cadence

SKILL++ Object System Functions Reference

Product Version 06.10
March 2003

Cadence Design Systems, Inc., 555 River Oaks Parkway, San Jose, CA 95134, USA

Contents

Before You Start

Overview information:

About This Manual

This manual is for the following users.

- Programmers beginning to program in SKILL
- CAD developers who have experience programming in SKILL, both Cadence internal users and Cadence® customers
- CAD integrators

Companion User Guide

The companion for this reference manual is Chapter 16, “SKILL+ Object System,”

in the *SKILL Language User Guide*, which

- Introduces the SKILL language to new users
- Leads users to understand advanced topics
- Encourages sound SKILL programming methods

About the SKILL Language

The SKILL programming language lets you customize and extend your design environment. SKILL provides a safe, high-level programming environment that automatically handles many traditional system programming operations, such as memory management. SKILL programs can be immediately executed in the Cadence environment.

SKILL is ideal for rapid prototyping. You can incrementally validate the steps of your algorithm before incorporating them in a larger program.

Storage management errors are persistently the most common reason cited for schedule delays in traditional software development. SKILL’s automatic storage management relieves your program of the burden of explicit storage management. You gain control of your software development schedule.

SKILL also controls notoriously error-prone system programming tasks like list management and complex exception handling, allowing you to focus on the relevant details of your algorithm or user interface design. Your programs will be more maintainable because they will be more concise.

The Cadence environment allows SKILL program development such as user interface customization. The SKILL Development Environment contains powerful tracing, debugging, and profiling tools for more ambitious projects.

SKILL leverages your investment in Cadence technology because you can combine existing functionality and add new capabilities.

SKILL allows you to access and control all the components of your tool environment: the User Interface Management System, the Design Database, and the commands of any integrated design tool. You can even loosely couple proprietary design tools as separate processes with SKILL's interprocess communication facilities.

Other Sources of Information

For more information about SKILL and other related products, you can consult the sources listed below.

Product Installation

The *Cadence Installation Guide* tells you how to install the product.

Other SKILL Development Documentation

The following are SKILL development-related documents. You can access this information directly using the CDSDoc SKILL menu.

SKILL Development Help
SKILL Development Functions Reference
SKILL Language User Guide
SKILL Language Functions Reference
Interprocess Communication SKILL Functions Reference

Related SKILL API Documentation

Cadence tools have their own application procedural interface functions. You can access the API manuals directly using the CDSDoc SKILL menu.

Design Framework II SKILL Functions contains APIs for the graphics editor, database access, design management, technology file administration, online environment, design flow, user entry, display lists, component description format, and graph browser.

User Interface SKILL Functions contains APIs for management of windows and forms.

Software Installation and License Management Reference and the *Cadence Configuration Guide* contains SKILL licensing functions.

Document Conventions

The conventions used in this document are explained in the following sections. This includes the subsections used in the definition of each function and the font and style of the syntax conventions.

Syntax Conventions

This section describes typographic and syntax conventions used in this manual.

`text`	Indicates text you must type exactly as it is presented.
`z_argument`	Indicates text that you must replace with an appropriate argument. The prefix (in this case, *`z_`*) indicates the data type the argument can accept. Do not type the data type or underscore.
`[ ]`	Denotes optional arguments. When used with vertical bars, they enclose a list of choices from which you can choose one.
`{ }`	Used with vertical bars and encloses a list of choices from which you must choose one.
`|`	Separates a choice of options; separates the possible values that can be returned by a Cadence® SKILL language function.
…	Indicates that you can repeat the previous argument.
=>	Precedes the values returned by a Cadence SKILL language function.
text	Indicates names of manuals, menu commands, form buttons, and form fields.

Important

The language requires many characters not included in the preceding list. You must type these characters exactly as they are shown in the syntax.

Data Types

The Cadence SKILL language supports several data types to identify the type of value you can assign to an argument. Data types are identified by a single letter followed by an underscore; for example, *t* is the data type in *t_viewNames*. Data types and the underscore are used as identifiers only: they are not to be typed.

The table below lists all data types supported by SKILL.

