The creation of a simplified subset if the Smalltalk-80 language

Edgardo Ortiz
Lehigh University

Follow this and additional works at: https://preserve.lehigh.edu/etd

Part of the Computer Sciences Commons, and the Electrical and Computer Engineering Commons

Recommended Citation
Ortiz, Edgardo, "The creation of a simplified subset if the Smalltalk-80 language" (1993). Theses and Dissertations. 233.
https://preserve.lehigh.edu/etd/233

This Thesis is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.
AUTHOR: Oritz, Edgardo

TITLE: The Creation of a Simplified Subset of the SmallTalk-80 Language

DATE: January 16, 1994
The creation of a simplified subset of the Smalltalk-80 language

by

Edgardo Ortiz

A Thesis
Presented to the Graduate and Research Committee of Lehigh University in Candidacy for the Degree of Master of Science

in
Computer Science

Lehigh University
December 1993
This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

12/9/93

Date

Thesis/Advisor

Chairperson of Department
Table of Contents

List of figures ................................ iv
Abstract ............................................. 1
I. Introduction ...................................... 3
II. The virtual image ................................. 10
   A. Reducing the virtual image ................. 10
   B. Class Index .................................. 22
III. Construction of new classes ................. 119
IV. A new user interface ......................... 121
   A. Interactive Editor design preconditions .. 121
   B. Beginning a LehighTalk run session ........ 122
V. Conclusion ........................................ 131
Bibliography ....................................... 134
Appendices .......................................... 135
   Appendix 1 Smalltalk Class Description .... 135
   Appendix 2 Primitives ......................... 136
Biography .......................................... 138
List of figures

Figure 1. Deleted classes ................................ 11
Figure 2. Amount of methods in each class before and after the reduction. ............... 21
Abstract

Smalltalk-80 is the programming language that best represents the Object-Oriented-Paradigm for programming. Smalltalk-80 pioneered the idea of integrated programming environments. This is a system in which the editor, compiler, and user interfaces -- windows, menus, and the mouse -- are integral parts of the system. The Smalltalk-80 system is composed of a large class library, and the interactive programming interface.

This thesis provides the foundations for the creation of a simplified subset of the Smalltalk-80 Language. This subset will be composed of a reduced class library and a considerably smaller and simplified user interface. The reduced system supports the fundamental concepts of Object-Oriented-Programming such as: Objects, Messages, Classes, Subclassing, and Inheritance. As a teaching tool, we want to provide future users with basic Classes, methods, and primitives needed to expand the system.

For the creation of the subset of Smalltalk-80, we reduce the number of basic classes, methods, and primitives in the Smalltalk-80 system. The reduction of the system is summarized as follow:

1-Originally, the number of basic classes in the Smalltalk-80 system were 78. They were reduced to 37.
2-Originally, in the group of 35 preselected classes, there were around 1089 methods. They were reduced to 679.

3-The systems primitives were reduced from 127 to 41.

We replace the programming environment in Smalltalk-80 with a new user interface. This new user interface added two new classes to our system. They are class File and class ClassEditor.

In general, we conclude that we achieved a significant part of our primary goal in the construction of a new system. We now that the system is quite large and incomplete, but we hope that soon we will see the first evaluation of 3+4 in our own system.
I. Introduction

Smalltalk-80 (Goldberg & Robson, 1983), a big and complex programming environment, is the result of more than ten years of research at Xerox Palo Alto Research Center. The creation of Smalltalk-80 also involved many other hardware manufacturers: Apple Computer, Digital Equipment Corporation, Hewlett-Packard, and Tektronix. Smalltalk-80 was preceded by other versions: Smalltalk-72, -74, -76, and -78, which culminated in Smalltalk-80.

The brief description of the evolution of the Smalltalk-80 that follows is drawn from Ingalls (1983, p. 10-22):

The very first Smalltalk evaluator was a thousand-line BASIC program which first evaluated 3+4 in October 1972. It was followed in two months by a Nova assembly code implementation which became known as the Smalltalk-72 system.

In 1974 a major redesign of the Smalltalk interpreter was produced with the aim of cleaning up its semantics and improving its performance. While the redesign was a mixed success, Smalltalk-74 was the site of several advances which fed into the later systems.

In 1976 another major redesign was performed to the Smalltalk language and implementation. It addressed most of the problems encountered in the previous four years of experience with Smalltalk:

- Classes and contexts became real objects;
- A class hierarchy provided inheritance;
- A simple yet flexible syntax for messages was introduced;
- The syntax eliminated message stream side-effects and could be compiled;
- A compact and efficient byte-encoded instruction set was introduced;
- A microcode emulator for this instruction set ran 4 to 100 times faster than previous Smalltalks;
OOZE provided storage for 65K objects—roughly the capacity of the Alto hardware.

In 1977 the creators of smalltalk began a project to build a portable computer capable of running the Smalltalk system. Known internally as NoteTaker, it began as a hand-held device for taking notes, but ended up as a suitcase-sized Smalltalk machine. The design challenge was significant. They were moving to an environment with less processing power, and the whole system had to fit in 1/4 Mbyte, since there was no swapping medium. Also they faced transporting 32K bytes of machine code which made up the Smalltalk-76 system. The result of these forces was the design of Smalltalk-78.

With Smalltalk-78 already in place, few changes were made to the Virtual Machine to produce Smalltalk-80. The main change was an increase in power from allowing blocks with arguments."

Smalltalk is an object Oriented Programming Environment that introduced many new terms that may be unfamiliar to traditional programmers. The following definitions are quoted from various references.

**Object.** A value of a class; also called an instance. In Smalltalk, everything is an object. (Kamin, 1990, p. 342).

**class.** The basic programming unit in Smalltalk. A class encapsulates the representation of a certain kind of object and defines the operations on objects of that kind. (Kamin, 1990, p. 342).

**instance.** One of the objects described by a class. (Goldberg & Robson, 1983, p. 16).

**subclass.** A class which is placed underneath another class in the inheritance hierarchy. (Kamin, 1990, p. 343).

**superclass.** A class which is placed above another class in the inheritance hierarchy. (Kamin, 1990, p. 343).

**instance variable.** A variable of which each object of a class has its own version. The instance variables of a class define
the representation of objects of that class. (Kamin, 1990, p. 342).

**class variable.** A variable that has one occurrence in a class (regardless of the number of instances of that class). Subclasses can refer to a class's class variables. (Kamin, 1990, p. 342)

**abstract class.** A class which has no instances. Abstract classes are defined in order to supply method definitions that can be inherited by subclasses. (Kamin, 1990, p. 341).

**method.** The Smalltalk term for a function defined within a class (and all functions in Smalltalk are defined in classes). Methods always have at least one argument, called the receiver of the message. In Smalltalk message-sending syntax, the message (i.e., method name) follows the expression defining the receiver. (Kamin, 1990, p. 342).

**message.** A request for an object to carry out one of its operations. (Goldberg & Robson, 1983, p. 16).

**overriding a method.** Specifying a method in a subclass for the same message as a method in a superclass. (Goldberg & Robson, 1983, p. 73).

**receiver.** The object to which a message is sent." (Goldberg & Robson, 1983, p. 37).

**message selector.** The name of the type of operation a message requests of its receiver. (Goldberg & Robson, 1983, p. 37).

**message argument.** An object that specifies additional information for an operation. (Goldberg & Robson, 1993, p. 37).

**unary message.** A message without arguments." (Goldberg & Robson, 1993, p. 37).

**message pattern.** A message selector and a set of argument names, one for each argument that a message with this selector must have. (Goldberg & Robson, 1983, p. 53).

**super.** A special variable that can be used as the receiver of a message in an expression, with the following meaning: The object that receives the message is actually self, but the method search begins in the superclass of the class in which the expression occurs. (Kamin, 1990, p. 343).
overloading. The defining of a single message name in more than one class, or, more generally, the use of a name in different ways. In this more general sense, many languages allow some form of overloading; for example, Pascal overloads the name "+" using it for both integer and floating-point addition. In Smalltalk, overloading is particularly important, as it permits the definition of polymorphic functions. (Kamin, 1990, p. 342).

protocol. The set of messages to which objects of a given class can respond. (Kamin, 1990, p. 343).

block. A Smalltalk object that represents a function. These values are "first class" in the sense that they can be passed as arguments to methods and returned as results, and can be assigned to variables. Blocks can have any number of arguments. To evaluate a zero-argument block, send it the value ,message; to evaluate a one-argument block, send it the value: message; to evaluate a two-argument block, send it the value:value: message; and so on. Blocks are used to implement control structures in Smalltalk. (Kamin, 1990, p. 341).

inheritance. The feature of Smalltalk whereby a class, being declared as a subclass of another, defines objects that have all the instance variables, and respond to all the messages, of that other class. New components can be added to the representation by the subclass, and new methods can be defined and inherited ones redefined. This is also known as single inheritance. (Kamin, 1990, p. 342).

method search. Inheritance is implemented by having the interpreter search for the method corresponding to a given message. This search begins in the class of the receiver of the message (with the exception of messages sent to super) and continues up the hierarchy until a definition for the message is found. (Kamin, 1990, p. 342).

Primitive. An operation that cannot be performed in the programming language, and thus must be accomplished with the aid of the underlying run-time system. (Budd, 1991, p. 377).

polymorphism. The use of a single function for different types of data. Since in Smalltalk the only notion of "type" is "protocol to which an object responds," any piece of code can be applied to any object that responds to the set of messages sent to it in that code; and since a message can be defined in more than a class, a piece of code can be applied to objects from different classes. (Kamin, 1991, p. 343).
self. A special variable that can be used in method definitions to refer to the receiver of the message. (Kamin, 1990, p. 343).

Smalltalk is not just a language, it is a programming environment. It is composed of a large class library, and the interactive programming environment. Developing programs in Smalltalk entails browsing and extending the class hierarchy.

Learning how to use Smalltalk-80 takes more time than other languages such as C, or Pascal. Mainly because:

1. Smalltalk is a large and complex system. LaLonde & Pugh (1990, p. xi) state:

"Developing programs in Smalltalk requires familiarity with the large class library, and the interactive programming environment. Programming in Smalltalk requires familiarity with all these components and the learning curve for programmers is therefore longer than for more traditional languages."

2. Smalltalk is a new way of viewing and thinking about computation. LaLonde & Pugh (1990, p. 3) state:

"Object-oriented programming is fundamentally different from traditional procedural or algorithmic approaches. Object-oriented programming describes a system in terms of the objects involved. Traditional programming approaches, on the other hand, describe systems in terms of their functionality."

We want to facilitate the process of learning how to program in Smalltalk by simplifying the interactive programming environment and the class library. In doing that, students can focus their attention on learning the Object-
oriented paradigm which is the most important step in the construction of high quality reusable code.

Our primary goal in this paper is to provide the foundations for the creation of a simplified subset of the Smalltalk-80 language. This subset will incorporate many of the object-oriented features of Smalltalk-80 such as Objects, Messages, Classes, Subclassing, and Inheritance.

Because the notion of a primitive is so important in this paper we include the following explanation quoted from "Tektronix Smalltalk-80 version t2.1.3b method whatIsAPrimitive."

"Some messages in the smalltalk system respond to primitives. A primitive response is performed directly by the interpreter rather than by evaluating expressions in a method. The methods for these messages indicate the presence of a primitive response by including <primitive:xx> before the first expression in the method.

Primitives exist for several reasons. Certain basic or 'primitive' operations cannot be performed in any other way. Smalltalk without primitives can move values from one variable to another, but cannot add two SmallIntegers together. Many methods for arithmetic and comparison between numbers are primitives. Some primitives exist only to make the system run faster; each does the same thing as a certain Smalltalk method, and its implementation as a primitive is optional.

When the Smalltalk interpreter begins to execute a method which specifies a primitive response, it tries to perform the primitive action and to return a result. If the routine in the interpreter for this primitive is successful, it will return a value and the expressions in the method will not be evaluated. If the primitive routine is not successful, the primitive 'fails', and the Smalltalk expressions in the method
are executed instead. These expressions are evaluated as though the primitive routine had not been called."

The second goal in this paper is to construct this subset with as few primitives, classes, and methods as possible. Our main concern is to minimize the size of the language without losing the concepts and spirit of Object Oriented Programming. Other goals are:

Support a language that is as close as possible to Smalltalk-80.

Learn about the Smalltalk-80 system, in particular the primitives and their relation to the classes.

Two major components of the Smalltalk-80 system can be distinguished: the virtual image and the virtual machine.

1. The virtual image consists of all the objects in the system.

2. The virtual machine consists of the hardware devices and machine language (or microcode) routines that give life to the objects in the virtual image.

In this paper we mostly emphasize reducing the virtual image. The following sections describes the efforts made in order to achieve our goals.
II. The virtual image
The virtual image consist of all the objects in the system.

A. Reducing the virtual image

The Smalltalk-80 system is composed of a set of classes. These classes provide functionality to the programming language. (Goldberg & Robson, 1983 p. 14) Figure 1. is a diagram of the Smalltalk-80 system classes. Figure 1. also shows the eliminated classes (see below). Parenthesized numbers after the class name indicate the reason for elimination. Underlined classes are included in the new system. Indentation indicates subclassing.
Figure 1. Deleted classes.

Parenthesized numbers indicate the reason for elimination. Please read next section. Underlined Classes are part of the proposed system.

<table>
<thead>
<tr>
<th>Object</th>
<th>File</th>
<th>Magnitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magnitude</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Character</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Date (5)</td>
<td>Random (7)</td>
<td></td>
</tr>
<tr>
<td>Time (5)</td>
<td>File (4)</td>
<td></td>
</tr>
<tr>
<td>FileDirectory (4)</td>
<td>FilePage (4)</td>
<td></td>
</tr>
<tr>
<td>Number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Float (3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fraction (3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LargeNegativeInteger (3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LargePositiveInteger (3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallInteger</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LookupKey</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Association</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Link (2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process (2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Collection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SequenceableCollection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LinkedList (2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Semaphore (2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ArrayedCollection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitmap (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DisplayBitmap (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RunArray (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>String</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Symbol</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Text (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ByteArray (5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interval</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OrderedCollection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SortedCollection (6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bag (6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MappedCollection (6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set (6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dictionary</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IdentityDictionary (6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PositionableStream</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ReadStream</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WriteStream</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ReadWriteStream (4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ExternalStream (4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UndefinedObject</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boolean</td>
<td></td>
<td></td>
</tr>
<tr>
<td>True (2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>False (2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ProcessorScheduler (2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delay (2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SharedQueue (2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Behavior</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ClassDescription</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MetaClass</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rectangle (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BitBlt (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CharacterScanner (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pen (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DisplayObject (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DisplayMedium (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Form (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cursor (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DisplayScreen (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>InfiniteForm (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpaqueForm (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Path (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arc (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Circle (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Curve (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LinearFit (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spline (1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We reduce the virtual image in the following ways:
1- First we didn't want to incorporate into the new system the classes that form the Programming Interface. Thus all classes and methods associated with the Programming Interface were removed from the system. This includes classes for Views, Browsers, Testing, and Error reporting. Graphics support is not included, consequently all the classes related to graphics were eliminated.
The classes are identified with the number (1) in figure 1.

2- Multiple processes are not allowed. The classes so eliminated are identified by the number (2) in figure 1.

3- Only smallintegers are supported by the system. This eliminates other subclasses of number -- Float, Fraction, LargeNegativeInteger, LargePositiveInteger -- The classes so eliminated are identified by the number (3) in figure 1.

