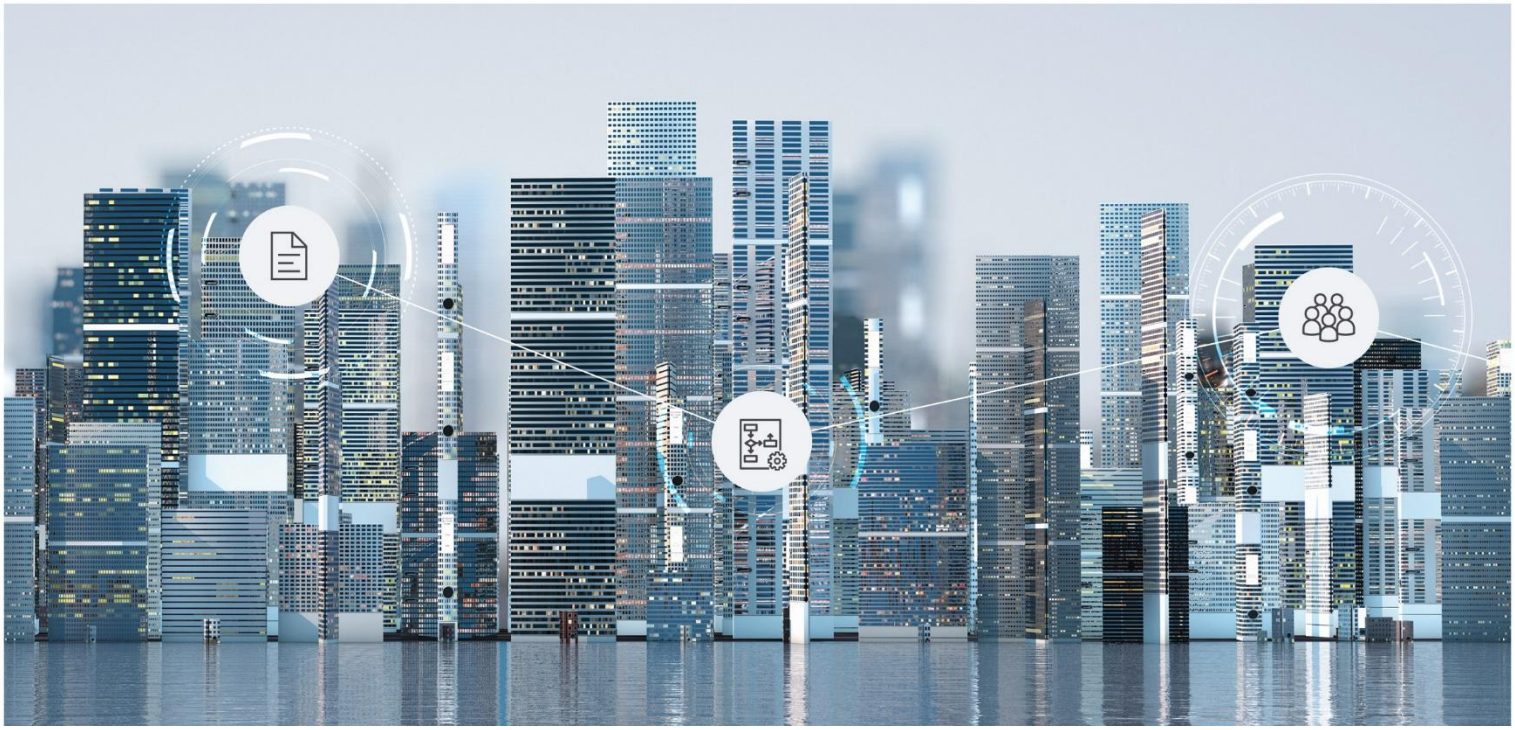
Fabasoft



White Paper

Fabasoft app.ducx Expressions

2025 July Release

Contents

1 Fabasoft app.ducx Expressions 6

1.1 General Remarks Concerning app.ducx Expression Language 6

1.1.1 Evaluating Expressions at Runtime 6

1.1.2 Testing Expressions 6

1.1.3 Tracing in app.ducx Expression Language 6

1.2 Names 7

1.3 Scopes 7

1.4 Types 8

1.4.1 Boolean 10

1.4.2 Integer 10

1.4.3 Float 10

1.4.4 String 10

1.4.5 Datetime/Date 11

1.4.6 Currency 13

1.4.7 Content 13

1.4.8 Dictionary 14

1.4.9 Object 14

1.4.10 Any 15

1.5 Variables 15

1.5.1 Redeclaration of Variables 16

1.5.2 Declaration of Available Variables 17

1.5.3 Declaration of Dictionary Members 17

1.6 Operators 17

1.6.1 Assignment Operators 17

1.6.2 Logical Operators 18

1.6.3 Calculation Operators 19

1.6.4 Comparison Operators 22

1.6.5 Conditional Operator 26

1.6.6 Selection Operator 26

1.6.7 $ Operator 27

1.6.8 # Operator 27

1.7 Predefined Variables and Functions 28

1.7.1 Predefined Variables 28
1.7.2 Popular Kernel Methods 29
1.7.3 Working With Contents 30
1.7.4 Working With Dictionaries 30
1.7.5 Getting the Data Type of an Expression 31
1.7.6 String Functions 31
1.7.7 List Functions 34
1.7.8 Mathematical Functions 35
1.7.9 Escape Sequences for Special Characters 36
1.8 Getting and Setting Property Values 37
1.9 Invoking Use Cases 38
1.10 Calculated Identifiers / Dynamic Invocation 39
1.11 Accessing the Transaction Context 40
1.12 Control Flow 44
1.12.1 Conditions 44
1.12.2 Loops 45
1.12.3 Exceptions and Error Handling 46
1.12.4 Creating New Transactions or Opening a Transaction Scope 47
1.12.5 Returning Values 48
1.12.6 Directives 49
1.13 Searching for Objects – app.ducx Integrated Query 50
1.13.1 FROM Clause 51
1.13.2 WHERE Clause 51
1.13.3 SELECT Clause 52
1.13.4 Options 52
1.14 Grammar of the app.ducx Expression Language 52
1.15 Kernel Interfaces: Searching for Objects 54
1.15.1 Options 55
1.15.2 Properties 56
1.15.3 Classes 56
1.15.4 Object List 57
1.15.5 Condition 57
1.15.6 Search Query Examples 58
1.15.7 Query Arguments 59
1.15.8 Grammar of the Kernel Interfaces Query Language 59

1.15.9 Grammar of the Kernel Interfaces Expression Language 60

1 Fabasoft app.ducx Expressions

app.ducx Expression Language is a proprietary interpreted script language that allows you to access the object model, and to invoke use cases. app.ducx expressions can be embedded in `expression` blocks and other language elements of the domain-specific languages of Fabasoft app.ducx. This chapter introduces the app.ducx Expression Language.

Note: The grammar of the app.ducx Expression Language can be found in chapter 1.14 “Grammar of the app.ducx Expression Language”. The syntax for search queries is available in chapter 1.15.8 “Grammar of the Kernel Interfaces Query Language”.

1.1 General Remarks Concerning app.ducx Expression Language

app.ducx Expression Language is a distinct domain-specific language of Fabasoft app.ducx. app.ducx expressions can be embedded inline in an `expression` block in other domain-specific languages. However, it is also possible to create separate `.ducx-xp` files containing app.ducx expressions. app.ducx expression language files can be referenced from other domain-specific languages using the `file` keyword.

The app.ducx expression language is processed by the Fabasoft app.ducx compiler and transformed into Fabasoft app.ducx Expressions, which are evaluated at runtime by the kernel.

Keywords, predefined functions and predefined variables are not case sensitive.

1.1.1 Evaluating Expressions at Runtime

In the first step, the expression code is parsed. An expression can be parsed at runtime by calling the `Parse` method of the runtime. The `Parse` method returns an expression object (which is not related to an object stored in the domain). In a second step, the `Evaluate` method is invoked on the expression object for evaluating the expression. The scopes to be used during the evaluation of the expression must be passed to the `Evaluate` method. The result of the evaluation is passed back in the return value of the `Evaluate` method.

1.1.2 Testing Expressions

By selecting some expression code in any app.ducx file, this code can be executed against the installation.

1.1.3 Tracing in app.ducx Expression Language

Using app.ducx expression language, you can write trace messages to the Fabasoft app.ducx Tracer.

For writing messages and variable values to the trace output, you can either use the `%%TRACE` directive or the `Trace` method of the runtime.

If you pass two arguments to the `Trace` method of the runtime, the first argument is interpreted as a message while the second argument is treated as the actual value.

The `%%TRACE` directive can also be used to trace special objects like `cootx` to output all transaction variables defined for a transaction or `coometh` to output all set parameters within the implementation of a method.

Note:

- Values traced using the `%%TRACE` directive are only written to the Fabasoft app.ducx Tracer if trace mode is enabled for the corresponding software component whereas values traced using the `Trace` method of the runtime are always written to the Fabasoft app.ducx Tracer.
- Keep in mind that the `%%TRACE` directive only works for method implementation and customization points.

Example

```
// Tracing string messages
coort.Trace("This message is written to the trace output");
// Tracing variable values
string mystrvar = "Hello world!";
object myobjvar = cooroot;
coort.Trace("Value of mystrvar", mystrvar);
coort.Trace("Value of myobjvar", myobjvar);
// Trace directives are only evaluated if the software component
// is in trace mode
%%TRACE("Value of mystrvar", mystrvar);
%%TRACE("Value of myobjvar", myobjvar);
// Tracing local and global scope
coort.Trace("Local variables", this);
coort.Trace("Global variables",::this);
```

1.2 Names

A name is a letter, followed by letters, digits and the underscore '_' symbol. Keywords of the app.ducx languages are generally not allowed as valid identifiers with some exceptions. Names are used to reference all types of objects, i.e. use cases, functions, variables, interfaces in the Fabasoft Folio environment, optionally prefixed with a software component.

1.3 Scopes

A scope is similar to a container holding a value that is accessible during the evaluation of an expression. The following distinct scopes are available to you when an expression is evaluated:

- local scope, which is accessed using the operator `:>`
- global scope, which is accessed using the operator `::`
- temporary scope, which is accessed using the operator `@`

You can use the keyword `this` along with the corresponding operator to access the value of a scope (e.g. `:>this` yields the value of the local scope, and `::this` the value of the global scope). However, in most cases the keyword `this` can be omitted. When accessing the local scope, you can also omit the operator `:>` in most cases.

Note: Inside the selection operator `[]` for selecting values of lists or compound properties, the `:>` operator is required to access the local scope.

The keyword `declare` is used to declare an identifier. The Fabasoft app.ducx compiler automatically generates identifier declarations for use case parameters to allow access to parameters over the local scope when implementing a use case in app.ducx expression language.

For the following example, assume that the local scope `this` contains an instance of object class `APPDUCXSAMPLE@200.200:Order`, and that the temporary variable `@customer` contains an

instance of object class `FSCFOLIO@1.1001:ContactPerson`.
`APPDUCXSAMPLE@200.200:customerorders` is an object list pointing to instances of object class `APPDUCXSAMPLE@200.200:Order`. Within the square brackets put after `APPDUCXSAMPLE@200.200:customerorders`, `this` has a different meaning as it refers to each instance stored in the object list, and not to the local scope.

Example

```
ContactPerson @customer;
// Returns a STRINGLIST containing the names of all orders in property
// APPDUCXSAMPLE@200.200:customerorders
@customer.customerorders[objname];
// Returns an OBJECTLIST containing the orders whose name is identical to
// the name of the order in the local scope
@customer.customerorders[objname == :>objname];
```

In the following example, two strings, `isbn` and `title`, are declared in the local scope `this`. The temporary variable `@publication` is initialized with a dictionary consisting of two properties, `isbn` and `title` that in turn are initialized using the two strings `isbn` and `title` from the local scope.

Note: Within the scope of the curly braces or square brackets if used with filter expressions, when using the assignment operator "`=`", `this` refers to the compound structure itself. To access the local scope, the `:>` operator must be used.

Example

```
string isbn = "000-0-00000-000-0";
string title = "An Introduction to Fabasoft app.ducx";
dictionary @publication = { isbn = :>isbn, title = :>title };
```

For better readability, the "JSON" notation could be used. In this notation, the scope does not change.

Example

```
string isbn = "000-0-00000-000-0";
string title = "An Introduction to Fabasoft app.ducx";
dictionary  @publication = { isbn : isbn, title : title };
```

The temporary scope `@this` is used for storing temporary values during the evaluation of an expression. Local scope `this` and global scope `::this` are similar to parameters, and can be populated with any valid value when the `Evaluate` method is called for evaluating an expression.

The scopes cannot be changed while an expression is evaluated. However, you can add or modify dictionary entries if a scope contains a dictionary.

1.4 Types

Types can be categorized into basic types and user-defined types (such as classes, structs, and enums). In addition, types can be categorized into scalar types and list types.

The following basic types are most common and are built into the language. These types have a corresponding type definition in the object model, as shown in following table.

Basic Type	Corresponding Type Object
`boolean`	`COOSYSTEM@1.1:BOOLEAN`
`integer`	`COOSYSTEM@1.1:INTEGER`
`float`	`COOSYSTEM@1.1:FLOAT`
`string`	`COOSYSTEM@1.1:STRING`
`date`, `datetime`	`COOSYSTEM@1.1:DATETIME`
`currency`	`COOSYSTEM@1.1:Currency`
`content`	`COOSYSTEM@1.1:CONTENT`
`dictionary`	`COOSYSTEM@1.1:DICTIONARY`
`object`	`COOSYSTEM@1.1:OBJECT`

Note: The special value `null` represents the absence of data for all types. It is commonly assigned to variables to indicate that they currently do not reference any valid object or value.

In addition to scalar types, list types group multiple values together. Some list types have corresponding type definitions in the object model, as shown in following table.

Keyword	Type Definition
`boolean[]`	`COOSYSTEM@1.1:BOOLEANLIST`
`integer[]`	`COOSYSTEM@1.1:INTEGERLIST`
`float[]`	`COOSYSTEM@1.1:FLOATLIST`
`string[]`	`COOSYSTEM@1.1:STRINGLIST`
`date[]`, `datetime[]`	`COOSYSTEM@1.1:DATETIMELIST`
`content[]`	`COOSYSTEM@1.1:CONTENTLIST`
`dictionary[]`	`COOSYSTEM@1.1:DICTIONARYLIST`
`object[]`	`COOSYSTEM@1.1:OBJECTLIST`

Elements of a list can be retrieved using the selection operator (see chapter 1.6.6 “Selection Operator”).

More information about operators can be found in chapter 1.6 “Operators”.

Some types support using kernel methods to work with values. The usage of the generic kernel methods may provide a more generic access to values (e.g. it is possible to set dictionary members with uncommon or reserved names).

All available kernel methods can be found in the reference documentation:

https://help.cloud.fabasoft.com/index.php?topic=doc/Reference-Documentation/interfaces-overview.htm

1.4.1 Boolean

The `boolean` data type is used to represent logical values. Values of this type can be `true` or `false`.