Data Types by Type

Prefix	Internal Name	Data Type
a	array	array
b	ddUserType	Boolean
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	gdm spec
h	hdbobject	hierarchical database configuration object
l	list	linked list
m	nmpIlUserType	nmpIl user type
M	cdsEvalObject	—
n	number	integer or floating-point number
o	userType	user-defined type (other)
p	port	I/O port

Data Types by Type, *continued*

Prefix	Internal Name	Data Type
q	gdmspecListIlUserType	gdm spec list
r	defstruct	defstruct
R	rodObj	relative object design (ROD) object
s	symbol	symbol
S	stringSymbol	symbol or character string
t	string	character string (text)
u	function	function object, either the name of a function (symbol) or a lambda function body (list)
U	funobj	function object
v	hdbpath	—
w	wtype	window type
x	integer	integer number
y	binary	binary function
&	pointer	pointer type

1

SKILL++ Object System Functions

Overview information:

For information on using the SKILL++ Object System functions, see Chapter 16, “SKILL++ Object System” of the *SKILL Language User Guide*.

Introduction

The central concepts of the SKILL++ Object System are class, instance, generic functions, and methods.

A *class* is a data structure template. A specific application of the template is termed an *instance*. All instances of a class have the same slots. SKILL++ Object System provides the following functions.

- `defclass` function to create a class.
- `makeInstance` function to create an instance of a class.

A *generic function* is a collection of function objects. Each element in the collection is called a *method*. Each method corresponds to a class. When you call a generic function, you pass an instance as the first argument. The SKILL++ Object System uses the class of the first argument to determine which methods to evaluate.

To distinguish them from SKILL++ Object System generic functions, SKILL functions are called **simple functions.** The SKILL++ Object System provides the following functions.

- `defgeneric` function to declare a generic function.
- `defmethod` function to declare a method.

The SKILL++ Object System provides for one class B to inherit structure slots and methods from another class A. You can describe the relationship between the class A and class B as follows:

- B is a subclass of A
- A is a superclass of B

New Functions

subclassesOf on page 38

SKILL++ Extension to SKILL

SKILL++ is the name of the second generation extension language for the CAD tools from Cadence. It combines the ease-of-use of the well-received SKILL environment with the power of the highly-acclaimed programming language, Scheme, to give users a more capable customization and extension-development platform.

For information about using SKILL++, refer to these *SKILL Language User Guide* chapters:

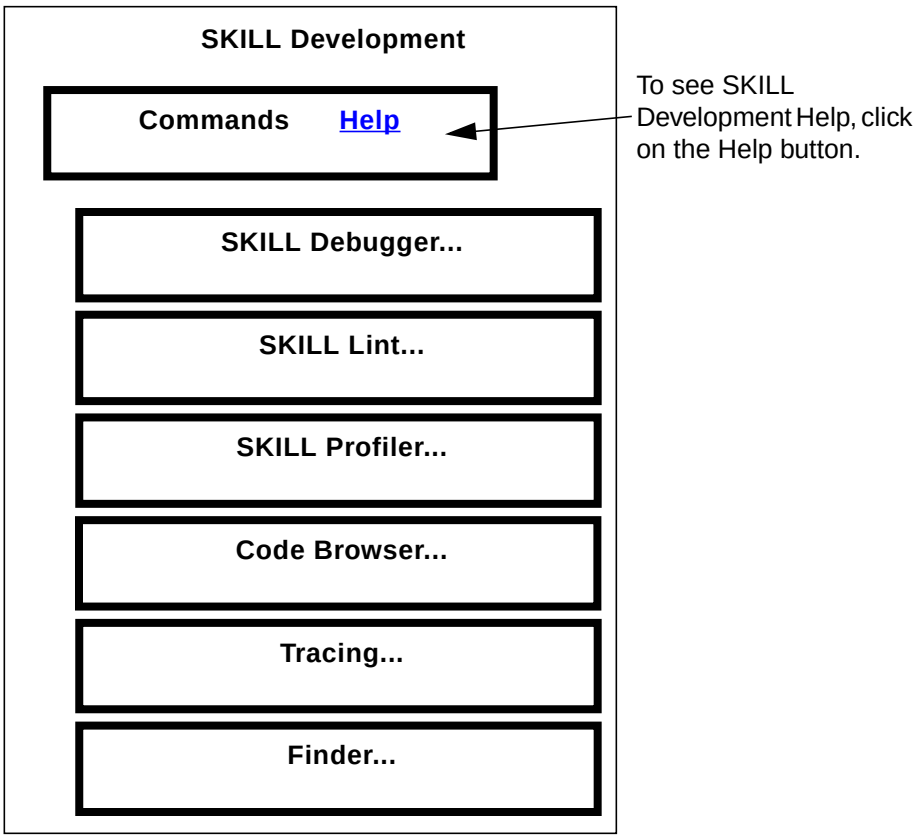
Chapter 13, “About SKILL and SKILL++” introduces the Cadence-supplied Scheme called SKILL++, which is the second generation extension language for the CAD tools from Cadence.

