4. METACLASSES

In object-oriented programming, every object is described by a class. Instances are described by classes and classes are, in turn, described by metaclasses. The methods that an instance inherits are defined in the class definition of that instance and the methods that the class inherits are defined in the metaclass definition of that class's metaclass. Sending a message to an instance invokes method in class. Similarly, sending a message to class invokes method in metaclass.

The two classes **Class** and **MetaClass** are metaclasses of other classes. If **Class** or **MetaClass** refers to the metaclass, it appears in a bold typeface.

One method defined by a class's metaclass is **New**, which returns a new instance of a class. Different classes can initialize their instances in different ways. For example, one class may need to have certain values assigned to instance variables at creation, while another does not. The different forms of **New** are defined in separate metaclasses.

A class's metaclass is assigned when the class is created. A new class is created by sending a metaclass the message **New** or by specializing an already existing class. In the latter case, the metaclass defaults to the metaclass of the class's super class. The class's metaclass can be changed by directly editing the class definition.

This chapter discusses the metaclasses provided with LOOPS, describes pseudoclasses, explains how to define new metaclasses, and discusses the root class **Tofu** .

4.1 Specific Metaclasses

This section describes the metaclasses provided by LOOPS: **Class**, **MetaClass, AbstractClass**, and **DestroyedClass**. These metaclasses are shown in Figure 4-1.

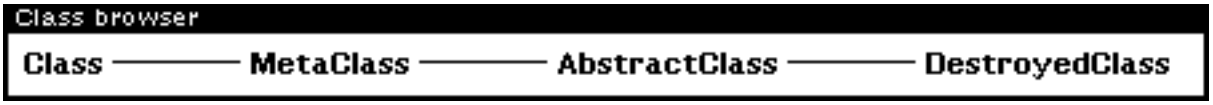

Figure 4-1. Class Browser Showing Metaclasses

4.1.1 Metaclass Class

Class is the default metaclass for LOOPS classes. When a class whose metaclass is **Class** receives the message **New**, it creates a new instance of itself and returns that instance. If this message is sent at the top level, the definition of the created instance is printed in the Executive window.

4.1.2 Metaclass MetaClass

MetaClass is the metaclass for all metaclasses and provides the message **New** to all metaclasses. For metaclasses, the result of sending the message **New** is the definition of a new metaclass. This is discussed in detail in Section 4.3, "Defining New Metaclasses."

4.1.3 Metaclass AbstractClass

If a class's metaclass is **AbstractClass**, then it cannot be instantiated. If an abstract class is sent the message **New**, the following message is printed to the TTY window.

```
#,($C className) Abstract Class cannot be instantiated
```

To make a class an **AbstractClass**, either send the metaclass **AbstractClass** the message **New** or change the metaclass of the class definition directly using the editor.

Use an abstract class to define a class which should not have any instances. For example, consider mixin classes. Mixins are always used in conjunction with another class to create a subclass. Instances are created from the new subclass that has the mixin as one of its parents. Because mixins never have instances, they have **AbstractClass** as their metaclass.

As an example, consider a circuit simulation module that contains various classes such as **Resistors, Inductors, Batteries,** and **Wires**. A possibility is to define a super class for these classes called **AnalogDevice** to contain all the information common to all such classes: current, impedance, voltage drop, etc. This super class also holds all the methods common to the classes, such as **ApplyOhmsLaw**. Since **AnalogDevice** is not itself intended to be instantiated (only its subclasses are), its metaclass can be **AbstractClass** so that an error occurs if it is accidentally instantiated.

Note: Whenever **AnalogDevice** is specialized to create a new subclass, be sure to change its metaclass.

4.1.4 Metaclass DestroyedClass

DestroyedClass is the metaclass for classes that have been sent the message **Destroy** or **Destroy!** Trying to instantiate a **DestroyedClass** causes an error. Attempts to destroy a **DestroyedClass** have no effect.

4.2 PSEUDOCLASSES

4.2 Pseudoclasses

Pseudoclasses provide an object interface to Lisp data types, which are also known as Lisp objects. Pseudoclasses associate a class with the type name of a Lisp object. When messages are sent to Lisp objects of the named type, the messages are actually sent to the pseudoclass. Lisp objects which have pseudoclasses are considered pseudoinstances.

Pseudoclasses provide two special cases in the message-sending mechanism: for lists whose first element is a class, or for ordinary Lisp data types.

In the first case, the list's first element is used as the class to look up the method to be used.

