3. CLASSES

Classes provide a description of instances within the object domain. The following information is contained within a class:

- The metaclass for this class. See Chapter 4, Metaclasses, for a discussion of metaclasses.
- Class Properties. xamples of class properties are an edit stamp and documentation.
- The supers list for this class. Classes exist in a hierarchy and the supers list places the class within that hierarchy. Instances of the class contain data and respond to messages that are described within the class and superclasses of the class.
- Class variables, their values, and their properties and vlues.
- Instance variables, their default values, and their properties and values.

This chapter covers creating and destroying classes, editing, accessing data stored in classes, inheritance, and related topics. Other chapters that contain information relevant to this chapter are Chapter 4, Metaclasses, since a metaclass is a class of classes, and Chapter 10, Browsers, since the primary user interface for manipulating classes is the browser.

3.1 Creating Classes

Several ways are available to create a class:

- Use the browser interface.
- Use function calling or message sending.
- Use dynamic mixins to dynamically create classes.

The rules for naming classes are the same as those for naming instances. Simply stated, a class name must be a litatom. One exception to this rule is the naming of dynamic mixin classes, which is discussed later in this chapter.

A class is generally referred to with this form: ($ className). See Chapter 2, Instances, for more details regarding LOOPS names.

As discussed in Chapter 2, Instances, the protocol that is followed when instances are created is for the LOOPS system to send the **NewInstance** message to the newly created instance. The **NewInstance** message can be specialized to incorporate behavior specific to the creation time of an instance. Similarly, the system follows a prototol when creating a class using the **New** message. After the class is created, it is sent the **NewClass** message.

3.1.1 Function Calling and Message Sending

The following table shows the items in this section.

Name	Type	Description
DefineClass	Function	Creates a new class.
New	Method	Creates a new class.
CreateClass	Method	Creates a new class.
NewClass	Method	Provides a placeholder for modifying the class creation protocol.

(**DefineClass** *name supers self*) [Function]

Purpose: Creates a new class.

Behavior: If *name* is not a litatom, a break occurs.

- If *supers* is non-NIL, it should be a list of classes or names of classes to be the supers for the newly created class. If the list contains multiple classes, this results in a class that has multiple super classes (see Section 3.3, "Inheritance"). The order of classes in the list specifies the order in which lookup will proceed. If one of the these classes is not a valid class, a break occurs.
- If *supers* is NIL and if *self* is ($ MetaClass), then the supers list is (Class).
- If both *supers* is NIL and *self* is NIL, the supers list is (Object).

If *self* is non-NIL, it is installed as the metaclass for the newly created class. See Chapter 4, Metaclasses.

A class is then built with an **Edited:** property containing the date and time and the value of variable **INITIALS**. (See the *Interlisp-D Reference Manual*.)

The newly created class has no class variables, instance variables, or methods.

The variable **LASTWORD** is set to *name*, which is added to **USERWORDS** for spelling escape completion. (See the *Interlisp-D Reference Manual* for information on **LASTWORD** and **USERWORDS**.)

Arguments:

name A LOOPS name to be given to the class.

supers A list of classes.

self A metaclass.

Returns: The class object.

Examples: The following command defines a subclass of the class **Object**.

```
(DefineClass 'ExampleClass)
```

The following command defines a subclass of the class **Window**.

```
(DefineClass 'MyClass '(Window))
```

The following command defines a class with multiple supers: **ExampleClass** and **Window**.

```
(DefineClass 'AnotherClass '(ExampleClass Window))
```

The following command defines a subclass of the class **Window** that has **AbstractClass** as its metaclass.

```
(DefineClass 'DontMakeMe '(Window) ($ AbstractClass))
```

(← *class* **New** *name supers init1 init2 init3*) [Method of Metaclass]

Purpose: Creates a new class.

Behavior: Sends the message **CreateClass** to *class*, passing the arguments *name* and *supers*. This returns a new class which is then sent the message **NewClass** passing the arguments *init1*, *init2*, and *init3*.

Arguments:
- *class* A pointer to a class.
- *name* A LOOPS name to be given to the class.
- *supers* A list of classes.
- *init1, init2, inti3* See Behavior.

Returns: The new class.

Categories: Object

Specializes: Class

Specializations: AbstractClass

Example: The following command creates the class, **AClass**, which is a subclass of the class **Window**. The metaclass of **AClass** is **Class**.

```
(← ($ Class) New 'AClass '(Window))
```

After **AClass** is created, the system sends the `following message:`

```
(← ($ AClass) NewClass)
```

(← *self* **CreateClass** *name supers*) [Method of Metaclass]

Purpose: Creates a new class.

Behavior: Method version of **DefineClass**.

Arguments:
- *self* A metaclass.
- *name* The name of the newly created class.
- *supers* A list of classes.

Returns: The clsss object.

Categories: MetaClass

(← *class* **NewClass** *init1 init2 init3*) [Method of Class]

Purpose: Provides a hook into class initialization. If you want special actions to occur when creating a class, specialize this method.

Arguments:
- *class* A pointer to a class.
- *init1, init2, init3* Dependent on user-defined functionality.