Chapter 14,”Using SKILL++” focuses in detail on the key areas in which SKILL++ semantics differ from SKILL semantics.

Chapter 15, “Using SKILL and SKILL++ Together” discusses the pragmatics of developing SKILL++ programs. It identifies several more factors to consider in creating applications which involve tightly integrated SKILL and SKILL++ components, such as, selecting a language, partitioning an application, cross calling between SKILL and SKILL++, and debugging a SKILL++ program.

Chapter 16, “SKILL++ Object System” describes a system that allows for object-oriented interfaces based on classes and generic functions composed of methods specialized on those classes. A class can inherit attributes and functionality from another class known as its superclass. SKILL++ class hierarchies result from this single inheritance relationship.

SKILL Development Help



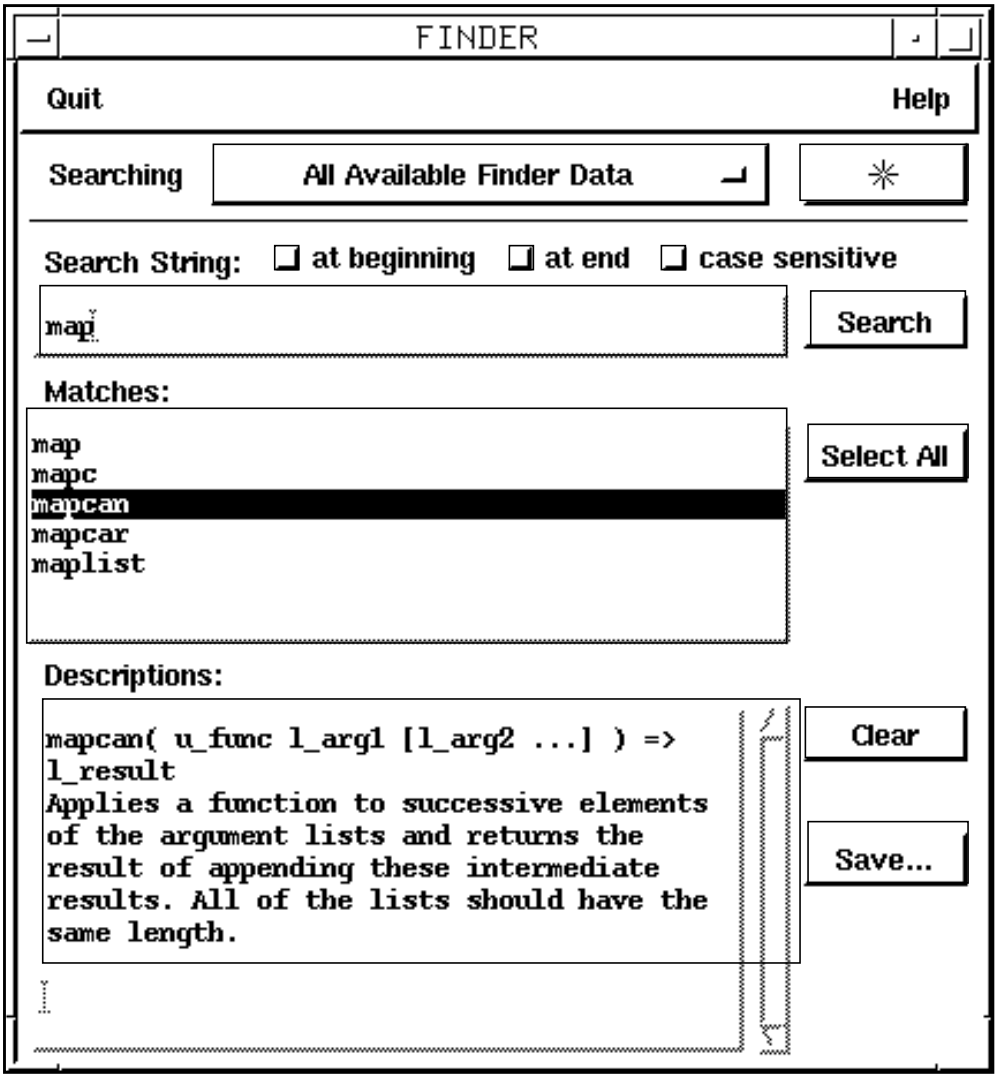
Information about the SKILL Development Toolbox is available in SKILL Development Help, which you access by clicking the *Help* button on the toolbox. Use this source for toolbox command reference information.

