cādence®

Cadence SKILL++ Object System Reference

Product Version ICADVM20.1
October 2020

Contents

Preface

This manual introduces the SKILL++ language to new users, leads users to understand advanced topics and encourages them to use sound SKILL programming methods.

This manual is intended for the following users:

- Programmers beginning to program in SKILL language
- CAD developers (internal users and customers) who have experience in SKILL programming
- CAD integrators

This preface contains the following topics:

Scope

Unless otherwise noted, the functionality described in this guide can be used in both mature node (for example, IC6.1.8) and advanced node and methodologies (for example, ICADVM20.1) releases.

Label	Meaning

(ICADVM20.1 Only)	Features supported only in the ICADVM20.1 advanced nodes and advanced methodologies releases.
(IC6.1.8 Only)	Features supported only in mature node releases.

Licensing Requirements

SKILL uses **Cadence Design Framework II** license (License Number 111), which is checked out at the launch of the `skill` executable or the workbench.

For information on licensing in the Cadence SKILL Language, see the *Virtuoso Software Licensing and Configuration User Guide*.

Related Documentation

What's New

- *Cadence SKILL Language What's New*

Installation, Environment, and Infrastructure

- *Cadence Installation Guide*
- *Virtuoso Design Environment SKILL Reference*
- *Cadence Application Infrastructure User Guide*
- *Virtuoso Software Licensing and Configuration Guide*

Other SKILL Books

- *Cadence SKILL IDE User Guide*
- *Cadence SKILL Development Reference*
- *Cadence SKILL Language User Guide*
- *Cadence SKILL Language Reference*
- *Cadence Interprocess Communication SKILL Reference*

Additional Learning Resources

Video Library

The Video Library on the Cadence Online Support website provides a comprehensive list of videos on various Cadence products.

To view a list of videos related to a specific product, you can use the *Filter Results* feature available in the pane on the left. For example, click the *Virtuoso Layout Suite* product link to view a list of videos available for the product.

You can also save your product preferences in the Product Selection form, which opens when you click the *Edit* icon located next to *My Products*.

Virtuoso Videos Book

You can access certain videos directly from Cadence Help. To learn more about the related features and to access the list of available videos, see Virtuoso Videos.

Rapid Adoption Kits

Cadence provides a number of Rapid Adoption Kits that demonstrate how to use Virtuoso applications in your design flows. These kits contain design databases and instructions on how to run the design flow.

In addition, Cadence offers the following training courses on the SKILL programming language:

- SKILL Language Programming Introduction
- SKILL Language Programming
- Advanced SKILL Language Programming

To explore the full range of training courses provided by Cadence in your region, visit Cadence Training or write to training_enroll@cadence.com.

Note: The links in this section open in a separate web browser window when clicked in Cadence Help.

Help and Support Facilities

Virtuoso offers several built-in features to let you access help and support directly from the software.

- The Virtuoso *Help* menu provides consistent help system access across Virtuoso tools and applications. The standard Virtuoso *Help* menu lets you access the most useful help and support resources from the Cadence support and corporate websites directly from the CIW or any Virtuoso application.
- The Virtuoso Welcome Page is a self-help launch pad offering access to a host of useful knowledge resources, including quick links to content available within the Virtuoso installation as well as to other popular online content.

  The Welcome Page is displayed by default when you open Cadence Help in standalone mode from a Virtuoso installation. You can also access it at any time by selecting *Help – Virtuoso Documentation Library* from any application window, or by clicking the *Home* button on the Cadence Help toolbar (provided you have not set a custom home page).

For more information, see Getting Help in *Virtuoso Design Environment User Guide.*

Customer Support

For assistance with Cadence products:

- Contact Cadence Customer Support

  Cadence is committed to keeping your design teams productive by providing answers to technical questions and to any queries about the latest software updates and training needs. For more information, visit https://www.cadence.com/support.
- Log on to Cadence Online Support

  Customers with a maintenance contract with Cadence can obtain the latest information about various tools at https://support.cadence.com.

Feedback about Documentation

You can contact Cadence Customer Support to open a service request if you:

- Find erroneous information in a product manual
- Cannot find in a product manual the information you are looking for

- Face an issue while accessing documentation by using Cadence Help

You can also submit feedback by using the following methods:

- In the Cadence Help window, click the *Feedback* button and follow instructions.
- On the Cadence Online Support Product Manuals page, select the required product and submit your feedback by using the *Provide Feedback* box.

Understanding Cadence SKILL

Cadence SKILL is a high-level, interactive programming language based on the popular artificial intelligence language, Lisp. It lets you customize and extend your design environment. Using SKILL you can validate the steps of your algorithm incrementally before incorporating them into a larger program.

For more information about the SKILL language, see Getting Started in the *SKILL Language User Guide*.

Using SKILL Code Examples

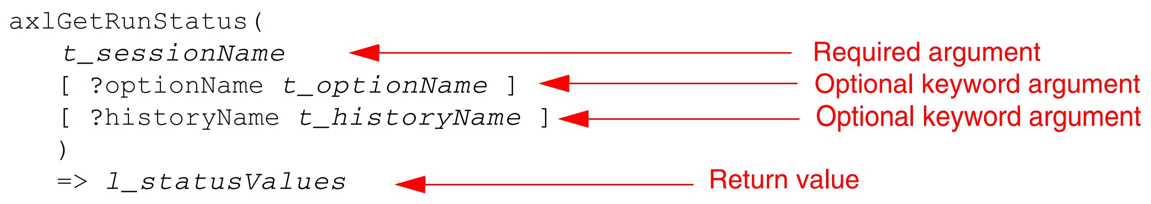
The SKILL APIs in this user manual are explained with illustrative code examples.

You can copy these examples from the manual and paste them directly into the Command Interpreter Window (CIW) or use the code in non-graphical SKILL mode.