Returns: *class*

Categories: Class

Example: Create a subclass of **Class** called **MyClass**:

```
(DefineClass 'MyClass '(Class))
```

Give it a method **NewClass:**

```
(DefineMethod ($ MyClass) 'NewClass '(init1 init2 init3)
'(PROGN (PutClass self init1 'prop1) self))
```

This looks like the following display editor window:

```
SEdit MyClass.NewClass  Package: INTERLISP
(Method ((MyClass NewClass) self init1 init2 init3)
        ;; This demonstrates the NewClass protocol
        (PutClass self init1 'prop1) self)
```

Now send the class **MyClass** the following command:

```
(← ($ MyClass) New 'testclass NIL "this is a test")
```

This results in the creation of the class shown in the following display editor window:

```
SEdit #,($C testclass)  Package: INTERLISP
((MetaClass MyClass Edited%:    ; Edited 2-Dec-87
                                ; 15:24 by
                                ; Martin.pasa
            prop1 "this is a test")
 (Supers Object) (ClassVariables)
 (InstanceVariables) (MethodFns))
```

To display the class, enter

```
(← ($ testclass) Edit)
```

3.1.2 Dynamic Mixins

In some programming situations, you may develop sets of mixins that are designed to be used together. (Mixins are classes that are used only in conjunction with another class to create a subclass, or provide some functionality useful in more than one class.) For example, the class **NamedClass** adds one instance variable **name** and specializes the **New** message to ensure that the instance variable **name** contains the name of the object.

```
(DefineClass 'NamedClass)
(← ($ NamedClass) AddIV 'name)
(DefineMethod ($ NamedClass) 'New '(self name)
'(←@ (←self NewInstance name) name name))
```

Other classes that want the names of their objects in an instance variable **name** can use **NamedClass** as a mixin.

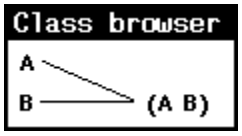
As another example, assume that you have one set consisting of **A1**, **A2**, **A3**, and **A4** and another set containing **B1**, **B2**, and **B3**. Formerly, to allow creation of an instance taking properties from arbitrary combinations of an element from each set, you had to create in advance all 12 combinations of classes with a super from **A** and a super from **B**. This was even more cumbersome if the **A**s and **B**s can also combine with any of a set of 5 **C**s.

What is desired is the ability to create combinations of these classes on the fly, without having to invent a name for each combination and without having each present in the system when only a few may be needed in any given application. To meet this need, LOOPS now provides the dynamic mixin class. The name of such a class is a list, in order, of the classes which are to be the supers of the class. Such a class is automatically created the first time it is referred to. Thus, the following sequence

```
(DefineClass 'A)
(DefineClass 'B)
(← ($ (A B)) New)
```

creates the class whose supers are **A** and **B** (if it did not already exist), and builds an instance of that class.

Dynamic mixins appear in browsers as shown in this sample window.



All of the browser operations still function on dynamic mixin classes.

These classes print as

```
#,($C (A B))
```


3.2 DESTROYING CLASSES

3.2 Destroying Classes

The following messages have been provided to destroy a class that has been created. Destroyed classes, if not being pointed to in some fashion, are eventually collected by the garbage collector.

The following table shows the methods in this section.

Name	Type	Description
Destroy	Method	Removes a class from the LOOPS system.
Destroy!	Method	Destroys a class and its subclasses.
DestroyClass	Method	Destroys a class by deleting its contents.

(← *class* **Destroy**) [Method of Class]

Purpose: Removes a class from the LOOPS system.

Behavior: If *self* has any subclasses, a break occurs and you are prompted to determine if you want to use **Destroy!**.

Sends the message **DestroyClass** to the metaclass of *self*.

Specializations of this method may be necessary to undo any actions that might have been performed by user specializations of the **NewClass** method. If you specialize **Destroy**, be sure to include a ←**Super** to guarantee that the functionality of the **Destroy** method is performed.

Arguments: *class* Must be a class.

Returns: NIL

Categories: Object

Specializes: Object

Specializations: DestroyedClass

Example: The following command destroys the class **Datum**:

```
(← ($ Datum) Destroy)
```

(← *class* **Destroy!**) [Method of Class]

Purpose: Destroys a class and its subclasses.

Behavior: Recursively sends the **Destroy** message to *self* and its subclasses.

Arguments: *class* Must be a class.

Returns: NIL

Categories: Object

Specializes: Object

Specializations: DestroyedClass

(← *class* **DestroyClass** *classToDestroy*) [Method of Class]

Purpose: Destroys *classToDestroy* by deleting its contents. This method is invoked by the LOOPS system and should generally not be called directly by user code. However, it can be specialized to change the way classes are destroyed.

Behavior: Performs the following actions:

- Removes *classToDestroy* from any files on **FILELST**.
- Sends the **Destroy!** message to all methods locally associated with *classToDestroy*.
- Removes *classToDestroy* from any subclass data contained in the supers of *classToDestroy*.
- Changes the class name of *classToDestroy* to ***aDestroyedClass***.
- Changes the supers list of *classToDestroy* to **DestroyedObject** and **Object**.
- Changes the metaclass of *classToDestroy* to **DestroyedClass**.
- Sets other fields of the internal class data structure to NIL.

Arguments: *class* Metaclass of *classToDestroy*.

classToDestroy
Class to destroy.

Returns: NIL

Categories: Class

Specializations: DestroyedClass

3.3 Inheritance

Classes exist in an ordered lattice or hierarchy. Information contained within a class - the supers list - defines where that class is located within the lattice. The supers list specifies the classes immediately above a given class. When an instance of a class is created, it contains not only the instance variables of the defining class, but also the instance variables of all of the classes above the defining class in the class hierarchy. When you try to determine the value of a class variable associated with an instance, all classes above the defining class may be searched. When you send a message to an instance, all classes above the defining class may be searched for the appropriate method.

There are two types of inheritance:

- Simple, in which a class has only one superclass.
- Multiple, in which a class has two or more classes on its supers list.

When an instance is created, it may contain an instance variable that is defined in more than one class. The default value for that instance variable depends on its inheritance. In the case of simple inheritance, the instance variable gets the value from the class that is lowest in the hierarchy. In multiple inheritance, the instance variable gets the value from the class that is lowest in an inheritance list. To create this list,

1. Put the first class that describes the instance.
2. Begin with the first class on its supers list and move up from it, making a list of classes which assume simple inheritance.
3. Build one of these lists for all successive super classes.
4. Append these lists together.
5. Remove all occurrences of any classes that appear in the list a multiplenumber of times except for the last entry.