In the second case, the class of the data type is found by using the **GetLispClass** function, which looks in an internal table based on the type name of the data type. If none is found, it is assumed to be **Tofu**. If found, the data type is considered a pseudoclass and instances of it pseudoinstances.

Pseudoclasses also provide special cases in the behavior of **GetValue** and **PutValue**, to allow simulation of variable or property access, as described below.

(**GetValue** *pseudoinstance varName propName*) [Function]

Purpose: A variation on the behavior of GetValue to simulate retrieving variable or property values on pseudoinstances.

Behavior: If **GetValue** is called with *self* bound to a pseudoinstance, then the method associated with the selector **GetValue** in the pseudoclass is called with the arguments:

pseudoinstance varName propName

Arguments:

pseudoinstance A Lisp object which has a pseudoclass.

varName The simulated variable name.

propName The simulated property name, or NIL.

Returns: The result of the call to the **GetValue** method in the pseudoclass.

(**PutValue** *pseudoinstance varName propName newValue*) [Function]

Purpose: A variation on the behavior of **PutValue** to simulate setting of variable or property values on pseudoinstances.

Behavior: If **PutValue** is called with *self* bound to a pseudoinstance, then the method associated with the selector **PutValue** in the pseudoclass is called with the arguments:

instance varName newValue propName

Arguments:

pseudoinstance A Lisp object which has a pseudoclass.

varName The simulated variable name.

propName The simulated property name, or NIL.

newValue The new value to be placed in the simulated slot.

Returns: The result of the call to the **PutValue** method in the pseudoclass.

(**GetLispClass** *obj*) [Function]

Purpose: Used by the system to compute a class corresponding to a Lisp data type.

Behavior: Gets the hash value for the key (**TYPENAME** *obj*) from an internal hash array.

- If this hash value is NIL, ($ Tofu) is returned.
- If the hash value is not NIL and it is a class, it is returned.

- In all other cases, the hash value, which should be a function, is applied to *obj* and the result is returned.

Arguments: *obj* A Lisp object.

Returns: Value depends on the hash value; see Behavior.

Example: The command

```
79←(GetLispClass (create annotatedValue))
```

returns

```
#,($C AnnotatedValue)
```

LispClassTable [Global Variable]

Purpose: Used by **GetLispClass** to map type names of Lisp objects to pseudoclasses.

Format: This hash table has EQ hashing. It contains pairs of symbol keys (a type name) and either classes, NIL, or a function object to be applied (see **GetLispClass**).

4.2.1 Example

This example creates a pseudoclass from the Lisp data type STRINGP.

1. Define a class **String** that receives its messages:

```
37←(DefineClass 'String)
#,($C String)
```

2. Place an entry in the LispClass hash table to link the Lisp data type STRING to the **String** class.

```
38←(PUTHASH 'STRINGP ($ String) LispClassTable)
#,($C String)
```

3. Add methods to **String** which will operate on Lisp STRINGPs, for example:

```
39←(DefineMethod ($ String) 'UpCase '(self)
    '(U-CASE self))
String.Upcase
```

This allows messages like the following:

```
40←(← "abc" UpCase)
"ABC"
```

4. Specialize **GetValue** and **PutValue** to allow element access in strings, for example:

```
41←(DefineMethod ($ String) 'GetValue '(index)
    '(NTHCHAR self index))
String.GetValue

42←(DefineMethod ($ String) 'PutValue '(index value)
    '(RPLSTRING self index value))
String.PutValue
```

This allows messages to access characters in strings, for example:

```
43←(← "abc" GetValue 2)
b

44←(← "abc" PutValue 2 'p)
"apc"
```


4.3 DEFINING NEW METACLASSES

4.3 Defining New Metaclasses

A new metaclass must be defined if you want to create several classes for which a class message, such as **Destroy**, needs to be specialized. To create a new metaclass, an object of the class **MetaClass** must be instantiated. This is done by sending **MetaClass** the message **New**.

(← (**$ MetaClass**) **New** *metaClassName supers*) [Method of Metaclass]

Purpose: Instantiates a new metaclass with **MetaClass** as its metaclass, *metaClassName* as its name, and *supers* as a list of its super classes.

Behavior: Evaluates *metaClassName*, which must evaluate to a symbol. The default for *supers* is **(Class)**. If used, *supers* must evaluate to a list of classes. The message returns the new metaclass.

Arguments: *metaClassName*
Name of the new metaclass; must evaluate to a symbol.

supers List of classes.