4-External file manipulation used in Smalltalk is replaced by a new class called File. The classes so eliminated are identified by the number (4) in figure 1.

5- Date and time are not supported. The classes so eliminated are identified by the number (5) in figure 1.
Classes that can be simulated or constructed from other classes. These classes have been eliminated in the interest of keeping the size of the system as small as possible. The classes so eliminated are identified by the number (6) in figure 1.

The first class of this last type (6) is SortedCollection. SortedCollection is a subclass of OrderedCollection in the Smalltalk system. LaLonde & Pugh (1990, p. 342) describe OrderedCollection as follows:

"OrderedCollection, the most general class, is capable of containing arbitrary objects and has operations for adding and removing elements anywhere in the sequence. It is a generalization of Array that permits automatic (transparent) growing and shrinking when additions and removals are performed."

Some operations include:

- `anOrderedCollection add: anElement` (adding an element)
- `remove: oldObject ifAbsent: absentBlock`

OrderedCollection is part of our proposed system. Class OrderedCollection provides the protocol needed to implement a specialized subclass like SortedCollection. We can use the methods inherited from class OrderedCollections for adding and removing elements, together with a sorting algorithm created by the programmer. In addition, the programmer can disallow all the methods provided for arbitrary insertion (for example, `addFirst:`, `addLast:`) into the collection and leave only the method `add:`. When the user adds or removes an object from the
collection, the complete collection will be sorted automatically.

The second class of type (6) is MappedCollection. Goldberg & Robson (1983, p. 168) describe MappedCollection as follows:

"Class MappedCollection is a subclass of Collection. It represents an access mechanism for referencing a subcollection of a collection whose elements are named. This mapping can determine a reordering or filtering of the elements of the collection. The basic idea is that a MappedCollection refers to a domain and a map. The domain is a Collection that is to be accessed indirectly through the external keys stored in the map. The map is a Collection that associates a set of external keys with another set of external keys. This second set of keys must be external keys that can be used to access the elements of the domain. The domain and the map, therefore, must be instances of Dictionary or of a subclass of SequenceableCollection."

In Smalltalk-80 a Collection is used for the domain and a Dictionary for the map in class MappedCollection. Since Collections and Dictionaries are supported in our system, we can create a class MappedCollection from them.

The third class of type (6) is IdentityDictionary. IdentityDictionary is a subclass of Dictionary. LaLonde & Pugh (1990, p. 304) describe class IdentityDictionary and Dictionary as follows:

"IdentityDictionaries and Dictionaries are array-like containers for objects that are associated with arbitrary keys. Only one object is associated with each key. Two different rules are used for deciding when two keys match. For identity dictionaries, two keys match if they are identical; for dictionaries, they match if they are equal. IdentityDictionary uses == for key comparisons whereas Dictionary uses =. We refer to == and = for the respective dictionaries as the key matching operations."

14
Since the main difference between these two classes is the key matching operation (as explained above) we can construct class IdentityDictionary from class Dictionary which is included in our system. 

7-Random numbers are not supported by the system. The class so eliminated is Random identified with number 7 in figure 1.
In the previous section we described how we reduced the number of classes in the system. In this section we describe how we reduced the methods inside the classes that are part of our proposed system. These classes are the ones that appear in boldface in figure 1.

For the elimination and inclusion of methods in our system we use the following strategy. The Smalltalk-80 system is composed of thousands of methods. We took all the methods included in the preselected classes (classes that appear in boldface in figure 1). We started by using the following criterion for exclusion or inclusion:

Step 1: If a subclass contains a method whose message pattern has the same selector as a method in the superclass (overridden method), this method is not a primary candidate for elimination. In this part of the elimination process we don't want to discard the method even if the method is not used for the preselected classes. In conclusion we include the method as a temporary candidate in our system.

Step 2: If the method exists only once in the system. This means the method is described in only one class. This method is primary candidate for elimination. All methods with this characteristic are groped together in a list.
Step 3: Using the list of methods obtained in step 2 we started a search for each call of the given method in the code of all the preselected classes. We accumulated the frequency of calls of the method in the code and with this information we construct another list. This list was composed of the name of the method and the frequency of its call in the code. For example if the method `at: put:` is called 5 times in the code, it will look in the list as follows:

```
5  at: put:
```

This list gives us a good idea about the use of the method in the code. If the method has been used many times that means that it has lower probability of deletion. If the method is used few times then it has higher probability of deletion.

Step 4: If the occurrence of the method obtained in step 3 is 0 or a small number like 1, 2, or 3, then this method has a very high probability for elimination, but independent of the number of occurrences, all the methods in the list are considered for elimination. There are various conditions for the elimination of methods inside the classes. The conditions are:

a- If the method is used for the Programming Interface or if it is used only for graphics support, delete the method.
b- If the method can be constructed from other methods inside the class and it is not extensively used in other classes, delete the method.

c- If the Method is used for mathematical functions or some arithmetic operations different from +, -, delete the method.

d- If the method only supports deleted classes (classes that are not part of our system) excluding methods related to the compiler, delete the method.

e- If the method appears inside the class but it is not used by other methods inside the same class or in any other class in the system, delete the method.

Step 5: We started looking at the overridden methods obtained in step 1. For the elimination of the methods in this category we use the same conditions used in step 4.

As we mentioned before, the classes in Smalltalk-80 that bring support to File manipulation and the programming interface were eliminated. This established the need for simplified classes to replace these classes.

The first class is File. Class File is a subclass of class Object created to support the manipulation of files in our system. The class include methods for opening disk files,
reading disk files, writing disk files, and for checking the size of the file.

The other class is ClassEditor. ClassEditor is a subclass of class Object. ClassEditor was created for support editing of classes as part of our user interface. The methods included in the class support the following areas: Editing of classes, and reading and writing Class descriptions on disk files.

The complete description for class File and class ClassEditor is provided in the section class index.

We would like to show all the methods that were eliminated but that would take a huge amount of space. Instead we describe what we found more useful and pertinent for the foundations of our new subset of Smalltalk-80. A complete list of classes and methods for our system is described in the following pages. The description for the classes and methods are quoted from McCullough P. L. (1986). In addition we show information related to the amount of methods included in the classes before and after the reduction process. This information is summarized in figure 2. Figure 2 contain four columns. The description for these four columns follows:

1- The first column includes the classes names. These are the names associated with each class in our system.
2- Second column shows the amount of methods in each class before the elimination process. This is the amount of methods contained in each class in the Smalltalk-80 system.

3- Third column shows the amount of methods in each class after the elimination process. This is the amount of methods contained in each class in our proposed system.

4- Column 4 represent the percent reduction. This shows how much we reduce the quantity of methods in each class. This percent is obtained from the evaluation of the formula 

\[ \frac{(A-B)}{A} \times 100 \]

where A is the second column and B is the third column.

By the addition of all the numbers in the second column we obtained the total of methods in the Smalltalk-80 system for the preselected classes before the reduction (Total 1089).

In the same way as in column two we found the total of all the methods in our system after the reduction (Total 679). If we use the formula \( \frac{(A-B)}{A} \times 100 \) where A=1089 and B=679 we found that the total reduction in the preselected classes is 37.65 percent.
Figure 2. Amount of methods in each class before and after the reduction.

<table>
<thead>
<tr>
<th>Classes</th>
<th>Quantity of methods before reduction</th>
<th>Quantity of methods after reduction</th>
<th>% reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>87</td>
<td>47</td>
<td>45.98</td>
</tr>
<tr>
<td>Magnitude</td>
<td>10</td>
<td>4</td>
<td>60.00</td>
</tr>
<tr>
<td>Character</td>
<td>37</td>
<td>16</td>
<td>56.76</td>
</tr>
<tr>
<td>Number</td>
<td>51</td>
<td>12</td>
<td>76.47</td>
</tr>
<tr>
<td>Integer</td>
<td>66</td>
<td>25</td>
<td>62.12</td>
</tr>
<tr>
<td>SmallInteger</td>
<td>44</td>
<td>27</td>
<td>38.64</td>
</tr>
<tr>
<td>Lookupkey</td>
<td>9</td>
<td>8</td>
<td>11.11</td>
</tr>
<tr>
<td>Association</td>
<td>5</td>
<td>5</td>
<td>0.00</td>
</tr>
<tr>
<td>Collection</td>
<td>37</td>
<td>29</td>
<td>21.62</td>
</tr>
<tr>
<td>SequenceableCollection</td>
<td>35</td>
<td>27</td>
<td>22.85</td>
</tr>
<tr>
<td>ArrayedCollection</td>
<td>12</td>
<td>6</td>
<td>50.00</td>
</tr>
<tr>
<td>Array</td>
<td>6</td>
<td>6</td>
<td>0.00</td>
</tr>
<tr>
<td>String</td>
<td>47</td>
<td>27</td>
<td>42.55</td>
</tr>
<tr>
<td>Symbol</td>
<td>31</td>
<td>25</td>
<td>19.35</td>
</tr>
<tr>
<td>Interval</td>
<td>22</td>
<td>18</td>
<td>18.18</td>
</tr>
<tr>
<td>OrderedCollection</td>
<td>43</td>
<td>34</td>
<td>20.93</td>
</tr>
<tr>
<td>Bag</td>
<td>14</td>
<td>9</td>
<td>35.71</td>
</tr>
<tr>
<td>Set</td>
<td>23</td>
<td>20</td>
<td>13.04</td>
</tr>
<tr>
<td>Dictionary</td>
<td>37</td>
<td>30</td>
<td>18.92</td>
</tr>
<tr>
<td>Stream</td>
<td>12</td>
<td>6</td>
<td>50.00</td>
</tr>
<tr>
<td>PositionableStream</td>
<td>20</td>
<td>16</td>
<td>20.00</td>
</tr>
<tr>
<td>ReadStream</td>
<td>4</td>
<td>4</td>
<td>0.00</td>
</tr>
<tr>
<td>WriteStream</td>
<td>24</td>
<td>13</td>
<td>45.83</td>
</tr>
<tr>
<td>ReadWriteStream</td>
<td>8</td>
<td>2</td>
<td>75.00</td>
</tr>
<tr>
<td>UndefinedObject</td>
<td>9</td>
<td>7</td>
<td>22.22</td>
</tr>
<tr>
<td>Boolean</td>
<td>15</td>
<td>10</td>
<td>33.33</td>
</tr>
<tr>
<td>False</td>
<td>10</td>
<td>8</td>
<td>20.00</td>
</tr>
<tr>
<td>True</td>
<td>10</td>
<td>8</td>
<td>20.00</td>
</tr>
<tr>
<td>Behavior</td>
<td>115</td>
<td>90</td>
<td>21.74</td>
</tr>
<tr>
<td>ClassDescription</td>
<td>63</td>
<td>27</td>
<td>57.14</td>
</tr>
<tr>
<td>Class</td>
<td>43</td>
<td>29</td>
<td>32.55</td>
</tr>
<tr>
<td>Metaclass</td>
<td>26</td>
<td>24</td>
<td>7.69</td>
</tr>
<tr>
<td>InstructionStream</td>
<td>16</td>
<td>16</td>
<td>0.00</td>
</tr>
<tr>
<td>ContextPart</td>
<td>79</td>
<td>25</td>
<td>68.35</td>
</tr>
<tr>
<td>BlockContext</td>
<td>19</td>
<td>19</td>
<td>0.00</td>
</tr>
<tr>
<td>ClassEditor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1089</td>
<td>679</td>
<td>37.65</td>
</tr>
</tbody>
</table>
# B. Class Index

<table>
<thead>
<tr>
<th>Object</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magnitude</td>
<td>30</td>
</tr>
<tr>
<td>Character</td>
<td>31</td>
</tr>
<tr>
<td>Number</td>
<td>33</td>
</tr>
<tr>
<td>Integer</td>
<td>35</td>
</tr>
<tr>
<td>SmallInteger</td>
<td>38</td>
</tr>
<tr>
<td>Lookupkey</td>
<td>42</td>
</tr>
<tr>
<td>Association</td>
<td>43</td>
</tr>
<tr>
<td>Collection</td>
<td>44</td>
</tr>
<tr>
<td>SequenceableCollection</td>
<td>48</td>
</tr>
<tr>
<td>ArrayedCollection</td>
<td>52</td>
</tr>
<tr>
<td>Array</td>
<td>53</td>
</tr>
<tr>
<td>String</td>
<td>54</td>
</tr>
<tr>
<td>Symbol</td>
<td>58</td>
</tr>
<tr>
<td>Interval</td>
<td>61</td>
</tr>
<tr>
<td>OrderedCollection</td>
<td>63</td>
</tr>
<tr>
<td>Bag</td>
<td>67</td>
</tr>
<tr>
<td>Set</td>
<td>68</td>
</tr>
<tr>
<td>Dictionary</td>
<td>70</td>
</tr>
<tr>
<td>Stream</td>
<td>73</td>
</tr>
<tr>
<td>PositionableStream</td>
<td>74</td>
</tr>
<tr>
<td>ReadStream</td>
<td>77</td>
</tr>
<tr>
<td>WriteStream</td>
<td>78</td>
</tr>
<tr>
<td>ReadWriteStream</td>
<td>80</td>
</tr>
<tr>
<td>UndefinedObject</td>
<td>81</td>
</tr>
<tr>
<td>Boolean</td>
<td>82</td>
</tr>
<tr>
<td>False</td>
<td>84</td>
</tr>
<tr>
<td>True</td>
<td>85</td>
</tr>
<tr>
<td>Behavior</td>
<td>86</td>
</tr>
<tr>
<td>ClassDescription</td>
<td>97</td>
</tr>
<tr>
<td>Class</td>
<td>101</td>
</tr>
<tr>
<td>Metaclass</td>
<td>105</td>
</tr>
<tr>
<td>InstructionStream</td>
<td>109</td>
</tr>
<tr>
<td>ContextPart</td>
<td>111</td>
</tr>
<tr>
<td>BlockContext</td>
<td>114</td>
</tr>
<tr>
<td>ClassEditor *</td>
<td>117</td>
</tr>
<tr>
<td>File *</td>
<td>118</td>
</tr>
</tbody>
</table>
Object

The class Object is a final class that has no superclass. In other words Object is a superclass of all the other classes in the system. The messages included in this class provide default behavior for the creation of new classes.

Object methods for initialize-release

release
Remove references to objects that may refer to the receiver. Answers self. This message should be overridden by subclasses with any cycles, in which case the subclass should also include the expression super release.

Object methods for accessing

at: index
Answer the value of an indexable field in the receiver. Fail if the argument index is not an Integer or is out of bounds. Essential. See documentation in Object metaclass.

<primitive: 60>

at: index put: value
Store the argument value in the indexable field of the receiver indicated by index. Fail if the index is not an Integer or is out of bounds. Or fail if the value is not of the right type for this kind of collection. Answer the value that was stored. Essential. See documentation in Object metaclass.

<primitive: 61>

basicAt: index
Answer the value of an indexable field in the receiver. Fail if the argument index is not an Integer or is out of bounds. Essential. See documentation in Object metaclass. Do not override this message in any subclass.

<primitive: 60>

basicAt: index put: value
Store the second argument value in the indexable field of the receiver indicated by index. Fail if the index is not an Integer or is out of bounds. Or
fail if the value is not of the right type for this kind of collection. Answer the value that was stored. Essential. See documentation in Object metaclass. Do not override in a subclass.

<primitive: 61>

basicSize
Answer the number of indexable fields in the receiver. This value is the same as the largest legal subscript. Essential. See documentation in Object metaclass. Do not override in any subclass.