Another way to think about this, which creates the same inheritance list, is the following:

1. Begin with the first super class and walk up the hierarchy until you reach a class where the inheritance paths merge.
2. Walk up each path leading from each successive super class to where paths merge.
3. Take the class where the paths merge and walk up from there.

As an example of simple inheritance, examine Figure 3-1 which shows some of the class variables and instance variables defined within each class.

Figure 3-1. Simple Inheritance Lattice

An instance of the class **ClassBrowser** has this as an inheritance list:

ClassBrowser
LatticeBrowser
Window
Object
Tofu

The instance variable values of this instance are as follows:

IV	Value	From Class
title	"Class browser"	**ClassBrowser**
width	64	**LatticeBrowser**
height	32	**LatticeBrowser**
menus	T	**Window**

Accessing the value of the class variable **LeftButtonItems** causes this value to come from the class **ClassBrowser**.

Figure 3-2 shows an example of multiple inheritance.

Unknown IMAGEOBJ type
GETFN: SKIO.GETFN.2

Figure 3-2. Multiple Inheritance Lattice

If the order of the supers for **Class5** is **Class3** and then **Class4** (that is, its supers list is (Class3 Class4)), then the inheritance list for an instance of **Class5** is as follows:

Class5
Class3
Class4
Class2
Class1

The instance variable and class variable values this instance are as follows:

IV	Value	From Class	CV	Value	From Class
iv1	11	**Class1**	**cv1**	A4	**Class4**
iv2	22	**Class2**	**cv2**	B	**Class1**
iv3	33	**Class3**	**cv3**	C	**Class3**
iv4	45	**Class5**	**cv4**	D4	**Class4**

3.4 Editing Classes

Changing the contents of a class typically involves using the display editor, although programmatical ways to make these changes are available. To edit a class structure, the LOOPS system first changes the structure to a list and then passes that list to the display editor. Upon exit from the display editor, the system translates the modified list back into the class structure.

The editor is most often called from the browser interface. (See Chapter 10, Browsers.) The following method provides a programmatical way to invoke the editor.

(← *class* **Edit** *commands*) [Method of Class)

Purpose: Edits a class definition.

Behavior: Calls **EDITE** (see the *Interlisp-D Reference Manual*) with the translated class structure passed as the **EXPR** argument and *commands* passed as the **COMS** argument.

This method binds the variable **LASTCLASS** to the class name of *self*.

Arguments: *class* Pointer to a class.

commands Commands passed to **EDITE**.

Returns: Name of the class.

Categories: Object

Specializes: Object

Example: The following command causes a display editor window to appear.

```
(← ($ LoopsIcon) Edit)
```

Calling the editor causes a structure to appear in a display editor window. At this time, you can change the structure of the class by using any of the following techniques:

- Changing the value of the class's metaclass. This is done by changing th class name after the word **MetaClass**.
- Changing the superclasses for the class. The form for this is :

  ```
  (Supers class1 class2 ...)
  ```

  At least one class must be in the supers list. The order of this list determines the order of inheritance; the first class after the word **Supers** on this list is the first class to search for inherited data and methods.
- Adding or removing class properties. Class properties occur within the same list as MetaClass, after the metaclass class name. The form for this is

  ```
  (MetaClass metaclassName classProp1 propVal1 classProp2
  propVal2 ...)
  ```
- Adding or removing class variables or associated properties. The form for class variables is:

  ```
  (ClassVariables
  (cvName1 cvVal1 prop1a propVal1a prop1b propVal1b ...)
  (cvName2 cvVal2 prop2a propVal2a prop2b propVal2b ...)
  ...)
  ```

  It is not necessary to have any properties for a class variable. If the length of each class variable list is not an even number, a break occurs under the editor. The message in the break window describes an odd length list the first time you try to exit from the editor.
- Adding or removing instance variables or associated properties. These have the same form as class variables with the distinction that the value listed for each instance variable is not its value, but only its default value for the purposes of instanciation.

For example, examine the display editor window in Figure 3-3.

```
SEdit #,($C IndirectVariable)  Package: INTERLISP
((MetaClass Class doc
            (* Active Value for redirecting references to another
               variable)
            Edited%: (* smL " 9-May-86 09:52"))
 (Supers ActiveValue) (ClassVariables)
 (InstanceVariables
       (object NIL doc (* The object with the "real" variable))
       (varName NIL doc (* The name of the "real" variable))
       (propName NIL doc (* The prop name of the "real" variable))
       (type NIL doc (* The type of the "real" variable)))
 (MethodFns IndirectVariable.GetWrappedValueOnly
            IndirectVariable.PutWrappedValueOnly
            IndirectVariable.WrappingPrecedence))
```

Figure 3-3. Sample Display Editor Window

This figure shows the following information:

- The title bar of the display editor window indicates the class being edited.
- The metaclass of the class **IndirectVariable** in this example is the class **Class**. **IndirectVariable** has two class properties. The first is a **doc** property. The second is an **Edited:** property.
- This class has one super class: **ActiveValue**.
- This class has no class variables. It has four instance variables: object, varName, propName, and type. Each has a **doc** property.
- The **MethodFns** are listed in this structure as a convenience. It is not possible to add or delete elements of this list from the editor and have any changes actually occur. Selecting one of the method function names and then selecting Edit (Meta-O in SEdit) allows you to edit that method either as its method code (METHOD-FNS), its method object (METHODS), or its Interlisp code (FNS).

3.5 MODIFYING CLASSES

3.5 Modifying Classes

In addition to the editing technique for changing a class, you can use programmatic means to modify the structure of a class. This section describes the functions and methods for modifying classes.

Name	Type	Description
Add	Method	Adds a component to a class.
Delete	Method	Deletes a component from a class.
DeleteClassProp	Function	Removes a class property from a class.
AddCV	Function	Adds a class variable to a class.
AddCV	Method	Adds a class variable to a class.
DeleteCV	Function	Deletes a class variable or one of its properties from a class.
AddCIV	Function	Adds an instance variable to a class; can also add properties to a class.