Sample SKILL Code

The following code sample shows the syntax of a SKILL API that accepts three arguments.

axlGetRunStatus

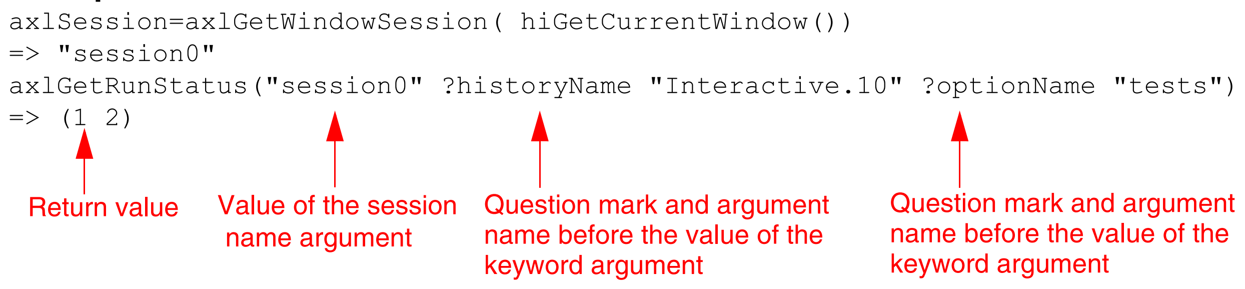


```
axlGetRunStatus(
    t_sessionName
    [ ?optionName t_optionName ]
    [ ?historyName t_historyName ]
    )
    => l_statusValues
```

The first argument `t_sessionName` is a required argument, where `t` signifies the data type of the argument. The second and third arguments `?optionName t_optionName` and `?historyName t_historyName` are optional keyword arguments (identified by a question mark), which are specified in name-value pairs and can be placed in any order during the function call.

The return value is the value that the SKILL API returns after evaluating the expression. In this case, it is a list of status values, *`l_statusValues`*.

Example



```
axlSession=axlGetWindowSession( hiGetCurrentWindow())
=> "session0"
axlGetRunStatus("session0" ?historyName "Interactive.10" ?optionName "tests")
=> (1 2)
```

Return value
Value of the session name argument
Question mark and argument name before the value of the keyword argument
Question mark and argument name before the value of the keyword argument

Accessing API Help

Quick reference information for SKILL APIs is available from the CIW and the SKILL API Finder. To access the reference information for a particular SKILL API, do one of the following:

- Type `help <function_name>` in the CIW.
- Type `startFinder ( [ ?funcName` *`t_functionName`* `] )` in the CIW.
- Start the SKILL API Finder from the CIW by choosing *Tools – Finder* or type `cdsFinder` on the UNIX command line.

  In the *Search in* field of the displayed Cadence SKILL API Finder window, type the SKILL API name for which you want to display the help information and click *Go*.

  The matches for the searched SKILL API appear in the *Results* area.

  To view the complete documentation of the searched SKILL API, select the API name in the *Results* area and click the *More Info* button. The complete documentation of the selected SKILL API appears in a new Cadence Help window.

Typographic and Syntax Conventions

The following typographic and syntax conventions are used in this manual.

text	Indicates names of manuals, menu commands, buttons, and fields.
`text`	Indicates text that you must type exactly as presented. Typically used to denote command, function, routine, or argument names that must be typed literally.
`z_argument`	Indicates text that you must replace with an appropriate argument value. The prefix (in this example, *`z_`*) indicates the data type the argument can accept and must not be typed.
|	Separates a choice of options.
`{ }`	Encloses a list of choices, separated by vertical bars, from which you **must** choose one.
`[ ]`	Encloses an optional argument or a list of choices separated by vertical bars, from which you **may** choose one.
`[ ?argName` *`t_arg`* `]`	Denotes a *key argument*. The question mark and argument name must be typed as they appear in the syntax and must be followed by the required value for that argument.
`...`	Indicates that you can repeat the previous argument. Used with brackets to indicate that you can specify zero or more arguments. Used without brackets to indicate that you must specify at least one argument.
`,...`	Indicates that multiple arguments must be separated by commas.
`=>`	Indicates the values returned by a Cadence$^{®}$ SKILL$^{®}$ language function.
`/`	Separates the values that can be returned by a Cadence SKILL language function.

If a command-line or SKILL expression is too long to fit within the paragraph margins of this document, the remainder of the expression is moved to the next line and indented. In code excerpts, a backslash (\ ) indicates that the current line continues on to the next line.

Identifiers Used to Denote Data Types

Data type identifiers are used to indicate the type of value required by an API argument. These data types are denoted by a single letter that is prefixed to the argument label and is separated from the argument by an underscore; for example, `t` is the data type in *`t_viewName`*. Data types and underscores are used only as identifiers; they must not be typed when specifying the argument in a function.

Prefix	Internal Name	Data Type
a	array	array
A	amsobject	AMS object
b	ddUserType	DDPI object
B	ddCatUserType	DDPI category object
C	opfcontext	OPF context
d	dbobject	Cadence database object (CDBA)
e	envobj	environment
f	flonum	floating-point number
F	opffile	OPF file ID
g	general	any data type
G	gdmSpecIlUserType	generic design management (GDM) spec object
h	hdbobject	hierarchical database configuration object
I	dbgenobject	CDB generator object
K	mapiobject	MAPI object
l	list	linked list
L	tc	Technology file time stamp
m	nmpIIUserType	nmpII user type
M	cdsEvalObject	cdsEvalObject
n	number	integer or floating-point number
o	userType	user-defined type (other)
p	port	I/O port
q	gdmspecListIlUserType	gdm spec list

Prefix	Internal Name	Data Type
`r`	defstruct	defstruct
`R`	rodObj	relative object design (ROD) object
`s`	symbol	symbol
`S`	stringSymbol	symbol or character string
`t`	string	character string (text)
`T`	txobject	transient object
`u`	function	function object, either the name of a function (symbol) or a lambda function body (list)
`U`	funobj	function object
`v`	hdbpath	hdbpath
`w`	wtype	window type
`sw`	swtype	subtype session window
`dw`	dwtype	subtype dockable window
`x`	integer	integer number
`y`	binary	binary function
`&`	pointer	pointer type

For more information, see *Cadence SKILL Language User Guide*.

1

Classes and Instances

allocateInstance

```
allocateInstance(
     u_class
     )
     => g_instance
```

Description

Creates and returns an empty instance of the specified class. All slots of the new instance are unbound.

Arguments

u_class	The specified class for which a new instance has to be created.

Value Returned

g_instance	An empty instance of `u_class`.

Example

```
defclass(A () ((slot1 @initform 1) (slot2 @initform 2)))
i = allocateInstance(findClass('A))
i->??
=> (slot1 \*slotUnbound\* slot2 \*slotUnbound\*)

i = allocateInstance('A)
i->??
=> (slot1 \*slotUnbound\* slot2 \*slotUnbound\*)
```

changeClass

```
changeClass(
     g_inst
     g_className
     [ g_initArgs ]
     )
     => g_updatedInst
```

Description

Changes the class of the given instance (*`g_inst`*) to the specified class (*`g_className`*).

The function `updateInstanceForDifferentClass()` is called on the modified instance to allow applications to deal with new or lost slots.

Arguments

`g_inst`	An instance of standardObject.
`g_className`	The new class for the instance.
`g_initArgs`	Additional arguments to be passed to the `updateInstanceForDifferentClass()` function.