Examples

```
boolean b1 = true;
boolean b2 = false;
boolean b3 = b1 || b2;
boolean b1 = null;
```

1.4.2 Integer

The `integer` data type represents 64 bit numbers without decimal points.

Example

```
integer i = 1;
integer i = null;
```

Useful explicit type conversions include the conversion from `string`, `float`, and `datetime`:

Example

```
integer i = integer("3"); // explicit conversion from string: string must be a
valid floating point number
integer i = 3.6; // explicit conversion from float: fractional digits are lost
during conversion
```

1.4.3 Float

The `float` data type is used to represent single-precision floating-point numbers with 9 digits of precision.

Example

```
float f = 3.14;
float f = null;
```

It implicitly supports the conversion from `integer` and the explicit conversion from `string`.

Example

```
float f = 1;
float f = float("3.14"); // explicit conversion from string: string must be a
valid floating point number
```

1.4.4 String

The `string` data type represents a sequence of characters and is used to store and manipulate textual data. Strings can be declared using single quotes (' '), or double quotes (" "). Strings can contain letters, numbers, symbols, and whitespace characters. They can also include special characters like newlines and tabs using escape sequences (link to XpGrammar). The implicit property `length` makes it possible to get the length of the string.

Examples

```
string firstname = "Jane";
string lastname = 'Doe';
string fullname = firstname + " " + lastname;
string fullname = null;
string stringwithnewline = "First line\nSecond line";
```

It implicitly supports the conversion from `integer` and `content`. Explicit conversions include conversions from float, object, datetime, and boolean.

Examples

```
string s = string(3.14); // "3.14"
string s = string(coouser); // results in the objaddress of the current user
string s = string(coonow); // current date and time in UTC format
string s = boolean(true); // "true"
```

1.4.5 Datetime/Date

The `datetime` data type refers to a built-in object that represents a specific date and time. It allows you to manipulate dates and times. To represent date only values you can use the keyword `date`. The following examples show how to assign different date values.

Examples

```
datetime now = coonow; // coonow holds the current date and time
datetime dt = 2012-01-02 14:15:22; // hours, minutes, and seconds are optional
datetime dt = 2012-01-02T14:15:22;
datetime dt = 2012-01-02;
datetime d = null;
date d = 2012-01-02;
date d = null;
```

It supports the explicit conversion from `string`:

Examples

```
datetime dt = datetime("2012-01-02T14:15:22"); // string needs to be a valid date literal
date d = date("2012-01-02");
```

The `datetime` data type exposes implicit properties that can be used to access and manipulate the individual date and time components. These implicit properties are listed in the following table. All of these implicit properties are of data type `integer`.

Property	Description
`year`	The `year` property can be used for getting and setting the year.
`month`	The `month` property can be used for getting and setting the month.
`day`	The `day` property can be used for getting and setting the day.
`hour`	The `hour` property can be used for getting and setting the hour.
`minute`	The `minute` property can be used for getting and setting the minutes.

`second`	The `second` property can be used for getting and setting the seconds.
`dayinweek`	The `dayinweek` property can be used for getting the day of the week, with the value `0` representing Sunday and `6` representing Saturday.
`dayinyear`	The `dayinyear` property can be used for getting the day of the year, with the possible values ranging from `1` to `366` (for a leap year).
`weekinyear`	The `weekinyear` property can be used for getting the week of the year, with the possible values ranging from `1` to `52`.
`local`	The `local` property returns the date and time value converted to local time.
`universal`	The `universal` property returns the date and time value converted to universal time.

Examples

```
// modify the date
dt.year = 2013;
dt.month = 2;
dt.day = 3;
// modify the time
dt.hour = 14;
dt.minute =15;
dt.second = 22;
// before two hours
datetime before2hours = coonow;
before2hours.hour -= 2;
datetime calcdate = 2024-03-03T00:00:00;
// Set time zone bias to -60 minutes, i.e. CET (Central European Time)
coort.SetThreadTimeZoneBias(-60);
// Get universal time value from calcdate (2024-03-02T00:00:00)
// utcdate will be 2024-03-01T23:00:00
datetime utcdate = calcdate.universal;
// A datetime value is not related to a time zone.
// So, when converting utcdate to the universal time once more
// utcdate will become 2024-03-01T22:00:00
utcdate = utcdate.universal;
// This is also the case when converting calcdate (2024-03-02T00:00:00)
// to local time. localdate will be 2024-03-02T01:00:00
datetime localdate = calcdate.local;
```

It is also possible to use some arithmetic and comparison operations with date values, as shown in the following examples.

Examples

```
datetime before2hours = coonow;
before2hours.hour -= 2;
coonow > before2hours; // dates can be compared
// caluclating time differences
datetime d1 = 2010-01-01 00:00:00;
datetime d2 = 2011-01-01 00:00:00;
integer diff = d2 - d1; // 1 year = 60*60*24*365 = 31536000 seconds
coonow % 86400; // set time to 00:00:00
```

```
// calculate the number of seconds since the last passed minute, hour, or day
d / 60; // equal to d.second
d / 60*60; // equal to d.second + d.minute*60
d / 60*60*24; // equal to d.second + d.minute*60 + d.hour*60*60
```

1.4.6 Currency

The `currency` data type is used to represent monetary values and allows to handle financial calculations related to currencies. It is composed of a currency value and a currency symbol. The currency value represents a specific amount of money. It can be a decimal or floating-point with an arbitrary number of digits. The currency symbol represents the specific currency unit.

Note: Converting a currency value to an integer or float can result in information loss since currency values can potentially have an arbitrary number of digits. To mitigate these issues, it is crucial to consider the range and precision of the target data type.

Examples

```
currency c1 = { currsymbol: EUR, currvalue: 3 };
currency c2 = { EUR, 3 };
currency c2 = null;
```

It supports implicit conversion from `integer` and `float` and it is also possible to apply basic arithmetic operations to currencies. Even with different currency symbols., as shown in following examples.

Examples

```
c1 + 3; // 6 EUR
c1 + c2; // 6 EUR
c1 + currency({ currsymbol: USD, currvalue: 3 }); // 6 EUR - the resulting
currency has the currency symbol of the left operand
```

1.4.7 Content

The `content` data type represents arbitrary data, such as text or binary files.

It may not to be confused with the struct `COOSYSTEM@1.1:Content`. The struct `Content` holds a property of type `content` along with additional metadata such as the file extension.

Examples

```
content mycont = "Hello world!"; // Implicit conversion
integer mycontsize = mycont.size; // 15 - 3 from the BOM and 12 from "Hello
world!"
content mycontfromhex = { base16: "4142" };
%%ASSERT(mycontfromhex.content == "AB");
content cont = coort.CreateContent();
cont.SetFile("/tmp/log.txt"); // read the content from a file
string log = cont.GetContent(cootx, COOGC_MULTIBYTEFILE, COOGC_UTF8); // decode
the content
log += " Some more text";
cont.SetContent(cootx, COOGC_MULTIBYTEFILE, COOGC_UTF8, log); // encode the
string
string filename = cont.GetFile(, true); // create a file with the new content
cont = null;
```

Note: When converting a `string` to a `content` as in the example above, be aware that this conversion also includes an UTF-8-byte order mark (BOM). Therefore, the value `mycontsize` is 15 (3 from the BOM and 12 from `"Hello world!"`).

1.4.8 Dictionary

The `dictionary` data type represents key-value maps to store and retrieve values based on unique string keys. It provides an efficient way to perform lookups, insertions, and deletions.

Dictionaries associate each key with a corresponding value. Keys are unique within the dictionary. Values can be of any data type, such as integers, strings, objects, or even other dictionaries. Moreover, values can be lists.

Dictionaries can be created in the following way:

Examples

```
dictionary mydictionary = {}; // create an empty dictionary
dictionary mydictionary = coort.CreateDictionary(); // create an empty dictionary
dictionary mydictionary = { key1: 1, key2: "2", key3: [1, 2] }; // create a pre-
initialized dictionary
```

To store a value in a dictionary, you provide a key-value pair and the dictionary associates the key with the corresponding value. If the key already exists in the map, the previous value associated with that key is overwritten. If the key is new, a new entry is created in the dictionary.

Examples

```
mydictionary.key1 = 2; // set value of key1
mydictionary.SetEntry("key1", 2); // set the value of key1 using a kernel method
mydictionary.key3 += 3; // add an element to the list under key3
mydictionary = null;
```

To retrieve a value from the map, you specify the key, and the map returns the corresponding value associated with that key. If the key is not found in the dictionary, a null value is returned.

Examples

```
integer i = mydictionary.key1; // get value of key1
integer i = mydictionary.GetEntry("key1"); // get value of key1 using kernel
method
integer[] integerlist = mydictionary.key3;
```

1.4.9 Object

The `object` data type represents objects in general. It is valid for instances of all object classes. It allows to access the most basic properties of objects, such as the name and address. These implicit properties are listed in the following table.

Property	Description
`address`	The `address` property can be used to get the unique key of the object.
`identification`	The `identification` property can be used to get the key of an object at a given time.

`name`	The `name` property can be used to get the name of the object without transaction context.
`reference`	The `reference` property can be used to get the reference of an object. This property is only defined for objects of classes derived from `ComponentObject`.

Examples

```
object myobject = coouser; // assign an existing object
object myobject = #User.ObjectCreate()[2]; // create an object
string objectAddress = mobject.address;
myobject = null;
```

It supports the explicit conversion from `string`:

Examples

```
object myobject = "COO.1.1.1.402"; // string must be a valid object address
object myobject = "COOSYSTEM@1.1:Object";// string must be a valid reference
```

1.4.10 Any

The `any` data type is a special type that represents a value of any type. When a variable is declared of type `any`, it allows that variable to hold any value, regardless of its type.

However, it's important to note that once a variable is assigned a specific type, it remains fixed and cannot be changed during runtime. This immutability can lead to type conversion errors if the variable is later used in operations or contexts that expect a different type.

Examples

```
any myinteger = 1;
myinteger = "2"; // Ok: conversion to integer(2)
myinteger = "a"; // Nok: runtime error: Could not convert 'a' to
'COOSYSTEM@1.1:INTEGER'
myinteger = null;
```

1.5 Variables

For all variables within an expression a type should be defined explicitly.

Syntax

```
// Declaring a variable
Type variable;
// Declaring and initializing a variable
Type variable = initialvalue;
```

Valid types are object classes and objects of the class `COOSYSTEM@1.1:TypeDefinition`, which includes base types as well as compound types and enumerations.

Additionally, attribute definitions (objects of type `COOSYSTEM@1.1:AttributeDefinition`) can be used as types for variables.

Shortcuts are provided for basic data types as defined by the object model language as well as kernel interfaces as listed below.

Type	Description
`runtime`	The type for a runtime interface, such as the predefined symbol `coort`.
`transaction`	The type for a transaction interface, such as the predefined symbol `cootx`. An interface of this type is returned, if you create a new transaction using `coort.CreateTransaction()`.
`method`	The type for a method interface, such as the predefined symbol `coometh`. An interface of this type is returned, if you obtain the implementation of a use case using `cooobj.GetMethod()`.
`searchresult`	The type for an interface for the result of an asynchronous search, as returned by `coort.SearchObjectsAsync()`.
`expression`	The type for a Fabasoft app.ducx Expression interface, as returned by `coort.Parse()`.
`aggregate`	The type for an aggregate interface, which can be used as a generic substitute for any compound type.
`interface`	A generic type for an interface.

Example

```
integer @bulksize = 150;
string @query = "SELECT objname FROM APPDUCXSAMPLE@200.200:Order";
searchresult @sr = coort.SearchObjectsAsync(cootx, @query);
Order[] @results = null;
while ((@results = @sr.GetObjects(@bulksize)) != null) {
  %%TRACE("Fetched chunk of search results", @results);
  @results.ProcessOrder();
}
```

1.5.1 Redeclaration of Variables

Sometimes the types of variables are not known or not specific enough to write expressions without warnings. This can be fixed using the keyword `assume`. `assume` tells the app.ducx compiler the correct type.

Example

```
usecase OnUserAndGroup(any memberof) {
  variant User, Group {
    expression {
      if (cooobj.HasAttribute(cootx, #usersurname)) {
        assume User cooobj;
        assume Group[] memberof;
      }
    }
```

```
  }
}
private class SimpleObject: COODESK@1.1:Folder {
  AdministrationObject[] server {
    filter = expression as attrfilterbooleanexpr {
      if (HasClass(#Group)) {
        assume Group this;
        return this.grshortname != null;
      }
    }
  }
}
```

1.5.2 Declaration of Available Variables

Some scopes can contain variables which are not documented. Should this be the case and the developer wants to access these variables, the keyword `assume` can be used, too. In this case the `assume` statement is written like a parameter declaration, specifying input/output modifier.

Example

```
usecase SetReturnURL() {
  variant User {
    application {
      expression {
        assume in string sys_branchvalue;
        assume out ::ru;
        ...
      }
    }
  }
}
```

1.5.3 Declaration of Dictionary Members

The keyword `assume` can also be used to declare the types of dictionary entries, especially the cardinality.

Example

```
usecase UpdateUser(dictionary datainput) {
  variant User {
    expression {
        assume string datainput.name;
        cooobj.objname = name;
        assume integer[] datainput.weights;
    }
  }
}
```

1.6 Operators

app.ducx expression language supports a wide range of operators.

1.6.1 Assignment Operators

Assignment operators allow you to set property and variable values. The following table contains a list of supported assignment operators.

Operator	Description
`=`	The `=` operator is used for simple assignments. The value of the right operand is assigned to the left operand.
`+=`	Both operands are added, and the result is assigned to the left operand. The `+=` operator can be used with strings, numeric data types, currencies and lists.
`-=`	The right operand is subtracted from the left operand, and the result is assigned to the left operand. The `-=` operator can be used with numeric data types, currencies, lists and dictionaries.
`*=`	Both operands are multiplied, and the result is assigned to the left operand. The `*=` operator can be used with numeric data types, currencies, lists and dictionaries.
`/=`	The left operand is divided by the right operand, and the result is assigned to the left operand. The `/=` operator can be used with numeric data types, currencies, lists and dictionaries.
`%=`	A modulus operation is carried out, and the result is assigned to the left operand. The `%=` operator can only be used with numeric data types, currencies, lists and dictionaries.
`<<=`	`<<=` is used for character- and bitwise shifting to the left. The `<<=` operator can be used with strings, integers, currencies and lists.
`>>=`	`>>=` is used for character- and bitwise shifting to the right. The `>>=` operator can be used with strings, integers, currencies and lists.
`??=`	If the left operand is null, the right operand is evaluated and assigned to the left operand.

Example

```
Order @order;
Customer @customer;
// A simple assignment operation
@order.orderdate = coonow;
// Adding an element to a list
@customer.customerorders += @order;
// Adding an element to a list only if it is not part of the list already
@customer.customerorders *= @order;
```

1.6.2 Logical Operators

Logical operators support short circuit evaluation semantics. The right operand is only evaluated if the result of the evaluation is not determined by the left operand already. The following table shows a list of the supported logical operators.

