
        raises(TypeMismatch, InvalidValue);
    CORBA::TCKind current_member_kind()
        raises(TypeMismatch, InvalidValue);
    NameValuePairSeq get_members();
    void set_members(in NameValuePairSeq value)
        raises(TypeMismatch, InvalidValue);
    NameDynAnyPairSeq get_members_as_dyn_any();
    void set_members_as_dyn_any(in NameDynAnyPairSeq value)
        raises(TypeMismatch, InvalidValue);
};
```

```
    FieldName current_member_name()
        raises(TypeMismatch, InvalidValue);
```

The **current_member_name** operation returns the name of the member at the current position. If the **DynStruct** represents an empty exception, the operation raises **TypeMismatch**. If the current position does not indicate a member, the operation raises **InvalidValue**.

This operation may return an empty string since the **TypeCode** of the value being manipulated may not contain the names of members.

```
    CORBA::TCKind current_member_kind()
        raises(TypeMismatch, InvalidValue);
```

current_member_kind returns the **TCKind** associated with the member at the current position. If the **DynStruct** represents an empty exception, the operation raises **TypeMismatch**. If the current position does not indicate a member, the operation raises **InvalidValue**.

```
    NameValuePairSeq get_members();
```

The **get_members** operation returns a sequence of name/value pairs describing the name and the value of each member in the struct associated with a **DynStruct** object. The sequence contains members in the same order as the declaration order of members as indicated by the **DynStruct**'s **TypeCode**. The current position is not affected. The member names in the returned sequence will be empty strings if the **DynStruct**'s **TypeCode** does not contain member names.

```
    void set_members(in NameValuePairSeq value)
        raises(TypeMismatch, InvalidValue);
```

The **set_members** operation initializes the struct data value associated with a **DynStruct** object from a sequence of name value pairs. The operation sets the current position to zero if the passed sequences has non-zero length; otherwise, if an empty sequence is passed, the current position is set to –1.

Members must appear in the **NameValuePairSeq** in the order in which they appear in the IDL specification of the struct. If one or more sequence elements have a type that is not equivalent to the **TypeCode** of the corresponding member, the operation raises **TypeMismatch**. If the passed sequence has a number of elements that disagrees with the number of members as indicated by the **DynStruct**'s **TypeCode**, the operation raises **InvalidValue**.

If member names are supplied in the passed sequence, they must either match the corresponding member name in the **DynStruct**'s **TypeCode** or must be empty strings, otherwise, the operation raises **TypeMismatch**. Members must be supplied in the same order as indicated by the **DynStruct**'s **TypeCode**. (The operation makes no attempt to assign member values based on member names.)

The **get_members_as_dyn_any** and **set_members_as_dyn_any** operations have the same semantics as their **Any** counterparts, but accept and return values of type **DynAny** instead of **Any**.

DynStruct objects can also be used for handling exception values. In that case, members of the exceptions are handled in the same way as members of a struct.

Comment: **Added exceptions to deal with situations where there cannot be a member or where the current position does not indicate a member. (Issue 1679) Changed member sequence from Any to DynAny (Issue 1671). Clarified semantics of operations and added TypeMismatch exception to avoid mangling several error conditions into a single exception.**

9.2.7 The DynUnion interface

DynUnion objects are associated with unions.

```
interface DynUnion : DynAny {
    DynAny get_discriminator();
    void set_discriminator(in DynAny d)
        raises(TypeMismatch);
    void set_to_no_active_member()
        raises(TypeMismatch);
    boolean has_no_active_member()
        raises(InvalidValue);
    CORBA::TCKind discriminator_kind();
    DynAny member()
        raises(InvalidValue);
    FieldName member_name()
        raises(InvalidValue);
    CORBA::TCKind member_kind()
        raises(InvalidValue);
```

```
};
```

The **DynUnion** interface allows for the insertion/extraction of an OMG IDL union type into/from a **DynUnion** object.

A union can have only two valid current positions: zero, which denotes the discriminator, and one, which denotes the active member. The **component_count** value for a union depends on the current discriminator: it is 2 for a union whose discriminator indicates a named member, and 1 otherwise.