Value Returned

`g_updatedInst`	The updated instance.

Example

```
(defclass A () ())
(defclass B (A) ((slot @initarg s)))
 x = (makeInstance 'A)
(changeClass x 'B ?s 1)
(classOf x)
    => class:B
```

className

```
className(
     us_class
     )
     => s_className
```

Description

Returns the class symbol denoting a class object.

For user-defined classes, *`s_className`* is the symbol passed to `defclass` in defining *`us_class`*.

Arguments

`us_class`	Must be a class object or a class symbol. Otherwise an error is signaled.

Value Returned

`s_className`	The class symbol.

Example

```
className( classOf( 5 )) => fixnum
```

```
defclass( GeometricObject
    ()      ;;; standardObject is the subclass by default
    ()      ;;; no slots
    )       ; defclass
className(findClass( 'GeometricObject))
=> GeometricObject
```

```
geom = makeInstance( 'GeometricObject )
className( classOf( geom))
=> GeometricObject
```

Reference

classOf, findClass

classOf

```
classOf(
     g_object
     )
     => u_classObject
```

Description

Returns the class object of which the given object is an instance.

Arguments

g_object	Any SKILL object.

Value Returned

u_classObject	Class object of which the given object is an instance.

Examples

```
classOf( 5 )
=> class:fixnum
```

```
className( classOf( 5 ))
=> fixnum
```

Reference

className, findClass

classp

```
classp(
     g_object
     su_class
     )
     => t / nil
```

Description

Checks if the given object is an instance of the given class or is an instance of one of its subclasses.

Arguments

g_object	Any SKILL object
su_class	A class object or a symbol denoting a class.

Value Returned

`t`	If the given object is an instance of the class or a subclass of the class.
`nil`	Otherwise.

Example

```
classp( 5 classOf( 5 ))  => t
classp( 5 'fixnum )      => t
classp( 5 'string )      => nil
classp( 5 'noClass )
*Error* classp: second argument must be a class - nil
```

Reference

classOf, className

defclass

```
defclass(
     s_className
     ( [ s_superClassName1 ]...[ s_superClassNameN ])
     ( [ ( s_slotName
     [ @initarg s_argName ]
     [ @reader s_readerFun ]
     [ @writer s_writerFun ]
     [ @initform g_exp ])
     ... ] )
     )
     => t
```

Description

Creates a class object with class name and optional super class name (or names) and slot specifications. This is a macro form.

If a super class is not given, the default super class is the `standardObject` class. For more information, see "Defining a Class (defclass)" in the *Cadence SKILL Language User Guide*.

Each slot specifier itself is a list composed of slot options. The only required slot option is the slot name.

Note: If you define a class with two slots that have the same name, as shown in the example given below, SKILL creates the class but also issues a warning.

```
defclass(A () ((slotA) (slotB) (slotA @initform 42)))
```

Arguments

`s_className`	Name of new class.
`s_superClassName1 ... s_superClassNameN`	Names of one or more super classes. Default is `standardObject`.
`s_slotName`	Name of the slot.
`@initarg` *`s_argName`*	Declares an initialization argument named *`s_argName`*. Calls to `makeInstance` can use *`s_argName`* as keyword argument to pass an initialization value.
`@reader` *`s_readerFun`*	Specifies that a method be defined on the generic function named *`s_readerFun`* to read the value of the given slot.
`@writer` *`s_writerFun`*	Specifies that a method be defined on the generic function named *`s_writerFun`* to change the value of the given slot.
`@initform` *`g_exp`*	The expression is evaluated every time an instance is created. The `@initform` slot option is used to provide an expression to be used as a default initial value for the slot. The form is evaluated in the class definition environment.

Value Returned

`t`	Always returns `t`.

Example 1

```
defclass( Point
    ( GeometricObject )
    (
        ( name @initarg name )
        ( x @initarg x )       ;;; x-coordinate
        ( y @initarg y )       ;;; y-coordinate
        )
    ) ; defclass => t
P = makeInstance( 'Point ?name "P" ?x 3 ?y 4 )
defclass(A (B C) ((slot1) (slot2) (slot2 @initform 42)))
```

findClass

```
findClass(
     s_className
     )
     => u_classObject / nil
```

Description

Returns the class object associated with a symbol. The symbol is the symbolic name of the class object.

Arguments

s_className	A symbol that denotes a class object.

Value Returned

u_classObject	Class object associated with a symbolic name.
`nil`	If there is no class associated with the given symbol.

Example

```
findClass( 'Point )         => funobj:0x1c9968
findClass( 'fixnum )        => funobj:0x1840d8
findClass( 'standardObject  => funobj:0x184028
findClass( 'fuzzyNumber )   => nil
```

Reference

defclass, className

initializeInstance

```
initializeInstance(
     g_instance
     [ u_?initArg1     value1 ]
     [ u_?initArg2     value2 ] ...
     )
     => t
```

Description

Initializes the newly created instance of a class. `initializeInstance` is called by the `makeInstance` function.

Arguments

g_instance	A symbol denoting an instance. The instance must be created using `makeInstance`.
`u_?initArg1` *value1* `u_?initArg2`*value2*	
	`initArg1`*value1* is the initial value for argument1 of the instance. Similarly for the pair `u_initArg2`*value2* and so forth.

Value Returned

t	The instance has been initialized.

Example

```
defclass( A () ())
=> t

defmethod( initializeInstance ((obj A) @key (a 3) @rest args))
      (printf "initializeInstance : A : was called with args - obj == '%L'
      a == '%L' rest == '%L'\n" obj a args)

(callNextMethod)
=> t

makeInstance( 'A ?a 7)
      initializeInstance : A : was called with args - obj == 'stdobj@0x83bf048' a
      == '7' rest == 'nil'
=> stdobj@0x83bf048
```

```
makeInstance( 'A)
       initializeInstance : A : was called with args - obj == 'stdobj@0x83bf054' a
      == '3' rest == 'nil'
=> stdobj@0x83bf054
```

isClass

```
isClass(
     g_object
     )
     => t / nil
```

Description:

Checks if the given object is a class object.

Arguments

g_object	Any SKILL object.

Value Returned

`t`	If the given object is a class object.
`nil`	Otherwise.

Example

```
isClass( classOf( 5 ) ) => t
isClass( findClass( 'Point ) ) => t
isClass( 'noClass ) => nil
```

Reference

classOf, findClass

makeInstance

```
makeInstance(
     us_class
     [ u_?initArg1     value1 ]
     [ u_?initArg2     value2 ] ...
     )
     => g_instance
```

Description

Creates an instance of a class, which can be given as a symbol or a class object.