Operator	Description

`and` (alternatively `&&`)	The `and` operator indicates whether both operands are `true`. If both operands have values of `true`, the result has the value `true`. Otherwise, the result has the value `false`. Both operands are implicitly converted to `BOOLEAN` and the result data type is `BOOLEAN`.
`or` (alternatively `||`)	The `or` operator indicates whether either operand is `true`. If either operand has a value of `true`, the result has the value `true`. Otherwise, the result has the value `false`. Both operands are implicitly converted to `BOOLEAN` and the result data type is `BOOLEAN`.
`not` (alternatively `!`)	The expression yields the value `true` if the operand evaluates to `false`, and yields the value `false` if the operand evaluates to `true`. The operand is implicitly converted to `BOOLEAN`, and the data type of the result is `BOOLEAN`.
`??`	The expression yields the value of the left operand if not null, otherwise the value of the right operand.

Example

```
if (@orderstate == OrderState(OS_SHIPPED) and @orderdate != null or
  @orderstate == OrderState(OS_COMPLETED) and @invoice == null) {
  throw #InvalidProcessingState;
}
```

1.6.3 Calculation Operators

The following table contains a list of supported calculation operators.

Operator	Description
`+ - * / %`	The `+`, `-`, `*`, `/` and `%` operators are supported for numeric data types and lists. `+`, `-`, `*` and `/` are also supported for currencies (`*` and `/` need one integer or float operand). Additionally, the `+` operator can be used to concatenate strings. **When used with lists, the following semantic applies:** `+` (concatenation): The right operand is concatenated to the end of the left operand. `-` (difference): Each element from the right operand is removed from the left operand. `*` (union): Each element from the right operand is appended to the left operand if the element does not occur in the left operand. `/` (symmetric difference): The resulting list is the union of the difference of the left and the right operand and the difference of the right and the left operand: `a / b == (a - b) * (b - a) == (a * b) - (b % a)`.

	`%` (intersection): The resulting list is the list of elements that exist in both the left and the right operand.
	
	Note: Each element in a list is treated as an individual element, even if the list contains other elements with the same value. So be careful if you use operators with lists that are not unique.
	
	Since there are two elements "`2`" in the left operand, these expressions are `true`:
	`[1, 2, 2] - [2] == [1, 2];`
	`[1, 2, 2] * [2] == [1, 2, 2];`
	
	When used with dictionaries, the following semantic applies:
	
	`-` (difference): Each entry from the right operand is removed from the left operand (regardless of the value of the entry)
	
	`*` (union): Each entry from the right operand is appended to the left operand if the entry does not occur in the left operand.
	
	`/` (symmetric difference): The resulting dictionary is the union of the difference of the left and the right operand and the difference of the right and the left operand: `a / b == (a - b) * (b - a) == (a * b) - (b % a)`.
	
	`%` (intersection): The resulting dictionary contains the entries that exist in both the left and the right operand. The values are taken from the left operand.

`++`	The `++` increment operator is a unary operator that adds 1 to the value of a scalar numeric operand. The operand receives the result of the increment operation. You can put the `++` before or after the operand. If it appears before the operand, the operand is incremented. The incremented value is then used in the expression. If you put the `++` after the operand, the value of the operand is used in the expression before the operand is incremented.
`--`	The `--` decrement operator is a unary operator that subtracts 1 from the value of a scalar numeric operand. The operand receives the result of the decrement operation. You can put the `--` before or after the operand. If it appears before the operand, the operand is decremented. The decremented value is then used in the expression. If you put the `--` after the operand, the value of the operand is used in the expression before the operand is decremented.
`<<`	The `<<` is used for character- and bitwise shifting to the left. The `<<` operator can be used with strings, integers, currencies and lists. When used with strings, the right operand specifies the number of characters removed from the beginning of the string. When used with lists, the right operand specifies the number of elements to be removed from the top of the list.
`>>`	The `>>` is used for character- and bitwise shifting to the right. The `>>` operator can be used with strings, integers, currencies and lists. When

	used with strings, the right operand specifies the number of spaces inserted on the left side of the string. When used with lists, the right operand specifies the number of elements to be removed from the end of the list.

Example

```
@aaa = ["John", "James", "Jim", "Jamie"];
@bbb = ["Jamie", "Jim"];
// Check if every element of list @bbb is included in list @aaa.
// You have to sort the both operands of the compare
// since for % the order is taken from the first operand @aaa:
sort(@aaa % @bbb) == sort(@bbb);
// So it is more efficient to use the difference:
@bbb - @aaa == [];
// Check if the last change of an object was carried out on the same date it was
// created. Using the "%86400" operation, the time portion of the datetime
// property "COOSYSTEM@1.1:objchangedat" is set to "00:00:00" in order to
compare
// the date portion only using the "==" operator.
objcreatedat % 86400 == objchangedat % 86400
```

Note: If two different currencies are added or subtracted an implicit conversion is carried out. Following evaluation order is defined: The conversion table of the transaction variable `TV_CURRCONVTAB` is used. If `TV_CURRCONVTAB` is not available, the conversion table of the left operand is used. If not available, the conversion table of the right operand is used. Otherwise, an error is generated.

Examples for List Operators

The `+`, `-`, `*`, `/` and `%` operators (concatenation, difference, union, symmetric difference, intersection) are supported for lists. The following example shows how list operators work.

Example

```
[]+[1] == [1];
[1, 2, 3] + [2, 3, 4] == [1, 2, 3, 2, 3, 4];
[1, 2, 3] + [] == [1, 2, 3];
[1, 2, 3] - [2, 3, 4] == [1];
[1, 2, 3] - [1, 2, 3] == [];
[1, 2, 2, 3] - [1, 2, 3] == [2];
[1, 2, 3] - [3, 2, 1] == [];
[1, 2, 3] - [4, 5, 6] == [1, 2, 3];
[1, 2, 2] - [2, 3, 4] == [1, 2]
[] - [1] == [];
[1, 2, 3] * [2, 3, 4] == [1, 2, 3, 4];
[1, 2, 2] * [1, 2, 2, 3, 3, 3, 4] == [1, 2, 2, 3, 3, 3, 4];
[] * [1, 2, 2] == [1, 2, 2];
[1, 2, 3] * [] == [1, 2, 3];
[1, 2, 3] * [3, 2, 1] == [1, 2, 3];
[1, 2, 2] * [2, 3, 4] == [1, 2, 2, 3, 4];
[1, 2, 3] / [4, 5, 6] == [1, 2, 3, 4, 5, 6];
[1, 2, 3] / [2, 3, 4] == [1, 4];
[1, 2, 3] / [1, 2, 3] == [];
[1, 2, 3] / [3, 2, 1] == [];
[1, 2, 2] / [2, 3, 4] == [1, 2, 3, 4]
[1, 2, 3, 4] % [2, 3, 4] == [2, 3, 4];
[1, 2, 3] % [2, 3, 4] == [2, 3];
```

```
[1, 2, 3] % [4, 5, 6] == [];
[1, 2, 3] % [3, 2, 1] == [1, 2, 3];
[] % [4, 5, 6] == [];
```

Examples for Dictionary Operators

The `-`, `*`, `/` and `%` operators (difference, union, symmetric difference, intersection) are supported for dictionaries. These operators work on an element level, the value of a dictionary entry is not relevant. The left operand dominates. The following example shows how dictionary operators work.

Example

```
({}) - ({}) == ({});
({}) - ({ a: 1, b: "x", c: true }) == ({});
({ a: 1, b: "x", c: true }) - ({}) == ({ a: 1, b: "x", c: true });
({ a: 1, b: "x", c: true }) - ({ a: 1, b: "x", c: true }) == ({});
({ a: 1, b: "x", c: true }) - ({ b: 2, c: "x", d: true }) == ({ a: 1 });
({ a: 1, b: "x", c: true }) - ({ a: 2, b: "x", c: 1 }) == ({});
({ a: 2, b: "x", c: 1 }) - ({ a: "x" }) == ({ b: "x", c: 1 });

({}) * ({}) == ({});
({}) * ({ a: 1, b: "x", c: true }) == ({ a: 1, b: "x", c: true });
({ a: 1, b: "x", c: true }) * ({}) == ({ a: 1, b: "x", c: true });
({ a: 1, b: "x", c: true }) * ({ a: 1, b: "x", c: true }) == ({ a: 1, b: "x", c:
true });
({ a: 1, b: "x", c: true }) * ({ b: 2, c: "x", d: true }) ==
                                                    ({ a: 1, b: "x", c: true, d:
true });
({ a: 1, b: "x", c: true }) * ({ a: 2, b: "x", c: 1 }) == ({ a: 1, b: "x", c:
true });
({ a: 2, b: "x", c: 1 }) * ({ a: "x" }) == ({ a: 2, b: "x", c: 1 });

({}) / ({}) == ({});
({}) / ({ a: 1, b: "x", c: true }) == ({ a: 1, b: "x", c: true });
({ a: 1, b: "x", c: true }) / ({}) == ({ a: 1, b: "x", c: true });
({ a: 1, b: "x", c: true }) / ({ a: 1, b: "x", c: true }) == ({});
({ a: 1, b: "x", c: true }) / ({ b: 2, c: "x", d: true }) == ({ a: 1, d: true
});
({ a: 1, b: "x", c: true }) / ({ a: 2, b: "x", c: 1 }) == ({});
({ a: 2, b: "x", c: 1 }) / ({ a: "x" }) == ({ b: "x", c: 1 });

({}) % ({}) == ({});
({}) % ({ a: 1, b: "x", c: true }) == ({});
({ a: 1, b: "x", c: true }) % ({}) == ({});
({ a: 1, b: "x", c: true }) % ({ a: 1, b: "x", c: true }) == ({ a: 1, b: "x", c:
true });
({ a: 1, b: "x", c: true }) % ({ b: 2, c: "x", d: true }) == ({ b: "x", c: true
});
({ a: 1, b: "x", c: true }) % ({ a: 2, b: "x", c: 1 }) == ({ a: 1, b: "x", c:
true });
({ a: 2, b: "x", c: 1 }) % ({ a: "x" }) == ({ a: 2 });
```

1.6.4 Comparison Operators

Comparison operators allow you to compare two operands. The following table provides a summary of the supported comparison operators. The data type of the result is always `BOOLEAN`.

Operator	Description
`==`	The equality operator compares two operands and indicates whether the value of the left operand is equal to the value of the right operand.

	The equality operator has a lower precedence than the relational operators (<, <=, >, >=).
!= (alternatively, <>)	The inequality operator compares two operands and indicates whether the value of the left operand is not equal to the value of the right operand. The inequality operator has a lower precedence than the relational operators (<, <=, >, >=).
<	The relational operator < compares two operands and indicates whether the value of the left operand is less than the value of the right operand.
<=	The relational operator <= compares two operands and indicates whether the value of the left operand is less than or equal to the value of the right operand.
>	The relational operator > compares two operands and indicates whether the value of the left operand is greater than the value of the right operand.
>=	The relational operator >= compares two operands and indicates whether the value of the left operand is greater than or equal to the value of the right operand.
<=>	The relational operator <=> compares two operands. It returns -1 if the left value is lower and +1 if the left value is greater than the right value. If the values are equal, the result is 0.
`contains`	The `contains` operator determines whether left operand contains the right operand. This operator can be used with string operands. It may be preceded by the `not` keyword.
`like`	The `like` operator determines whether the left string matches the right string. The `%` and _ wildcards can be used in the right string operand and cannot be escaped. The `like` operator can be preceded by the `sounds` keyword for a phonetic comparison. Furthermore, it can also be preceded by the `not` keyword.

`in`	The `in` operator determines whether the value of the left operand is an element of the list provided in the right operand. The `in` operator can also be used with a list in the left operand. It may be preceded by the `not` keyword.
	When using lists in both operands, the semantic is:
	`[a1, a2, ... an] in [b1, b2, ... bm]`
	`->`
	`(a1 == b1 or a1 == b2 or ... or a1 == bm) or`
	`(a2 == b1 or a2 == b2 or ... or a2 == bm) or`
	`.`
	`.`

	.
	(an == b1 or an == b2 or ... or an == bm)
	
	This means, that the expression is `true`, if **any** element from the **left** operand is in the list of the right operand.
	
	Note:
	
	• **Version 2021 and later releases**
	When the left operand is `null` (i.e. an empty list) the evaluation of the `in` operator is `false`.
	• **Version 2020 and prior releases**
	When the left operand is `null` (i.e. an empty list) the evaluation of the `in` operator is `true`.
	This behavior was changed as it led to typical errors in cases like this:
	`if (obj in objlist) {`
	`  obj.GetName(cootx);`
	`}`

`includes`	The `includes` operator determines whether the value of the right operand is an element of the list provided in the left operand. It may be preceded by the `not` keyword.
	
	When using lists in both operands, the semantic is:
	
	`[a1, a2, ... an] includes [b1, b2, ... bm]`
	
	`->`
	
	`(a1 == b1 or a1 == b2 or ... or a1 == bm)` **and**
	`(a2 == b1 or a2 == b2 or ... or a2 == bm)` **and**
	.
	.
	.
	`(an == b1 or an == b2 or ... or an == bm)`
	
	This means, that the expression is `true`, if **all** elements from the **right** operand are in the list of the left operand.
	
	Note:
	
	• **Version 2021 and later releases**
	When the right operand is `null` (i.e. an empty list) the evaluation of the `includes` operator is `false`.
	• **Version 2020 and prior releases**
	When the right operand is `null` (i.e. an empty list) the evaluation of the `includes` operator is `true`.
	This behavior was changed as it led to typical errors in cases like this:
	`if (objlist includes obj) {`
	`  obj.GetName(cootx);`
	`}`

`between and`	The `between and` operator determines whether the value of the first operand is in the range between the values of the operands provided after the keywords `between` and `and`. If the first operand is a list, then all values of the list must be between the second and the third operand.
`is null`	The `is null` operator returns `true` if the value of the left operand is undefined.

Example

```
if (@points < 100) {
  @memberstatus = "MS_SILVER";
}
else if (@points between 100 and 1000) {
  @memberstatus = "MS_GOLD";
}
else {
  @memberstatus = "MS_PLATINUM";
}
// If @memberstatus is null this evaluates to true
if (@memberstatus in ["MS_GOLD", "MS_PLATINUM"]) {
  @expressshipping = true;
}
if (@nickname like "Bob%" or @nickname in ["Dick", "Rob"]) {
  @firstname = "Robert";
}
```

Note:

- When comparing aggregates or dictionaries, the values of all attributes or entries are compared. The comparison is recursive for nested values.
- Aggregate types can specify a comparator method in `COOSYSTEM@1.1:typecompare`. When comparing such aggregates this method is used to calculate the result of the comparison. Standard comparison of aggregates only allows a check for equality of values (operators `==`, `!=`, `in`, and `includes`). Using `COOSYSTEM@1.1:typecompare` allows implementation of greater or less operators for aggregates. If no `COOSYSTEM@1.1:typecompare` is specified, the default comparison for aggregates uses the properties in `COOSYSTEM@1.1:typesort` first, then the properties in `COOSYSTEM@1.1:typecompattrs`.
- Objects are compared in an internal order.
- Dictionaries only allow a check for equality of values (operators `==`, `!=`, `in`, and `includes`).
- Contents and COM interfaces cannot be compared by their value, when comparing using the operators `==`, `!=`, `in`, and `includes`, the internal identity of these objects is used.
- String operands are compared using the setting `COOSYSTEM@1.1:domaincisqry` in your current domain object. The default for this setting is true, meaning that comparison is case insensitive by default. This is also relevant for `min`/`max`/`sort`/`unique`/`find` and the list operators `-`, `*`, `/`, and `%`.