```
DynAny get_discriminator()
    raises(InvalidValue);
```

The **get_discriminator** operation returns the current discriminator value of the **DynUnion**.

```
void set_discriminator(in DynAny d)
    raises(TypeMismatch);
```

The **set_discriminator** operation sets the discriminator of the **DynUnion** to the specified value. If the **TypeCode** of the **d** parameter is not equivalent to the **TypeCode** of the union's discriminator, the operation raises **TypeMismatch**.

Setting the discriminator to a value that is consistent with the currently active union member does not affect the currently active member. Setting the discriminator to a value that is inconsistent with the currently active member deactivates the member and activates the member that is consistent with the new discriminator value (if there is a member for that value) by initializing the member to its default value.

Setting the discriminator of a union sets the current position to 0 if the discriminator value indicates a non-existent union member (**has_no_active_member** returns true in this case). Otherwise, if the discriminator value indicates a named union member, the current position is set to 1 (**has_no_active_member** returns false and **component_count** returns 2 in this case).

```
void set_to_default_member()
    raises(TypeMismatch);
```

The **set_to_default_member** operation sets the discriminator to a value that is consistent with the value of the **default** case of a union; it sets the current position to zero and causes **component_count** to return 2. Calling **set_to_default_member** on a union that does not have an explicit **default** case raises **TypeMismatch**.

```
void set_to_no_active_member()
    raises(TypeMismatch);
```

The **set_to_no_active_member** operation sets the discriminator to a value that does not correspond to any of the union's case labels; it sets the current position to zero and causes **component_count** to return 1. Calling **set_to_no_active_member** on a union that has an explicit **default** case or on a union that uses the entire range of discriminator values for explicit **case** labels raises **TypeMismatch**.

```
boolean has_no_active_member();
```

The **has_no_active_member** operation returns true if the union has no active member (that is, the union's value consists solely of its discriminator because the discriminator has a value that is not listed as an explicit **case** label). Calling this operation on a union that has a **default** case returns false. Calling this operation on a union that uses the entire range of discriminator values for explicit **case** labels returns false.

```
CORBA::TCKind discriminator_kind();
```

The **discriminator_kind** operation returns the **TCKind** value of the discriminator's **TypeCode**.

```
CORBA::TCKind member_kind()
    raises(InvalidValue);
```

The **member_kind** operation returns the **TCKind** value of the currently active member's **TypeCode**. Calling this operation on a union that does not have a currently active member raises **InvalidValue**.

```
DynAny member()
    raises(InvalidValue);
```

The **member** operation returns the currently active member. If the union has no active member, the operation raises **InvalidValue**. Note that the returned reference remains valid only for as long as the currently active member does not change. Using the returned reference beyond the life time of the currently active member raises **OBJECT_NOT_EXIST**.

```
FieldName member_name()
    raises(InvalidValue);
```

The **member_name** operation returns the name of the currently active member. If the union's **TypeCode** does not contain a member name for the currently active member, the operation returns an empty string. Calling **member_name** on a union without an active member raises **InvalidValue**.

```
CORBA::TCKind member_kind()
    raises(InvalidValue);
```

The **member_kind** operation returns the **TCKind** value of the **TypeCode** of the currently active member. If the union has no active member, the operation raises **InvalidValue**.

Comment: **Mostly rewrote the DynUnion interface because there were too many problems with it. Issues 747, 1120, 1157, 1158, 1159, 1675, 1974.**

9.2.8 The DynSequence interface

DynSequence objects are associated with sequences.