Arguments

`us_class`	Class object or a symbol denoting a class object. The class must be either `standardObject` or a subclass of `standardObject`.

`u_?initArg1` *`value1`*
`u_?initArg2` *`value2`*
	The symbol `u_initArg1` is specified in one of the slot specifiers in the `defclass` declaration of either *`us_class`* or a superclass of *`us_class`*. *`value1`* is the initial value for that slot. Similarly for the pair `u_initArg2` *`value2`* and so forth.

Value Returned

`g_instance`	The instance. The print representation of the instance resembles `stdobj:`*`xxxxx`*, where *`xxxxx`* is a hexadecimal number.

Example

```
defclass( Circle ( GeometricObject )
    (( center @initarg c ) ( radius @initarg r )) ) => t
P = makeInstance( 'Point ?name "P" ?x 3 ?y 4 )
    => stdobj:0x1d003c
C = makeInstance( 'Circle ?c P ?r 5.0 ) => stdobj:0x1d0048
makeInstance( 'fixnum )
*Error* unknown: non-instantiable class - fixnum
```

Reference

defclass

printself

```
printself(
     g_object
     )
     => g_result
```

Description

A generic function which is called to print a `stdObject` instance.

Arguments

g_object	An instance of a class.

Value Returned

g_result	A string or symbol representing information about *g_object*.

Example

```
defmethod( printself ((obj myClass))
sprintf(nil "#{instance of myClass:%L}" obj) ; returns a string
)
i = makeInstance('myClass)
=> #{instance of myClass:stdobj@0x83ba018}
; prints all instances of myClass
```

setSlotValue

```
setSlotValue(
     g_standardObject
     s_slotName
     g_value
     )
     => g_value
```

Description

Sets the *`s_slotName`* slot of *`g_standardObject`* to *`g_value`*.

An error is signaled if there is no such slot for the *`g_standardObject`*. This function bypasses any `@writer` generic function for the slot that you specified in the `defclass` declaration for the *`g_standardObject`*'s class.

Arguments

`g_standardObject`	An instance of the *`standardObject`* class or a subclass of *`standardObject`*.
`s_slotName`	The slot symbol used as the slot name in the `defclass` slot specification.
`g_value`	Any SKILL data object.

Value Returned

`g_value`	The value assigned to the slot.

Example

```
defclass( GeometricObject ()
      (
            ( x @initarg x )
            ( y @initarg y )
      )
)     => t
geom = makeInstance( 'GeometricObject ?x 0 )
                          => stdobj:0x34b018
slotValue( geom 'y )      => \*slotUnbound\*
setSlotValue( geom 'y 2 ) => 2
slotValue( geom 'y )      => 2
```

sharedInitialize

```
sharedInitialize(
     g_object
     g_slotList
     @rest l_initargs
     )
     => g_object / error
```

Description

This is a generic function, which is called when an instance is created, re-initialized, updated to conform to a redefined class, or updated to conform to a different class. It is called from the `initializeInstance`, `updateInstanceForRedefinedClass`, and `updateInstanceForDifferentClass` functions to initialize slots of the instance *`g_object`* using the corresponding `initforms`.

If the function is successful, the updated instance is returned.

Arguments

`g_object`	An instance of a class.
`g_slotList`	`t` or a list of slot names (symbols). If the argument is `t`, it initializes all uninitialized slots. If it is a list of slot names, it initializes only the uninitialized slots in the list.
`@rest` *`l_initargs`*	List of optional `initargs`.

Value Returned

`g_object`	The updated instance (`g_object`).
`error`	Otherwise.

Example

```
defclass( A () ((a @initform 1)))
=> t

defmethod( sharedInitialize ((obj A) slots @key k @rest args)
(printf "sharedInitialize A: obj->?? == '%L' k == '%L' args == '%L'\n" obj->?? k
args)
(callNextMethod)

)
```

```
=> t
defclass( B () ((b @initform 2)))
=> t
x = makeInstance( 'A ?k 9)
sharedInitialize A: obj->?? == '(a \*slotUnbound\*)' k == '9' args == 'nil'
=> stdobj@0x83bf018
defclass( A () ((a @initform 1)
(c @initform 3)))
*WARNING* (defclass): redefinition of class A updating stdobj@0x83bf018
sharedInitialize A: obj->?? == '(a 1 c \*slotUnbound\*)' k == 'nil' args == 'nil'
=> t
changeClass( x 'B ?k 7)
updating stdobj@0x83bf018
stdobj@0x83bf018
x->??
(b 2)
changeClass( x 'A ?k 7)
updating stdobj@0x83bf018
sharedInitialize A: obj->?? == '(a \*slotUnbound\* c \*slotUnbound\*)' k == '7'
args == 'nil'
stdobj@0x83bf018
x->??
(a 1 c 3)
```

slotBoundp

```
slotBoundp(
     obj
     t_slotName
     )
     => t / nil
```

Description

Checks if a named slot is bound to an instance or not.

Note: For compatibility with previous releases, an alias to this function with the name `ilSlotBoundp` exists.

Arguments

`obj`	An instance of some class.
`t_slotName`	Slot name.

Value Returned

`t`	If the slot is bound.
`nil`	If the slot is unbound.

Note: It throws an error if *`obj`* or *`t_slotName`* is invalid.

Example

```
myObject => slotX = 20
slotBoundp(myObject "slotX") => t
```

slotUnbound

```
slotUnbound(
     u_class
     g_object
     s_slotName
     )
     => g_result
```

Description

This function is called when the `slotValue` function attempts to reference an unbound slot. It signals that the value of the slot *`s_slotName`* of *`g_object`* has not been set yet. In this case, `slotValue` returns the result of the method.

Arguments

`u_class`	A class object. The class must be either `standardObject` or a subclass of `standardObject`.
`g_object`	An instance of *`u_class`*.
`s_slotName`	The name of the unbound slot.

Value Returned

`g_value`	Value contained in the slot *`s_slotName`*. The default value is '*slotUnbound*.

Example

```
defclass( A () ((a)))
=> t

x = (makeInstance 'A)
=> stdobj@0x83bf018

defmethod( slotUnbound (class (obj A) slotName) (printf "slotUnbound : slot '%L'is
unbound\n" slotName) (setSlotValue obj slotName 6)
)
=> t

x->a
=> slotUnbound : slot 'a' is unbound
=> 6

x->a
=> 6
```

```
defmethod( slotUnbound (class (obj A) slotName) (printf "slotUnbound : slot '%L'
is unbound\n" slotName) (setSlotValue obj slotName 6)
8
)
=> t
```

```
*WARNING* (defmethod): method redefined generic:slotUnbound class:(t A t)
```

```
x->a = '\*slotUnbound\*
```

```
\*slotUnbound\*
```

```
x->a
=> slotUnbound : slot 'a' is unbound
=> 8 ;; the return value of slotUnbound method, not a new value of the slot
```

```
x->a
=> 6
```

slotValue

```
slotValue(
     g_standardObject
     s_slotName
     )
     => g_value
```

Description

Returns the value contained in the slot `slotName` of the given `standardObject`.