<primitive: 62>

readFromString: aString
Create an object based on the contents of aString.

size
Answer the number of indexable fields in the receiver. This value is the same as the largest legal subscript. Essential. See documentation in Object metaclass.

<primitive: 62>

yourself
Answer self.

Object methods for testing

isInteger
Coerces Integers to true and everything else to false. Integer overrides with ^true.

isNil
Coerces nil to true and everything else to false. UndefinedObject overrides with ^true.

notNil
Coerces nil to false and everything else to true. UndefinedObject overrides with ^false.

Object methods for comparing

= anObject
Answer true if the receiver and the argument represent the same object and false otherwise. If = is redefined in any subclass, consider also redefining the message hash.

24
== anObject
Answer true if the receiver and the argument are
the same object (have the same object pointer) and
false otherwise. Do not redefine the message == in
any other class Essential. No Lookup. See
documentation in Object metaclass.

<primitive: 110>

hash
Answer a SmallInteger whose value is half of the
receiver’s object pointer (interpreting object
pointers as 16-bit signed quantities). Fails if the
receiver is a SmallInteger. Essential. See
documentation in Object metaclass.

<primitive: 75>

hashMappedBy: map
Answer what my hash would be if oops changed
according to map.

Object methods for copying

copy
Answer another instance just like the receiver.
Subclasses typically override this method; they
typically do not override shallowCopy

deepCopy
Answer a copy of the receiver with its own copy of
each instance variable.

shallowCopy
Answer a copy of the receiver which shares the
receiver’s instance variables.

Object methods for printing

isLiteral
Answer whether the receiver has a literal text form
recognized by the compiler.
printOn: aStream  
Append to the argument aStream a sequence of  
characters that identifies the receiver.

printString  
Answer a String whose characters are a description  
of the receiver.

storeOn: aStream  
Append to the argument aStream a sequence of  
characters that is an expression whose evaluation  
creates an object similar to the receiver.

storeString  
Answer a String representation of the receiver from  
which the receiver can be reconstructed.

Object methods for class membership

class  
Answer the object which is the receiver’s class.  
Essential. See documentation in Object metaclass.

<primitive: 111>

isKindOf: aClass  
Answer a Boolean as to whether the class, aClass,  
is a superclass or class of the receiver.

isMemberOf: aClass  
Answer a Boolean as to whether the receiver is an  
instance of the class, aClass.

respondsTo: aSymbol  
Answer a Boolean as to whether the method  
dictionary of the receiver’s class contains aSymbol  
as a message selector.

Object methods for message handling

perform: selector withArguments: anArray  
Send the receiver the keyword message indicated by  
the arguments. The argument selector is the  
selector of the message. The arguments of the  
message are the elements of anArray. Invoke  
messageNotUnderstood: if the number of arguments  
expected by the selector is not the same as the
length of anArray. Essential. See documentation in Object metaclass.

<primitive: 84>

Object methods for error handling

error: aString
The default behavior for error: is the same as halt:. This additional message is the one a subclass should override in order to change the handling of errors.

primitiveFailed
Announce that a primitive has failed and there is no appropriate Smalltalk code to run.

shouldNotImplement
Announce that although the receiver inherits this message, it should not implement it.

subclassResponsibility
This message sets up a framework for the behavior of the class’ subclasses. Announce that the subclass should have implemented this message.

Object methods for system primitives

asOop
Answer an Integer which is unique to me. See below. Essential. See also documentation in Object metaclass.

<primitive: 75>

instVarAt: index
Answer a fixed variable in an object. The numbering of the variables corresponds to the named instance variables. Fail if the index is not an Integer or is not the index of a fixed variable. Essential. See documentation in Object metaclass.

<primitive: 73>

instVarAt: anInteger put: anObject
Store a value into a fixed variable in the receiver. The numbering of the variables
corresponds to the named instance variables. Fail if the index is not an Integer or is not the index of a fixed variable. Answer the value stored as the result. Using this message violates the principle that each object has sovereign control over the storing of values into its instance variables. Essential. See documentation in Object metaclass.

<primitive: 74>

**nextInstance**

Answer the next instance after the receiver in the enumeration of all instances of this class. Fails if all instances have been enumerated.

<primitive: 78>

Object methods for private

**errorImproperStore**

Create an error notification that an improper store was attempted.

**errorNonIntegerIndex**

Create an error notification that an improper object was used as an index.

**errorSubscriptBounds: index**

Create an error notification that an improper integer was used as an index.

**mustBeBoolean**

Catches attempts to test truth of non-Booleans. This message is sent from the interpreter.

**nilFields**

Store nil into all pointer fields of the receiver.

**species**

Answer the preferred class for reconstructing the receiver. For example, collections create new collections whenever enumeration messages such as collect: or select: are invoked. The new kind of collection is determined by the species of the original collection. Species and class are not always the same. For example, the species of Interval is Array.
Object methods for converting

-> aValue
   Answer an Association with the receiver as key and aValue as the value.

Object methods for 'compiler access'

bindingOf: aName forStore: aBoolean
   The compiler cannot find a binding for aName. Answer an association, if any, of aName and its value. Answer nil if none. aBoolean is true if aName is the object of a store (i.e. appears on the left of an assignment.) All models should respond to this message. It is located here (in Object) because this is the only common superclass of all models.

Object class methods for class initialization

initialize

Object class methods for instance creation

readFrom: aStream
   Create an object based on the contents of aStream.
Object

**Magnitude**
The class Magnitude is the abstract class that provides protocol for objects that have the ability to be compared along a linear dimension.

Magnitude methods for comparing

Note: Other methods like >, >= can be created easily from <, =, that are provided in the system.

< aMagnitude
  Compare the receiver with the argument and answer with true if the receiver is less than the argument. Otherwise answer false.

= aMagnitude
  Compare the receiver with the argument and answer with true if the receiver is equal to the argument. Otherwise answer false.

hash
  Hash must be redefined whenever = is redefined.

hashMappedBy: map
  My hash is independent of my oop.
This class represents characters by storing their associated ASCII (extended to 256 codes) code. The instances of this class are created uniquely, so that all instances $R$ (for instance) are identical.

Character methods for comparing

$<$ aCharacter
  Answer true if the receiver’s value $<$ aCharacter’s value.

= aCharacter
  Answer true if the receiver and the argument are the same object (have the same object pointer) and false otherwise.

  <primitive: 110>

hash

hashMappedBy: map

Character methods for accessing

asciiValue
  Answer the value of the receiver.

Character methods for copying

copy
  Answer with me because Characters are unique.

deepCopy
  Answer with self because Characters are unique.

Character methods for printing

isLiteral

printOn: aStream

storeOn: aStream
  Character literals are preceded by ‘$’.
Character methods for converting

asCharacter
  Answer the receiver itself.

asSymbol

Character class methods for class initialization

initialize
  Create the table of unique Characters. This code is not shown so that the user cannot destroy the system by trying to recreate the table.

Character class methods for instance creation

digitValue: x
  Answer the Character whose digit value is x. For example, answer $9 for x=9, $0 for x=0, $A for x=10, $Z for x=35.

value: anInteger
  Answer the Character whose value is anInteger. Characters are unique; they are stored in the class variable CharacterTable.

Character class methods for constants

characterTable
  Answer the class variable in which unique Characters are stored.
Object
Magnitude
Number
The abstract class Number defines the protocol of all numeric objects. Its messages support arithmetic operations and comparisons.

Number methods for arithmetic

+ aNumber
   Answer the sum of the receiver and aNumber.
   self subclassResponsibility

- aNumber
   Answer the difference between the receiver and aNumber.
   self subclassResponsibility

Number methods for coercing

coerce: aNumber
   Answer with a number representing the argument, aNumber, that is represented the same kind of Number as is the receiver. Must be defined by all Number classes.

generality
   Answer the number representing the ordering of the receiving in the generality hierarchy. A number in this hierarchy coerces to numbers higher in hierarchy (i.e., with larger generality numbers).

retry: aSymbol coercing: aNumber
   Arithmetic represented by the symbol, aSymbol, could not be performed with the receiver and the argument, aNumber, because of the differences in representation. Coerce either the receiver or the argument, depending on which has higher generality, and try again. If the symbol is the equals sign, answer false if the argument is not a Number. If the generalities are the same, create an error message.

Number methods for converting

asInteger
   Answer an integer nearest the receiver toward zero.

Number methods for intervals

to: stop
Answer an Interval from the receiver up to the argument, stop, incrementing by 1.

to: stop by: step
Answer an Interval from the receiver up to the argument, stop, incrementing by step.

to: stop by: step do: aBlock
Create an Interval from the receiver up to the argument, stop, incrementing by step. For each element of the interval, evaluate the block, aBlock.

to: stop do: aBlock
Create an Interval from the receiver up to the argument, stop, incrementing by 1. For each element of the interval, evaluate the block, aBlock.

Number methods for printing

storeOn: aStream
Numbers print in a form which is recognized by the compiler.

Number class methods for instance creation

readFrom: aStream
Answer an instance of Number as described on the stream, aStream.
The class Integer provides protocol for objects with integer values.

Integer methods for testing

**isInteger**

Integer methods for comparing

`< anInteger`

- Compare the receiver with the argument and return true if the receiver is less than the argument. Otherwise return false.

`= anInteger`

- Compare the receiver with the argument and return true if the receiver is equal to the argument. Otherwise return false.

**hash**

**hashMappedBy:** map

Integer methods for arithmetic

`+ aNumber`

- Add the receiver to the argument and answer with the result.

`- aNumber`

- Subtract the argument from the receiver and answer with the result.

Integer methods for enumerating

**timesRepeat:** aBlock

- Evaluate the argument, aBlock, the number of times represented by the receiver.
Integer methods for bit manipulation

anyMask: mask
Treat the argument as a bit mask. Answer true if any of the bits that are 1 in the argument are 1 in the receiver.

bitAnd: aNumber
Logical AND the twos-complement representation of the receiver with the twos-complement representation of the argument and return the result.

bitOr: aNumber
Logical OR the twos-complement representation of the receiver with the twos-complement representation of the argument and return the result.

bitShift: anInteger
Logical SHIFT the twos-complement representation of the receiver with the twos-complement representation of the argument and return the result. Shift left if the argument is positive, right if the argument is negative. Zeros are shifted in from the right in left shifts. The sign bit is extended in right shifts.

bitXor: aNumber
Logical XOR the twos-complement representation of the receiver with the twos-complement representation of the argument and return the result.

highBit
Answer with the index of the high order bit of the binary representation of the receiver.

Integer methods for printing

isLiteral

printOn: aStream

printOn: aStream base: b

Integer methods for converting
asInteger
  Answer with the receiver itself.

Integer methods for coercing

coerce: aNumber

generality

Integer methods for system primitives

lastDigit
  Answer the last digit of the integer.

lastDigitGet: digit
  Store the argument, digit, as the last digit of the integer.

Integer methods for private

Integer class methods for instance creation

new: length neg: neg
  Answer an instance of a large integer whose size is length. neg is a flag determining whether the integer is negative or not.

readFrom: aStream
  Answer a new Integer as described on the stream, aStream.

readFrom: aStream radix: radix
  Answer an instance of one of my concrete subclasses. Initial minus sign accepted, and bases > 10 use letters A-Z. imbedded radix specifiers not allowed - use Integer class readFrom: for that. Answer zero (not an error) if there are no digits.
SmallInteger methods for arithmetic

+ aNumber
Add the receiver to the argument and answer with the result if it is a SmallInteger. Fail if the argument or the result is not a SmallInteger.

<primitive: 1>

- aNumber
Subtract the argument from the receiver and answer with the result if it is a SmallInteger. Fail if the argument or the result is not a SmallInteger.

<primitive: 2>

SmallInteger methods for bit manipulation

bitAnd: arg
Logical AND the twos-complement representation of the receiver with the argument and answer with the result. Fail if the argument is not a SmallInteger.

<primitive: 14>

bitOr: arg
Logical OR the twos-complement representation of the receiver with the argument and answer with the result. Fail if the argument is not a SmallInteger.

<primitive: 15>

bitShift: arg
Answers with a SmallInteger whose value (in twos-complement representation) is the receiver’s value (in twos-complement representation) shifted left by the number of bits indicated by the argument. Negative arguments shift right. Zeros are shifted in from the right in left shifts. The
sign bit is extended in right shifts. Fail if the result cannot be represented as a SmallInteger.

<primitive: 17>

**bitXor:** arg

Exclusive OR the twos-complement representation of the receiver with the argument and answer with the result. Fail if the argument is not a SmallInteger.

<primitive: 16>

**highBit**

SmallInteger methods for comparing

< aNumber

Compare the receiver with the argument and answer with true if the receiver is less than the argument. Otherwise answer false. Fail if the argument is not a SmallInteger. Essential. No Lookup. See Object documentation whatIsAPrimitive.

<primitive: 3>

= aNumber

Compare the receiver with the argument and answer true if the receiver is equal to the argument. Otherwise answer false. Fail if the argument is not a SmallInteger.

<primitive: 7>

**hash**

hashMappedBy: map

SmallInteger methods for copying

deepCopy

shallowCopy

SmallInteger methods for coercing

coerce: n

generality
SmallInteger methods for printing

printOn: aStream base: b

SmallInteger methods for system primitives

asObject
  <primitive: 76>

asOop
  Answer an integer which is unique to SmallInteger, consistent with Object.asOop.

digitAt: n
  Answer the value of an apparent indexable field. This is provided for compatibility with LargeInteger.

digitLength
  Answer with the number of indexable fields in the receiver. This value is the same as the largest legal subscript. Included so that a SmallInteger can behave like a LargeInteger.

instVarAt: i
  Small integer has to be specially handled

SmallInteger methods for private

subtractOrFail: aNumber
  This is a private copy of the subtraction primitive, used by SmallInteger class initialize to discover the correct value of SmallInteger minVal.

SmallInteger class methods for class initialization

initialize
  Initialize the digit buffer, and discover the range of SmallIntegers on the system.

SmallInteger class methods for constants

maxBits
  Answer N such that maxVal is (2^N)-1.

maxBytes

40
Answer $N$ such that $\text{maxVal} < 256^N$.

**maxVal**

Answer the maximum value for a SmallInteger.

**minVal**

Answer the minimum value for a Small Integer.
Object
  Magnitude
  Lookupkey
I serve as the key for looking up entries in a Dictionary. Subclasses of me are dictionary entries. An available example is Association.

LookupKey methods for comparing

< aLookupKey
= aLookupKey

hash

hashMappedBy: map
  Answer what my hash would be if oops changed according to map.

LookupKey methods for accessing

key
  Answer the lookup key of the receiver.

key: anObject
  Store the argument, anObject, as the lookup key of the receiver.

LookupKey methods for printing

printOn: aStream

LookupKey class methods for instance creation

key: aKey
  Answer a new instance of me with the argument as the lookup up.
Object
  Magnitude
  LookupKey

Association
Association is a pair of associated objects—a key and a value. It can serve as an entry in a dictionary. Think of It’s key as the left-hand side of the dictionary and It’s value as the right-hand side.

Association methods for accessing

key: aKey value: anObject
  Store the arguments as the variables of the receiver.

value
  Answer the value of the receiver.

value: anObject
  Store the argument, anObject, as the value of the receiver.

Association methods for printing

printOn: aStream

Association class methods for instance creation

key: newKey value: newValue
  Answer a new instance of Association with the arguments as the key and value of the association.
Object

Collection
Collection is the abstract class of all collection classes. It provides protocol for groups of objects such as Sets or Arrays.