AddIV	Method	Adds an instance variable to a class.
DeleteCIV	Function	Removes an instance variable or property from a class.
ReplaceSupers	Method	Changes the super classes of a class.

(← *class* **Add** *type name value prop*) [Method of Class]

Purpose/Behavior: Adds a component to a class.

Arguments:

class	Pointer to a class.
type	One of IV, IVPROP, CV, CVPROP, METACLASS, or METHOD.
name	The name of the item to be added.
value	The value, or default value if *type* is one of IV or IVPROP.
prop	The name of the property, if a property is to be added.

Returns: NIL

Categories: Class

Example: The following command adds a new instance variable **color** to class **Datum**:

```
(← ($ Datum) Add 'IV 'color)
```

(← *class* **Delete** *type name prop*) [Method of Class)

Purpose: Deletes a component from a class.

Behavior: Varies according to the arguments.

- If *type* is one of IV, IVPROP, or NIL, this calls (**DeleteCIV** *class name prop*).
- If *type* is one of CV or CVPROP, this calls (**DeleteCV** *class name prop*).
- If *type* is META, METACLASS, or CLASS, and if *prop* is NIL, then the metaclass of *self* is changed to the class **Class**.
- If *type* is META, METACLASS, or CLASS, and if *prop* is non-NIL, then this calls (**DeleteClassProp** *class prop*).
- If *type* is METHOD or SELECTOR, this calls (**DeleteMethod** *class name prop*).

Arguments:

class	A pointer to a class.
type	See Behavior.
name	IV, CV, or selector name.
prop	A property name.

Returns: NIL

Categories: Class

Example: The following command deletes the instance variable **color** from the class **Datum**:

```
(← ($ Datum) Delete 'IV 'color)
```

(**DeleteClassProp** *classRec propName*) [Function]

Purpose: Removes a class property from a class.

Behavior: Marks *classRec* as changed.

Arguments: *classRec* Pointer to a class.

propName Property to be deleted.

Returns: NIL is *propName* is not found; otherwise *propName*.

(**AddCV** *class varName newValue*) [Function]

Purpose: Adds a class variable to a class.

Behavior: Varies according to the arguments.

- If *varName* is NIL, you are prompted to enter a name.
- If *varName* is already a class variable, its value is changed to *newValue*. NIL is returned.
- If *varName* is not a class variable of *class*, it is added to *class* with the value *newValue*. Also, a **doc** property is added with the following value:

```
'(* CV added by , (USERNAME NIL T))
```

varName is returned in this case.

Arguments: *class* A pointer to a class.

varName Name of the new variable.

newValue The new value.

Returns: Value depends on the arguments; see Behavior**.**

(← *class* **AddCV** *varName newValue*) [Method of Class]

Purpose: Adds a class variable to a class.

Behavior: Provides a method version of the function **AddCV**.

Arguments: See the function **AddCV**.

Returns: NIL

Categories: Class

(**DeleteCV** *class varName prop*) [Function]

Purpose: Deletes a class variable or one of its properties from a class.

Behavior: Marks *class* as changed.

Arguments: *class* Pointer to a class.

varName Class variable name to be deleted.

prop Property to be deleted.

Returns: NIL, if *varName* is not found, else *varName*.

(**AddCIV** *class varName defaultValue otherProps*) [Function]

Purpose: Adds an instance variable, and perhaps properties, to a class.

Behavior: If the length of *otherProps* is odd, an error occurs.

The remaining behavior varies according to the arguments.

- If *varName* is NIL, you are prompted to enter a name.
- If *varName* is already an instance variable of *class*, then change its default value to *defaultValue*. Properties on *otherProps* are added or changed as necessary. NIL is returned.
- If *varName* is not an instance variable of *class*, it is added to *class* and its default value is *defaultValue*. Properties on *otherProps* are also added. If there is no **doc** property, it is added and given the following value:

  ```
  '(* IV added by , (USERNAME NIL T))
  ```

  varName is returned in this case.

Arguments:

class Must be a pointer to a class.

varName New instance variable name.

defaultValue New default value.

otherProps NIL or a list in property list format.

Returns: Value depends on the arguments; see Behavior.

(← *class* **AddIV** *varName defaultValue otherProps*) [Method of Class]

Purpose: Adds an instance variable to a class.

Behavior: Provides a method version of the function **AddCIV**.

Arguments: See the function **AddCIV**.

Returns: NIL

Categories: Object

Specializes: Object

Example: Define a new class **TestClass**, add an instance variable **testIV** with two properties **testProp1** and **testProp2**, all with initial values, and then prettyprint the class's variables.

```
64←(DefineClass 'TestClass)
#,($C TestClass)

65←(← ($ TestClass) AddIV 'testIV 1234
'(testProp1 1 testProp2 2))
testIV

66←(← ($ TestClass) PPV! T)
#,($ TestClass)
MetaClass and its Properties
  Class  Edited: (* edited: 24-Sep-87 08:41 by mcgill)
Supers
```

```
  (Object Tofu)
Instance Variable Descriptions
  testIV 1234 doc (* IV added by MCGILL)
testProp2 2 testProp1 1
Class Variables
```

(**DeleteCIV** *class varName prop*) [Function]

Purpose: Removes an instance variable or property from a class.

Behavior: If *class* does not have *varName*, a break occurs.

Marks *class* as changed.

Arguments:

class	Pointer to a class.
varName	Instance variable to be deleted.
prop	If non-NIL, property to be deleted.

Returns: Value depends on the arguments.

- NIL for removing an instance variable if successful.
- *prop* for removing a property if successful.
- NIL if *prop* is not a property.

(← *class* **ReplaceSupers** *supers*) [Method of Class]

Purpose: Changes the super classes of a class.