1.6.5 Conditional Operator

The conditional operator `?:` has three operands. It tests the result of the first operand, and then evaluates one of the other two operands based on the result of the evaluation of the first operand. If the evaluation of the first operand yields `true`, the second operand is evaluated. Otherwise, the third operand is evaluated.

Example

```
@orders = (@customer != null) ?
  @customer.customerorders[objname] :
  null;
```

1.6.6 Selection Operator

The selection operator `[]` can be used for the following purposes:

- as a list constructor to define a list of values,
- to select elements from a list of values
- to filter elements of a list,
- to specify a parameter as the return value when invoking a use case, and
- for calculated identifiers (see chapter 1.10 “Calculated Identifiers”).

Note:

- **Version 2021 February Release and later releases**
  When using the selection operator to filter elements of a list, the following optimization is implemented: if the result of the expression is used in a Boolean context or checked for `null`, processing stops, once the Boolean result is available.
  ```
  if (folder.objchildren[IsUsable()]) {
    // list processing stopped after the first matching object
  }
  boolean visisble = folder.objchildren[IsUsable()] != null;
  ```
- **Version 2020 and prior releases**
  When using the selection operator to filter elements of a list, all elements are processed.

Example

```
// Constructing an empty list
@productcategories = [];
// Constructing a list of string values
@productcategories = ["Fish", "Meat", "Poultry"];
// Selecting elements from a list
// The result is a single element
@fish = @productcategories[0];
@meat = @productcategories[1];
// Selecting multiple elements from a list
// The result is a list again
@nofish = @productcategories[1,2];
// Selecting elements starting from the end of the list by specifying negative
indices
@poultry = @productcategories[-1];
// Example for filtering a list: This expression returns the orders that
// do not have a valid order date.
// The result is again a list if more than one item is selected
@customer.customerorders[orderdate is null];
```

```
// Selecting a sub list
// The result is a list again
@nofish = @productcategories[1:2];
// Selecting a sub list with negative elements
// The result is a list again
@nofish = @productcategories[-2:-1];
// Specifying a parameter as the return value when invoking a use case
@neworder = #Order.ObjectCreate()[2];
// Results in the method object of the call
@meth = #Order.ObjectCreate()[...];
// Results in all entries of the customization point CPSymbols
@list = coouser.CPSymbols()[...];
```

1.6.7 $ Operator

The $ operator can be used in following cases.

1.6.7.1 Identifier

By default, the kernel tries to interpret identifiers as references when evaluating expressions. In order to use an identifier that could also be a reference as name, it must be prefixed with `$`.

Note: The Fabasoft app.ducx compiler will attempt to automatically insert the symbol `$` when serializing the expression, if it can determine the context in which the identifier is used. If it can't calculate a definitive type, you will receive a warning.

For the following example, assume that the local scope contains a dictionary. `objname` can only be used as a variable when prefixed with `$`.

Example

```
// Assuming the local scope contains a dictionary:
$objname = "Hello world";
// When omitting the "$", the expression is interpreted as follows:
this.COOSYSTEM@1.1:objname = "Hello world";
```

1.6.7.2 String Interpolation

If strings are prefixed with `$`, they can contain arguments with the syntax `{~ argument ~}`. All arguments in the string are evaluated, converted to string type and embedded in the string instead of the pattern.

Example

```
// '--Administrator, System--'
string username = $"--{~ coouser.GetName() ~}--";
// '3'
string displaycount = $"{~ count([0,1,2]) ~}";
```

1.6.8 # Operator

If you need to retrieve a component object in an expression, you must prefix its reference with `#`. In order to use an identifier that could also be a variable as reference, it must be prefixed with `#`.

Note: The Fabasoft app.ducx compiler will attempt to automatically fully qualify an identifier to a reference when serializing the expression, if it can determine the context in which the identifier is used. If it can't calculate a definitive type, you will receive a warning.

For instance, when referring to a property definition or an object class, you must prefix the reference with `#` in order to get the corresponding component object.

Example

```
// Accessing property definition COOSYSTEM@1.1:objname
@objnameprop = #objname;
@ordername = @order.GetAttributeValue(cootx, @objnameprop);
// Accessing object class APPDUCXSAMPLE@200.200:Order
@orderclass = #APPDUCXSAMPLE@200.200:Order;
@neworder = @orderclass.ObjectCreate();
// Accessing property of undeclared object
@neworder.#objname = "New Name";
```

1.7 Predefined Variables and Functions

The app.ducx expression language comes with a set of predefined variables and functions to make programming as convenient as possible.

1.7.1 Predefined Variables

The following table shows the list of predefined variables that are provided automatically by the kernel when an expression is evaluated.

Variable	Description
`coort`	The `coort` variable can be used to access the runtime.
`cootx`	The current transaction context is accessible by the `cootx` variable.
`cooobj`	The `cooobj` variable holds the current object on which the evaluation of the expression is invoked.
`coometh`	For use case implementations in app.ducx expression language, the `coometh` variable holds the so-called method context.
`coouser`	The `coouser` variable holds the user object of the current user.
`coogroup`	The `coogroup` variable holds the group object of the role of the current user.
`cooposition`	The `cooposition` variable holds the position object of the role of the current user.
`cooenv`	The `cooenv` variable holds the user environment of the current user.
`cooroot`	The `cooroot` variable holds the desk of the current user.
`coolang`	The `coolang` variable holds the language of the current user.

`coodomain`	The `coodomain` variable holds the current domain.
`coonow`	The `coonow` variable holds the current date and time.

Example

```
// Using the runtime
User @user = coort.GetCurrentUser(); // or coouser
object @desk = coort.GetCurrentUserRoot();// or cooroot
datetime @currentdate = coonow;
// Accessing transaction variables using the generic interface
cootx.SetVariableValue(#APPDUCXSAMPLE@200.200, 1, #BOOLEAN, 0, true);
boolean @txvarval = cootx.GetVariableValue(#APPDUCXSAMPLE@200.200, 1);
// Using the current object
string @name = cooobj.GetName(cootx);
ObjectClass @objcls = cooobj.GetClass();
Object[] @orders = cooobj.APPDUCXSAMPLE@200.200:customerorders;
// Using the method context to call the super method
cooobj.CallMethod(cootx, coometh);
```

1.7.2 Popular Kernel Methods

The following table contains a list of implicit pseudo functions supported by app.ducx expression language. These functions can be invoked on an object.

Function	Description
`IsClass(class)`	`IsClass` determines whether the object class it is invoked on is derived from or identical to `class`.
`HasClass(class)`	`HasClass` determines whether the object is an instance of or derived from `class`.
`GetClass()`	`GetClass` returns the object class of the object.
`GetName(tx)`	`GetName` returns the *Name* (`COOSYSTEM@1.1:objname`) of the object.
`GetAddress()`	`GetAddress` returns the *Address* (`COOSYSTEM@1.1:objaddress`) of the object.
`GetReference()`	`GetReference` returns the *Reference* (`COOSYSTEM@1.1:reference`) of a component object.
`GetIdentification()`	`GetIdentification` returns the full identification of the object, which is a combination of the *Address* (`COOSYSTEM@1.1:objaddress`) and a version timestamp.

Example

```
@objectclass = cooobj.GetClass();
@objectname = cooobj.GetName(cootx);
@objectaddress = cooobj.GetAddress();
cooobj.HasClass(#APPDUCXSAMPLE@200.200:Order) ?
  cooobj.APPDUCXSAMPLE@200.200:ProcessOrder() : null;
```

1.7.3 Working With Contents

The following table lists methods that can be invoked on variables of type `CONTENT`.

Method	Description
`GetFile(name, generatetemp)`	`GetFile` copies the `CooContent` to a file and returns the file name. `name` denotes the name of the file to be created.
`SetFile(name, removeonrelease)`	`SetFile` stores the content of a file in a `CooContent`. `name` denotes the name of the file to be used as a source.
`GetContent(tx, flags, codepage)`	`GetContent` returns a `CooContent` as a `String` and can only be used for retrieving text-based contents.
`SetContent(tx, flags, codepage, string)`	`SetContent` stores a `String` in a `CooContent`.

Example

```
// Assuming that the global scope contains a dictionary
if (::HasEntry("description")) {
  // Initialize a new content
  content @description = {};
  // Set the string contained in ::description into the content
  // (use 65001 as "codepage" for UTF-8 encoding)
  @description.SetContent(cootx, 1, 65001, ::description);
  // Remove the description string from the global scope dictionary
  ::ClearEntry("description");
}
```

1.7.4 Working With Dictionaries

The following table lists methods that can be invoked on variables of type `DICTIONARY`.

Method	Description
`GetEntry(key)`	`GetEntry` returns the list of values stored under `key`. Use this method for retrieving lists from a dictionary.
`GetEntryValue(key)`	`GetEntryValue` returns value stored under `key`. Use this method for retrieving scalar values from a dictionary.
`GetEntryValueCount(key)`	`GetEntryValueCount` returns the number of values of the entry specified by `key`.

`GetEntryCount()`	`GetEntryCount` returns the number of entries in a dictionary.
`GetEntryKey(index)`	`GetEntryKey` returns the key of the entry of the specified `index`.
`SetEntry(key, values)`	`SetEntry` creates an entry under `key` for the specified `values`. Use this method for storing lists of values in a dictionary.
`SetEntryValue(key, inx, value)`	`SetEntryValue` creates an entry under `key` for the specified `value`. Use this method for storing a scalar value in a dictionary.
`TestEntry(key)`	`TestEntry` checks whether a dictionary contains an entry under `key`. This method returns `true` if the value stored under `key` is `null`.
`HasEntry(key)`	`HasEntry` checks whether a dictionary contains an entry under `key`. This method returns `false` if the value stored under `key` is `null`.
`ClearEntry(key)`	`ClearEntry` removes the entry stored under `key` from a dictionary.
`Reset()`	`Reset` removes all entries from a dictionary.
`Backup()`	`Backup` serializes the contents of a dictionary to a string.
`Restore(string)`	`Restore` rebuilds a dictionary from a serialized `string`.

1.7.5 Getting the Data Type of an Expression

The `typeof` function allows you to determine the data type of an expression. The result is a type or property definition object.

Example

```
// Determining the data type of the local scope
@localtype = typeof(this);
// Determining the data type of the global scope
@globaltype = typeof(::this);
// Determining the data type of a variable
@myvalue = "Hello world!";
@resulttype = typeof(@myvalue);
// Determining the data type of an expression
@resulttype = typeof(cooobj.APPDUCXSAMPLE@200.200:customerorders);
```

1.7.6 String Functions

The following table contains the list of string utility functions.

Function	Description
`upper(string)`	The `upper` function converts all characters of a string to upper case.
`lower(string)`	The `lower` function converts all characters of a string to lower case.
`indexof(string, pattern)`	The `indexof` function returns the character-based index of `pattern` within `string`. If the pattern is not found the function returns -1
`strlen(string)`	The `strlen` function returns the length of `string`.
`strtrim(string)`	The `strtrim` function trims white space at the beginning and at the end of `string`.
`strhead(string, index)`	The `strhead` function extracts the leftmost `index` characters from a string and returns the extracted substring. `index` is zero-based. If a negative value is supplied in `index`, absolute value of `index` is subtracted from the length of the string. If the absolute value of a negative `index` is larger than the length of the string, `0` is used for `index`.
`strtail(string, index)`	The `strtail` function extracts the characters from a string starting at the position specified by `index` and returns the extracted substring. `index` is zero-based. If a negative value is supplied in `index`, the absolute value of `index` is subtracted from the length of the string. If the absolute value of a negative `index` is larger than the length of the string, `0` is used for `index`.
`strsplit(string, separator)`	The `strsplit` function identifies the substrings in `string` that are delimited by `separator`, and returns a list containing the individual substrings.
`strjoin(list [, separator])`	The `strjoin` function concatenates the `list` of strings and inserts `separator` between the individual elements yielding a single concatenated string. If `separator` is not specified or `null` then the list elements are concatenated directly.
`strreplace(string, from [, to])`	The `strreplace` function replaces all occurrences of string `from` with string `to` in `string`. If `to` is not specified or `null` then all occurrences of `from` are deleted from the string.

Example

```
@value = strhead("ABC", 1);                    // yields "A"
@value = strtail("ABC", 1);                    // yields "BC"
@value = strsplit("ABC", "B");                 // yields ["A","C"]
@value = strjoin(["A", "B", "C"], "");         // yields "ABC"
```

```
@value = strjoin(strsplit("A-B-C", "-"), "+"); // yields "A+B+C"
@value = strreplace("ABC", "B", "");           // yields "AC"
@value = strreplace("ABCABC", "B", "X");       // yields "AXCAXC"
@value = strlen("ABC");                        // yields 3
```

In addition to the string functions provided by app.ducx expression language, the actions listed in the following table are useful for manipulating strings. For further information, refer to the Fabasoft reference documentation.

Function	Description
`COOSYSTEM@1.1:Format(value, pattern, symbols, result)`	This action takes a single value (any type) as first parameter and a formatting pattern as second parameter. The third parameter is for advanced options (code page, custom symbols for separators or the decimal point). The result is returned in the fourth parameter. Refer to the Fabasoft reference documentation for a description of the supported formatting patterns.
`COOSYSTEM@1.1:Print(string, ...)`	Processes a format string or prints the object to a resulting string. If the `string` parameter contains a non-empty format string, this is used regardless of the object of the action. If the object is a string object, the property `Print COOSYSTEM@1.1:string` is used as format string. If the object is an error message, the property `COOSYSTEM@1.1:errtext` is used as format string. In all other cases the name of the object is used as format string. If the string contains formatting patterns starting with the "%" character these patterns are replaced by the additional parameters of the `Print` action. Refer to the Fabasoft reference documentation for a description of the supported formatting patterns.
`COOSYSTEM@1.1:PrintEx(string, arguments)`	Uses `COOSYSTEM@1.1:Print` to print a format string or an object to a resulting string. Parameters for formatting are passed in a string list in the parameter `arguments`. Each line in the string list in the `arguments` parameter is evaluated as an expression.

Example

```
// Get the modification date of the object as formatted string
string @formattedstr = cooobj.Format(cooobj.objmodifiedat, "dT");
```

```
// Format an integer value with leading zeroes to yield "000123"
integer @intval = 123;
string @intvalstr = cooobj.Format(@intval, "000000");
// Format an integer value using digit grouping to yield "123,456"
integer @intval2 = 123456;
string @intvalstr2 = cooobj.Format(@intval2, "#,###");
// Get the object's name and subject as formatted string
string @titlestr = cooobj.Print("%s (%s)", cooobj.objname, cooobj.objsubject);
// Assuming that StrCustomFormat is a string object containing the format pattern
// "%s iteration %03d", the following expression will write the name of the
current object
// and the current loop iteration number padded with leading zeroes to the tracer
for (integer @idx = 0; @idx < 1000; @idx++) {
  %%TRACE(#StrCustomFormat.Print(null, cooobj.objname, @idx));
}
```

1.7.7 List Functions

The following table shows the list of functions provided for working with lists.