Collection methods for ‘accessing’

size
Answer how many elements the receiver contains.

Collection methods for testing

includes: anObject
Answer whether anObject is one of the receiver’s elements.

isEmpty
Answer whether the receiver contains any elements.

occurrencesOf: anObject
Answer how many of the receiver’s elements are equal to anObject.

Collection methods for adding

add: newObject
Include newObject as one of the receiver’s elements. Answer newObject. This message should not be sent to instances of subclasses of ArrayedCollection.

addAll: aCollection
Include all the elements of aCollection as the receiver’s elements. Answer aCollection.

Collection methods for removing

remove: oldObject
Remove oldObject as one of the receiver’s elements. Answer oldObject unless no element is equal to oldObject, in which case, create an error message.

remove: oldObject ifAbsent: anExceptionBlock
Remove oldObject as one of the receiver’s elements. If several of the elements are equal to oldObject,
only one is removed. If no element is equal to oldObject, answer the result of evaluating anExceptionBlock. Otherwise, answer the argument, oldObject. SequenceableCollections can not respond to this message. self subclassResponsibility

Collection methods for enumerating

**collect: aBlock**
Evaluate aBlock with each of the receiver’s elements as the argument. Collect the resulting values into a collection that is like the receiver. Answer the new collection.

**detect: aBlock ifNone: exceptionBlock**
Evaluate aBlock with each of the receiver’s elements as the argument. Answer the first element for which aBlock evaluates to true.

**do: aBlock**
Evaluate aBlock with each of the receiver’s elements as the argument.

**inject: thisValue into: binaryBlock**
Accumulate a running value associated with evaluating the argument, binaryBlock, with the current value and the receiver as block arguments. The initial value is the value of the argument, thisValue. For instance, to sum a collection, use: collection inject: 0 into: [:subTotal :next | subTotal + next].

**reject: aBlock**
Evaluate aBlock with each of the receiver’s elements as the argument. Collect into a new collection like the receiver, only those elements for which aBlock evaluates to false. Answer the new collection.

**select: aBlock**
Evaluate aBlock with each of the receiver’s elements as the argument. Collect into a new collection like the receiver, only those elements for which aBlock evaluates to true. Answer the new collection.

Collection methods for printing

45
printOn: aStream
storeOn: aStream

Collection methods for converting

asBag
   Answer a new instance of Bag whose elements are the elements of the receiver.

asSet
   Answer a new instance of Set whose elements are the unique elements of the receiver.

Collection methods for private

emptyCheck

errorEmptyCollection

errorNoMatch

errorNotFound

errorNotKeyed

growSize
   Answer an amount by which the receiver should grow to make room for more elements (in response to the message 'grow').

maxPrint
   Answer the maximum number of characters to print with printOn: .

maxSize
   Answer the largest basicSize which is valid for the receiver's class.
Collection class methods for class initialization

**initialize**
Initialize the class variables for the 16 bit 4404.

Collection class methods for instance creation

**with: anObject**
Answer a new instance of me containing anObject.

Collection class methods for constants

**maxSize**
Answer the largest basicSize which is valid for the receiver.
Object
  Collection
    SequenceableCollection
Class SequenceableCollection represents collections whose elements are ordered and are externally named by integer indices. SequenceableCollection is a subclass of Collection and provides the protocol for accessing, copying, and enumerating elements of a collection when it is known that there is an ordering associated with the elements.

SequenceableCollection methods for comparing

= otherCollection
Answer whether the species of the receiver is the same as otherCollection’s species, and the receiver’s size is the same as otherCollection’s size, and each of the receiver’s elements equal the corresponding element of otherCollection.

hash
Equal has been redefined -- hash is redefined.

hashMappedBy: map

SequenceableCollection methods for accessing

first
Answer the first element of the receiver. Create an error if the receiver contains no elements.

indexOf: anElement
Answer the index of anElement within the receiver. If the receiver does not contain anElement, answer 0.

indexOf: anElement ifAbsent: exceptionBlock
Answer the index of anElement within the receiver. If the receiver does not contain anElement, answer the result of evaluating the exceptionBlock.

indexOfSubCollection: aSubCollection startingAt: anIndex
Answer the index of the receiver’s first element, such that that element equals the first element of aSubCollection, and the next elements equal the rest of the elements of aSubCollection. Begin the search at element anIndex of the receiver. If no such match is found, answer 0.

indexOfSubCollection: aSubCollection startingAt: anIndex ifAbsent: exceptionBlock
Answer the index of the receiver’s first element, such that that element equals the first element of aSubCollection, and the next elements equal the rest of the elements of aSubCollection. Begin the search at element anIndex of the receiver. If no such match is found, answer the result of evaluating exceptionBlock.

last

Answer the last element of the receiver. Create an error if the receiver contains no elements.

replaceFrom: start to: stop with: replacement startingAt: repStart

This destructively replaces elements from start to stop in the receiver starting at index, repStart, in the collection, replacement. Answer the receiver. No range checks are performed - this may be primitively implemented.

size

SequenceableCollection methods for adding

grow

The receiver becomes bigger--this is not a copy of the receiver, so all shared references survive.

SequenceableCollection methods for removing

remove: oldObject ifAbsent: anExceptionBlock

SequencableCollections cannot implement removing.

SequenceableCollection methods for copying

, aSequenceableCollection

Answer with a copy of the receiver concatenated with the argument, a SequencableCollection.

copyFrom: start to: stop

Answer a copy of a subset of the receiver, starting from element at index start until element at index stop.
copyReplaceFrom: start to: stop with: replacementCollection
Answer a copy of the receiver satisfying the following conditions: If stop is less than start, then this is an insertion; stop should be exactly start-1, start = 1 means insert before the first character, start = size+1 means append after last character. Otherwise, this is a replacement; start and stop have to be within the receiver’s bounds.

copyWith: newElement
Answer a copy of the receiver that is 1 bigger than the receiver and has newElement at the last element.

copyWithout: oldElement
Answer a copy of the receiver in which all occurrences of oldElement have been left out.

shallowCopy

SequenceableCollection methods for enumerating

collect: aBlock

do: aBlock

reverse
Answer with a new collection like me with its elements in the opposite order.

reverseDo: aBlock
Evaluate aBlock with each of the receiver’s elements as the argument, starting with the last element and taking each in sequence up to the first. For SequenceableCollections, this is the reverse of the enumeration in do:.

with: aSequenceableCollection do: aBlock
Evaluate aBlock with each of the receiver’s elements along with the corresponding element from aSequenceableCollection.

SequenceableCollection methods for converting

asArray

50
Answer a new instance of Array whose elements are the elements of the receiver, in the same order.

SequenceableCollection methods for private

toOutOfBounds

swap: oneIndex with: anotherIndex
    Move the element at oneIndex to anotherIndex, and vice-versa.
ArrayedCollection is a subclass of SequenceableCollection. It represents a collection of elements with a fixed range of integers as external keys.

ArrayedCollection methods for accessing

size
Answer the number of indexable fields in the receiver. This value is the same as the largest legal subscript. Primitive is specified here to override SequenceableCollection size.

<primitive: 62>

ArrayedCollection methods for printing

storeOn: aStream

ArrayedCollection methods for 'private'

defaultElement

ArrayedCollection class methods for instance creation

new
Answer a new instance of me, with size = 0.

with: anObject
Answer a new instance of me, containing only anObject.

ArrayedCollection class methods for 'constants'

maxSize
Answer the largest basicSize which is valid for the receiver.
Instances of the class Array are perhaps the most commonly used data structure in Smalltalk programs. Arrays are represented textually by a pound sign preceding the list of array elements.

Array methods for comparing

hash

hashMappedBy: map
   Answer what my hash would be if oops changed according to map.

Array methods for converting

asArray
   Answer with the receiver itself.

Array methods for printing

isLiteral

printOn: aStream

storeOn: aStream
   Use the literal form if possible.
Object
Collection
SequenceableCollection
ArrayedCollection

String
The String class is an indexed collection of Characters. Instances of the class String are similar to Arrays except that the individual elements must be Characters.

String methods for comparing

< aString
Answer true if and only if the receiver collates before aString. The collation sequence is ascii with case differences ignored.

= aString
Compare two strings. Primitive 148 handles LargePositiveInteger indices as well as SmallIntegers.

<primitive:148>

hash

hashMappedBy: map

match: text
Answer whether text matches the pattern in the receiver. Matching ignores upper/lower case differences. Where the receiver contains #, text may contain any single character. Where the receiver contains *, text may contain any sequence of characters.

spellAgainst: aString
Answer an integer between 0 and 100 indicating how similar the argument is to the receiver. No case conversion is done.

String methods for accessing

at: index
Answer the Character stored in the field of the receiver indexed by the argument. Fail if the index argument is not an Integer or is out of bounds.
<primitive: 63>
at: index put: aCharacter
Store the Character in the field of the receiver indicated by the index. Fail if the index is not an Integer or is out of bounds, or if the argument is not a Character.
<primitive: 64>

basicAt: index
Answer the Character stored in the field of the receiver indexed by the argument. Fail if the index argument is not an Integer or is out of bounds.
<primitive: 63>

basicAt: index put: aCharacter
Store the Character in the field of the receiver indicated by the index. Fail if the index is not an Integer or is out of bounds, or if the argument is not a Character.
<primitive: 64>

findString: subString startingAt: start
Answer the index of subString within the receiver, starting at start. If the receiver does not contain subString, answer 0.

replaceFrom: start to: stop with: replacement startingAt: repStart
This destructively replaces elements from start to stop in the receiver starting at index, repStart, in the collection, replacement. Answer the receiver.

size
Answer the number of indexable fields in the receiver. This value is the same as the largest legal subscript.
<primitive: 62>

string
Answer the receiver itself. This is for compatibility with other text classes.
String methods for 'copying'

deepCopy
DeepCopy would otherwise mean make a copy of the character; since characters are unique, just return a shallowCopy.

String methods for printing

isLiteral

printOn: aStream
Print inside string quotes, doubling imbedded quotes.

storeOn: aStream
Print inside string quotes, doubling imbedded quotes.

String methods for converting

asLowercase
Answer a string made up from the receiver whose characters are all lowercase.

asString
Answer the receiver itself.

asSymbol
Answer the unique symbol whose characters are the characters of the string.

asUppercase
Answer a string made up from the receiver whose characters are all uppercase.

String methods for 'private'

compare: s
primReplaceFrom: start to: stop with: replacement startingAt: repStart

This destructively replaces elements from start to stop in the receiver starting at index, repStart, in the collection, replacement. Answer the receiver. The range errors cause the primitive to fail.

<primitive: 105>

stringhash

String class methods for instance creation

fromString: aString
Answer a new String that is a copy of the argument, aString.

readFrom: inStream
Answer a new String that is determined by reading the stream, inStream. Embedded double quotes become the quote Character.
Symbols are Strings which are created uniquely.

Symbol methods for comparing

\(\text{=} \ \text{anObject}\)

\text{hash}

Answer with a SmallInteger whose value is half of the receiver’s object pointer (interpreting object pointers as 16-bit signed quantities).

<primitive: 75>

\text{hashMappedBy: map}

Answer what my hash would be if oops changed according to map.

Symbol methods for copying

\text{copy}

Answer with me, because Symbols are unique.

\text{shallowCopy}

Answer with me, because Symbols are unique.

Symbol methods for printing

\text{isLiteral}

\text{printOn: aStream}

\text{storeOn: aStream}

Symbol methods for converting

\text{asString}

\text{asSymbol}
Symbol methods for 'private'

errorNoModification

species

string: aString

stringhash

Symbol methods for system primitives

classPart

isCompound

keywords

Answer an array of the keywords that compose the receiver.

numArgs

Answer the number of arguments that the receiver requires if it is interpreted as a message selector.

selectorPart

Return just my part after the class name if the receiver is a compound selector (otherwise the whole thing).

Symbol class methods for class initialization

initialize

Symbol class methods for instance creation

intern: aString

Answer a unique Symbol whose characters are those of aString.

internCharacter: aCharacter

Answer with, and create if necessary, a unique Symbol whose characters are just this character.
Symbol class methods, for private

hasInterned: aString ifTrue: symBlock
Answer with false if aString hasn't been interned (into a Symbol), otherwise supply the symbol to symBlock and return true.

rehash
Rebuild the hash table that holds all the unique Symbols.

table
Access for SystemTracer

Symbol initialize
Object
  Collection
    SequenceableCollection
      Interval
Is a collection of numbers representing a mathematical progression.

Interval methods for comparing

= anInterval
  Answer true if Interval species and anInterval species are equal, and if Interval starts, steps and sizes are equal.

hash

hashMappedBy: map

Interval methods for accessing

at: anInteger
  Answer the anInteger’th element.

first

increment
  Answer the receiver’s interval increment.

last

size

Interval methods for copying

copy
  Return a copy of Interval.

Interval methods for enumerating

collect: aBlock

do: aBlock

reverseDo: aBlock
  Evaluate aBlock for each element of my interval, in reverse order.
Interval methods for printing

`printOn: aStream`

`storeOn: aStream`

This is possible because we know numbers store and print the same.

Interval methods for 'private'

`setFrom: startInteger to: stopInteger by: stepInteger`

`species`

Interval class methods for instance creation

`from: startInteger to: stopInteger by: stepInteger`

Answer a new instance of me, starting at startInteger, ending and stopInteger, and with an interval increment of stepInteger.

`new`

Create and answer with a new instance of the receiver (a class) with no indexable fields. Fail if the class is indexable. Override SequenceableCollection new.

`<primitive: 70>`
OrderedCollections are ordered by the sequence in which objects are added and removed from them. The elements are accessible by external keys that are indices. The accessing, adding, and removing protocols are augmented to refer to the first and last elements, and to elements preceding or succeeding other elements.

OrderedCollection methods for accessing

**at: anInteger**
Answer with my element at index anInteger. at: is used by a knowledgeable client to access an existing element.

**at: anInteger put: anObject**
Put anObject at element index anInteger. at:put: can not be used to append, front or back, to an orderedcollection; it is used by a knowledgeable client to replace an element.

**first**
Answer the first element. If the receiver is empty, create an error message. This is a little faster than the implementation in the superclass self emptyCheck.

**last**
Answer the last element. If the receiver is empty, create an error message. This is a little faster than the implementation in the superclass self emptyCheck.

**size**

OrderedCollection methods for copying

**copyEmpty**
Answer a copy of the receiver that contains no elements.

**copyFrom: startIndex to: endIndex**
Answer a copy of the receiver that contains elements from position startIndex to endIndex.
copyReplaceFrom: start to: stop with: replacementCollection
Answer a copy of the receiver with replacementCollection’s elements in place of the receiver’s start’th to stop’th elements. This does not expect a 1-1 map from replacementCollection to the start to stop elements, so it will do an insert or append.

copyWith: newElement
Answer a copy of the receiver that is 1 bigger than the receiver and includes the argument, newElement, at the end.

copyWithout: oldElement
Answer a copy of the receiver that does not contain any elements equal to oldElement.

OrderedCollection methods for adding

add: newObject

addAll: anOrderedCollection
Add each element of anOrderedCollection at my end. Answer anOrderedCollection.

addAllLast: anOrderedCollection
Add each element of anOrderedCollection at the end of the receiver. Answer anOrderedCollection.

addFirst: newObject
Add newObject to the beginning of the receiver. Add newObject.

addLast: newObject
Add newObject to the end of the receiver. Answer newObject.

grow
Become larger. Typically, a subclass has to override this if the subclass adds instance variables.