Behavior: Checks that no circular lists can be made in the inheritance lattice.

- If the super class of *class* is Tofu, no change occurs.
- If supers is different from the current supers, the supers list of *class* is changed and *class* is marked as changed.

Arguments:

class	Pointer to a class.
supers	A list of class names or classes.

Returns: NIL

Categories: Class

3.6 METHODS FOR MANIPULATING CLASS NAMES

3.6 METHODS FOR MANIPULATING CLASS NAMES

3.6 Methods for Manipulating Class Names

LOOPS classes must have one and only one LOOPS name. The following functions and methods allow you to change and rename class names.

Name	Type	Description
Rename	Method	Changes the name of a class. Prompts for name if not provided, then calls **SetName**.
SetName	Method	Changes the name of a class.

UnSetName	Method	Unnames a class.
ClassName	Function	Finds the class name of an object.

(← *class* **Rename** *newName*) [Method of Class]

Purpose: Changes the name of a class. Prompts for name if not provided, then calls **SetName**.

Behavior: Varies according to the argument.

- If *newName* is NIL, this causes a break and prompts you for a name. **Rename** then sends the message **SetName** passing this name as an argument
- If *newName* is non-NIL, **Rename** sends the message **SetName** passing *newName* as an argument.

Arguments: *class* Pointer to a class.

newName A litatom.

Returns: NIL

Categories: Object

Specializes: Object

Example: The following command renames class **Datum** to **Thing**:

```
(← ($ Datum) Rename 'Thing)
```

(← *class* **SetName** *newClassName*) [Method of Class]

Purpose: Changes the name of a class.

Behavior: Removes the old name of *self* from **ObjNameTable**.

SetName uses the Interlisp-D function **EDITCALLERS** to rename references to the class name or any file that contains the class. If **EDITCALLERS** cannot succeed, for example, when a file is not RANDACCESSP, a message is printed that the class cannot be renamed on that file. For complete information on **EDITCALLERS**, see the *Interlisp-D Reference Manual*.

The names of the method functions of *class* are changed to use *newClassName*.

Arguments: *class* Pointer to a class.

newClassName A litatom.

Returns: NIL

Categories: Object

Specializes: Object

(← *class* **UnSetName**) [Method of Class]

Purpose: Unnames a class, but does not destroy it. Has limited usefulness for keeping a class name from being typed in.

Behavior: Removes *class* from the LOOPS name hash table and from any files on **FILELST**. This method is intended to be used internally only; it is not recommended to create an unnamed class.

Arguments: *class* Pointer to a class.

Returns: NIL

Categories: Object

Specializes: Object

(**ClassName** *self*) [Function]

Purpose: Finds the class name of an object.

Behavior: Varies according to the arguments.

- If *self* is a class, this returns the name of that class.
- If *self* is an instance, this returns the name of the class that describes that instance.
- If *self* is neither a class or an instance, this returns **Tofu**.

Arguments: *self* See Behavior.

Returns: Value depends on the arguments; see Behavior.

Example: Given that

```
(← ($ Window) New 'w1)
```

the commands

```
(ClassName ($ w1))
(ClassName ($ Window))
```

both return **Window**.

3.7 QUERYING THE STRUCTURE OF A CLASS

3.7 Querying the Structure of a Class

The following functions and methods allow you to query what is contained in a class.

Name	Type	Description
GetClassProp	Method	Obtains a class's metaclass or properties.
HasAttribute	Method	Determines whether *self* has an attribute name.
HasAttribute!	Method	Recursive form of **HasAttribute**, but works only on classes.
HasCV	Method	Determines if a class has a class variable with a specified property.
HasItem	Method	Determines if a class has an item of a given type.
HasIV	Method	Determines if a class has an instance variable with a specified property.

HasIV!	Method	Same as **HasIV**, except that **HasIV!** also searches up the supers chain.
ListAttribute	Method	Lists the elements of a class that are local to the class.
ListAttribute!	Method	Lists all the items associated with a class.
WhoHas	Function	Determines what classes contains a specified item.

(← *class* **GetClassProp** *propname*) [Method of Class]

Purpose: Obtains a class's metaclass or properties by following metaclass links.

Behavior: Varies according to the arguments.

- If *propname* is NIL, this returns the *class*'s metaclass.
- If *propname* is non-NIL, this looks first in *class* for that property. If it cannot find it there, it looks through *class*'s metaclass links.
- If no property is found, the value of the variable **NotSetValue** is returned.

Arguments: *class* A pointer to a class.

prop Property name.

Returns: Value depends on the arguments; see Behavior.

Categories: Class

Example: The following commands show the variety of responses.

```
51←(← ($ Window) GetClassProp)
#,($C Class)

52←(← ($ Window) GetClassProp 'doc)
"A LOOPS object which represents a window"

53←(← ($ IconWindow) GetClassProp 'doc)
"An icon window that appears as an irregular shaped image
on the screen -- See the ICONW Library utility"
```

(← *self* **HasAttribute** *type name propname*) [Method of Class]

Purpose: Determines whether *self* has an attribute name, with a property *propname* if supplied.

Behavior: *self* can be an instance or a class. Remaining behavior depends on *type*, which is converted to uppercase on entry:

- If *type* is IV, IVPROP, or NIL, this returns T if *self* has an instance variable of *name*, with a property called *propname* (if *propname* is non-NIL), otherwise it returns NIL.
- If *type* is CV or CVPROP, this returns T if *self* has a CV called *name*, with a property of *propname* (if *propname* is non-NIL), otherwise it returns NIL.
- If *type* is METHOD or SELECTOR, this returns NIL or the name of the method implementing *name*.

HasAttribute applied to an instance reports on the actual state of the instance; it sees all instance variables and class variables whether local,

inherited, or specially added to the instance. If only local attributes are required, use (← (Class instance) HasAttribute ...).