Function	Description
`count(list)`	The `count` function returns the number of elements in `list`. **Note:** Do not use `count(list) > 0` or `count(list) == 0` to check whether a list is full or empty. Use the Boolean type cast `list` or `!list` instead, this is much more efficient.
`insert(list, index, value)`	The `insert` function inserts `value` into `list` at position `index`. The parameter `list` is modified by that function; therefore, it must be assignable. `index` is zero-based. If a negative value is supplied in `index`, absolute value of `index` is subtracted from the length of the list. If the absolute value of a negative `index` is larger than the length of the list, `0` is used for `index`. If `index` is greater than the number of elements in `list`, `value` is appended at the end of `list`.
`delete(list, index [, count])`	The `delete` function deletes the value at `index` from `list`. The parameter `list` is modified by that function; therefore, it must be assignable. `index` is zero-based. If a negative value is supplied in `index`, absolute value of `index` is subtracted from the length of the list. If the absolute value of a negative `index` is larger than the length of the list, `0` is used for `index`. If `count` is not specified or `null` then one element is deleted, otherwise `count` specifies the number of elements following `index` that should be deleted. If less than `count` elements are available, the list is truncated at `index`. If `count` is negative, the elements before `index` are deleted.
`find(list, value)`	The `find` function searches `list` for the element `value`, and returns the index of the first occurrence within the entire list. If `value` is not found in `list`, the number of elements in `list` is returned.

`sort(list)`	The `sort` function sorts the elements in `list`. Lists of contents, dictionary or interfaces cannot be sorted. Lists of aggregates can only be sorted, if the aggregate type defines a compare action in `COOSYSTEM@1.1:typecompare`. Lists of objects are sorted by a defined internal order. This makes `sort` useful in combination with the `unique` function, since `sort(unique(objlist))` is much faster than `unique(objlist)`. When sorting aggregates, all properties of `COOSYSTEM@1.1:typesortattrs` are used for comparing in the listed order. If the aggregate contains additional properties in `COOSYSTEM@1.1:typecompattrs`, these properties are used as well. If the aggregate has no `COOSYSTEM@1.1:typesortattrs`, only the properties of `COOSYSTEM@1.1:typecompattrs` are used. Elements of type `COOSYSTEM@1.1:CONTENT` or `COOSYSTEM@1.1:COMINTERFACE` cannot be sorted or compared.
`unique(list)`	The `unique` function makes the elements in `list` unique.
`revert(list)`	The `revert` function reverts the elements in `list`.

Example

```
insert(@orders, count(@orders), @neworder);
delete(@orders, find(@orders, @canceledorder));
unique(sort(@orders));
```

1.7.8 Mathematical Functions

The following table shows the list of mathematical functions supported by app.ducx expression language.

Function	Description
`sum(list)`	The `sum` function returns the sum of all values in `list`. Values can also be passed to `sum` as individual arguments. The `sum` function can only be used with numeric data types.
`avg(list)`	The `avg` function returns the average of all values in `list`. Values can also be passed to `avg` as individual arguments. The `avg` function can only be used with numeric data types.
`min(list)`	The `min` function returns the smallest value in `list`. Values can also be passed to `min` as individual arguments. The `min` function can be used with strings and numeric data types.
`max(list)`	The `max` function returns the largest value in `list`. Values can also be passed to `max` as individual arguments. The `max` function can be used with strings and numeric data types.

Example

```
@avgorderamount = avg(@customer.APPDUCXSAMPLE@200.200:customerorders.
  APPDUCXSAMPLE@200.200:ordertotal.currvalue);
```

Additional mathematical functions are provided by the #Math object. For further information, see https://help.cloud.fabasoft.com/index.php?topic=doc/Reference-Documentation/class-fscexpext-MathFunctions.htm.

1.7.9 Escape Sequences for Special Characters

The following table contains a list of supported escape sequences for special characters.

Character	ASCII representation	ASCII value	Escape sequence
New line	NL (LF)	10 or 0x0a	\n
Horizontal tab	HT	9	\t
Vertical tab	VT	11 or 0x0b	\v
Backspace	BS	8	\b
Carriage return	CR	13 or 0x0D	\r
Form feed	FF	12 or 0x0C	\f
Alert	BEL	7	\a
Backslash	\	92 or 0x5C	\\
Question mark	?	63 or 0x3F	\?
Single quotation mark	'	39 or 0x27	\'
Double quotation mark	"	34 or 0x22	\"
Hexadecimal number	hh		\xhh
Null character	NUL	0	\0
Unicode character	hhhh		\uhhhh
Unicode sequence	hh hhhh hhh		\u{hh,hhhh,hhh}

Note: Please be aware that strings in expressions are UTF-8. If you use hexadecimal escape sequences be sure that the resulting byte sequence is a valid UTF-8 character.

Example

```
@fullname = "Samuel \"Sam\" Adams";
```

1.8 Getting and Setting Property Values

Syntax

```
// Getting property values using the assignment operator
variable = object.property;
// Setting property values using the assignment operator
object.property = expression;
// Setting values of a compound property list using the assignment operator
// The values are assigned based on their position
object.property = [{ val1, val2, ... }, { vala, valb, ... }, ...];
// Simple initialization of compound properties
CompoundType compoundtype = { val1, val2, ... };
CompoundType({ val1, val2, ... });
// Setting values of a compound property list using the assignment operator
// The values are assigned based on their position
object.property = [{ val1, val2, ...}, { vala, valb, ...}, ...];
// Setting values of a compound property list using the assignment operator
// The values are assigned based on the properties of the compound type
object.property = [{ reference1 = val1, reference2 = val2, ... }, ...];
// Setting values of a compound property list using the "JavaScript" notation
// The values are assigned based on the properties of the compound type
// The right sight
object.property = [{ reference1 : val1, reference2 : val2, ... }, ...];


// Getting property values using the Get... functions
variable = object.GetAttributeValue(transaction, property);
variable = object.GetAttribute(transaction, property);
variable = object.GetAttributeString(transaction, property, language);
variable = object.GetAttributeStringEx(transaction, property, language,
  additionallist, displayflags);
// Setting property values using the Set... functions
object.SetAttributeValue(transaction, property, index, value);
object.SetAttribute(transaction, property, value);
```

Note: You can set the value (`currvalue`) of a currency property (compound type) by just assigning a string, float or integer to the currency property (`currency @cur = 7.56;`). In this case it is not necessary to specify explicitly the `currvalue` property (`@cur.currvalue = 7.56;`).

You can use the assignment operator `=` to get and set the value of a property. Even though using the assignment operator is recommended, you may also use the following functions to get and set property values:

- The `GetAttributeValue` function is used for retrieving a scalar property value.
- The `GetAttribute` function is used for retrieving a list.
- The `GetAttributeString` function is used for retrieving a string representation of the property value. This function can be used for retrieving multilingual strings in the desired language.
- The `GetAttributeStringEx` function is used for retrieving a string representation of the property value. This function allows you to specify so-called display flags (see the following table) for the value to be retrieved.
- The `SetAttributeValue` function is used to store a scalar value in a property.

- The `SetAttribute` function is used to store a list.

Value	Description
`0x0001`	Object address
`0x0002`	Object reference
`0x0004`	Enumeration reference
`0x0010`	XML escaped
`0x0020`	HTML escaped
`0x1000`	Row list
`0x2000`	Column list
`0x8000`	No list indicator

Example

```
@orders = @customer.APPDUCXSAMPLE@200.200:customerorders;
@orders = @customer.GetAttribute(cootx, #APPDUCXSAMPLE@200.200:
  customerorders);
@orderdate = @order.GetAttributeValue(cootx, #APPDUCXSAMPLE@200.200:
  orderdate);
@order.APPDUCXSAMPLE@200.200:orderdate = coonow;
@order.SetAttributeValue(cootx, #APPDUCXSAMPLE@200.200:orderdate, 0,
  coonow);
@engname = @product.GetAttributeString(cootx, #mlname, #LANG_ENGLISH);
@gername = @product.GetAttributeString(cootx, #mlname, #LANG_GERMAN);
@description = @product.GetAttributeStringEx(cootx, #APPDUCXSAMPLE@200.200:
  productdescription, null, null, 0x1000);
```

1.9 Invoking Use Cases

Syntax

```
object.usecase(value, value,… , name: value, name: value, …)
```

A use case can only be invoked on an object. The full reference of the use case to be invoked must be provided, followed by the list of parameters, which must be enclosed in parentheses. Multiple parameters must be separated by commas.

You do not need to specify optional parameters. They can be omitted in the parameter list.

The parameter list consists of two parts. The first parameters are supplied by position. After that, `name:value` pairs can be written to set the parameters by name.

There are four methods for retrieving output parameters in app.ducx expression language:

- If the method supplies has a parameter which is marked as return value, the method call can be used like a function.

- If you need to retrieve only one output parameter, the selection operator `[]` specifying the result parameter position can be used.
- If you need to retrieve the method object of the call, the selection operator `[...]` can be used.
- If you need to retrieve multiple output parameters, variables must be specified for the output parameters in the parameter list and prefixed with an ampersand (`&`).

Example

```
// Optional parameters can be omitted
@order.COODESK@1.1:ShareObject(, , #APPDUCXSAMPLE@200.200:customerorders,
  @customer);
// "null" can also be passed in for optional parameters
@order.COODESK@1.1:ShareObject(null, null, #APPDUCXSAMPLE@200.200:
  customerorders, @customer);
// using parameter names
@order.COODESK@1.1:ShareObject(view:  #APPDUCXSAMPLE@200.200:
  customerorders, target: @customer);
// Retrieving an output parameter: method 1 - using the return value
@neworder = #APPDUCXSAMPLE@200.200:Order.ObjectCreate();
// Retrieving an output parameter: method 2 - specify the return value
@neworder = #APPDUCXSAMPLE@200.200:Order.ObjectCreate()[2];
// Retrieving an output parameter: method 3
method @meth = #APPDUCXSAMPLE@200.200:Order.ObjectCreate()[...];
@neworder = @meth.GetParameter(2);
// Retrieving an output parameter: method 4
#APPDUCXSAMPLE@200.200:Order.ObjectCreate(null, &@neworder);
```

1.10 Calculated Identifiers / Dynamic Invocation

Syntax

```
// Accessing a property using a calculated identifier
object.[expression]
// Invoking a use case using a calculated identifier
object.[expression](parameter, ...)
```

The selection operator `[]` can be used to specify an expression yielding a calculated identifier for accessing a property or invoking a use case.

For a calculated identifier, the expression specified in square brackets is evaluated, and then the result is interpreted as a property definition, entry key in a dictionary, an action or a use case.

Example

```
// Assigning a value to a calculated property
@attrdef = #APPDUCXSAMPLE@200.200:orderdate;
@order.[@attrdef] = coonow;
// Invoking a use case using a calculated identifier
@createinvoice = #APPDUCXSAMPLE@200.200:CreateInvoice;
@customer.[@createinvoice](@orders, &@invoice);
```

If the expression of the selection operator `[]` is a list of property definitions, then the list is interpreted as a path:

Example

```
// Accessing a value to a calculated property path
@attrdefs = [#objlock, #objlockedby, #objname]
string lockingusername = cooobj.[@attrdefs]
```

The selection operator `[]` can also be used with dictionaries, then the expression of the selection operator `[]` contains one or more strings identifying the entry keys in the dictionary.

Note:

- It is not possible to mix property definitions and strings in the expression of the selection operator `[]` to access object/aggregate properties and dictionary entries in one step.
- Only one action or use case can be specified, not a list.
- Dynamic invocation can also be used with the -> operator invoking applications or dialogs.

1.11 Accessing the Transaction Context

The `cootx` variable can be used to access the current transaction context. The most important methods of a transaction are listed in the following table.

Method	Description
`Abort()`	`Abort` aborts the current transaction and rolls back any changes.
`Commit()`	`Commit` closes the current transaction and stores any changes.
`CommitEx(flags)`	`CommitEx` closes the current transaction and stores any changes. Additionally, the following flags are supported: • `COOCF_NORMAL` Normal commit • `COOCF_KEEPONFAILURE` If commit fails, all transaction data is kept • `COOCF_KEEPREFRESHINFO` After commit the refresh info of all touched object is kept • `COOCF_KEEPSEARCHINFO` After commit the search info of the searches executed in this transaction is kept • `COOCF_KEEPVARIABLES` After commit all transaction variables are kept • `COOCF_KEEPLOCKS` After commit all locks of objects are kept • `COOCF_KEEPOBJECTS` After commit all modified objects are stored in

	the transaction variable `COOSYSTEM@1.1:TV_COMMITTEDOBJECTS` • `COOCF_NOTELESS` If specified the properties `COOSYSTEM@1.1:objmodifiedat` and `COOSYSTEM@1.1:objchangedby` are not set. This flag is only allowed, if the current user has the role `COOSYSTEM@1.1:SysAdm` and the current user is registered in `COOSYSTEM@1.1:domainmasterusers` of the current domain object
`Persist(object)`	`Persist` temporarily stores the state of the current transaction without committing the changes.
`Clone()`	`Clone` returns a clone of the current transaction.
`HasVariable(swc, id)`	`HasVariable` checks whether transaction variable `id` of software component `swc` contains a value. This method returns `false` if if the value stored is `null`.
`TestVariable(swc, id)`	`TestVariable` checks whether transaction variable `id` of software component `swc` contains a value. This method returns `true` even if the value stored is `null`.
`ClearVariable(swc, id)`	`ClearVariable` removes transaction variable `id` of software component `swc` from the transaction.
`GetVariable(swc, id)`	`GetVariable` retrieves the list of values stored in transaction variable `id` of software component `swc`.
`SetVariable(swc, id, type, values)`	`SetVariable` stores the specified `values` in transaction variable `id` of software component `swc`.
`GetVariableValueCount(swc, id)`	`GetVariableValueCount` returns the number of values stored in transaction variable `id` of software component `swc`.
`HasVariableValue(swc, id)`	`HasVariableValue` returns `true` if a transaction variable `id` of software component `swc` is available.
`GetVariableValue(swc, id)`	`GetVariableValue` retrieves a scalar `value` stored in transaction variable `id` of software component `swc`.
`SetVariableValue(swc, id, type, value)`	`SetVariableValue` stores the specified scalar `value` in transaction variable `id` of software component `swc`.
`GetVariableString`	`GetVariableString` retrieves a scalar `value` stored in transaction variable `id` of software component `swc` in printable form.