If there is no slot with the given name an error is signalled. This function bypasses any `@reader` generic function for the slot that you specified in the `defclass` declaration for the *`g_standardObject`*'s class.

Arguments

`g_standardObject`	An instance of the `standardObject` class or a subclass of `standardObject`.
`s_slotName`	The slot symbol used as the slot name in the *defclass* slot specification.

Value Returned

`g_value`	Value contained in the slot *`s_slotName`* of the given `standardObject`.

Example

```
defclass( GeometricObject ()
       (
              ( x @initarg x )
              ( y @initarg y )
       )
)
=> t
```

2

Generic Functions and Methods

ansiDefmethod

```
ansiDefmethod(
     s_name
     l_spec
     g_body
     )
     => t
```

Description

A SKILL++ `defmethod` macro for supporting lexical scoping in `callNextMethod`. It creates a closure for a method.

Arguments

s_name	A method name.
l_spec	A list of specializers for the specified method.
g_body	Body of the method.

Value Returned

`t`	Always returns `t`.

Example

```
(defclass Parent () ())
(defclass Child (Parent) ())


(defmethod Printer ((self Parent) function)
  (error "This line is never reached"))
```

```
(defmethod Printer ((self Child) function)
  (printf "The function returns %L\n" (funcall function)))


(defmethod Caller ((self Parent))
'Parent)


(ansiDefmethod Caller ((self Child))
(Printer self callNextMethod))


(Caller (makeInstance 'Child))
The function returns Parent
=> t
```

callAs

```
callAs(
     us_class
     s_genericFunction
     g_arg1
     [ g_arg2 ... ]
     )
     => g_value
```

Description

Calls a method specialized for some super class of the class of a given object directly, bypassing the usual method inheritance and overriding of a generic function.

It is an error if the given arguments do not satisfy the condition `(classp g_obj us_class)`.

Arguments

us_class	A class name or class object.
s_genericFunction	A generic function name.
g_arg1	A SKILL object whose class is *us_class* or a subclass of *us_class*.
g_arg2	Arguments to pass to the generic function.

Value Returned

g_value	The result of applying the selected method to the given arguments.

Example

```
defclass( GeometricObj () ())
=> t
defclass( Point (GeometricObj ) () )
=> t
defgeneric( whoami (obj) println("default"))
=> t
defmethod( whoami  (( obj Point )) println("Point"))
=> t
defmethod( whoami  (( obj GeometricObj))
                        println( "GeometricObj"))
=> t
```

```
p = makeInstance( 'Point )
=> stdobj:0x325018
whoami(p)                                  ;prints "Point"
=> nil
callAs( 'GeometricObj 'whoami p )          ;prints "GeometricObj"
=> nil
```

Reference

nextMethodp, callNextMethod

callNextMethod

```
callNextMethod(
     [ g_arg ... ]
     )
     => g_value
```

Description

Calls the next applicable method for a generic function from within the current method. Returns the value returned by the method it calls.

This function can only be (meaningfully) used in a method body to call the next more general method in the same generic function.

You can call `callNextMethod` with no arguments, in which case all the arguments passed to the calling method will be passed to the next method. If arguments are given, they will be passed to the next method instead.

Arguments

g_arg	Optional arguments to pass to the next method.

Value Returned

g_value	Returns the value returned by the method it calls.

Example

If you call the `callNextMethod` function outside a method you get:

```
ILS-<2> procedure( example() callNextMethod() )
example
ILS-<2> example()
*Error* callNextMethod: not in the scope of any generic function call
```

This example also shows the effect of incrementally defining methods:

```
ILS-<2> defgeneric( HelloWorld ( obj )
        printf( "Generic Hello World\n" )
        )
=> t
ILS-<2> HelloWorld( 5 )
Generic Hello World
=> t
; t is the superclass of all classes
```

```
ILS-<2> defmethod( HelloWorld ((obj t ))
        printf( "Class: %s says Hello World\n" 't )
        )
=> t
ILS-<2> HelloWorld( 5 )
Class: t says Hello World
=> t
; systemObject is a subclass of t
ILS-<2> defmethod( HelloWorld ((obj systemObject  ))
        printf( "Class: %s says Hello World\n" 'systemObject )
        callNextMethod()
        )
=> t
ILS-<2> HelloWorld( 5 )
Class: systemObject says Hello World
Class: t says Hello World
=> t
; primitiveObject is a subclass of systemObject
ILS-<2> defmethod( HelloWorld (( obj primitiveObject ))
        printf( "Class: %s says Hello World\n" 'primitiveObject )
        callNextMethod()
        )
=> t
ILS-<2> HelloWorld( 5 )
Class: primitiveObject says Hello World
Class: systemObject says Hello World
Class: t says Hello World
=> t
; fixnum is a subclass of primitiveObject
ILS-<2> defmethod( HelloWorld (( obj fixnum ))
        printf( "Class: %s says Hello World\n" 'fixnum )
        callNextMethod()
        )
=> t
ILS-<2> HelloWorld( 5 )
Class: fixnum says Hello World
Class: primitiveObject says Hello World
Class: systemObject says Hello World
Class: t says Hello World
=> t
ILS-<2> HelloWorld( "abc" )
Class: primitiveObject says Hello World
Class: systemObject says Hello World
Class: t says Hello World
=> t
```

Reference

nextMethodp, callAs

defgeneric

```
defgeneric(
     s_functionName
     ( s_arg1
     [ s_arg2 ... ]
     )
     [ g_exp ... ]
     )
     => t
```

Description

Defines a generic function with an optional default method. This is a macro form. Be sure to leave a space after the function name. See the *Cadence SKILL Language User Guide* for a discussion of generic functions.

Arguments

s_functionName	Name of the generic function. Be sure to leave a space after the function name.
s_arg1	Any valid argument specification for SKILL functions, including `@key`, `@rest`, and so forth.
g_exp	The expressions that compose the default method. The default method is specialized on the class `t` for the first argument. Because all SKILL objects belong to class `t`, this represents the most general method of the generic function and is applicable to any argument.

Value Returned

`t`	Generic function is defined.