OrderedCollection methods for removing

remove: oldObject ifAbsent: absentBlock
removeFirst
Remove the first element of the receiver. If the receiver is empty, create an error message.

removeLast
Remove the last element of the receiver. If the receiver is empty, create an error message.

OrderedCollection methods for enumerating

collect: aBlock
Evaluate aBlock with each of my elements as the argument. Collect the resulting values into a collection that is like me. Answer with the new collection. Override superclass in order to use add:, not at:put:.

do: aBlock
override the superclass for performance

reverse
Answer with a new collection like me with its elements in the opposite order. Override superclass in order to use add:, not at:put:.

reverseDo: aBlock
override the superclass for performance

select: aBlock
Evaluate aBlock with each of my elements as the argument. Collect into a new collection like me, only those elements for which aBlock evaluates to true. Override superclass in order to use add:, not at:put:.

OrderedCollection methods for private

errorFirstObject

errorNoSuchElement

errorNotFound

find: oldObject
makeRoomAtFirst
makeRoomAtLast
setIndices

OrderedCollection class methods for instance creation

new

new: anInteger
    If a subclass adds fields, then it is necessary for
    that subclass to reimplement new:

OrderedCollection class methods for constants

maxSize
    Answer the largest basicSize which is valid for the
    receiver.
Object
   Collection
      Bag
Bag is the simplest kind of collection. It represents collections whose elements are unordered and have no external keys.

Bag methods for accessing

size

Bag methods for testing

includes: anObject

occurrencesOf: anObject

Bag methods for adding

add: newObject

add: newObject withOccurrences: anInteger
   Add the element newObject to the elements of the receiver. Do so as though the element were added anInteger number of times. Answer newObject.

Bag methods for removing

remove: oldObject ifAbsent: exceptionBlock

Bag methods for enumerating

do: aBlock

Bag methods for private

setDictionary

Bag class methods or instance creation

new
Set is an unordered collection of elements that does not permit duplication.

Set methods for accessing

size

Set methods for testing

includes: anObject

occurrencesOf: anObject

Set methods for adding

add: newObject

grow

The receiver becomes twice as big--this is not a copy of the receiver, so all shared references survive.

Set methods for removing

remove: oldObject ifAbsent: aBlock

Set methods for enumerating

collect: aBlock

Evaluate aBlock with each of the receiver's elements as the argument. Collect the resulting values into another Set. Answer the new Set. We override the general method, so that we make a big enough set and avoid growing.

do: aBlock

Set methods for private

atNewIndex: index put: anObject
find: anObject ifAbsent: aBlock
findElementOrNil: anObject
fixCollisionsFrom: index
noCheckAdd: anObject
rehash
setTally
swap: oneElement with: otherElement

Set methods for copying
deepCopy

Set class methods for instance creation
ew
new: anInteger

Set class methods for constants
maxSize Answer the largest basicSize which is valid for the receiver.
Object
  Collection
    Set
      Dictionary
Class Dictionary represents a set of associations between keys and values.

Dictionary methods for accessing

associationAt: key ifAbsent: aBlock
  Answer the association at key. If key is not found, answer the result of evaluating aBlock.

associations
  Answer an OrderedCollection containing the receiver’s associations in an arbitrary order.

at: key
  Answer the value at key. If key is not found, create an error message.

at: key ifAbsent: aBlock
  Answer the value at key. If key is not found, answer the result of evaluating aBlock.

at: key put: anObject
  Set the value at key to be anObject. If key is not found, create a new entry for key and set its value to anObject. Answer anObject.

keyAtValue: value
  Answer the key whose value equals the argument, value. If there is none, cause an error.

keyAtValue: value ifAbsent: exceptionBlock
  Answer the key whose value equals the argument, value. If there is none, answer the result of evaluating exceptionBlock.

keys
  Answer a set containing the receiver’s keys.

values
  Answer a Bag containing the receiver’s values.
Dictionary methods for testing

includes: anObject

occurrencesOf: anObject
   Answer how many of the receiver’s elements are equal to anObject.

Dictionary methods for adding

add: anAssociation

declare: key from: aDictionary
   Add key to the receiver. If key already exists, do nothing. If aDictionary includes key, then remove it from aDictionary and use its association as the entry to the receiver.

grow

Dictionary methods for enumerating

collect: aBlock
   Evaluate aBlock with each of my values as the argument. Collect the resulting values into a collection that is like me. Answer with the new collection.

do: aBlock

select: aBlock
   Evaluate aBlock with each of my values as the argument. Collect into a new dictionary, only those associations for which aBlock evaluates to true.

Dictionary methods for converting

asSortedCollection

Dictionary methods for printing

printOn: aStream

storeOn: aStream
Dictionary methods for dictionary testing

**includesKey: key**
Answer whether the receiver has a key equal to the argument, key.

Dictionary methods for dictionary removing

**removeKey: key**
Remove key from the receiver. If key is not in the receiver, create an error message. Otherwise, answer the value associated with key.

**removeKey: key ifAbsent: aBlock**
Remove key from the receiver. If key is not in the receiver, answer the result of evaluating aBlock. Otherwise, answer the value associated with key.

Dictionary methods for dictionary enumerating

**associationsDo: aBlock**
Evaluate aBlock for each of the receiver’s key/value associations.

**keysDo: aBlock**
Evaluate aBlock for each of the receiver’s keys.

Dictionary methods for private

**errorKeyNotFound**

**errorValueNotFound**

**findKey: key ifAbsent: aBlock**

**findKeyOrNil: key**

**rehash**
Smalltalk rehash.
Object
    Stream

Stream is an abstract class representing objects that stream over a self typing data structure.

Stream methods for 'accessing'

contents
    Answer the contents of the receiver.

next
    Answer the next object in the receiver.

nextPut: anObject
    Insert the argument, anObject, at the next position in the receiver.

new: newSize.

Stream methods for 'testing'

atEnd
    Answer whether the position is greater than or equal to the limit.

Stream methods for 'enumerating'

do: aBlock
    Evaluate aBlock for each of the remaining elements of the receiver.
Object
  Stream
  PositionableStream

I assume that my contents is an indexable collection and that, in support of accessing the elements of my contents, I can reposition a pointer. I am abstract in that I do not implement the messages next and nextPut: which are inherited from my superclass Stream.

PositionableStream methods for 'accessing'

contents
  Answer with a copy of my collection from 1 to readLimit.

next: anInteger
  Answer the next anInteger elements of the receiver.

peek
  Answer what would be returned with a self next, without changing position. If the receiver is at the end, answer nil.

peekFor: anObject
  Answer false and do not move the position if self next \= anObject or if the receiver is at the end. Answer true and increment position if self next = anObject.

through: anObject
  Answer a subcollection from position to the occurrence (inclusive) of anObject. If not there, answer everything.

PositionableStream methods for 'testing'

atEnd
  Answer true if the position is greater than or equal to the limit, otherwise answer false. Fail if position or readLimit is not a SmallInteger. Optional. See Object documentation whatIsAPrimitive.
isNotEmpty
Answer whether the receiver has no elements.

PositionableStream methods for 'positioning'

position
Answer the current position of accessing the stream.

position: anInteger
Set position to anInteger as long as anInteger is within the bounds of the receiver’s contents. If it is not, cause an error.

reset
Set the receiver’s position to 0.

skip: anInteger
Set position to position+anInteger. A subclass might choose to be more helpful and select the minimum of self size and position+anInteger or maximum of 1 and position+anInteger for the repositioning.

skipTo: anObject
Position the receiver past the next occurrence of anObject. Answer true if anObject is found, false otherwise.

PositionableStream methods for 'private'

on: aCollection

positionError
Since I am not necessarily writable, it is up to my subclasses to override position: if expanding the collection is preferable to giving this error.
PositionableStream class methods for 'instance creation'  

**on: aCollection**

Answer a new instance of me, streaming over aCollection.

**on: aCollection from: firstIndex to: lastIndex**

Answer a new instance of me, streaming over a copy of aCollection from firstIndex to lastIndex.
Object
Stream
PositionambleStream
ReadStream

I am a reader of an indexable collection. I only define the message next.

ReadStream methods for 'accessing'

next
Answer with the next object in the Stream represented by the receiver. Fail if the collection of this stream is not an Array or a String. Fail if the stream is positioned at its end, or if the position is out of bounds in the collection. Optional. See Object documentation whatIsAPrimitive.

<primitive: 65>

nextPut: anObject

ReadStream methods for 'private'

on: aCollection from: firstIndex to: lastIndex

ReadStream class methods for 'instance creation'

on: aCollection from: firstIndex to: lastIndex
Answer with a new instance streaming over a copy of aCollection from firstIndex to lastIndex.
Object
  Stream
    PositionableStream
      WriteStream
I am a writer of an indexable collection. writeLimit marks the farthest that has been written into the collection, not the actual size of the collection.

WriteStream methods for 'accessing'

contents

nextPut: anObject
  Insert the argument at the next position in the Stream represented by the receiver. Fail if the collection of this stream is not an Array or a String. Fail if the stream is positioned at its end, or if the position is out of bounds in the collection. Fail if the argument is not of the right type for the collection. Optional. See Object documentation whatIsAPrimitive.

  <primitive: 66>

size

WriteStream methods for 'positioning'

position: anInteger
reset

WriteStream methods for 'printing'

print: anObject
  Have anObject print on the receiver.

store: anObject
  Have anObject print on me for rereading.
WriteStream methods for 'private'

on: aCollection

on: aCollection from: firstIndex to: lastIndex

pastEndPut: anObject

with: aCollection

WriteStream class methods for 'instance creation'

on: aCollection from: firstIndex to: lastIndex
  Answer with a new instance streaming over a copy of aCollection from firstIndex to lastIndex.

with: aCollection
  Answer a new instance of me streaming on aCollection. Assume that the collection is already full so the position and the limits are set to the end.
Object
Stream
PositionableStream
WriteStream
ReadWriteStream

I am a positionable stream into which we can both read and write.

ReadWriteStream methods for 'accessing'

contents
Answer with a copy of my collection from 1 to readLimit.

next
Return the next object in the Stream represented by the receiver. Fail if the collection of this stream is not an Array or a String. Fail if the stream is positioned at its end, or if the position is out of bounds in the collection. Optional. See Object documentation whatIsAPrimitive.

<primitive: 65>
The purpose of including class UndefinedObject in the system is to handle error messages.

UndefinedObject methods for initialize-release

release
   Nil release is a no-op.

UndefinedObject methods for copying

deepCopy
   Only one instance of UndefinedObject should ever be made, so answer with self.

shallowCopy
   Only one instance of UndefinedObject should ever be made, so answer with self.

UndefinedObject methods for printing

printOn: aStream

storeOn: aStream

UndefinedObject methods for testing

isNil
   Overrides method found in Object. Return true.

notNil
   Overrides method found in Object. Return false.
Object

Boolean
The class Boolean provides protocol for logical values it provide common behavior to true and false.

Boolean methods for logical operations

& aBoolean
Evaluating conjunction -- Evaluate the argument. Then answer true if both the receiver and the argument are true.

not
Negation-- answer true if the receiver is false, answer false if the receiver is true.

| aBoolean
Evaluating disjunction (OR) -- Evaluate the argument. Then answer true if either the receiver or the argument is true.

Boolean methods for controlling

and: alternativeBlock
Nonevaluating conjunction -- if the receiver is true, answer the value of the argument, alternativeBlock; otherwise answer false without evaluating the argument.

ifFalse: alternativeBlock
If the receiver is true (i.e., the condition is true), then the value is the true alternative, which is nil. ' Otherwise answer the result of evaluating the argument, alternativeBlock. Create an error if the receiver is nonBoolean.

ifTrue: alternativeBlock
If the receiver is false (i.e., the condition is false), then the value is the false alternative, which is nil. Otherwise answer the result of evaluating the argument, alternativeBlock. Create an error if the receiver is nonBoolean.
or: alternativeBlock
    Nonevaluating disjunction -- if the receiver is false, answer the value of the argument, alternativeBlock; otherwise answer true without evaluating the argument.

Boolean methods for copying

deepCopy
    Receiver has two concrete subclasses, True and False. Only one instance of each should be made, so return self.

shallowCopy
    Receiver has two concrete subclasses, True and False. Only one instance of each should be made, so return self.

Boolean methods for printing

storeOn: aStream
    Print True or False.
**Object**

**Boolean**

**False**

**False** is used to implement conditional transfer of control. It describe behavior for it’s sole instance, false.

False methods for logical operations

& **alternativeObject**

Evaluating conjunction -- answer false since receiver is false.

not

Negation -- answer true since the receiver is false.

| **aBoolean**

Evaluating disjunction (OR) -- answer with the argument, aBoolean.

False methods for controlling

and: **alternativeBlock**

Nonevaluating conjunction -- answer with false since the receiver is false.

**ifFalse: alternativeBlock**

Answer the value of alternativeBlock.

**ifTrue: alternativeBlock**

Since the condition is false, answer the value of the false alternative, which is nil.

or: **alternativeBlock**

Nonevaluating disjunction -- answer value of alternativeBlock.

False methods for printing

**printOn: aStream**

Print false.
Object
  Boolean
    True
**True** is used to implement conditional transfer of control. It
describe behavior for it's sole instance, True.

True methods for logical operations

& alternativeObject
  Evaluating conjunction -- answer alternativeObject
  since receiver is true.

not
  Negation--answer false since the receiver is true.

| aBoolean
  Evaluating disjunction (OR) -- answer true since
  the receiver is true.

True methods for controlling

and: alternativeBlock
  Nonevaluating conjunction -- answer the value of
  alternativeBlock since the receiver is true.

ifFalse: alternativeBlock
  Since the condition is true, the value is the true
  alternative, which is nil.

ifTrue: alternativeBlock
  Answer the value of alternativeBlock.

or: alternativeBlock
  Nonevaluating disjunction -- answer true since the
  receiver is true.

True methods for printing

printOn: aStream
  Print True.
Object
  Behavior

Behavior provides the minimum state necessary for compiling methods, and creating and running instances. Most objects are created as instances of the more fully supported subclass, Class, but Behavior is a good starting point for providing instance-specific behavior (as in Metaclass).

Behavior methods for 'initialize-release'

obsolete
  Invalidate and recycle local messages. Remove the receiver from its superclass' subclass list.

Behavior methods for 'accessing'

format
  Answer an Integer that encodes the kinds and numbers of variables of instances of the receiver.

Behavior methods for 'testing'

instSize
  Answer the number of named instance variables (as opposed to indexed variables) of the receiver.

isBits
  Answer whether the receiver contains just bits (not pointers).

isBytes
  Answer whether the receiver has 8-bit instance variables.

isPointers
  Answer whether the receiver contains just pointers (not bits).

isVariable
  Answer whether the receiver has a variable (indexable) part.
isWords
Answer whether the receiver has 16-bit instance variables.

Behavior methods for ‘copying’

**copy**
Make a copy of the receiver without a list of subclasses.

Behavior methods for ‘printing’

**printOn:** aStream

Behavior methods for ‘creating class hierarchy’

**addSubclass:** aSubclass
Make the argument, aSubclass, be one of the subclasses of the receiver.

**removeSubclass:** aSubclass
If the argument, aSubclass, is one of the receiver’s subclasses, remove it.

**superclass:** aClass
Change the receiver’s superclass to be aClass.

Behavior methods for ‘creating method dictionary’

**addSelector:** selector **withMethod:** compiledMethod
Add the message selector with the corresponding compiled method to the receiver’s method dictionary.