`GetVariableStringEx`	`GetVariableStringEx` retrieves a scalar `value` stored in transaction variable `id` of software component `swc` in printable form.
`GetVariableTypeDefinition(swc, id)`	`GetVariableTypeDefinition` returns the type definition for the variable stored in transaction variable `id` of software component `swc`.
`IsClone()`	`IsClone` returns `true` if the transaction is a clone transaction.
`IsModified()`	`IsModified` checks whether objects were modified within the transaction.
`IsModifiedEx()`	`IsModifiedEx` checks whether any data was modified within the transaction.
`IsCreated(object)`	`IsCreated` checks whether `object` was created in this transaction.
`IsDeleted(object)`	`IsDeleted` checks whether `object` was deleted in this transaction.
`IsChanged(object)`	`IsChanged` checks whether `object` was changed in this transaction.
`IsAttributeChanged(object, property)`	`IsAttributeChanged` checks whether `property` of `object` was changed in this transaction.
`GetTransactionFlags()`	`GetTransactionFlags` retrieves the flags of the transaction: • `COOTXF_ROLECHANGED` During the transaction an automatic role change has been performed • `COOTXF_NOREFRESH` Objects are not automatically refreshed when accessed with this transaction • `COOTXF_NOAUTOVERSION` During commit of the transaction no automatic version will be created
`SetTransactionFlags(flags)`	`SetTransactionFlags` allows you to set the transaction flags (see G`etTransactionFlags)`.
`Backup()`	`Backup` returns a dump of the transaction as printable string.
`Restore(data)`	`Restore` restores the transaction from `data` created by `Backup()`.

`OpenScope()`	`OpenScope` starts a new transaction scope.
`CloseScope()`	`CloseScope` closes the current transaction scope.
`GetMaster()`	`GetMaster` returns the top most transaction of the transaction.

In some scenarios it is necessary to carry out operations in a separate transaction. Any changes that have been made in a new transaction can be committed or rolled back separately from the main transaction.

Example

```
usecase CreateInvoice() {
  variant Order {
    impl = expression {
      // Create a new transaction
      interface @extension = coort.GetExtension();
      transaction @localtx = @extension.CreateTransaction();
      transaction @backuptx = cootx;
      try {
        coort.SetThreadTransaction(@localtx);
        Invoice @invoice = #Invoice.ObjectCreate();
        @invoice.APPDUCXSAMPLE_200_300_InitializeInvoice();
        @localtx.Commit();
      }
      catch (error) {
        @localtx.Abort();
      }
      finally {
        // Restore original transaction context
        coort.SetThreadTransaction(@backuptx);
        // Clear the variables holding the transactions
        @backuptx = null;
        @localtx = null;
      }
    }
  }
}
```

Note: A better and simpler way to create transactions is using the `try new transaction` statement.

Working With Transaction Variables

Syntax

```
// Retrieving the value stored in a transaction variable
value = #TV.reference;
// Storing a value in a transaction variable
#TV.reference = value;
```

The `#TV` object is a special object that provides access to transaction variables.

Note: Transaction variables can also be accessed using the `cootx` variable. Please refer to chapter 1.7.1 “Predefined Variables” for further information.

Example

```
// Retrieving the value stored in a transaction variable
@printinvoice = #TV.TV_PRINTINVOICE;
// Storing a value in a transaction variable
#TV.TV_INVOICE = cooobj;
```

1.12 Control Flow

The app.ducx expression language supports common language constructs for control flow and expression evaluation.

1.12.1 Conditions

Syntax

```
if (expression) {
  ...
}
else if (expression) {
  ...
}
else {
  ...
}

switch (expression) {
case constant:
  ...
  break;
case constant:
  ...
  break;
default:
  ...
  break;
}
```

You can use `if` statements in app.ducx expression language. The `if` keyword must be followed by parentheses enclosing a conditional expression, and non-optional curly braces. An `if` block can be followed by multiple `else if` blocks and an optional `else` block.

Example

```
@orderstate = @order.orderstate;
if (@orderstate == OrderState(OS_PENDING)) {
  @order.ProcessPendingOrder();
}
else if (@orderstate == OrderState(OS_SHIPPED)) {
  @order.ProcessShippedOrder();
}
else {
  throw #OrderAlreadyProcessed;
}
// lists as conditional expression are evaluated true, if the list contains at
least
// one not null element
if (["", "", "a"]) {
  true;
}
```

Note: It is not necessary that `OS_PENDING` is explicitly casted (e.g. `@orderstate == OS_PENDING` works, too).

The `switch` - `case` - `default` statement can be used to evaluate the switch expression and execute the appropriate case.

Example

```
OrderState @orderstate;
switch (@orderstate){
case OS_PENDING:
  @state = 1;
  break;
case OS_SHIPPED:
  @state = 2;
  break;
default:
  @state = 0;
  break;
}
```

Note: Enumeration items like `OS_PENDING` are determined by `@orderstate` in the `switch` statement.

1.12.2 Loops

Syntax

```
for (expression) {
  ...
}
while (expression) {
  ...
}
do {
  ...
} while (expression);
```

The app.ducx expression language contains loops with both a constant and a non-constant number of iterations. The statements `for`, `while`, and `do-while` are supported. All loops have a mandatory block.

The `break` statement can be used to exit a loop.

The `continue` statement can be used to skip the remainder of the loop body and continue with the next iteration of the loop.

Note: `continue` inside a catch block does not apply to any enclosing loop.

Example

```
currency totalvalue = 0;
object[] orderpositions;
// Iteration with index (inefficient)
for (integer idx = 0; idx < count(orderpositions); idx++) {
  Product product = positions[idx].product;
  totalvalue += product.unitprice * positions[idx].quantity;
}
currency total = 0;
// Iteration with iterator (efficient)
```

```
for (OrderPosition position : orderpositions) {
  Product product = position.product;
  if (product != null) {
    total += product.unitprice * position.quantity;
  }
}
// Iteration of a searchresult
searchresult result = FROM User;
string[] names;
for (User u : result) {
  names += u.userfirstname;
}
integer stock = product.itemsinstock;
integer threshold = product.productionthreshold;
while (stock <= threshold) {
  product.ProduceItem();
  stock++;
}
OrderState orderstate;
do {
  order.ProcessOrder(&orderstate);
} while (@orderstate != OS_COMPLETED);
```

Note: Avoid iterating through lists using a `for` loop with an index, since this is very inefficient in large lists.

1.12.3 Exceptions and Error Handling

Exceptions are used to handle exceptional or unexpected situations that may occur during the execution of a program. When an error situation arises, the code can throw an exception to indicate that something unexpected has occurred. This allows the kernel to transfer control to an appropriate exception handler, which can handle the exception gracefully.

Use `try-catch-finally` statements to handle exceptions. A `try` block must be followed by at least one `catch` block, followed by an optional `finally` block.

If an exception occurs when processing the `try` block, the kernel tries to locate a `catch` block with a condition matching the error message object of the exception. If no matching `catch` block is found by the kernel, the error text is handled by default (e.g. shown in the UI).

A `catch` block can have three distinct types of conditions:

- An error message object can be specified to handle only matching exceptions.
- A variable of type `integer` can be specified. If an exception occurs, the error code is stored in the specified variable, and the corresponding `catch` block is executed. The variable can be used to access the error text (`coort.GetErrorText(errorcode)`) and the corresponding error message (`coort.GetErrorMessage(errorcode)`).
- The `...` operator can be specified to handle all exceptions without considering the error code.

The optional `finally` block is executed after the `try` and `catch` blocks have been processed, whether an exception occurred or not.

Example

```
try {
  User usr = #User.ObjectCreate(,,newaddress);
}
catch(#COOERR_INVADDR) {
```

```
  /*
   * handle an invalid object address format
   */
}
catch(errorcode) {
  /*
   * handle any other error raised during object creation
   */
  string errortext = coort.GetErrorText(errorcode);
  ErrorMessage errorobject = coort.GetErrorMessage(errorcode);
}
finally {
  ...
}
```

Using the `throw` keyword, the app.ducx expression language allows you to raise an error:

- To rethrow an exception in an error handler (e.g. in a `catch` block), you just need to specify the error code of the exception after the `throw` keyword.
- To raise a custom error, specify the error object and the arguments. If arguments are listed, the action `COOSYSTEM@1.1:RaiseError` is called, which allows you to format the message text.

Syntax

```
// Raising a custom error
throw errormessage, argument;

// Rethrowing an exception
throw errorcode;
```

Example

```
// Raising a custom error using a "throw" statement
throw #InvalidInvoice;

// Raising a custom error, assuming that the
// error message APPDUCXSAMPLE@200.200:InvalidInvoice contains a formatting
pattern like
// "Invoice '%s' (no. %d) is not valid!"
throw #InvalidInvoice, cooobj.objname, cooobj.invoicenumber;
```

To continue the execution directly after the statement raising the error the keyword `continue` can be used inside the `catch` block.

1.12.4 Creating New Transactions or Opening a Transaction Scope

Similar to the `try` statement for error handling it is possible to execute a block using a separate transaction context:

Example

```
try new transaction {
  // The statements in this block are executed in a new transaction context.
  // This new transaction context is also available in the cootx built-in
variable.
  Invoice @invoice = #Invoice.ObjectCreate();
  @invoice.APPDUCXSAMPLE_200_300_InitializeInvoice();
}
// After the try block an implicit Commit() or Abort() is executed on the new
// transaction context created for the try block.
```

```
// If the code in the try block throws an exception, Abort() is called, else
// Commit().
// If the implicit Commit() itself throws an exception this can be handeled by
// the catch block below.
catch (...) {
  // Here you can catch exceptions that occurred during the try block or during
the
  // implicit Commit() after the try block.
  // This code is executed in the original transaction context so the
  // changes made during the try block are not available any more.
}
finally {
  // Here you can perform some additional cleanup.
  // This code is executed in the original transaction context.
}
```

If the `new` keyword is omitted, a transaction scope is opened. A transaction scope is a sub transaction of the current transaction. When a transaction scope is committed, the changes of that scope are propagated to the surrounding transaction. These changes are only persisted if the surrounding transaction context is committed.

Example

```
try transaction {
  // The statements in this block are executed in a new transaction scope.
  // This new transaction scope is also available in the cootx built-in
variable.
  Invoice @invoice = #Invoice.ObjectCreate();
  @invoice.APPDUCXSAMPLE_200_300_InitializeInvoice();
}
// After the try block an implicit Commit() or Abort() is executed on the new
// transaction scope created for the try block.
// If the code in the try block throws an exception, Abort() is called, else
// Commit().
// Afterwards, the transaction scope is closed.
// If the implicit Commit() itself throws an exception this can be handeled by
// the catch block below.
catch (...) {
  // Here you can catch exceptions that occurred during the try block or during
the
  // implicit Commit() after the try block.
  // This code is executed in the original transaction scope so the
  // changes made during the try block are not available any more.
}
finally {
  // Here you can perform some additional cleanup.
  // This code is executed in the origina transaction scope.
}
```

1.12.5 Returning Values

Syntax

```
return expression;
```

The `return` statement can be used to stop the evaluation of an expression at any time. Each expression has a return value, which is calculated by the expression following the `return` keyword.

Note: It is not allowed to use `return` with a value in a method implementation.

Example

```
if (@order != null) {
  return @order;
}
```

1.12.6 Directives

A directive is a special statement that does not influence the semantic of the expression.

Syntax

```
%%NAME(parameters);
```

1.12.6.1 %%TRACE

The `%%TRACE` directive can be used to conditionally write trace messages to the Fabasoft app.ducx Tracer.

Syntax

```
%%TRACE(message);
%%TRACE(value);
%%TRACE(message, value);
```

Example

```
%%TRACE("Hello World!");
%%TRACE(cooobj);
%%TRACE("Current Object", cooobj);
%%TRACE(cooobj.objname + " locked?", cooobj.objlock.objlocked);
```

1.12.6.2 %%FAIL

The `%%FAIL` directive can be used to generate a failure. The message is written to the Fabasoft app.ducx Tracer and an error (`EXPRERR_FAIL`) is raised.

Note: Like the `%%TRACE` directive, the `%%FAIL` directive is only evaluated if trace mode is activated for your software component.

Syntax

```
%%FAIL;
%%FAIL(message);
```

Example

```
%%FAIL;
%%FAIL("Unexpected!");
```

1.12.6.3 %%ASSERT

The %%ASSERT directive can be used to check conditions. In case the condition returns false, a message is written to the Fabasoft app.ducx Tracer and an error (EXPRERR_ASSERT) is raised.

Note: Like the %%TRACE directive, the %%ASSERT directive is only evaluated if trace mode is activated for your software component.

Syntax

```
%%ASSERT(condition);
%%ASSERT(message, condition);
%%ASSERT(message, expectedvalue, actualvalue);
```

Example

```
%%ASSERT(cooobj.objlock.objlocked);
%%ASSERT("'cooobj' should not be locked.", cooobj.objlock.objlocked);
@expect = "Test";
@actual = cooobj.objname;
%%ASSERT(@expected != @actual);
%%ASSERT("Expecting " + @expect + ", but actual value is '" + @actual +
  "'.", @expect, @actual);
```

1.12.6.4 %%DEBUGGER

The %%DEBUGGER directive can be used to set a breakpoint in a Fabasoft app.ducx Expression.

Syntax

```
%%DEBUGGER;
```

1.12.6.5 %%LOG

The %%LOG directive can be used to log messages to Fabasoft app.telemetry. app.telemetry provides the log level LOG, IPC, NORMAL, DETAIL, and DEBUG.

Syntax

```
%%LOG(message);
%%LOG(level, message);
```

Example

```
%%LOG("Object created by " + cooobj.objcreatedby.objname); // Detail level
%%LOG("DEBUG", "Object created by " + cooobj.objcreatedby.objname); // Debug
level
```

1.13 Searching for Objects – app.ducx Integrated Query

app.ducx Integrated Queries can be used to search for objects. Integrated queries allow you to formulate your queries directly in the expression language. Integrated queries are composed of a series of clauses that define the data source and conditions to filter it. The basic clauses are:

- `FROM`
  This clause specifies the data source to be queried, and may introduce a condition variable to represent each element in the data source.
- `WHERE`
  This clause is optional and filters the data source based on a specified condition, and returns only those elements that satisfy the condition.
- `SELECT`
  This clause is optional and specifies which properties of the resulting objects should be loaded into the cache. Please be aware that omitting the `SELECT` clause will cause the runtime to only load the `COOSYSTEM@1.1:objclass` property into the cache.

Note: Integrated queries result in an asynchronous search, which means the result is of type `searchresult`. You can access `searchresult` either using the kernel interface (https://help.cloud.fabasoft.com/index.php?topic=doc/Reference-Documentation/interface-CooSearchResult.htm) or by iteration (see chapter 1.12.2 “Loops”).

A complete reference of the grammar can be found in chapter 1.14 “Grammar of the app.ducx Expression Language”.

1.13.1 FROM Clause

The `FROM` clause is the sole mandatory element in integrated queries. Below is an example of the basic structure of a query to retrieve all users of a domain:

Example

```
searchresult allusers = FROM User
```

You can also retrieve objects of multiple classes, as demonstrated in the following example:

Example

```
searchresult allusersandgroups = FROM User, Group
```

Additionally, you can query object list properties using the keyword `IN`, as demonstrated in the example below.

Example

```
searchresult folders = FROM COODESK@1.1:Folder IN cooroot.objchildren
```

Specifying the object class in the `FROM` clause limits the result to objects of only that class. For example, if you specify `COODESK@1.1:Folder` as the object class in the `FROM` clause, the query result will only contain objects of that class (including derived classes). You can use the property `COOSYSTEM@1.1:objclass` in the `WHERE` clause to exclude derived classes.