Arguments:
- *self* — Can be an instance or a class.
- *type* — See Behavior.
- *name* — A symbol which is looked up as the variable or method name.
- *propname* — A symbol which is looked up as the property name.

Returns: See Behavior.

Categories: Object

Specializations: Class

Example: The command

```
(← ($ LoopsIcon) HasAttribute 'IV 'icon)
```

returns T.

(← *class* **HasAttribute!** *type name propname*) [Method of Class]

Purpose: Recursive form of **HasAttribute**; only works on classes

Behavior: Similar to **HasAttribute**, but will also search through *class* 's supers.

Arguments:
- *class* — A class.
- *type* — See Behavior under **HasAttribute**.
- *name* — A symbol which is looked up as the variable or method name.
- *propname* — A symbol which is looked up as the property name.

Returns: See Behavior.

Categories: Object

Specializations: Class

Example: The command

```
(← ($ LoopsIcon) HasAttribute 'IV 'left)
```

returns NIL, but

```
(← ($ LoopsIcon) HasAttribute! 'IV 'left)
```

returns T.

(← *class* **HasCV** *cvName prop*) [Method of Class]

Purpose: Determines if a class has a class variable *cvName* with a property *prop*.

Note: The preferred form of this method is **HasAttribute** or **HasAttribute!**.

Behavior: Varies according to the arguments.

- If *prop* is NIL, this returns T if *class* contains a class variable called *cvName*, else NIL.

- If *prop* is non-NIL, this returns T if *class* contains a class variable called *cvName* with the property *prop*, else NIL.

Note: **HasCV** does not distinguish between locally defined class variables and inherited class variables. If you need to test a class to see if it has a class variable defined locally, you can use the **HasAttribute** method. For example, the form (←MyClass HasAttribute 'CV 'ABC) will return a non-NIL value if and only if the class **MyClass** has a local definition of the class variable ABC.

Arguments:
class A pointer to a class.
cvName A class variable name.
prop Property name.

Returns: NIL or T; see Behavior.

Categories: Object

Specializes: Object

Example: The command

```
(← ($ Window) HasCV 'TitleItems)
```

returns T.

(← *class* **HasItem** *itemName prop itemType*) [Method of Class]

Purpose: Determines if a class has an item of a given type.

Note: The preferred form of this method is **HasAttribute** or **HasAttribute!**.

Behavior: Varies according to the arguments.

- If *itemType* is IV or IVS, this sends the message (← *class* **HasIV** *itemName prop*).
- If *itemType* is CV or CVS, this sends the message (← *class* **HasCV** *itemName prop*).
- If *itemType* is SELECTOR, METHOD, SELECTORS, or METHODS, this finds the corresponding local method of *class*.
- If *itemType* is not one of the above, this returns NIL.

Arguments:
class Pointer to a class.
prop Property name.
itemType See Behavior.

Returns: Value depends on the arguments; see Behavior.

Categories: Class

(← *class* **HasIV** *IVName prop*) [Method of Class]

Purpose: Determines if a class has an instance variable *IVName* with a property *prop*.

Note: The preferred form of this method is **HasAttribute** or **HasAttribute!**.

Behavior: *class* should point to a class.

- If *prop* is NIL, this returns T if *IVName* is contained in *class*.
- If *prop* is non-NIL, this returns T if *IVName* is contained in *class*, and *prop* is a property of *IVName* in *class* or one of its supers.

Arguments:
class Pointer to a class.
IVName Instance variable name.
prop Property name.

Returns: Value depends on the arguments; see Behavior.

Categories: Object

Specializes: Object

(← *class* **HasIV!** *IVName prop*) [Method of Class]

Purpose/Behavior: Same as **HasIV**, except that **HasIV!** also searches up the supers chain.

Note: The preferred form of this method is **HasAttribute** or **HasAttribute!**.

Arguments: See the method **HasIV**.

Returns: Value depends on the arguments; see Behavior.

Categories: Class

(← *class* **ListAttribute** *type name*) [Method of Class]

Purpose: Lists the elements of a class that are local to the class.

Behavior: *type* is converted to uppercase on entry. The remaining behavior varies according to the arguments.

- If *type* is IVS, this returns the instance variable names (not values) local to *class*. *name* is ignored.
- If *type* is IV, IVPROPS, or NIL, *name* should be bound to an instance variable of *class*. This returns the property names (not values) of the instance variable *name*. If *name* is not an instance variable of *class*, this returns NIL.
- If *type* is CVS, this returns the class variables local to *class*. *name* is ignored.
- If *type* is CV or CVPROPS, *name* should be bound to a class variable of *class*. This returns the property names of the class variable *name*. If *name* is not a class variable of *class*, this returns NIL.
- If *type* is METHODS or SELECTORS, this returns the selectors for the class. *name* is ignored.

Arguments:
class Pointer to a class.
type See Behavior.
name See Behavior.

Returns: Value depends on the arguments; see Behavior.

Categories: Object

Specializes: Object

Example: The following commands show the variety of responses.

```
55←(← ($ SupersBrowser) ListAttribute 'IVs)
(title)

56←(← ($ Window) ListAttribute 'iv 'menus)
(DontSave Title LeftButtonItems MiddleButtonItems TitleItems doc)

57←(← ($ IconWindow) ListAttribute 'METHODS)
(GetMenuItems)
```

(← *class* **ListAttribute!** *type name verboseFlg*) [Method of Class]

Purpose: Lists all items associated with a class.

Behavior: Provides a recursive version of **ListAttribute**.

If *verboseFlg* is NIL, items that are inherited from Tofu, Object, or Class are omitted, unless *class* is one of Tofu, Object, or Class.

type is converted to uppercase on entry.