The Walkthrough topic in this help system identifies and explains the tasks you perform when you develop SKILL programs using the SKILL Development Toolbox. Using a demonstration program, it explains the various tools available to help you measure the performance of your code and also look for possible errors and inefficiencies in your code. It includes a section on working in the non-graphical environment.

For a list of SKILL lint messages, and message groups, refer to the *SKILL Development Help*.

Quick Reference Tool - Finder

Quick reference information for syntax and abstract statements for SKILL language functions and application procedural interfaces (APIs) is available using the Finder, a new tool accessible from the SKILL Development Toolbox or from UNIX.



For more information refer to “Finder” in *SKILL Development Help*.

Copying and Pasting Code Examples

You can copy examples from CDSDoc windows and paste the code directly into the CIW or use the code in nongraphics SKILL mode.

To select text,

- Press `Control-drag left mouse` to select a text segment of any size.
- Press `Control-double click left mouse` to select a word.
- Press `Control-triple click left mouse` to select an entire section.

SKILL Functions

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 args.

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

className

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

Description

Returns the class symbol denoting a class object.

For user-defined classes, `s_classNam`e 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

defclass, 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.

Example

```
classOf( 5 )               => funobj:0x1840d8
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_superClassName ] )
         ( [ ( 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 and slot specifications. This is a macro form.

If the super class is not given, the default super class is the `standardObject` class. Each slot specifier itself is a list composed of slot options. The only required slot option is the slot name.

Arguments

s_className	Name of new class.
s_superClassName	Name of super class. 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

```
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 )
```

Reference

`makeInstance`

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 *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, defclass

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

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

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

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.

Prequisites

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

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`

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.

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
```

Reference

slotValue

slotValue

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

Description

Returns the value contained in the slot *s_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

defclass( Point
      ( GeometricObject)
      (
            ( name @initarg name )
      )
)                                => t

p = makeInstance( 'Point ?name "P" ?x 0 ?y 10 )
                                 => stdobj:0x355024
```

```
slotValue( p 'name )            => "P"
slotValue( p 'x )               => 0
slotValue( p 'y )               => 10
```

Reference

setSlotValue

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
```

Reference

classNameclassOffindClass

subclassp

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

Description

Predicate function that checks if *u_classObject1* is a subclass of *u_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

defclass, findClass, 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.

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

```
L = superclassesOf( findClass( 'fixnum ))
=> (funobj:0x1840d8 funobj:0x1840c8
        funobj:0x184048funobj:0x184038 funobj:0x184018)

foreach( mapcar classObject L
    className( classObject )
    )
=> (fixnum number primitiveObject systemObject t)

superclassesOf( 5 ) => nil
superclassesOf( classOf( 5 )) == L
=> t
```

Reference

className, classOf, findClass