Categories: **MetaClass**

Specializes: Class.New

Specializations: AbstractClass.New

Example: Assume the following **MetaClass** definition:

```
42←(←($ MetaClass) New 'ListMetaClass '(Class))
#,($C ListMetaClass)
```

The message **New** can then be defined for the metaclass, **ListMetaClass.** In this example, it saves all the instances created of a class with the metaclass **ListMetaClass**. The instances are stored as the value of the class property **AllInstances**. Define the message **New** using **DefineMethod** as follows:

```
DefineMethod ($ ListMetaClass) 'New '(name arg1 arg2 arg3
arg4 arg5)
    '((* * Create an instance and add it to
          list in class)
    (LET ((newObj (←Super)))

            (* * newObj is the instance returned by
                sending the New message provided by
                the class CLASS.)

           (PutClass
             self
             (CONS newObj (LISTP (GetClassHere
```

```
                                    self 'AllInstances)))
                    'AllInstances)

                    (* * LISTP returns list or
                        NIL if not a list)
                    (* * GetClassHere returns the value
                        of a property of a class.)

                newObj]
```

Now a new class can be defined by sending #,($C ListMetaClass) the message **New**. The result is a new class with **ListMetaClass** as the metaclass.

```
43←(←($ ListMetaClass) New 'Book)
#,($C Book)     [New class called Book]

44←(←($ Book) New 'Book1)
#,($C Book1)

45←(←($ Book) New 'Book2)
#,($C Book2)

46←(GetClass ($ Book) 'AllInstances)
#,($C Book2) #,($C Book1)  [List of all instances created so far]
```


4.4 Tofu

The highest class in the LOOPS hierarchy is **Tofu**, which is an acronym for Top of the Universe. It is the simplest class, having no instance variables and only three defined messages:

- **MessageNotUnderstood**
- **MethodNotFound**
- **SuperMethodNotFound**

Figure 4-2 shows specializations of **Tofu**. The most familiar specialization of **Tofu** is the class **Object**, which is the root of the most of the other classes. Another specialization of **Tofu** is **AnnotatedValue**. **AnnotatedValue** is used with active values (see Chapter 8, Active Values).

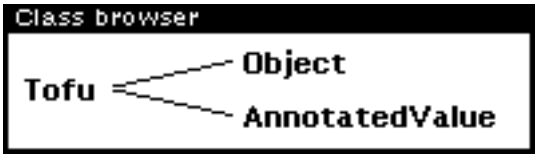

Figure 4-2. Specializations of Tofu

If another evaluation protocol or scheme for catching error conditions is needed, specialize **Tofu** and define all the methods required for handling data, usually some subset of the methods of **Object**. Specializing Tofu should only be necessary on very rare occasions.

The following table shows the methods in this section.

Functionality	Type	Description

MessageNotUnderstood	Method	Provides the error handling mechanism for when a message is sent to an object that cannot respond to that message.
MethodNotFound	Method	Provides some intermediate checking before sending the message MessageNotUnderstood.
SuperMethodNotFound	Method	Provides some intermediate checking before sending the message **MessageNotUnderstood**.

(← *self* **MessageNotUnderstood** *selector messageArguments superFlg*) [Method of Tofu]

Purpose/Behavior: Provides the error handling mechanism for when a message is sent to an object that cannot respond to that message.

Calls **ERROR** with a list which includes *self*, *selector*, and "not understood."

Arguments:
- *self* — An object receiving a message with the selector *selector*.
- *selector* — A selector.
- *messageArguments* — A list of the arguments to the message.
- *superFlg* — Used internally.

Returns: See Behavior.

Categories: Tofu

Specializations: Object

(← *self* **MethodNotFound** *selector*) [Method of Tofu]

Purpose/Behavior: Provides some intermediate checking before sending the message **MessageNotUnderstood**.

Arguments:
- *self* — An object receiving a message with the selector *selector*.
- *selector* — A selector.

Returns: Used for side effect only.

Categories: Tofu

(← *self* **SuperMethodNotFound** *selector classOfSendingMethod*) [Method of Tofu]

Purpose/Behavior: Provides some intermediate checking before sending the message **MessageNotUnderstood**.

Arguments:
- *self* — An object receiving a message with the selector *selector*.
- *selector* — A selector.
- *classOfSendingMethod* — The class with the method that contains a ←**Super.**

Returns: Used for side effect only.

Categories: Tofu

[This page intentionally left blank]