Example

```
ILS-<2> defgeneric( whatis ( object )
            printf(
            "%L is an instance of %s\n"
            object className( classOf( object))
                  )
            ) ; defgeneric
ILS-<2> whatis( 5 )
5 is an instance of fixnum
```

```
t
ILS-<2> whatis( "abc" )
"abc" is an instance of string
t
```

Reference

defmethod

defmethod

```
defmethod(
     s_name
     (
     ( s_arg1
       s_class
     )
     s_arg2 ...
     )
     g_exp1 ...
     )
     => t
```

Description

Defines a method for a given generic function. This is a macro form. Be sure to leave a space after `s_name`.

The method is specialized on the `s_class`. The method is applicable when `classp( s_arg1 s_class )` is true.

Arguments

`s_name`	Name of the generic function for which this method is to be added. Be sure to leave a space after `s_name`.
`( s_arg1 s_class)`	List composed of the first argument and a symbol denoting the class. The method is applicable when `s_arg1` is bound to an instance of `s_class` or one of its subclasses.
`g_exp1 ...`	Expressions that compose the method body.

Value Returned

`t`	Always returns `t`.

Example

```
defmethod( whatis (( p Point ))
    sprintf( nil "%s %s @ %n:%n"
        className( classOf( p ))
        p->name
        p->x
        p->y
        )
    ) ; defmethod
=> t
```

Reference

defgeneric, `procedure`, `defun`

getMethodSpecializers

```
getMethodSpecializers(
     s_genericFunction
     )
     => l_classNames / nil
```

Description

Returns the specializers of all methods currently associated with the given generic function, in a list of class names. The first element in the list is `t` if there is a default method.

Arguments

s_genericFunction	A symbol that denotes a generic function object.

Value Returned

l_classNames	List of method specializers that are currently associated with *s_genericFunction*. The first element in the list is `t` if there is a default method.
`nil`	*s_genericFunction* is not a generic function.

Example

```
defmethod( met1 ((obj number)) println(obj))
=> t
getMethodSpecializers('met1)
=>(number)
defclass( XGeometricObj () () )
=> t
defgeneric( whoami (obj) printf("Generic Object\n"))
=> t
defmethod( whoami (( obj XGeometricObj)) printf( "XGeometricObj, which is also
a\n"))
=> t
getMethodSpecializers('whoami)
=> (t XGeometricObj)
getMethodSpecializers('car)
=> *Error* getMethodSpecializers: first argument must be a generic function - car
nil
getMethodSpecializers(2)
=> *Error* getMethodSpecializers: argument #1 should be a symbol (type template =
"s") - 2
```

isGeneric

```
isGeneric(
     g_function
     )
     => t / nil
```

Description

Checks if the specified symbol (function name) or funobj (function object) represents a generic SKILL++ function.

Arguments

g_function	A symbol (function name) or funobj (function object)..

Value Returned

`t`	Returns `t`, if the specified symbol (function name) or funobj (function object) is a generic SKILL++ function.
`nil`	Returns `nil`, if the specified symbol (function name) or funobj (function object) is not a generic SKILL++ function.

Example

```
defgeneric(f1 (x y))

defun(f2 (x y) x + y)

isGeneric('f2)
=> nil

isGeneric('f1)
=> t
```

getGFbyClass

```
getGFbyClass(
     s_className
     [ g_nonExistent ]
     )
     => l_methods
```

Description

Displays the list of all generic functions specializing on a given class.

Arguments

s_className	Name of the class for which you want view the list of specializing functions.
g_nonExistent	If set to t, lists the list of generic functions specializing on non-defined classes only.

Value Returned

l_methods	A list of generic functions.

Example

```
getGFbyClass('systemObject)
  => (printObject)
```

getApplicableMethods

```
getApplicableMethods(
     s_gfName
     l_args
     )
     => l_funObjects
```

Description

Returns a list of applicable methods (*funObjects*) for the specified generic function for a given set of arguments. The returned list contains methods in the calling order.

Arguments

s_gfName	Specifies the name of the generic function
l_args	Specifies a list of arguments for which you want to retrieve the applicable methods

Values Returned

l_funObjects	Returns a list of methods in the calling order **Note:** If there are no applicable methods for the given arguments then an error is raised.

Example

```
getApplicableMethods('testMethod list("test" 42))
=> (funobj@0x83b76d8 funobj@0x83b76f0 funobj@0x83b76a8 funobj@0x83b7678
funobj@0x83b7690 funobj@0x83b7630 funobj@0x83b7600 funobj@0x83b76c0 )
```

getMethodName

```
getMethodName(
     U_funObject
     )
     => s_name
```

Description

Returns the method name for the given function object

Arguments

U_funObject	Specifies the name of the function object for which you want to retrieve the method name

Values Returned

s_name	Returns the method name for the specified generic function object

Example

```
getMethodName(funobj@0x0182456)
=> testMethod
```

getMethodRole

```
getMethodRole(
     U_funObject
     )
     => s_role / nil
```

Description

Returns the method role for the given function object. *`U_funObject`* should be a valid generic function object.

Arguments

`U_funObject`	Specifies the name of the function object for which you want to retrieve the method role. This should be a valid generic function object.

Values Returned

`s_role`	Returns the role of the specified generic function object
`nil`	Returns `nil` if the method is a primary method

Example

```
getMethodRole(funobj@0x0182456)
=> @before
```

getMethodSpec

```
getMethodSpec(
     U_funObject
     )
     => l_spec
```

Description

Returns the list of specializer for the given funobject. `U_funObject` should be a valid generic method object.

Arguments

U_funObject	Specifies the name of the function object for which you want to retrieve the list of specializers. This should be a valid generic method object.

Values Returned

l_spec	Returns a list of specializers for the specified generic function object

Example

```
getMethodSpec(funobj@0x0182456)
=> (string number)
```

getGFproxy

```
getGFproxy(
     s_gfName
     )
     => U_classObj / nil
```

Description

Returns a proxy instance from the specified generic function object

Arguments

s_gfName	Specifies a symbol that denotes the name of a generic function object

Value Returned

U_classObj	Returns the associated proxy instance
`nil`	Returns `nil` if a generic function does not exist

Example

```
getGFproxy('niTest); niTest is the name of the generic function
  => stdobj@0x83c0018
classOf(getGFproxy('niTest))
  => class:niGF
classOf(getGFproxy('printself))  ;; class of standard generic function (printself)
  => class:ilGenericFunction  ;; default
getGFproxy('abc)
  => nil ;; non-existing generic function
```

nextMethodp

```
nextMethodp(
     )
     => t / nil
```

Description

Checks if there is a next applicable method for the current method's generic function. The *current method* is the method that is calling `nextMethodp`.

`nextMethodp` is a predicate function which returns `t` if there is a next applicable method for the current method's generic function. This next method is specialized on a superclass of the class on which the current method is specialized.