1.13.2 WHERE Clause

The `WHERE` clause follows the `FROM` clause and makes it possible to filter objects using Boolean conditions. In order to utilize `WHERE` clauses you will need to introduce a condition variable (e.g. "u" in the example below) within the `FROM` clause that represents each element in the data source. You can then use the condition variable within the `WHERE` clause to access properties.

Example

```
searchresult allusers = FROM User u WHERE u.active
searchresult someusers = FROM User u WHERE u.userfirstname == "John"
```

Note: It is not possible to reference properties of condition variables which specify `AttrNoSearchPossible`. Further, it is not possible to access properties of objects retrieved from condition variables (e.g. `u.objowner.objname`).

1.13.3 SELECT Clause

In integrated queries, the `SELECT` clause precedes the `FROM` clause. The `SELECT` clause is used to specify properties that should be loaded into the cache. Per default, if you omit the `SELECT` clause, only the value of `COOSYSTEM@1.1:objclass` will be loaded.

Example

```
searchresult u1 = SELECT * FROM User u WHERE u.active;
searchresult u2 = SELECT userfirstname, usersurname FROM User u WHERE u.active;
```

1.13.4 Options

You can specify options at the beginning of an integrated query to restrict the search. Integrated queries support the following options:

- `LIMIT`
  Restricts the search result to the defined number of objects.
- `IGNORECASE`
  A case-insensitive search is carried out, even if the search is configured as case-sensitive in the domain and database.
- `SCOPE`
  The scope allows to define a query scope object (reference or object address) that defines the location the search is carried out.
  Example: `SCOPE(#COOSYSTEM@1.1:LoginQuery), SCOPE('COO.1.1.1.2686')`

Example

```
searchresult al10 = LIMIT 10 SELECT * FROM User u WHERE u.active
```

1.14 Grammar of the app.ducx Expression Language

The grammar of the app.ducx Expression Language is formally defined as described below.

Grammar

```
Expression := { ( [ Statement ] ";" ) | BlockStatement } [ Statement ].
Statement := VarDecl | Assume | StatementExpression | Directive |
  "break" | "continue".
BlockStatement := If | While | DoWhile | For | Switch | Try | SimpleBlock.
VarDecl := Datatype [ "[]" ] [ "@" | "::" ] [ "$" ] Name [ "=" XpExpression ].
Datatype := Identifier.
Assume := "assume" [ "optional" ] [ "in" | "out" | "ref" ]
  Datatype [ "[]" ] [ "@" | "::" ] [ "$" ] { Identifier "." } Name.
StatementExpression := XpExpression | Return | Throw.
Return := "return" XpExpression.
Throw := "throw" Argument { "," Argument }.
Directive := "%%" Identifier [ Arguments | XpExpression ].
If := "if" "(" XpExpression ")" "{" Expression "}"
```

```
  [ "else" ( If | ( "{" Expression "}" ) ) ].
While := "while" "(" XpExpression ")" "{" Expression "}".
DoWhile := "do" "{" Expression "}" "while" "(" XpExpression ")".
For := "for" "(" [ VarDecl | XpExpression ]
  ( ( ";" [ XpExpression ] ";" [XpExpression] ) | ( ":" XpExpression ) ) ")"
  "{" Expression "}".
Switch := "switch" "(" XpExpression ")" "{" { Case } "}".
Case := ( "default" | ( "case" XpExpression ) ) ":" [ Expression ].
Try := "try" [ "new" "transaction" ] "{" Expression "}" { CatchBlock }
  [ "finally" "{" Expression "}" ].
CatchBlock := "catch" "(" ( ( [ ":>" | "::" | "@" ] [ "$" ] Name ) |
  Primary | "..." ) ")"
  "{" Expression "}".
SimpleBlock := "{" Expression "}".
XpExpression := Assignment | Query.
Query := [ "LIMIT" Assignment | "SCOPE" "(" Assignment ")" | "IGNORECASE" ]
  [ "SELECT" ("*" | Identifier { "," Identifier}) ]
  "FROM" Identifier { "," Identifier } [ Name ]
  [ "IN" MemberOperation ]
  [ "WHERE" Or ].
AssignOp := "=" | "&=" | "^=" | "|=" | "+=" | "-=" |
  "*=" |"/=" | "%=" | "<<=" | ">>=" | "??=".
Assignment := Ternary [ AssignOp Query ].
Ternary := Coalesce { "?" XpExpression ":" XpExpression }.
Coalesce := Or { "??" Or }.
Or := And { ("or" | "OR" | "||") And }.
And := Equal { ("and" | "AND" | "&&" ) Equal }.
Equal := Comparison { ("==" | "!=" | "<>") Comparison }.
CompOp := "<" | ">" | "<=" | ">=" | "<=>" |
  ( [ "sounds" | "SOUNDS" ] [ "not" | "NOT" ] ( "like" | "LIKE" ) |
  ( [ "not" | "NOT" ] ( "in" | "contains" | "includes" ) ) |
  ( [ "not" | "NOT" ] ( "IN" | "CONTAINS" | "INCLUDES" ) ) |
  "is" | "IS".
Comparison := Bit { ( CompOp Bit ) |
  ( [ "not" | "NOT" ] ( "between" | "BETWEEN" ) Bit ( "and" | "AND" ) Bit ) }.
Bit := Add { ( '|' | '&' | '^' | '<<' | '>>' ) Add }.
Add := Mul { ( "-" | "+" ) Mul }.
Mul := Unary { ( "*" | "/" | "%" ) Unary }.
UnaryOp = "!" | "not" | "NOT" | "-" | "+" | "&" | "~".
Unary := ( UnaryOp Unary ) | Prefix.
Prefix := [ "++" | "--" ] Postfix.
Postfix := MemberOperation [ "++" | "--" ].
MemberOperation := Primary { ( "." | "->" ) Selector }.
Primary := ( Literal | Detach | Reference | Cast | ArrayInit | StructInit |
  "(" XpExpression ")" ) [ IndexOperations ].
Detach := "->" ( Identifier | ( "[" XpExpression "]" ) ) Arguments.
Cast := Identifier [ "[]" ] Arguments.
Reference := [ ":>" | "::" | "@" ] [ "$" ] Identifier.
Selector := ( ( "#" "$" Identifier) | "[" XpExpression "]" ) [ Arguments ]
  [ IndexOperations ].
Literal := Integer | Float | Hex | Date | String | StringTemplate | Null | Bool |
  ObjectLiteral | Address.
IndexOperations := { "[" IndexOperation "]" }.
IndexOperation := Slice | XpExpression | "...".
Arguments := "(" [ Argument { "," [ Argument ] } ] ")".
Argument := XpExpression | NamedArgument.
NamedArgument := Identifier ":" Argument.
Integer := Digit { Digit }.
Float := Digit { Digit } "." Digit { Digit }.
Hex := ( "0x" | "0X" ) HexDigit { HexDigit }.
String := ( "'" { Char } "'" ) | ( '"' { Char } '"' ).
StringTemplate := '$"' { ( "{~" XpExpression "~}" ) | Char } '"'.
Null := "null" | "NULL".
Bool := "false" | "true".
ObjectLiteral := "#" ( Identifier | Component ).
Component := "SWC" | ( Identifier "@" Integer "." Integer ).
Address := "COO." Integer "." Integer "." Integer "." Integer [ "@" Date ].
```

```
Slice := [ XpExpression ] ":" [ XpExpression ].
Digit := "0" .. "9".
Digits := Digit { Digit }.
HexDigit := Digit | "a" .. "f" | "A" .. "F".
Identifier := [ Component ":" ] Name.
Name := ( 'a'..'z' | 'A'..'Z' | '_' ) { 'a'..'z' | 'A'..'Z' | '_' | '0'..'9' }.
Date := Digits "-" Digits "-" Digits
  [ ( "T" | " " ) Digits [ ":" Digits ] [ ":" Digits ] ].
```

In general, the rule `Char` represents all printable characters and the following escape sequences:
`'\\' ('a' | 'b' | 'f' | 'n' | 'r' | 't' | 'v' | 'u' | 'x' | '"' | "'" |'\\')`

Moreover, the start- and end symbols (e.g., `'"'`, `"'"`) used for strings must be escaped in order to use them as character values inside strings. Additionally, `"{"` must be escaped inside string templates.

The following reserved words cannot be used as variable names.

Reserved Words

```
AND, and, ASC, assume, BETWEEN, between, break, BY, case,
catch, contains, CONTAINS, continue, default, DESC, do,
else, false, finally, for, FROM, if, IGNORECASE, import,
in, IN, includes, INCLUDES, is, IS, like, LIKE, LIMIT,
new, NOT, not, NULL, null, optional, or, OR, ORDER, out, ref,
return, SCOPE, SELECT, SOUNDS, sounds, switch, throw,
TIMEOUT, transaction, true, try, WHERE, while, yield
```

1.15 Kernel Interfaces: Searching for Objects

The Kernel Interface Query Language can be used to search for objects. To carry out a search the runtime methods `SearchObjects` and `SearchObjectsAsync` can be used. `SearchObjects` returns the search result array at once (10,000 objects at the maximum) whereas `SearchObjectsAsync` returns a `searchresult`, which can be used to step through the result (without limit). Additionally, the runtime method `SearchValues` can be used. `SearchValues` returns an aggregated value using `COUNT`, `SUM`, `MIN` or `MAX`. Using `SearchValues`, the evaluation of the query conditions occurs only in the database. For security reasons this method is only available for privileged users.

The following example shows a Fabasoft app.ducx expression that illustrates how to search for orders at once and asynchronously.

Example

```
integer @bulksize = 150;
string @query = "SELECT objname FROM APPDUCXSAMPLE@200.200:Order";
// Performs a search with SearchObjects
Order[] @results = coort.SearchObjects(cootx, @query);
%%TRACE("Number of hits", count(@results));
// Performs an asynchronous search with SearchObjectsAsync()
searchresult @sr = coort.SearchObjectsAsync(cootx, @query);
Order[] @resultsasync = null;
// Steps through the search result
while ((@resultsasync = @sr.GetObjects(@bulksize)) != null) {
  %%TRACE("Fetched chunk of search results", @resultsAsync);
  for (Order @order : @resultsasync) {
    %%TRACE("Result entry", @order.objaddress);
```

```
  }
}
// Count objects with SearchValues
@query = "SELECT COUNT(*) FROM APPDUCXSAMPLE@200.200:Order";
integer @objcnt = coort.SearchValues(cootx, @query);
```

A search query is built up by following parts:

- Options (optional)
  Options can be used to restrict the search.
- `SELECT` clause
  In the `SELECT` clause properties can be defined which should be loaded in the cache.
- `FROM` clause
  Defines the object classes that should be searched for.
- `IN` clause (optional)
  Defines an object list that should be searched instead of the whole database.
- `WHERE` clause (optional)
  The `WHERE` clause is used to restrict the search result by defining conditions.

Syntax

```
{Options} SELECT Properties FROM Classes [IN ObjectList] [WHERE Condition]
```

A complete reference of the grammar can be found in chapter 1.15.8 “Grammar of the Kernel Interfaces Query Language”.

1.15.1 Options

In most cases, no options will be required.

- `LIMIT`
  Restricts the search result to the defined number of objects. This setting can only be used with `SearchObjects`. The maximum value is 10,000.
- `PRELOAD`
  In case of an asynchronous search the `PRELOAD` value defines how many objects are fetched in advance when stepping through the search result.
- `TIMEOUT`
  Restricts the search time to the specified value (seconds).
  Example: `TIMEOUT 3`
- `NOCHECK`
  By default it is checked whether the defined properties in the SELECT clause belong to the object classes in the FROM clause. This option disables the check.
- `NOEXEC`
  Only a syntax check of the search query takes place, but the search itself gets not executed.
- `NOHITPROPERTIES`
  In case of a full-text search several hit properties (hit rank, hit count, hit display) may be displayed in the search result. With this option no hit properties are returned.
  Note: A full-text search is triggered when using `CONTAINS` or `LIKE '%%something'` in the `WHERE` clause.
- `HITPROPERTIES`
  In case of a full-text search hit properties (`COOSYSTEM@1.1:contenthitrank`,

`COOSYSTEM@1.1:contenthitcount`, `COOSYSTEM@1.1:contenthitdisplay`) can be displayed in the search result. This option can be used to define which hit properties are returned.
Example: `HITPROPERTIES(COOSYSTEM@1.1:contenthitrank)`

- `IGNORECASE`
  A case-insensitive search is carried out, even if the search is configured as case-sensitive in the domain and database.
- Location
  If no location is specified the search is carried out in the COO stores of the user's local domain.
  - `LOCAL`
    Restricts the search to the COO stores of the user's local domain.
  - `GLOBAL`
    The search is carried out in all known domains.
  - `DOMAINS`
    Restricts the search to the defined domains (list of addresses of the domain objects).
    Example: `DOMAINS('COO.200.200.1.1','COO.200.200.1.7')`
  - `CACHE`
    Restricts the search to the kernel cache.
  - `TRANSACTION`
    Restricts the search to objects belonging to the current transaction.
  - `SCOPE`
    The scope allows to define a query scope object (reference or object address) that defines the location the search is carried out.
    Examples: `SCOPE(#LoginQuery)` / `SCOPE('COO.1.1.1.2686')`
  - `SERVICES`
    Restricts the search to the defined COO services.
  - `STORES`
    Restricts the search to the defined COO stores.
  - `ARCHIVES`
    Restricts the search to the defined archive stores.

1.15.2 Properties

It is useful to define properties that are accessed later on because these properties are loaded in the cache. When accessing these objects, no further server request is necessary to read the defined properties.
`SELECT *` loads all properties in the cache and therefore should only be used if many properties are used further on.

1.15.3 Classes

Objects of the defined object classes (and derived object classes) are searched. If derived object classes should not be found use the property `objclass` in the `WHERE` clause.

Example

```
SELECT objname FROM User WHERE .objclass = #User
```

1.15.4 Object List

In case an `IN` clause is supplied the search is restricted to the objects which are contained in the list. The conditions are evaluated within the database, the `FROM` clause is optional. This feature can be used to filter object lists without loading all objects.

Example

```
// Returns all children starting with the letter A.
SELECT objname IN COO.1.2.3.4.objchildren WHERE .objname LIKE 'A%'
// Returns all child folders.
SELECT objname FROM COODESK@1.1:Folder IN COO.1.2.3.4.objchildren
```

Restrictions: The `CACHE` keyword is not supported for list queries, this also means that only objects stored within the COO Service can be found. In case of a container-based installation, the component objects are stored within the Kernel cache of the container image, and therefore also cannot be found.