- If *type* is META or METACLASS, this returns the same as **ListAttribute.**
- If *type* is IVS or NIL, this returns the instance variables an instance of *class* would have.
- If *type* is SUPERS or SUPERCLASSES, this returns the ordered list of super classes of *class*.
- If *type* is SUBS or SUBCLASSES, this returns all of the subclasses of *class*.
- If *type* is any other option that can be passed to **ListAttribute**, this returns all local and inherited values.

Arguments:

class	Pointer to a class.
type	See Behavior.
name	A litatom.
verboseFlg	See Behavior.

Returns: Value depends on the arguments; see Behavior.

Categories: Object

Specializes: Object

(**WhoHas** *name type files editFlg*) [Function]

Purpose: Determines what classes contain a specified item.

Behavior: Returns a list of classes on *files* that contain *name*. If *editFlg* is non-NIL, then edit the methods (if *type* is METHOD), or the classes before returning.

Arguments:

name	The item specified.
type	One of IV, CV, METHOD, or Method. If *type* is NIL, it defaults to METHOD.
files	A file or a list of files. If *files* is NIL, it defaults to **FILELST**.

editFlg T or NIL.

Returns: A list of classes on *files* that contain *name*.

3.8 COPYING CLASSES AND THEIR CONTENTS

3.8 Copying Classes and Their Contents

Inheritance lets classes be described in terms of other classes in a hierarchical manner. When it is preferable to duplicate a class description in different parts of a lattice these methods provide the capability.

The following table shows the methods in this section.

Name	Type	Description
Copy	Method	Copies a class.
CopyCV	Method	Copies a class variable to another class.
CopyIV	Method	Copies an instance variable to another class.

(← *class* **Copy** *name*) [Method of Class]

Purpose: Makes a copy of a class.

Behavior: If *name* is NIL, you are prompted to supply a name for the new class. This copies variables and properties and methods.

Arguments: *class* The class being copied.

name The name of the copy.

Returns: The new class.

Categories: Class

Example: Given that

```
(DefineClass 'Datum)
(← ($ Datum) AddIV 'someThing)
```

the following command makes a copy of class **Datum** and names it **Thing**:

```
(← ($ Datum) Copy 'Thing)
```

(← *class* **CopyCV** *cvName toClass*) [Method of Class]

Purpose/Behavior: Copies a class variable to another class. This also copies the properties of *cvName* to *toClass*.

Arguments: *class* The source class.

cvName The name of the class variable to copy.

toClass The destination class.

Returns: NIL

Categories: Class

(← *class* **CopyIV** *ivName toClass*) [Method of Class]

Purpose/Behavior: Copies an instance variable to another class. This also copies the properties of *ivName* to *toClass*.

Arguments:
- *class* The source class.
- *ivName* The name of the instance variable to copy.
- *toClass* The destination class.

Returns: NIL

Categories: Class

3.9 ENUMERATING INSTANCES OF CLASSES

3.9 Enumerating Instances of Classes

New instances may be created without names, or without being tracked. These methods allow you to produce a list of instances according to their classes. **Prototype** instances are a convenience used where the methods defined for a class must be used, but there is no logical instance for the class.

The following table shows the items in this section.

Name	Type	Description
AllInstances	Method	Finds all instances of a class.
AllInstances!	Method	Finds all instances of a class or its subclasses.
IndexedObject	Class	Keeps track of instances so that **AllInstances** searches can proceed more rapidly.
PrintOn	Method	Modifies how instances of **IndexedObject** that do not have LOOPS names will be printed.
Prototype	Method	Returns an instance of a class that is stored on the class's class variable **Prototype**.

(← *class* **AllInstances**) [Method of Class]

Purpose: Finds all instances of a class.

Behavior: Checks if *class* is a subclass of **IndexedObject**. If so, a faster search is used to find all of the instances of *class*. If not, this checks if each object is an instance of *class*. Instances that do not yet have a UID will not be found.

Arguments:
- *class* A class.

Returns: A list of the instances found.

Categories: Class

Example: The following command produces a list of all the LOOPS window instances:

```
61←(← ($ Window) AllInstances)
```

(← *class* **AllInstances!**) [Method of Class]

Purpose: Finds all instances of a class or its subclasses.

Behavior: Returns a list of instances that are instances of *class* or any of its subclasses. Instances that do not have the class **IndexedObject** as a super class, or that do not yet have a UID are not found. (See Chapter 18, Reading and Printing, for more information on UIDs.)

Arguments: *class* A pointer to a class.

Returns: A list of the instances found.

Categories: Class

IndexedObject [Class]

Purpose: Keeps track of instances so that **AllInstances** searches can proceed more rapidly.

Behavior: This class is to be used as a **Mixin** (an addition superclass), and should be the first class on a supers list for a class.

IndexedObject provides **NewInstance** and **Destroy** protocols that cause instances to be added to or removed from a global list when they are created or destroyed. This global list allows the **AllInstances** protocols to search more quickly.

IndexedObject also provides a **PrintOn** protocol that modifies how instances will be printed if they have no LOOPS name.

MetaClass: Class

Supers: Object

Class Variables: **IdentifierVar**
The name of an instance variable which will contain a string which could provide some identification to the user. Used in **PrintOn** if variable is in object and filled. **shortName**, the value of this class variable, is the default variable name which is used.

(← *self* **PrintOn**) [Method of IndexedObject]

Purpose: Modifies how instances of **IndexedObject** that do not have LOOPS names will be printed.

Behavior: If *self* has a LOOPS name, or if *self* does not have an instance variable with a name equal to (@ *self* ::IdentifierVar), then do a (←**Super**). Otherwise, build a form that incorporates the value of the instance variableIV referenced by (@ *self* ::IdentifierVar).

Arguments: *self* An instance.