Prerequisites

This function should only be used within the body of a method to determine whether a next method exists.

Caution

The return value and the effect of this function are unspecified if called outside of a method body.

Arguments

None.

Value Returned

`t`	There is a next method
`nil`	There is no next method.

Example

```
defclass( GeometricObj () ())
=> t
defclass( Point ( GeometricObj ) () )
=> t
defmethod( whoami (( obj Point ))
              if( nextMethodp()
            then printf("Point, which is a " )
            callNextMethod()
            else printf("Point")))
=> t
p = makeInstance( 'Point )
=> stdobj:0x325030
whoami(p)
=> t
```

Prints `Point.`

```
defmethod( whoami  (( obj GeometricObj))
                println( "GeometricObj"))
=> t
whoami( p)
=> nil
```

Prints `Point, which is a "GeometricObj"`

Reference

defmethod, callNextMethod

removeMethod

```
removeMethod(
     s_genFunction
     g_className
     [ g_method ]
     )
     => t / nil
```

Description

Removes a given method from a generic function.

Note: For compatibility with previous releases, an alias to this function with the name, `ilRemoveMethod` exists.

Arguments

`s_genFunction`	Name of the generic function from which the method needs to be removed.
`g_className`	Name of the class or list of classes to which the generic function belongs.
`g_method`	Specifies the method qualifier. It can have one of the following values: '`@before`, '`@after`, and '`@around`. If this value is not provided or is specified as `nil`, then the primary method is removed.

Value Returned

`t`	Returns `t`, if the method is successfully removed.
`nil`	Returns `nil`, if the method is not removed.

Example

```
removeMethod('my_function 'my_class '@before)
```

```
removeMethod('myFunB '(classX classY) '@after)
```

updateInstanceForDifferentClass

```
updateInstanceForDifferentClass(
     g_previousObj
     g_currentObj
     @rest initargs
     )
     => t
```

Description

A generic function, which is called from `changeClass` to update the specified instance (`g_currentObj`).

Arguments

g_previousObj	A copy of the `ilChangeClass` argument. It keeps the old slot values of the specified instance.
g_currentObj	The instance to be updated.
initargs	Additional arguments for the instance.

Value Returned

`t`	Always returns `t`

updateInstanceForRedefinedClass

```
updateInstanceForRedefinedClass(
      obj
      l_addedSlots
      l_deletedSlots
      l_dplList
      )
      => t
```

Description

It is a generic function, which is called to update all instances of a class, when a class redefinition occurs.

The primary method of `updateInstanceForRedefinedClass` checks the validity of `initargs` and throws an error if the provided `initarg` is not declared. It then initializes slots with values according to the `initargs`, and initializes the newly added-slots with values according to their `initform` forms.

When a class is redefined and an instance is being updated, a property-list is created that captures the slot names and values of all the discarded slots with values in the original instance. The structure of the instance is transformed so that it conforms to the current class definition.

The arguments of `updateInstanceForRedefinedClass` are the transformed instance, a list of slots added to the instance, a list of slots deleted from the instance, and the property list containing the slot names and values of slots that were discarded. This list of discarded slots contains slots that were local in the old class and are shared in the new class.

Arguments

obj	Instance of the class being redefined.
l_addedSlots	A list of slots added to the class.
l_deletedSlots	A list of slots deleted from the class.
l_dplList	A list of slots that were discarded, with their values.

Value Returned

`t`	Always returns `t`

Example

Define a method for the class `myClass` (to be applied to all instances of `myClass` if it is redefined):

```
(defmethod updateInstanceForRedefinedClass ((obj myClass) added deleted
dplList @rest initargs)
;;callNextMethod for obj and pass ?arg "myArg" value for slot arg
(apply callNextMethod obj added deleted dplList ?arg "myArg" initargs)
```

3 Generic Specializers

ilArgMatchesSpecializer

```
ilArgMatchesSpecializer(
     U_genericfuncObj
     s_specClass
     s_specArg
     )
     => t / nil
```

Description

Checks if the given argument matches generic specializer

Arguments

U_genericfuncObj	Specifies a generic function object
s_specClass	Specifies the generic specializer class
s_specArg	Specifies the specializer argument

Value Returned

`t`	Returns `t` if the given argument matches
`nil`	Returns `nil` if the given argument does not match

Example

```
(defmethod ilArgMatchesSpecializer (gf (theClass mySpec) arg)
(eq arg->type 'polygon))
```

ilEquivalentSpecializers

```
ilEquivalentSpecializers(
     U_genericfuncObj
     s_spec1
     s_spec2)
     => t / nil
```

Description

Defines a method to check if two specializers are equal (required during method redefinition).

Arguments

U_genericfuncObj	Specifies the generic function object
s_spec1	Specifies the first specializer
s_spec2	Specifies the second specializer

Value Returned

`t`	Returns `t` if the two specializers are equal
`nil`	Returns `nil` if the two specializers are not equal

Example

```
(defmethod ilEquivalentSpecializers(gf spec1 spec2)
classOf(spec1) == classOf(spec2))
```

ilGenerateSpecializer

```
ilGenerateSpecializer(
     U_genericfuncObj
     s_specClass
     s_specArg)
     => g_expression
```

Description

Returns a SKILL expression that makes an instance of the given specializer class and optionally set the slots. In the generated SKILL expression, *`s_specArg`* can be used to initialize the slots.

Arguments

`U_genericfuncObj`	Specifies the generic function object
`s_specClass`	Specifies the generic specializer class
`s_specArgs`	Specifies the evaluated specializer arguments that are defined in `defmethod`

Value Returned

`g_expression`	Returns a SKILL expression that is to be evaluated inside `defmethod`

Example

```
; create an instance without any slot
(defmethod ilGenerateSpecializer (gf (specName t) specArgs)
    `(makeInstance ',specName)
```

ilSpecMoreSpecificp

```
ilSpecMoreSpecificp(
     U_genericfuncObj
     s_spec1
     s_spec2
     s_specArg)
     => t / nil
```

Description

Checks if *spec1* is more specific than *spec2*. You need to define all required `ilSpecMoreSpecificp` methods for all existing custom specializers (so that the system can find a method to compare any pair of custom specializers).

Arguments

U_genericfuncObj	Specifies a generic function object
s_spec1	Specifies the first generic specializer class
s_spec2	Specifies the second generic specializer class
s_specArg	Specifies the specializer argument

Value Returned

`t`	Returns `t` if spec1 is more specific than spec2
`nil`	Returns `nil` to indicate that the custom specializer definition is inconsistent

Example

```
(defmethod ilSpecMoreSpecificp (gf (spec1 classSpec1) (spec2 classSpec2)
args)spec1->value > spec2->value)
```

4

Subclasses and Superclasses

subclassesOf

```
subclassesOf(
     u_classObject
     )
     => l_subClasses
```

Description

Returns the ordered list of all (immediate) subclasses of *`u_classObject`*. Each element in the list is a class object.