```
typedef sequence<any> AnySeq;
typedef sequence<DynAny> DynAnySeq;

interface DynSequence : DynAny {
    unsigned long get_length();
    void set_length(in unsigned long len)
        raises(InvalidValue);
    AnySeq get_elements();
    void set_elements(in AnySeq value)
        raises(TypeMismatch, InvalidValue);
    DynAnySeq get_elements_as_dyn_any();
    void set_elements_as_dyn_any(in DynAnySeq value)
        raises(TypeMismatch, InvalidValue);
};
```

```
    unsigned long get_length();
```

The **get_length** operation returns the current length of the sequence.

```
    void set_length(in unsigned long len)
        raises(TypeMismatch, InvalidValue);
```

The **set_length** operation sets the length of the sequence. Increasing the length of a sequence adds new elements at the tail without affecting the values of already existing elements. Newly added elements are default-initialized.

Increasing the length of a sequence sets the current position to the first newly-added element if the previous current position was –1. Otherwise, if the previous current position was not –1, the current position is not affected.

Increasing the length of a bounded sequence to a value larger than the bound raises **InvalidValue**.

Decreasing the length of a sequence removes elements from the tail without affecting the value of those elements that remain. The new current position after decreasing the length of a sequence is determined as follows:

- If the length of the sequence is set to zero, the current position is set to –1.
- If the current position is –1 before decreasing the length, it remains at –1.
- If the current position indicates a valid element and that element is not removed when the length is decreased, the current position remains unaffected.
- If the current position indicates a valid element and that element is removed, the current position is set to –1.

```
    DynAnySeq get_elements();
```

The **get_elements** operation returns the elements of the sequence.

```
void set_elements(in AnySeq value)
    raises(TypeMismatch, InvalidValue);
```

The **set_elements** operation sets the elements of a sequence. The length of the **DynSequence** is set to the length of **value**. The current position is set to zero if **value** has non-zero length and to –1 if **value** is a zero-length sequence.

If value contains one or more elements whose **TypeCode** is not equivalent to the element **TypeCode** of the **DynSequence**, the operation raises **TypeMismatch**. If the length of value exceeds the bound of a bounded sequence, the operation raises **InvalidValue**.

Comment: **Cleaned up semantics of length (Issue 660). Removed attribute and replaced with operations to permit proper exception handling (Issue 1119).**

The **get_elements_as_dyn_any** and **set_elements_as_dyn_any** operations have the same semantics, but accept and return values of type **DynAny** instead of **Any**.

9.2.9 The DynArray interface

DynArray objects are associated with arrays.

```
interface DynArray : DynAny {
    AnySeq get_elements();
    void set_elements(in AnySeq value)
        raises(TypeMismatch, InvalidValue);
    DynAnySeq get_elements_as_dyn_any();
    void set_elements_as_dyn_any(in DynAnySeq value)
        raises(TypeMismatch, InvalidValue);
};

    DynAnySeq get_elements();
```

The **get_elements** operation returns the elements of the **DynArray**.

```
    void set_elements(in DynAnySeq value)
        raises(TypeMismatch, InvalidValue);
```

Comment: **Added InvalidValue exception. It was missing here, even though it appeared in the interface IDL above. Added TypeMismatch exception because otherwise, I have no idea what is wrong (length of sequence or type of element).**

The **set_elements** operation sets the **DynArray** to contain the passed elements. If the sequence does not contain the same number of elements as the array dimension, the operation raises **InvalidValue**. If one or more elements have a type that is inconsistent with the **DynArray**'s **TypeCode**, the operation raises **TypeMismatch**.

The **get_elements_as_dyn_any** and **set_elements_as_dyn_any** operations have the same semantics as their **Any** counterparts, but accept and return values of type **DynAny** instead of **Any**.

Note that the dimension of the array is contained in the **TypeCode** which is accessible through the **type** attribute. It can also be obtained by calling the **component_count** operation.

9.2.10 The DynValue interface

DynValue objects are associated with value types.