Returns: A list ; see example

Categories: Object

Specializes: Object

Example: Create a class, **IndexedObjectTest**, that has this structure.

```
62←(DefineClass 'IndexedObjectTest '(IndexedObject))
#,($C IndexedObjectTest)

63←(← ($ IndexedObjectTest) AddIV 'shortName 'ioTest)
shortName
```

Create an instance.

```
64←(SETQ test (← ($ IndexedObjectTest) New))
#,($& IndexedObjectTest (YMW0.0X%:.>T4.n18 . 36))

65←(←@ test shortName 'changeName)
changeName

66←(← test PrintOn)
("#," $& IndexedObjectTest (changeName (YMW0.0X%:.>T4.n18 . 36)))
```

(← *class* **Prototype** *newProtoFlg*) [Method of Class]

Purpose: Returns a prototype instance of a class.

Behavior: Varies according to the arguments.

- If *class* has a class variable **Prototype** and the variable's value is an instance of *class*, return the value (assuming *newProtoFlg* is NIL).
- If there is no class variable **Prototype**, or if there is a class variable **Prototype** but its value is not an instance of *class*, or if *newProtoFlg* is non-NIL, then create a new instance of *class*, store the instance on the class variable **Prototype**, and return the instance.

See **Proto** in Chapter 7, Message Sending Forms, for more information.

Arguments: *class* A class.

newProtoFlg
If non-NIL, create a new prototype instance.

Returns: The prototype.

Categories: Class

Example: LOOPS defines an icon to make it easy to bring up class browsers and file browsers. The icon is the **Prototype** instance of the class **LoopsIcon**.

To move the icon to the center of the bottom of the screen, enter

```
71←(←Proto ($ LoopsIcon) Move (QUOTIENT SCREENWIDTH 2) 0)
(576 . 0)
```

This places the left edge of the icon at the center of the screen. To move the icon to the center of the screen, enter

```
72←(LET ((icon (← ($ LoopsIcon) Prototype)))
       (← icon Move (QUOTIENT (DIFFERENCE SCREENWIDTH
                                          (@ icon width))
                              2)
                      0))
(544 . 0)
```


3.10 DEALING WITH INHERITANCE

3.10 Dealing with Inheritance

The inheritance lattice for classes shows how methods and variables are shared (see Chapter 10, Browsers, for details on how to graph the lattice on the screen). To programmatically inspect and add to this lattice via **Specialize**, use the following functions and methods:

Name	Type	Description
Fringe	Method	Finds the leaves of a branch of an inheritance tree.
Specialize	Method	Creates a subclass of a class.
SubClasses	Method	Returns a list of subclasses.
Subclass	Method	Determines if a class is a subclass of another class.
AllSubClasses	Function	Computes the subclasses of a class.
SubsTree	Function	Computes all the names of the subclasses of a class.

(← *class* **Fringe**) [Method of Class]

Purpose: Finds the leaves of a branch of an inheritance tree.

Behavior: Returns a list of subclasses of *class*, whether close or distant, that have no subclasses.

Arguments:
class A class, the root of the tree to explore.

Returns: Names of subclasses of *class* that have no subclasses.

Categories: Class

Example: The following commands show the variety of responses.

```
73←(← ($ Window) Fringe)
(InstanceBrowser MetaBrowser SupersBrowser FileBrowser
LoopsIcon IconWindow)

74←(← ($ ClassBrowser) Fringe)
(MetaBrowser SupersBrowser FileBrowser)
```

(← *class* **Specialize** *newName*) [Method of Class]

Purpose: Creates a subclass of a class.

Behavior: Creates a class with class as its only super.

- If *newName* is non-NIL, this is the name of the new class.
- If *newName* is NIL, this creates a name consisting of the name of *class* followed by an integer.

Arguments:
class Pointer to a class.

newName Name of the new subclass.

Returns: The new class.

Categories: Class

Example: Given that

```
(DefineClass 'Datum)
```

the following command creates a specialization of the class **Datum** called **DatumX**:

```
(← ($ Datum) Specialize 'DatumX)
```

(← *class* **SubClasses**) [Method of Class]

Purpose: Returns a list of subclasses.

Behavior: The classes returned by this are the immediate subclasses of *class*.

Arguments: *class* A pointer to a class.

Returns: A list of subclasses.

Categories: Class

Specializations: DestroyedClass

Example: The following command gets a list of the subclasses of the class **Window**:

```
(← ($ Window) SubClasses)
```

(← *class* **Subclass** *super*) [Method of Class]

Purpose: Determines if a class is a subclass of another class.

Behavior: If *class* is a subclass of *super*, *super* is returned, else NIL.

Arguments: *class* Pointer to a class.

super Either the LOOPS name of a class or a pointer to a class.

Returns: Value depends on the arguments; see Behavior.

Categories: Class

Example: The command

```
(← ($ DestroyedClass) Subclass 'Class)
```

returns

```
#,($C Class)
```

(**AllSubClasses** *class currentSubs*) [Function]

Purpose: Computes the subclasses of a class.

Behavior: This is a recursive function that computes (without duplicates) all of the subclasses of *class*.

Arguments: *class* Must be a pointer to a class, for example, ($ Window).

currentSubs Used by LOOPS; NIL when called by the user.

Returns: A list of classes.

Example: The command

```
(AllSubClasses ($ LatticeBrowser))
```

returns

```
(#,($C FileBrowser) #,($C SupersBrowser)
#,($C MetaBrowser) #,($C ClassBrowser)
#,($C InstanceBrowser))
```

(**SubsTree** *class currentList*) [Function]

Purpose: Computes the names of the subclasses of a class.

Behavior: Provides a recursive function that computes (without duplicates) all of the names of the subclasses of *class*.

Arguments:

class Can be a class name or a pointer to a class

currentList Used internally by **SubsTree**; it should be NIL when called by the user.

Returns: A list of class names.

Example: The command

```
(SubsTree 'LatticeBrowser)
```

returns

```
(InstanceBrowser ClassBrowser MetaBrowser SupersBrowser
FileBrowser)
```

[This page intentionally left blank]