**addSelectorUnchecked:** selector **withMethod:** compiledMethod
Add the message selector with the corresponding compiled method to the receiver’s method dictionary. Do not check for effect on (multiple) inheritance.

**methodDictionary:** aDictionary
Store the argument, aDictionary, as the method dictionary of the receiver.

**removeSelector:** selector
Assuming that the message selector is in the receiver’s method dictionary, remove it. If the selector is not in the method dictionary, create an error notification.

**removeSelectorUnchecked: selector**
Assuming that the message selector is in the receiver’s method dictionary, remove it. If the selector is not in the method dictionary, create an error notification. Do not check for effect on (multiple) inheritance.

**tryCopyingCodeFor: selector**
Check if ‘selector’ is compound, and if so, try to copy down the appropriate code.

Behavior methods for ‘instance creation’

**basicNew**
Answer a new instance of the receiver (which is a class) with no indexable variables. Fail if the class is indexable.

<primitive: 70>

**basicNew: anInteger**
Answer a new instance of the receiver (which is a class) with the number of indexable variables specified by the argument, anInteger. Fail if the class is not indexable or if the argument is not a positive Integer. Essential. See Object documentation whatIsAPrimitive.

<primitive: 71>

**new**
Answer a new instance of the receiver (which is a class) with no indexable variables. Fail if the class is indexable.

<primitive: 70>

**new: anInteger**
Answer a new instance of the receiver (which is a class) with the number of indexable variables specified by the argument, anInteger. Fail if the class is not indexable or if the argument is not a positive Integer.

<primitive: 71>
Behavior methods for 'accessing class hierarchy'

**allDynamicSuperclasses**
Answer an OrderedCollection of the receiver and the receiver's ancestor's dynamic superclasses; ordered with immediate superclass first.

**allSubclasses**
Answer an OrderedCollection of the receiver's subclasses and the receiver's ancestor's subclasses in breadth-first order, with the immediate subclasses first.

**allSuperclasses**
Answer an OrderedCollection of the receiver's superclasses and the receiver's ancestor's superclasses in breadth-first order, with the immediate superclasses first.

**allSuperclassesInto: orderedCollection**
Add all my superclasses to orderedCollection if not already there. Use breadth-first order.

**hasMultipleSuperclasses**

**subclasses**
Answer the receiver's subclasses. Return a copy so that callers who add or delete subclasses won't get confused.

**superclass**
Answer the receiver's superclass. Only returns the first one - use 'superclasses' to find them all.

**superclasses**
Answer with an array of all the receiver's superclasses. superclass == nil ifTrue: [^#()].

**withAllSubclasses**
Answer an OrderedCollection of subclasses including this class in breadth first order.

**withAllSuperclasses**
Answer an OrderedCollection of superclasses including this class in breadth first order.

Behavior methods for 'accessing method dictionary'

89
allSelectors
Answer a set of all the message selectors that instances of the receiver can understand.

compiledMethodAt: selector
Answer the compiled method associated with the message selector in the receiver’s method dictionary. If the selector is not in the dictionary, create an error notification.

methodDescriptionAt: selector
Return a method description for the method for 'selector'

selectorAtMethod: method setClass: classResultBlock
Answer both the message selector associated with the compiled method and the class in which that selector is defined.

selectors
Answer a Set of all the message selectors specified in the receiver’s method dictionary.

sourceCodeAt: messageSelector
Answer the string corresponding to the source code for the argument.

sourceCodeForMethod: method at: messageSelector
Answer the string corresponding to the source code for the argument.

sourceMethodAt: selector
Answer the paragraph corresponding to the source code for the argument.

superMethodDescriptionAt: selector
Return a method description for the method for 'selector' inherited from my superclasses.

Behavior methods for 'accessing instances and variables'

allClassVarNames
Answer a Set of the names of the receiver’s and the receiver’s ancestor’s class variables.

allInstances
Answer a collection of all instances of this class.

allInstVarNames
Answer an Array of the names of the receiver's instance variables.

**allSharedPools**

Answer a Set of the pools, dictionaries, that the receiver and the receiver's ancestors share. Subclasses, such as class Class, override this message.

**allVarNamesSelect: selectBlock**

Answer a collection of all the static variable names defined for the receiver which satisfy the condition in selectBlock. Test class and pool variables, including superclass variables. Also include global variables.

**classVarNames**

Answer a Set of the receiver's class variable names. Since the receiver does not retain knowledge of class variables, the method fakes it by creating an empty set.

**instVarNames**

Answer an Array of the instance variable names. Behaviors must make up fake local instance variable names because Behaviors have instance variables for the purpose of compiling methods, but these are not named instance variables.

**sharedPools**

Answer a Set of the pools, dictionaries, that the receiver shares. Since the receiver does not retain knowledge of pool dictionaries, the method fakes it by creating an empty array. Subclasses, such as class Class, override this message.

**someInstance**

Answer the first instance of this receiver. See Object nextInstance. Fails if there are none. `<primitive: 77>`

Behavior methods for 'testing class hierarchy'

**inheritsFrom: aClass**

Answer whether the argument, aClass, is on the receiver's superclass chain.
Behavior methods for 'testing method dictionary'

**canUnderstand: selector**

Answer true if the receiver can respond to the message whose selector is the argument, false otherwise. The selector can be in the method dictionary of the receiver’s class or any of its superclasses.

**hasMethods**

Answer whether the receiver has any methods in its method dictionary.

**includesSelector: aSymbol**

Answer whether the message whose selector is the argument is in the method dictionary of the receiver’s class.

**scopeHas: varName ifTrue: assocBlock**

Look up varName in this class, its superclasses, and Smalltalk. If it is there, pass the association to assocBlock, and answer true; else answer false.

**whichClassIncludesSelector: aSymbol**

Answer the class on the receiver’s superclass chain where the argument, aSymbol (a message selector), will be found.

**whichSelectorsAccess: instVarName**

Answer a set of selectors whose methods access the argument, instVarName, as a named instance variable.

**whichSelectorsReferTo: literal**

Answer a set of selectors whose methods access the argument as a literal.

**whichSelectorsReferTo: literal special: specialFlag byte: specialByte**

Answer a collection of selectors whose methods access the argument as a literal.

Behavior methods for 'compiling'

**compile: code notifying: requestor trailer: bytes**
Compile the argument, code, as source code in the context of the receiver. Use the default fail code [^nil]. Does not save source code. The second argument, requestor, is to be notified if an error occurs. The argument code is either a string or an object that converts to a string or a PositionableStream on an object that converts to a string. The third argument, bytes, is a trailer, that is, an array of three bytes that should be added to the end of the compiled method. These point to the location of the source code (on a file).

compile: code notifying: requestor trailer: bytes ifFail: failBlock

Compile the argument, code, as source code in the context of the receiver and install the result in the receiver’s method dictionary. The argument requestor is to be notified if an error occurs. The argument code is either a string or an object that converts to a string or a PositionableStream on an object that converts to a string. The trailer is an array of three bytes that should be added to the end of the compiled method. These point to the location of the source code (on a file). This method does not save the source code. Evaluate the failBlock if the compilation does not succeed.

compileAllFrom: oldClass

Compile all the methods in oldClass’s method dictionary. See recompile:from: regarding oldClass, which is normally just self.

compilerClass

Return a compiler class appropriate for source methods of this class.

compileUnchecked: code

Compile the argument, code, and install the result in the receiver’s method dictionary. Do not check for possible effect on inheritance, since that’s what this is doing.

decompile: selector
Find the compiled code associated with the argument, selector, as a message selector in the receiver’s method dictionary and decompile it. Answer the resulting source code as a string. Create an error if the selector is not in the receiver’s method dictionary.

decompilerClass
Return a decompiler class appropriate for compiled methods of this class.

parserClass
Return a parser class to use for parsing methods in this class.

poolHas: varName ifTrue: assocBlock
Behaviors have no pools

recompile: selector

recompile: selector from: oldClass
Recompile the method associated with selector in the receiver’s method dictionary. Take care not to write out any new source code - just generate new bytes. oldClass may differ from self in order to decompile right (if sourceFiles == nil) when adding or removing fields of a class.

Behavior methods for ‘enumerating’

allAccessesTo: instVarName
Return a list of all methods in my hierarchy that refer to the named instance variable.

allCallsOn: aLiteral
Answer a SortedCollection of all the methods that call on aLiteral.

allInstancesDo: aBlock
Evaluate the argument, aBlock, for each of the current instances of the receiver.

allSubclassesDo: aBlock
Evaluate the argument, aBlock, for each of the receiver’s subclasses.

browseAllCallsOn: aSymbol
Create and schedule a message browser on each method that calls on aSymbol. For example, Number browseAllCallsOn: #/.

crossReference
Answer an array of arrays of size 2 whose first element is a message selector in the receiver’s method dictionary and whose second element is a set of all message selectors in the method dictionary whose methods send a message with that selector. Subclasses are not included.

Behavior methods for ‘private’

accumulateInstVarNames: names traversedClasses: classSet
accumulat.instance variable names in ‘names’. Do this in depth-first, left-to-right order. This will give the ordering of instance variable names expected by the compiler and other parts of the system.

checkMethodFor: selector
copy method from superclass if necessary. Answer true if no conflict detected

defaultSelectorForMethod: aMethod
Given a method, invent an appropriate selector, that is, one that will parse with the correct number of arguments.

flushCache
Tell the interpreter to remove the contents of its method lookup cache, if it has one.

<primitive: 89>

format: nInstVars variable: isVar words: isWords pointers: isPointers
Set the format for the receiver (a Class).

printSubclassesOn: aStream callingSuperclass: whichSuper level: level

95
As part of the algorithm for printing a description of the receiver, print the subclass on the file stream, aStream, indenting level times.

removeClass: aClass selector: selector in: aDictionary

removeFromInheritanceTable: table
  I have been deleted -- remove me from the given inheritance table

removeFromInheritanceTables
  I have been deleted. Remove me from multiple inheritance tables.

unmovedVarsFrom: sup
  Answer with an Array with true for fields with the same offset in this class as in super

updateInheritanceTable: table oldSelf: oldSelf
  I have replaced an old behavior or class. Update the given multiple inheritance table

updateInheritanceTables: oldSelf
  I have replaced an old behavior or class. Update the multiple inheritance tables.

Behavior class methods for 'initialization'

init
  Behavior init.
Object
  Behavior
    ClassDescription

ClassDescription adds a number of facilities to basic Behavior:
  - named instance variables
  - category organization for methods
  - the notion of a name of this class (implemented as subclass responsibility)
  - the maintenance of the Changes set, and logging changes on a file
  - most of the mechanism for fileOut

ClassDescription is an abstract class: its facilities are intended for inheritance by the two subclasses, Class and Metaclass.

ClassDescription methods for 'initialize-release'

obsolete
  Make the receiver obsolete.

subclassOf: newSuper oldClass: oldClass instanceVariableNames:
newInstVarString variable: v
words: w pointers: p ifBad: badBlock

Basic initialization message for creating classes using the information provided as arguments. Answer whether old instances will be invalidated.

updateInstancesFrom: oldClass
  Recreate any existing instances of the argument, oldClass, as instances of the receiver, which is a newly changed class. Permute variables as necessary.

validateFrom: oldClass in: environ instanceVariableNames:
invalidFields methods: invalidMethods

Recompile the receiver, a class, and redefine its subclasses if necessary.

ClassDescription methods for 'accessing'

name
  Answer a String that is the name of the receiver.
ClassDescription methods for 'copying'

**copy: sel from: class classified: cat**
Install the method associated with the first argument, sel, a message selector, found in the method dictionary of the second argument, class, as one of the receiver’s methods. Classify the message under the third argument, cat.

**copyAll: selArray from: class classified: cat**
Install all the methods found in the method dictionary of the second argument, class, as the receiver’s methods. Classify the messages under the third argument, cat.

ClassDescription methods for 'testing'

**isMeta**

ClassDescription methods for 'printing'

**classVariablesString**
Answer a string of my class variable names separated by spaces, in alphabetical order.

**definition**
Answer a string that defines the receiver.

**notNil**

**instanceVariablesString**
Answer a string of my instance variable names separated by spaces.

**printOn: aStream**

**sharedPoolsString**
Answer a string of my class variable names separated by spaces.

**storeOn: aStream**
Classes and Metaclasses have global names.

**superclassesString**
Answer a string of my superclass names separated by spaces.

ClassDescription methods for 'instance variables'

**instVarNames**
Answer an Array of the names of instance variables defined in the receiver.

**removeInstVarName: aString**
Remove the argument, aString, as one of the receiver’s instance variables.

ClassDescription methods for 'method dictionary'

**removeSelector: aSymbol**
Remove the message whose selector is aSymbol from the method dictionary of the receiver, if it is there. Answer nil otherwise.

ClassDescription methods for 'organization'

**category**
Answer the system organization category for the receiver.

**category: cat**
Categorize the receiver under the system category, cat, removing it from any previous categorization.

**organization**
Answer the instance of ClassOrganizer that represents the organization of the messages of the receiver.

ClassDescription methods for 'compiling'

**compile: code classified: heading**
Compile the argument, code, as source code in the context of the receiver and install the result in the receiver’s method dictionary under the classification indicated by the second argument, heading. nil is to be notified if an error occurs.
The argument code is either a string or an object that converts to a string or a PositionableStream on an object that converts to a string.

compile: code classified: heading notifying: requestor
Compile the argument, code, as source code in the context of the receiver and install the result in the receiver's method dictionary under the classification indicated by the second argument, heading. The third argument, requestor, is to be notified if an error occurs. The argument code is either a string or an object that converts to a string or a PositionableStream on an object that converts to a string.

compile: code notifying: requestor trailer: bytes ifFail: failBlock
Intercept this message in order to remember system changes.

ClassDescription methods for 'private'

errorCategoryName

kindOfSubclass
Answer a string that describes what kind of subclass the receiver is, i.e., variable, variable byte, variable word, nonVariable or regular subclass.
My instances describe the representation and behavior of objects. I add more comprehensive programming support facilities to the basic attributes of Behavior and the descriptive facilities of ClassDescription. An example is accessing shared (pool) variables.

Class methods for 'initialize-release'

declare: varString
Declare class variables common to all instances. Answer whether recompilation is advisable.

obsolete
Change the receiver to an obsolete class by changing its name to have the prefix -AnObsolete-, and nilling the fields of any instances.

sharing: poolString
Set up sharedPools. Answer whether recompilation is advisable.

superclass: sup methodDict: md format: ft name: nm organization: org instVarNames: nilOrArray classPool: pool sharedPools: poolSet
Answer an instance of me, a new class, using the arguments of the message as the needed information.

validateFrom: oldClass in: environ instanceVariableNames: invalidFields methods: invalidMethods
Recompile the receiver and redefine its subclasses if necessary.

Class methods for 'accessing'

classPool
Answer the dictionary of class variables.

name
Answer the name of the receiver.
Class methods for 'accessing class hierarchy'

hasMultipleSuperclasses

isObsolete
   Answer whether the receiver is an obsolete class.

Class methods for 'testing method dictionary'

hasMethods
   Answer a Boolean as to whether any methods are defined for the receiver (includes whether there are methods defined in the receiver's metaclass).

Class methods for 'copying'

copy

copyForValidation
   Make a copy of the receiver (a class) but do not install the created class as a new class in the system. This is used for creating a new version of the receiver in which the installation is deferred until all changes are successfully completed.

Class methods for 'instance variables'

addInstVarName: aString
   Add the argument, aString, as one of the receiver's instance variables.

removeInstVarName: aString
   Remove the argument, aString, as one of the receiver's instance variables.