```
interface DynValue : DynAny {
    FieldName current_member_name()
        raises(TypeMismatch, InvalidValue);
    CORBA::TCKind current_member_kind()
        raises(TypeMismatch, InvalidValue);
    NameValuePairSeq get_members();
    void set_members(in NameValuePairSeq value)
        raises(TypeMismatch, InvalidValue);
    NameDynAnyPairSeq get_members_as_dyn_any();
    void set_members_as_dyn_any(in NameDynAnyPairSeq value)
        raises(TypeMismatch, InvalidValue);
};
```

Operations on the **DynValue** interface have semantics as for **DynStruct**.

Comment: **Removed skipping of private members here because that doesn't help to preserve invariants at all. (There may be invariants that depend on both private and public members.) Other solutions are possible, such as preventing construction of a DynValue and access to its public members completely if there are private members in the value.**

9.3 Usage in C++ language

9.3.1 Dynamic creation of CORBA::Any values

9.3.1.1 Creating an any which contains a struct

Consider the following IDL definition:

```
// IDL
struct MyStruct {
    long member1;
    boolean member2;
};
```

The following example illustrates how a `CORBA::Any` value may be constructed on the fly containing a value of type **MyStruct**:

```
// C++
CORBA::ORB_var orb = ...;
DynamicAny::DynAnyFactory_var dafact
   = orb->resolve_initial_references("DynAnyFactory");
CORBA::StructMemberSeq mems(2);
CORBA::Any_var result;
CORBA::Long    value1 = 99;
CORBA::Boolean value2 = 1;
mems.length(2);
mems[0].name = CORBA::string_dup("member1");
mems[0].type = CORBA::TypeCode::_duplicate(CORBA::_tc_long);
mems[1].name = CORBA::string_dup("member2");
mems[1].type
   = CORBA::TypeCode::_duplicate(CORBA::_tc_boolean);

CORBA::TypeCode_var new_tc = orb->create_struct_tc(
         "IDL:MyStruct:1.0",
         "MyStruct",
         mems
);

// Construct the DynStruct object. Values for members are
// the value1 and value2 variables

DynamicAny::DynAny_ptr dyn_any
   = dafact->create_dyn_any(new_tc);
DynamicAny::DynStruct_ptr dyn_struct
   = DynamicAny::DynStruct::_narrow(dyn_any);
CORBA::release(dyn_any);
dyn_struct->insert_long(value1);

dyn_struct->next();
dyn_struct->insert_boolean(value2);
result = dyn_struct->to_any();
dyn_struct->destroy();
CORBA::release(dyn_struct);
```

9.3.2 Dynamic interpretation of CORBA::Any values

9.3.2.1 Filtering of events

Suppose there is a CORBA object which receives events and prints all those events which correspond to a data structure containing a member called `is_urgent` whose value is true.

The following fragment of code corresponds to a method which determines if an event should be printed or not. Note that the program allows several struct events to be filtered with respect to some common member.

```
// C++
CORBA::Boolean Tester::eval_filter(
   DynamicAny::DynAnyFactory_ptr dafact,
   const CORBA::Any &            event
)
{
   CORBA::Boolean success = FALSE;

   // First, convert the event to a DynAny.
   // Then attempt to narrow it to a DynStruct.
   // The _narrow only returns a reference
   // if the event is a struct.
   //
   DynamicAny::DynAny_var dyn_var
      = dafact->create_dyn_any(event);
   DynamicAny::DynStruct_var dyn_struct
      = DynamicAny::DynStruct::_narrow(dyn_any);
   if (!CORBA::is_nil(dyn_struct)) {
      CORBA::Boolean found = FALSE;
      do {
         CORBA::String_var member_name
            = dyn_struct->current_member_name();
         found = (strcmp(member_name, "is_urgent") == 0);
      } while (!found && dyn_struct->next());


      if (found) {
      // We only create a DynAny object for the member
      // we were looking for:
         DynamicAny::DynAny_var dyn_member
            = dyn_struct->current_component();
         success = dyn_member->get_boolean();
      }
      dyn_struct->destroy();
   }
   return success;
}
```