1.15.5 Condition

Supplying values for properties restricts the results further. Following general rules apply:

- Fully qualified references are used to define the properties. `COOSYSTEM@1.1` may be omitted for properties belonging to this software component.
- It is good practice to start the reference with a period to make clear that the property belongs directly to the object and is not part of a compound type.
- Compound types can be accessed using a property path.
  Example: `.COOMAPI@1.1:emailinformation.COOMAPI@1.1:emailaddress`
- As object constants use the object addresses.
- String constants are defined with double quotes " or single quotes '. Special characters like " and ' can be escaped with a backslash \.
- Date/Times have to be provided this way: `yyyy-mm-dd hh:mm:ss`
- Dates can also be provided using the short form: `yyyy-mm-dd`
- Expression keywords can be used as values
  Example: `.objowner = coouser`

Following keywords can be used to specify a condition:

- `NOT`
  The expression yields the value `true` if the operand evaluates to `false`, and yields the value `false` if the operand evaluates to `true`.
- `AND`
  Indicates whether both operands are true.
- `OR`
  Indicates whether either operand is true.
- <, <=, >, >=, =, <>
  Compares two operands: less, less equal, greater, greater equal, equal, not equal
- `[SOUNDS] [NOT] LIKE`
  `LIKE` determines whether the left string matches the right string. The `%`, `*`, `?`, and _ wildcards can be used in the right string operand and cannot be escaped. The `LIKE` operator can be preceded by the `SOUNDS` keyword for a phonetic comparison.

Example: `WHERE COOMAPI@1.1:emailinformation.COOMAPI@1.1:emailaddress LIKE "*fabasoft.com"`

- `[NOT] CONTAINS`
  Triggers a full text search.
  Example: `WHERE COOSYSTEM@1.1:contcontent CONTAINS 'Workflow'`
- `[NOT] IN`
  Determines whether the value is in the defined list.
- `[NOT] INCLUDES`
  Determines whether the value of the right operand is an element of the list provided in the left operand.
- `[NOT] BETWEEN ... AND ...`
  Determines whether the value is between the specified boundaries.
- `IS [NOT] NULL`
  Determines whether the property has a value.
- `UPPER`
  Converts all characters of a property to upper case (string data type).
- `LOWER`
  Converts all characters of a property to lower case (string data type).
- `SUM`
  Calculates the sum of all property values (numeric data type).
- `AVG`
  Calculates the average of all property values (numeric data type).
- `COUNT`
  Calculates the number of elements of a property (any data type).
- `MIN`
  Calculates the smallest value of all property values (numeric, string, date data type).
- `MAX`
  Calculates the largest value of all property values (numeric, string, date data type).

1.15.6 Search Query Examples

The following example shows a variety of search queries.

Example

```
// Returns all Note objects
SELECT objname FROM NOTE@1.1:NoteObject
// Returns contact persons with "Jefferson" in COOSYSTEM@1.1:usersurname
SELECT objname FROM FSCFOLIO@1.1001:ContactPerson WHERE .usersurname =
'Jefferson'
// The settings in the query scope object restrict the search
// Account objects are returned that reference the current user as owner
SCOPE (#FSCFOLIOCRM@1.1001:CRMQueryScope)
SELECT * FROM FSCFOLIOCRM@1.1001:Account WHERE .objowner = coouser
// The search is restricted to the domain with object address COO.1.1900.1.1
DOMAINS ('COO.1.1900.1.1') SELECT .objname FROM CurrentDomain
// Returns users that are created between the last hour and last half-hour
SELECT objname FROM User
WHERE (.objcreatedat >= coonow-60*60) AND (.objcreatedat < coonow-30*60)
// Returns users with a task in the task list
SELECT objname FROM User WHERE .COOAT@1.1001:attasklist IS NOT NULL
```

```
// A query scope object is used and the search is restricted to 100 result
entries
SCOPE (#FSCLEGALHOLD@1.1001:LegalHoldQueryScope)
LIMIT 100 SELECT objname FROM Object
WHERE .FSCLEGALHOLD@1.1001:objlegalholds.objowner = coouser
```

The following example shows a value query using `SearchValues`.

Example

```
// Returns the biggest content size of all content objects
coort.SearchValues(cootx, "SELECT MAX(.content.contsize) FROM ContentObject")
```

1.15.7 Query Arguments

It is often necessary to use query conditions which are supplied by method parameters or even user input. Therefore, it is possible to use variables within the query to avoid building the query string which requires the use of the correct escaping rules. The values are supplied as key/value pairs (`DICTIONARY`), so the key can be used as variable name to reference its value. Variable names are prefixed with a dollar sign (`$`). Variables cannot be used to replace attributes, operators or query keywords.

Example

```
// Returns the first ten objects found which are named "Test"
coort.SearchObjects(
  cootx,
  "LIMIT $limit SELECT * FROM $class WHERE .objname = $name",
  { limit: 10, class: #Object, name = "Test" });


coort.SearchObjects(
  cootx,
  "SELECT * IN $folder.objchildren WHERE .objowner = $user",
  { folder: COO.1.2.3.4, user: coouser });
```

Variables can be used at the following places:

```
LIMIT $limit
PRELOAD $preload
TIMEOUT $timeout
SERVICES($services)
SCOPE($scope)
DOMAINS($domains)
STORES($stores)
EVALUATE $objects
FROM $classes
IN $object.attrdef
WHERE .attrdef = $value
```

1.15.8 Grammar of the Kernel Interfaces Query Language

The grammar of the app.ducx Query Language is formally defined as described below.

Grammar

app.ducx Query Language

```
Statement := { Options } ( Query | ValueQuery | Evaluation ).
Query := "SELECT" Attributes
  ( "FROM" Classes [ "IN" ObjectList ] | "EXECUTE" Procedure )
  [ "WHERE" Condition ].
```

```
ValueQuery := "SELECT" ( AggregateExpression | ColumnExpression )
  "FROM" Classes [ "WHERE" Condition ].
Evaluation := "EVALUATE" Sequence "WHERE" Condition.
Options := ( "LIMIT" Integer | "PRELOAD" Integer | "TIMEOUT" Integer |
  "NOCHECK" | "NORESTRICTIONS" | "NOEXEC" |
  "NOHITPROPERTIES" | "HITPROPERTIES" "(" Attributes ")" |
  "IGNORECASE" | Location ).
Location := ( "CACHE" | "TRANSACTION" | "LOCAL" | "INSTALLATION" | "GLOBAL" |
  "SERVICES" "(" Service { "," Service } ")" |
  "SCOPE" "(" Scope ")" |
  "DOMAINS" "(" Domain { "," Domain } ")" |
  "STORES" "(" Store { "," Store } ")" ).
Attributes := ( "*" | Attribute { "," Attribute } | Expression).
Classes := Class { "," Class }.
ObjectList := ( Reference | Object | Ident ) { "." Attribute }.
Condition := Term { "OR" Term }.
Term := Factor { "AND" Factor }.
Factor := [ "NOT" ] Primary.
Primary := ( Predicate | "(" Condition ")" ).
Predicate := Expression
  [ ( ( "<" | "<=" | ">" | ">=" | "=" | "<>" ) Expression |
  [ "SOUNDS" ] [ "NOT" ] "LIKE" Shift { "," Shift } |
  [ "NOT" ] "CONTAINS" Shift { "," Shift } |
  [ "NOT" ] "IN" "(" ( Sequence | Query ) ")" |
  [ "NOT" ] "INCLUDES" "(" ( Sequence | Query ) ")" |
  [ "NOT" ] "BETWEEN" Shift "AND" Shift |
  "IS" [ "NOT" ] "NULL" ) ].
Expression := ( Identifier |
  [ ( "UPPER" | "LOWER" | "SUM" | "AVG" | "COUNT" | "MIN" | "MAX" ) ]
  "(" ( Identifier | Shift ) ")" |
  Shift ).
AggregateExpression := ( "SUM" | "COUNT" | "MIN" | "MAX" )
  "(" Identifier ")".
ColumnExpression := [ "DISTINCT" ] Identifier.
Attribute := ( Reference | Object ).
Class := ( Reference | Object ).
Procedure := ( Reference | Object ).
Identifier := [ "." ] { Reference "." } Reference ).
Domain := ( String | Object ).
Service := ( String | Object ).
Scope := ( String | Object ).
Letter := "a" ... "z" "A" ... "Z".
Digit := "0" ... "9".
HexDigit := "0" ... "9" "a" ... "f" "A" ... "F".
Digits := Digit { Digit }.
HexDigits := HexDigit { HexDigit }.
Name := Letter { ( Letter | Digit | "_" ) }.
Reference := Name [ "@" Digits "." Digits [ ":" Name ] ].
String := ( "'" ... "'" | '"' ... '"' ).
Integer := ( Digits | "0" ( "x" | "X" ) HexDigits ).
Float := Digits "." [ Digits ]
  [ ( "d" | "D" | "e" | "E" ) [ ( "+" | "-" ) ] Digits ].
DateTime := Digits "-" Digits "-" Digits
  [ ( "T" | " " ) Digits [ ":" Digits [ ":" Digits ] ] ].
Object := ( Address | "#" Reference ).
Address := "COO." Digits "." Digits "." Digits "." Digits [ "@" DateTime ].
```

The grammar of the app.ducx Query Language partially refers to the grammar of the Kernel Interfaces Expression Language.

1.15.9 Grammar of the Kernel Interfaces Expression Language

Grammar

```
Expression := { Statement }.
Statement := ( ";" | Directive ";" | Block | Declare ";" |
  If | For | While | Switch | Try |
  Do ";" | Break ";" | Continue ";" | Return ";" | Throw ";" |
  Sequence [ ";" ] ).
Directive := "%%" Name [ ( Sequence | "(" [ Sequence ] ")" ) ].
Block := "{" Expression "}".
Declare := "declare" [ ">" | "&" | "<" ] Ident { "," [ ">" | "&" | "<" ] Ident }.
If := "if" "(" Cond ")" Block [ "else" ( If | Block ) ].
For := "for" "(" Sequence ( ";" [ Cond ] ";" Sequence | ":" Statement ) ")"
  Block.
While := "while" "(" Cond ")" Block.
Do := "do" Block "while" "(" Cond ")".
Switch := "switch" "(" Cond ")" "{"
  { ( "case" ( Ident | Const ) | "default" ) ":" [ Expression ] } "}".
Break := "break".
Continue := "continue".
Return := "return" [ Assign ].
Try := "try" [ [ "new" ] "transaction" ] Block {
  "catch" "(" ( Object | [ ( "@" | "::" | ":>" ) ] Ident | "..." ) ")" Block }
  [ "finally" Block ].
Throw := "throw" Cond Arguments.
Items := [ Init ] { "," [ Init ] }.
Sequence := [ Assign ] { "," [ Assign ] }.
Arguments := [ Assign ] { "," [ Assign ] }.
Init := Cond [ ( ":" | "=" ) Assign ].
Assign :=
  Cond [ [ "[]" ] [ ( "@" | "::" | ":>" ) ] Ident ]
  [ ( "=" | "&=" | "^=" | "|=" | "<<=" | ">>=" |
     "+=" | "-=" | "*=" | "/=" | "%=" | "??=" ) Assign ].
Cond := Coalesce [ "?" Sequence ":" Coalesce].
Coalesce := Or { "??" Or }.
Or := And { ( "||" | "or" ) And }.
And := BitOr { ( "&&" | "and" ) BitOr }.
BitOr := BitXOr { "|" BitXOr }.
BitXOr := BitAnd { "^" BitAnd }.
BitAnd := Equal { "&" Equal }.
Equal := Rel [ ( "==" | "!=" | "<>" ) Rel ].
Rel := Shift [ ( ( "<" | "<=" | ">" | ">=" | "<=>" ) Shift |
  [ "sounds" ] [ "not" ] "like" Shift |
  [ "not" ] "contains" Shift |
  [ "not" ] "in" Shift |
  [ "not" ] "includes" Shift |
  [ "not" ] "between" Shift "and" Shift |
  "is" [ "not" ] "null" ) ].
Shift := Add { ( "<<" | ">>" ) Add }.
Add := Mul { ( "+" | "-" ) Mul }.
Mul := Prefix { ( "*" | "/" | "%" ) Prefix }.
Prefix := ( Postfix |
  ( "&" | "++" | "--" | "!" | "not" | "~" | "+" | "-" ) Prefix ).
Postfix := ( Primary { "." Qualifier |
  "(" Arguments ")" [ "[" ( "..." | Cond ) "]" ] |
  "[" Sequence "]" |
  "++" | "--" |
  "->" Qualifier "(" Arguments ")" [ "[" Cond "]" ] } |
  "->" Qualifier [ "(" Arguments ")" [ "[" Cond "]" ] ] ).
Qualifier := ( Ident | Reference | "[" Sequence "]" ).
Primary := (
  "@" ( "this" | Ident ) |
  "::" ( "this" | Ident | Reference ) |
  ":>" ( "this" | Ident | Reference ) |
  "this" | Ident | Reference |
  "[]" | "null" | "true" | "false" |
  "coort" | "cootx" | "cooobj" | "coometh" | "coocontext" |
  "coouser" | "coogroup" | "cooposition" |
```

```
  "cooenv" | "cooroot" | "coohome" | "coolang" | "coodomain" | "coonow" |
  ( [ "upper" | "lower" | "count" | "sum" | "avg" | "min" | "max" ]
    "(" Sequence ")" ) |
  "insert" "(" Assign "," Assign "," Assign ")" |
  "delete" "(" Assign "," Assign [ "," Assign ] ")" |
  "find" "(" Assign "," Assign ")" |
  "sort" "(" Assign ")" |
  "unique" "(" Assign ")" |
  "revert" "(" Assign ")" |
  "super" "(" ")" |
  "typeof" "(" Assign ")" |
  "indexof" "(" Assign "," Assign ")" |
  "strlen" "(" Assign ")" |
  "strtrim" "(" Assign ")" |
  "strhead" "(" Assign "," Assign ")" |
  "strtail" "(" Assign "," Assign ")" |
  "strsplit" "(" Assign "," Assign ")" |
  "strjoin" "(" Assign [ "," Assign ] ")" |
  "strreplace" "(" Assign "," Assign [ "," Assign ] ")" |
  "(" Sequence ")" |
  "{" Items "}" |
  "[" Sequence "]" |
  '$"' ... '{~' Assign '~}' { ... '{~' Assign '~}' } ... '"'
  Const ).
Const := ( String | DateTime | Object |
  [ "+" | "-" ] Integer | [ "+" | "-" ] Float ).
Ident := ( "$" | [ "$" ] Name ).
Letter := "a" ... "z" "A" ... "Z".
Digit := "0" ... "9".
HexDigit := "0" ... "9" "a" ... "f" "A" ... "F".
Digits := Digit { Digit }.
HexDigits := HexDigit { HexDigit }.
Name := Letter { ( Letter | Digit | "_" ) }.
Reference := Name [ "@" Digits "." Digits [ ":" Name ] ].
String := ( "'" ... "'" | '"' ... '"' ).
Integer := ( Digits | "0" ( "x" | "X" ) HexDigits ).
Float := Digits "." [ Digits ]
  [ ( "d" | "D" | "e" | "E" ) [ ( "+" | "-" ) ] Digits ].
DateTime := Digits "-" Digits "-" Digits
  ( "T" | " " ) Digit Digit ":" Digit Digit ":" Digit Digit.
Object := ( Address | "#" Reference ).
Address := "COO." Digits "." Digits "." Digits "." Digits [ "@" DateTime ].
```