Class methods for 'class variables'

addClassVarName: aString
   Add the argument, aString, as a class variable of the receiver.

allClassVarNames
Answer a Set of the names of the receiver’s class variables, including those defined in the superclasses of the receiver.

classVarNames
Answer a Set of the names of the class variables defined in the receiver.

initialize
Typically used for the initialization of class variables and metaclass instance variables. Does nothing, but may be overridden in Metaclasses.

removeClassVarName: aString
Remove the class variable whose name is the argument, aString, from the names defined in the receiver, a class. Create an error notification if aString is not a class variable or if it is still being used in the code of the class without being declared in a superclass.

Class methods for 'pool variables'

allSharedPools
Answer a Set of the pools the receiver shares, including those defined in the superclasses of the receiver.

sharedPools
Answer a Set of the pool dictionaries declared in the receiver.

Class methods for 'compiling'

compileAllFrom: otherClass

poolHas: varName ifTrue: assocBlock
Look up the first argument in the context of the receiver. If it is there, pass the association to assocBlock, and answer true, else answer false.

Class methods for 'subclass creation'
subclass: t instanceVariableNames: f classVariableNames: d poolDictionaries: s category: cat

This is the standard initialization message for creating a new class as a subclass of an existing class (the receiver).

subclass: t otherSupers: others instanceVariableNames: f classVariableNames: d category: cat

This is the standard initialization message for creating a new class as a subclass of an existing class (the receiver).

variableByteSubclass: t instanceVariableNames: f classVariableNames: d poolDictionaries: s category: cat

This is the standard initialization message for creating a new class as a subclass of an existing class (the receiver) in which the subclass is to have indexable byte-sized nonpointer variables.

variableSubclass: t instanceVariableNames: f classVariableNames: d poolDictionaries: s category: cat

This is the standard initialization message for creating a new class as a subclass of an existing class (the receiver) in which the subclass is to have indexable pointer variables.

variableWordSubclass: t instanceVariableNames: f classVariableNames: d poolDictionaries: s category: cat

This is the standard initialization message for creating a new class as a subclass of an existing class (the receiver) in which the subclass is to have indexable word-sized nonpointer variables.

Class class methods for 'instance creation'

getSuperclasses: superNames
Metaclasses add instance-specific behavior to various classes in the system. This typically includes messages for initializing class variables and instance creation messages particular to that class. There is only one instance of a metaclass, namely the class (thisClass) which is being described. A metaclass shares the class variables of its instance.

[Subtle] In general, the superclass hierarchy for metaclasses parallels that for classes. Thus,

    Integer superclass == Number, and
    Integer class superclass == Number class.

However there is a singularity at Object. Here the class hierarchy terminates, but the metaclass hierarchy must wrap around to Class, since ALL metaclasses are subclasses of Class. Thus,

    Object superclass == nil, and
    Object class superclass == Class.'

Metaclass methods for 'initialize-release'

instanceVariableNames: instVarString
    Declare additional variables for my instances.

ewNamed: aSymbol
    Answer a new instance of me whose name is the argument, aSymbol.

newNamed: aSymbol otherSupers: others
    Answer a new instance of me whose name is the argument, aSymbol.

obsolete
    Invalidate and recycle local messages. Remove the receiver from its superclass' subclass list.

subclassOf: superMeta
    Change the receiver to be a subclass of the argument, superMeta, a metaclass. Reset the receiver's method dictionary and properties.

superclass: superMeta
    Change the receiver's superclass to be the argument, superMeta, a metaclass.
Metaclass methods for 'accessing'

name
Answer a String that is the name of the receiver, either Metaclass or the name of the receiver’s class followed by the ' class'.

Metaclass methods for 'testing'

isMeta

isObsolete
Answer whether the receiver is an obsolete metaclass.

Metaclass methods for 'copying'

copy
Make a copy of the receiver without a list of subclasses. Share the reference to the sole instance.

copyForValidation
Special copy for ClassDescription validateFrom:in:fields:methods:. Answer a copy of the receiver without the subclasses.

Metaclass methods for 'instance creation'

new
The receiver can only have one instance. Create it or complain that one already exists.

Metaclass methods for 'instance variables'

addInstVarName: aString
Add the argument, aString, as one of the receiver’s instance variables.

removeInstVarName: aString
Remove the argument, aString, as one of the receiver’s instance variables.
Metaclasse methods for 'class variables'

addClassVarName: aString

classPool
  Answer the dictionary of class variables.

Metaclasse methods for 'class hierarchy'

instHasMultipleSuperclasses

name: newName inEnvironment: environ subclassOf: sup and:
  others instanceVariableNames: instVarString variable: v words:
  w pointers: p classVariableNames: classVarString
  poolDictionaries: poolString category: categoryName comment:
  commentString changed: changed

Create a new metaclass from the information provided in the arguments. Create an error if the name does not begin with an uppercase letter or if a class of the same name already exists.

name: newName inEnvironment: environ subclassOf: sup
instanceVariableNames: instVarString variable: v words: w
pointers: p classVariableNames: classVarString
poolDictionaries: poolString
category: categoryName comment: commentString changed: changed

Create a new metaclass from the information provided in the arguments. Create an error if the name does not begin with an uppercase letter or if a class of the same name already exists.

subclasses
  Answer the receiver's subclasses.

Metaclasse methods for 'compiling'

scopeHas: name ifTrue: assocBlock

Metaclasse methods for 'printing'
definition
Answer with a string that defines me

Metaclass class methods for 'instance creation'

subclassOf: superMeta
Answer a metaclass that is a subclass of metaclass superMeta.
Object

**InstructionStream**

Instance Variables: sender <Context>, sometimes <CompiledMethod> pc <Integer> pointing into my method As a superclass of contexts, I store the return pointer in sender, and the current position in my method in pc. This class by itself has the ability to interpret the byte-encoded Smalltalk instruction set and maintains a program counter (pc) for streaming through CompiledMethods. Contexts thus inherit all this capability and, for other users, sender can hold a method to be similarly interpreted. The unclean re-use of sender to hold the method was to avoid a trivial subclass for the stand-alone scanning function.

InstructionStream methods for ‘testing’

**willReturn**
Answer whether the next bytecode is a return.

**willSend**
Answer whether the next bytecode is a message-send.

**willStorePop**
Answer whether the next bytecode is a store-pop.

InstructionStream methods for ‘decoding’

**interpretJump**

**interpretNextInstructionFor: client**
Send to the argument, client, a message that specifies the type of the the next instruction.

InstructionStream methods for ‘scanning’

**addFieldlndexTo: set**
If this instruction is an instVar reference, add its index (offset+1) to set.

**addSelectorTo: set**
If this instruction is a send, add its selector to set.

**followingByte**
Answer the following bytecode.

**method**

Answer the compiled method that supplies the receiver's bytecodes.

**nextByte**

Answer the next bytecode.

**pc**

Answer the index of the next bytecode.

**scanFor:** **scanBlock**

Answer the index of the first bytecode for which scanBlock answer true when supplied with that bytecode.

InstructionStream methods for 'private'

**interpretExtension:** **offset in:** **method for:** **client**

**method:** **method pc:** **startpc**

InstructionStream class methods for 'class initialization'

**initialize**

Initialize an array of special constants returned by single-bytecode returns.

InstructionStream class methods for 'instance creation'

**on:** **method**

Answer a new InstructionStream on the argument, method.

InstructionStream initialize
Object

InstructionStream

ContextPart

Instance Variables: stackp <Integer> indicating the offset of the top of my temporary value stack To the instruction parsing ability of InstructionStream I add the actual semantics for execution. The execution state is stored in my subclasses indexable fields, which store temporary variables and a stack of values used in evaluating expressions. The actual semantics of execution can be found in my category simulator, which exactly parallels the operation of the Smalltalk machine itself.

ContextPart methods for 'accessing'

`home`

Answer the context in which the receiver was defined.

`method`

`receiver`

Answer the receiver of the message that created this context.

`sourceCode`

Answer the source form of the receiver’s method.

ContextPart methods for 'temporaries'

`tempAt: index`

Answer the value of the temporary variable whose index is the argument, index.

`tempAt: index put: value`

Store the argument, value, as the temporary variable whose index is the argument, index.

`tempNames`

Answer an OrderedCollection of the names of the receiver’s temporary variables, which are strings.

ContextPart methods for 'controlling'

`activateMethod: newMethod withArgs: args receiver: rcvr class: class`
Answer a new context initialized with the arguments.

pop

push: t1

top
Answer the top of the receiver’s stack.

ContextPart methods for ‘printing’

printOn: aStream

ContextPart methods for ‘implementation dependent accessing’

at: t1

at: index put: aValue
Put aValue at the specified location in the receiver unless the stack pointer is nil or the index is larger than the stack pointer.

basicAt: t1

basicAt: t1 put: t2

instVarAt: index put: aValue

primDecrementStackp
<primitive: 131>

primIncrementStackp
<primitive: 130>

stackp: aValue
Set the stack pointer to the argument by primitively incrementing or decrementing the stack pointer.

ContextPart methods for ‘copying’

copy
Answer a copy of the receiver, copy the instance variables and if the stack pointer is not nil, copy the indexable part.
ContextPart class methods for 'class initialization'

**initPrimitives**

The methods (from class Object) that are cached in tryPrimitiveMethods are used by the simulator to catch failures when simulating primitives.

ContextPart class methods for 'instance creation'

**basicNew: anInteger**

Answer a new instance of the receiver with the number of indexable variables specified by the argument, anInteger.

<primitive: 141>

**new: anInteger**

Answer a new instance of the receiver with the number of indexable variables specified by the argument, anInteger.

<primitive: 141>
My instances function similarly to instances of MethodContext, but they hold the dynamic state for execution of a block in Smalltalk. They access all temporary variables and the method sender via their home pointer, so that those values are effectively shared. Their indexable part is used to store their independent value stack during execution. My instance must hold onto its home in order to work. This can cause circularities if the home is also pointing (via a temp, perhaps) to the instance. In the rare event that this happens (as in SortedCollection sortBlock:) the message fixTemps will replace home with a copy of home, thus defeating the sharing of temps but, nonetheless, eliminating the circularity.

BlockContext methods for 'initialize-release'

home: t1 startpc: t2 nargs: t3

BlockContext methods for 'accessing'

fixTemps
Fix the values of the temporary variables used in the block that are ordinarily shared with the method in which the block is defined.

hasMethodReturn

home
Answer the context in which the receiver was defined.

method
Answer the compiled method in which the receiver was defined.

receiver

BlockContext methods for 'temporaries'

tempAt: index
tempAt: index put: value

BlockContext methods for 'evaluating'

value
Evaluate the block represented by the receiver. Fail if the block expects any arguments or if the block is already being executed. Optional. No Lookup. See Object documentation whatIsAPrimitive.

<primitive: 81>

value: arg
Evaluate the block represented by the receiver. Fail if the block expects other than one argument or if the block is already being executed. Optional. No Lookup. See Object documentation whatIsAPrimitive.

<primitive: 81>

value: arg1 value: arg2
Evaluate the block represented by the receiver. Fail if the block expects other than two arguments or if the block is already being executed. Optional. See Object documentation whatIsAPrimitive.

<primitive: 81>

value: arg1 value: arg2 value: arg3
Evaluate the block represented by the receiver. Fail if the block expects other than three arguments or if the block is already being executed. Optional. See Object documentation whatIsAPrimitive.

<primitive: 81>

valueWithArguments: anArray
Evaluate the block represented by the receiver. The argument is an Array whose elements are the arguments for the block. Fail if the length of the Array is not the same as the the number of arguments that the block was expecting. Fail if the block is already being executed. Essential. See Object documentation whatIsAPrimitive.

<primitive: 82>

BlockContext methods for 'controlling'
whileFalse
Evaluate the receiver once and then repeatedly as long as the value returned by the evaluation is false.

whileFalse: aBlock
Evaluate the argument, aBlock, as long as the value of the receiver is false. Ordinarily compiled in-line. But could also be done in Smalltalk.

whileTrue
Evaluate the receiver once and then repeatedly as long as the value returned by the evaluation is true.

whileTrue: aBlock
Evaluate the argument, aBlock, as long as the value of the receiver is true. Ordinarily compiled in-line. But could also be done in Smalltalk.

BlockContext methods for 'printing'

printOn: aStream

BlockContext methods for 'private'

valueError
ClassEditor provides the capabilities for View and Edit classes.

editClass:aClass

EditClass(one argument). Let the user make changes to classes through an editor. Exit the editor implies reparsed and inclusion of the class description.

<primitive: 210>

viewClass:aClass

ViewClass. Let the user make changes or view the class description. Exit the editor does not implies inclusion.

<primitive: 211>

saveaClass:aClass file:filename
Save aClass as a disk file.

readClass:aClass file:filename
Read a class description from a disk file

saveEditorcontent: filename
Save the editor contents as a disk file.
Object

File *
A Class File is included to bring support to external files.
Every read returns a single line as a String.

open: filename for: 'r', 'w'
Open the indicated file for reading or writing.
<primitive: 200>

read: aString from: filename
Return the next line from the file.
<primitive: 201>

write: aString into: filename
Write aString into the file filename.
<primitive: 202>

size
Return the size of the file.
<primitive: 203>

eof
answer if end of file.
III. Construction of new classes

Every object in the system is an instance of a class. Programming in smalltalk consists of extending the existing class library. In this section we explain how one can create new classes that can expand the functionality of the system.

In our system we can create new classes in the same way as in Smalltalk-80, with the only difference that we are not using the same user interface. Every class is a subclass of another class with the exception of class Object. This means that in reality what we are doing is creating subclasses of already constructed classes. Here we want to describe the creation of new classes in a syntactical way.

There are four different kinds of messages for creating subclasses in the system. The messages are specified in class Class. The messages are:

subclass: t instanceVariableNames: f classVariableNames: d poolDictionaries: s category: cat

This is the standard initialization message for creating a new class as a subclass of an existing class (the receiver).

variableByteSubclass: t instanceVariableNames: f classVariableNames: d poolDictionaries: s category: cat

This is the standard initialization message for creating a new class as a subclass of an existing class (the receiver) in which the subclass is to have indexable byte-sized nonpointer variables.
variableSubclass: t  instanceVariableNames: f
classVariableNames: d  poolDictionaries: s  category: cat
This is the standard initialization message for creating a new class as a subclass of an existing class (the receiver) in which the subclass is to have indexable pointer variables.

variableWordSubclass: t  instanceVariableNames: f
classVariableNames: d  poolDictionaries: s  category: cat
This is the standard initialization message for creating a new class as a subclass of an existing class (the receiver) in which the subclass is to have indexable word-sized nonpointer variables.

The programmer can use the one that is best for its implementation. For instance if we want to inherit from the class Object and we want to create a new class Addition the message will be:

Object subclass: #Addition
  instanceVariableNames:''
classVariableNames:''
  poolDictionaries:
category:'Example'
After this message the programmer can add different methods to solve the specific problem. How we incorporate new classes into the system is the topic of the next section. For more information about how to create new classes see (Goldberg & Robson, 1983).
IV. A new user interface

In Smalltalk-80 programmers add new classes and test the behavior of them using the Smalltalk-80 programming environment. This environment is composed of Views, Browsers, Testing, and Error reporting. The following is a description of how the user interacts with the programming environment Goldberg & Robson (1983, p. 292):

"A user and the Smalltalk-80 programming environment interact through a bitmap display screen, a keyboard, and a pointing device. The display is used to present graphical and textual views of information to the user. The keyboard is used to present textual information to the system. The pointing device is used to select information on the display screen. Smalltalk-80 uses an indirect pointing device called a mouse. A cursor on the screen shows the location currently being pointed to by the mouse. The cursor is moved by moving the mouse over a flat surface. The mouse has three buttons, which are used to make different kinds of selection."