The list is sorted so that each element of the list is a subclass of the remaining elements.

Arguments

`u_classObject`	A class object.

Value Returned

`l_subClasses`	The list of subclasses. If the argument is not a class object, then *`l_subClasses`* is `nil`.

Example

```
L = superclassesOf( findClass( 'fixnum ))
subclassesOf( findClass( ’primitiveObject ) )
      => (class:list class:port class:funobj class:array class:string
             class:symbol class:number
         )


      subclassesOf( 5 ) => nil
```

subclassp

```
subclassp(
     u_classObject1
     u_classObject2
     )
     => t | nil
```

Description

Predicate function that checks if *`classObject1`* is a subclass of *`classObject2`*.

A class *`C1`* is a subclass of class *`C2`* if *`C2`* is a (direct or indirect) superclass of *`C1`*.

Arguments

`u_classObject1`	A class object.
`u_classObject2`	A class object.

Value Returned

`t / nil`	*`s_class2`* is a (direct or indirect) superclass of *`s_class1`*.

Example

```
subclassp( findClass( 'Point ) findClass( 'standardObject )) => t
subclassp(
    findClass( 'fixnum )
    findClass( 'primitiveObject ))
=> t
subclassp(
    findClass( 'standardObject )
    findClass( 'primitiveObject )
    )
=> nil
```

Reference

superclassesOf

superclassesOf

```
superclassesOf(
     u_classObject
     )
     => l_superClasses
```

Description

Returns the ordered list of all super classes of `u_classObject`. Each element in the list is a class object.

The list is sorted so that each element of the list is a subclass of the remaining elements.

Note: If a class is inherited from multiple classes, `superclassesOf()` traverses the entire inheritance tree and returns the linearized class list.

Arguments

`u_classObject`	A class object.

Value Returned

`l_superClasses`	The list of super classes. If the argument is not a class object, then *`l_superClasses`* is `nil`.

Example

```
defclass(basicA () ())
defclass(basicB () ())
defclass(derived1 (basicA) ())
defclass(derived2 (basicA basicB) ())
superclassesOf(findClass('derived1))
=> (class:derived1 class:basicA class:standardObject class:t)
superclassesOf(findClass('derived2))
=> (class:derived2 class:basicA class:basicB class:standardObject class:t)
```

5

Dependency Maintenance Protocol Functions

The dependency maintenance protocol provides a way to register an object that is notified whenever a class or generic function on which it is set is modified. The registered object is called a *dependent* of the class or generic function metaobject. SKILL uses the `addDependent` and `removeDependent` methods to maintain the dependents of a class or a generic function metaobject. The dependents can be accessed using the `getDependents` method. The dependents are notified about a modified class or generic function by calling the `updateDependent` method.

These methods are described in the following section.

addDependent

```
addDependent(
     g_object
     g_dependent
     )
     => t | nil
```

Description

Registers a dependent object for given object. SKILL checks if *`g_dependent`* already exists as a dependent of *`g_object`* (using the `eqv` operator), then *`g_dependent`* is not registered again and `nil` is returned.

Arguments

`g_object`	Specifies a SKILL object, which could be a class or a generic function on which the dependent object needs to be set.
`g_dependent`	Specifies the dependent object that you want to set on the given object.

Value Returned

`t`	Returns `t` if the dependent object was successfully registered.
`nil`	Returns `nil` if the dependent object is already registered for the given object.

Example

```
addDependent( findClass('class) 'dep1)
```

This example registers the dependent object, `dep1`, for an object of class, `class`

getDependents

```
getDependents(
     g_object
     )
     => l_dependents
```

Description

Returns a list of dependents registered for the given SKILL object, which could be a class or a generic function

Arguments

g_object	Specifies a SKILL object, which could be a class or a generic function on which the dependent object needs to be added.

Value Returned

l_dependents	Returns a list of dependents registered for the given object.

Example

```
getDependents( findClass('class))
=> (dep1 dep2)
```

removeDependent

```
removeDependent(
     g_object
     g_dependent
     )
     => t | nil
```

Description

Removes a dependent object from the given object.

Note: An object can be a dependent of multiple SKILL meta objects. If an attempt is made to remove an object from a given meta object of which the object is not a dependent, `removeDependent` will return `nil` but not display any error.

Arguments

g_object	Specifies a SKILL object, which could be a class or a generic function from which the dependent object needs to be removed.
g_dependent	Specifies the dependent object that you want to remove.

Value Returned

`t`	Returns `t` if the dependent object is removed.
`nil`	Returns `nil` if the dependent object is not removed.

Example 1

```
removeDependent( findClass('class) 'dep1)
```

This example removes the dependent object, `dep1`, from the object of class, `class`.

updateDependent

```
updateDependent(
     u_class
     g_dependent
     s_notifType
     u_classObj
     )
     => t
```

Description

Updates the dependents of a SKILL object, which could be a class or a generic function, when the SKILL object is modified. The SKILL engine calls this method for each *`g_dependent`* at different times. For example, if *`g_dependent`* is a method, the SKILL engine calls `updateDependent` at the time of adding or removing the method; whereas, for dependent classes the SKILL engine calls the `updateDependent` method at the end of class creation.

Note: Your applications can implement methods on this generic function.

Arguments

`u_class`	Specifies a SKILL object, which could be a generic function or a class, for which the dependents need to be updated. Depending on the SKILL object specified, different arguments are passed. For example, in case the specified SKILL object is a generic function then the dependent object could be a generic function object or a proxy object and in case of the SKILL object is a class, then `class:class` can be specified as the dependent object.
`g_dependent`	Specifies a dependent object that you want to update.
`s_notifType`	Specifies the type of update that has occurred using the following qualifiers: `add_method`, `remove_method`, `add_class`, `redef_class`, `add_generic`, and `redef_generic`.
`u_classObj`	Specifies the class object when a new class is defined. This argument is `nil` when a class is redefined.

Value Returned

The return value is ignored.

Example 1

```
defmethod( ilUpdateDependent((proxy class) obj dep type)
  printf("updateDependent called for CLASS -- %L" classOf(proxy))
  printf(" obj : %L type : %L\n" obj type)
  printf("Dependents : %L\n" get(className(proxy) '\*dependents\*))
  printf("Dependent : %L\n" dep)
  t
)
```