It is not our intention to describe the complete Smalltalk-80 programming environment in this section. More information can be found about the Smalltalk-80 programming environment in "The Orange Book" by Goldberg. (Goldberg, 1984).

A. Interactive Editor design preconditions

Our system does not support the Smalltalk-80 user interface. This implies that an alternate user interface must be developed in order to provide a medium of interaction with the system. The following are the preconditions for the design of this new user interface:
1- The system assumes nothing more than ASCII terminals, consequently no pointing device will be used. The system will use the complete screen.

2- The system has to be interactive. This means that the user will be able to write expressions directly on the screen and the system will respond with the result of the evaluation of the expression.

3- The system most be able to include class descriptions from disk files or interactively. The files most be in ASCII.

4- Interaction with the system will be through commands; these commands tell the system what to do with the classes.

B. Beginning a LehighTalk run session.

In this section we describe the components of the new user interface through hypothetical examples. The material in this session covers four activities that are fundamental to using our system:

1- Loading and quitting the LehighTalk system.
2- Creating and viewing Classes.
3- Saving Classes specifications and Smalltalk statements.
4- Evaluating Smalltalk statements.
To begin a run session, turn on the computer, the operating system is loaded into memory and begins execution. If your computer has its own hard disk, the operating system will be on drive C and will issue the prompt

C>

to inform you that it is ready. Assuming that the computer is at the directory in which our system is stored, at the dos prompt C> the user should enter the command

C>LehighTalk

*LehighTalk* typed at the Dos prompt will start the execution of our version of smalltalk. Each time the user loads the system into the computer’s memory, LehighTalk displays a new prompt ">" indicating that is ready for execute commands or Smalltalk Statements.

Ex. 1.

C>LehighTalk

LehighTalk

>-Quit

C>
To execute a LehighTalk command the user needs to write the command name after the LehighTalk prompt >.
All the commands are preceded by the character " ~ "; this tells the system that the word after the character " ~ " is a command and not a Smalltalk statement. In all cases when the character " ~ " appears at the beginning of the line the system will check if the word following the character is a command. If it's not a command it will be parsed as an expression. A list of commands and examples follow:

~Quit. Will terminate the execution of the LehighTalk (see example 1 above).

~IE ["filename",Class, clear]
The letters IE stand for Interactive Editor.
The ~IE command provides the user with an editor for making changes to the system. The system will stop during the editing session. After completion of changes the file or class that is been edited will be parsed. If the class or classes in the file are syntactically correct new instances of the classes will be added to the system. If it is a group of Smalltalk statements they will be ready for execution. If the command is invoked without the parameters this will open the editor with its actual contents. The "clear" parameter will erase the contents of the editor. To exit the editor press Esc.
Ex. 2.  

C>LehighTalk
LehighTalk
>~IE "test"

3+4
10-5

>~IE clear
>~IE

>~IE Magnitude

This command call a file named test into the editor. In this case test contain Smalltalk statements.

This simbolizes the editor working. The user can make changes or simply Exit the editor mode by pressing <Esc>

After the execution of the clear command the contents of the editor is erased.

This command calls class Magnitude from the system for editing purposes.

Object subclass:#Magnitude
  instanceVariableNames:'
  classVariableNames:'
  poolDictionaries:'
  category:'Numeric-Magnitudes'

Magnitude methodsFor:'comparing'

< aMagnitude

>

125
The interactive editor can be used for creation of new classes the following example illustrate this capability.

Ex. 3.

C>LehighTalk
LehighTalk
>~IE

Object subclass:#Test
  instanceVariableNames:"
  classVariableNames:"
  poolDictionaries:"
  category:'Test-Test'

At this time we can write a class description. Upon leaving the editor the class will be parsed and included in the system.

~s [ClassName] "filename"

Save a description of a class called ClassName or the content of the editor (if className is not provided) on a disk file named "filename".

Ex. 4.

>~s Bank "Bank.st"
Save class Bank in a disk file called Bank.st"

>~s "test"

When the command is invoked without the ClassName parameter the system will save the contents of the editor. If the editor has a class description it will save the class description on file. If the editor contains a group of Smalltalk statements those will be saved instead.

~LC

126
List the classes included in the system.

Ex. 5.

\texttt{~LC}

Object, Magnitude, Number,

ClassDescription, Class, Metaclass

\texttt{~run}

This command will execute the content of the editor as a group of smalltalk statements.

Ex. 6.

\texttt{C>LehighTalk}
\texttt{LehighTalk}
\texttt{~IE "test"}

\texttt{3+4}
\texttt{10-5}

\texttt{~run}
\texttt{3+4}
\texttt{7}
\texttt{10-5}
\texttt{5}
\texttt{~Quit}
\texttt{C>}

127
The last part of this section is devoted to the evaluation of Smalltalk statements in our system. We can evaluate Smalltalk statements interactively in two ways:

1- We can write the Smalltalk statement directly after the system prompt followed by <Enter>. The system responds by evaluating the statement and typing the result.

Ex. 7.
C>LehighTalk
LehighTalk
>7-3
4
>

2- Use command -run to process a group of Smalltalk statement.

The next page illustrates the use of this alternative (Ex. 9) together with other examples.
Sample run session
Ex. 8.
   C>LehighTalk
   LehighTalk >
   >1 + 9
   10
   >Quit
   C>

Ex. 9.
   C>LehighTalk
   LehighTalk >
   ~>IE "addition"

   3+4
   10-5

   ~>run
   3+4
   7
   10-5
   5
   >
   ~>IE

   3+4
   10-5
V. Conclusion

First we summarize the objectives of this thesis:

1- Our primary goal in this thesis is to provide the foundations for the creation of a simplified subset of the Smalltalk-80 language. This subset will incorporate many of the object-oriented features of Smalltalk-80 such as Objects, Messages, Classes, Subclassing, and Inheritance.

2- The second goal is to construct this subset with as amount of primitives, classes, and methods as possible. Our main concern is to minimize the size of the language without losing the concepts and spirit of Object Oriented Programming.

3- This system should support a language that is as close as possible to Smalltalk-80.

4- Learn about the Smalltalk-80 system, in particular the primitives and their relation to the classes.

Our main contribution to the first goal was the reduction of the virtual image and the creation of a simplified user interface. We did not attempt to reduce the virtual machine, because this was beyond the scope of this work. In the
reduction of the virtual image we did not exclude in any way parts that implement characteristics of the object oriented philosophy like Objects, Messages, Classes, Subclassing and Inheritance.

In trying to attain our second goal we eliminated many classes. Most of them were related to the Smalltalk-80 programming environment. The other classes were related to file manipulation, processes, numerical values, magnitude and collections. We reduced the number of classes in the system from 78 to 37. Two of these 37 classes are new in our system, File and ClassEditor. They were created to replace the file support and the user interface in Smalltalk.

The reduction of methods inside the preselected classes were significant. Originally in the group of 35 preselected classes there were around 1089 methods, we reduced them to 679. Forty one system primitives are used in our system. According to (Goldberg & Robson, 1983, p. 612-616) there are 127 Smalltalk primitives. The reduction in the number of primitives comes from the fact that there are fewer classes in our system and consequently less functionality.

We met our third goal by not changing the Smalltalk-80 syntax as described by (Goldberg & Robson, 1983).
We met our final goal by studying all the primitives related to the classes in the system. We know exactly in which class and method a primitive is used and for what purpose.

In general we conclude that we achieved a significant part of our primary goals in the construction of a new system. We know that the system is quite large and incomplete but we hope that soon we will see the first evaluation of 3+4 in our own system.
Bibliography


Appendices
Appendix 1 Smalltalk Class Description

This is the list of all the classes. Underlines imply that the class and its methods together with the description is included in the system, indentation is used to imply subclassing.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td></td>
</tr>
<tr>
<td>Magnitude</td>
<td></td>
</tr>
<tr>
<td>Character</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td></td>
</tr>
<tr>
<td>Number</td>
<td></td>
</tr>
<tr>
<td>Float</td>
<td></td>
</tr>
<tr>
<td>Fraction</td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>LargeNegativeInteger</td>
<td></td>
</tr>
<tr>
<td>LargePositiveInteger</td>
<td></td>
</tr>
<tr>
<td>SmallInteger</td>
<td></td>
</tr>
<tr>
<td>LookupKey</td>
<td></td>
</tr>
<tr>
<td>Association</td>
<td></td>
</tr>
<tr>
<td>Link</td>
<td></td>
</tr>
<tr>
<td>Process</td>
<td></td>
</tr>
<tr>
<td>Collection</td>
<td></td>
</tr>
<tr>
<td>SequenceableCollection</td>
<td></td>
</tr>
<tr>
<td>LinkedList</td>
<td></td>
</tr>
<tr>
<td>Semaphore</td>
<td></td>
</tr>
<tr>
<td>ArrayedCollection</td>
<td></td>
</tr>
<tr>
<td>Array</td>
<td></td>
</tr>
<tr>
<td>Bitmap</td>
<td></td>
</tr>
<tr>
<td>DisplayBitmap</td>
<td></td>
</tr>
<tr>
<td>RunArray</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td></td>
</tr>
<tr>
<td>Symbol</td>
<td></td>
</tr>
<tr>
<td>Text</td>
<td></td>
</tr>
<tr>
<td>ByteArray</td>
<td></td>
</tr>
<tr>
<td>Interval</td>
<td></td>
</tr>
<tr>
<td>OrderedCollection</td>
<td></td>
</tr>
<tr>
<td>SortedCollection</td>
<td></td>
</tr>
<tr>
<td>Bag</td>
<td></td>
</tr>
<tr>
<td>MappedCollection</td>
<td></td>
</tr>
<tr>
<td>Set</td>
<td></td>
</tr>
<tr>
<td>Dictionary</td>
<td></td>
</tr>
<tr>
<td>IdentityDictionary</td>
<td></td>
</tr>
<tr>
<td>Stream</td>
<td></td>
</tr>
<tr>
<td>PositionableStream</td>
<td></td>
</tr>
<tr>
<td>ReadStream</td>
<td></td>
</tr>
<tr>
<td>WriteStream</td>
<td></td>
</tr>
<tr>
<td>ReadWriteStream</td>
<td></td>
</tr>
<tr>
<td>ExternalFileStream</td>
<td></td>
</tr>
<tr>
<td>FileStream</td>
<td></td>
</tr>
<tr>
<td>Random</td>
<td></td>
</tr>
<tr>
<td>File</td>
<td></td>
</tr>
<tr>
<td>FileDirectory</td>
<td></td>
</tr>
<tr>
<td>FilePage</td>
<td></td>
</tr>
<tr>
<td>UndefinedObject</td>
<td></td>
</tr>
<tr>
<td>Boolean</td>
<td></td>
</tr>
<tr>
<td>False</td>
<td></td>
</tr>
<tr>
<td>True</td>
<td></td>
</tr>
<tr>
<td>ProcessorScheduler</td>
<td></td>
</tr>
<tr>
<td>Delay</td>
<td></td>
</tr>
<tr>
<td>SharedQueue</td>
<td></td>
</tr>
<tr>
<td>Behavior</td>
<td></td>
</tr>
<tr>
<td>ClassDescription</td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td></td>
</tr>
<tr>
<td>Metaclass</td>
<td></td>
</tr>
<tr>
<td>Point</td>
<td></td>
</tr>
<tr>
<td>Rectangle</td>
<td></td>
</tr>
<tr>
<td>BitBlt</td>
<td></td>
</tr>
<tr>
<td>CharacterScanner</td>
<td></td>
</tr>
<tr>
<td>Pen</td>
<td></td>
</tr>
<tr>
<td>DisplayObject</td>
<td></td>
</tr>
<tr>
<td>DisplayMedium</td>
<td></td>
</tr>
<tr>
<td>Form</td>
<td></td>
</tr>
<tr>
<td>Cursor</td>
<td></td>
</tr>
<tr>
<td>DisplayScreen</td>
<td></td>
</tr>
<tr>
<td>InfiniteForm</td>
<td></td>
</tr>
<tr>
<td>OpaqueForm</td>
<td></td>
</tr>
<tr>
<td>Path</td>
<td></td>
</tr>
<tr>
<td>Arc</td>
<td></td>
</tr>
<tr>
<td>Circle</td>
<td></td>
</tr>
<tr>
<td>Curve</td>
<td></td>
</tr>
<tr>
<td>Line</td>
<td></td>
</tr>
<tr>
<td>LinearFit</td>
<td></td>
</tr>
<tr>
<td>Spline</td>
<td></td>
</tr>
</tbody>
</table>
Appendix 2 Primitives

<table>
<thead>
<tr>
<th>Primitive Index</th>
<th>Class-Selector Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SmallInteger +</td>
</tr>
<tr>
<td>2</td>
<td>SmallInteger -</td>
</tr>
<tr>
<td>3</td>
<td>SmallInteger &lt;</td>
</tr>
<tr>
<td>7</td>
<td>SmallInteger =</td>
</tr>
<tr>
<td>14</td>
<td>SmallInteger bitAnd:</td>
</tr>
<tr>
<td>15</td>
<td>SmallInteger bitOr:</td>
</tr>
<tr>
<td>16</td>
<td>SmallInteger bitXor:</td>
</tr>
<tr>
<td>17</td>
<td>SmallInteger bitShift:</td>
</tr>
<tr>
<td>60</td>
<td>LargeNegativeInteger digitAt:</td>
</tr>
<tr>
<td>61</td>
<td>LargePositiveInteger digitAt:</td>
</tr>
<tr>
<td>62</td>
<td>Object at:</td>
</tr>
<tr>
<td>63</td>
<td>Object basicAt:</td>
</tr>
<tr>
<td>64</td>
<td>LargeNegativeInteger digitAt:put:</td>
</tr>
<tr>
<td>65</td>
<td>LargePositiveInteger digitAt:put:</td>
</tr>
<tr>
<td>66</td>
<td>Object basicAt:put</td>
</tr>
<tr>
<td>67</td>
<td>Object at:put:</td>
</tr>
<tr>
<td>68</td>
<td>ArrayedCollection size</td>
</tr>
<tr>
<td>69</td>
<td>LargeNegativeInteger digitLength</td>
</tr>
<tr>
<td>70</td>
<td>LargePositiveInteger digitLength</td>
</tr>
<tr>
<td>71</td>
<td>Object basicSize</td>
</tr>
<tr>
<td>72</td>
<td>Object size</td>
</tr>
<tr>
<td>73</td>
<td>String size</td>
</tr>
<tr>
<td>74</td>
<td>String at:</td>
</tr>
<tr>
<td>75</td>
<td>String basicAt:</td>
</tr>
<tr>
<td>76</td>
<td>String basicAt:put:</td>
</tr>
<tr>
<td>77</td>
<td>String at:put:</td>
</tr>
<tr>
<td>78</td>
<td>Behavior basic^New Behavior new</td>
</tr>
<tr>
<td>79</td>
<td>Interval class new</td>
</tr>
<tr>
<td>80</td>
<td>Behavior new:</td>
</tr>
<tr>
<td>81</td>
<td>Behaviour basicNew:</td>
</tr>
<tr>
<td>82</td>
<td>Object instVarAt:</td>
</tr>
<tr>
<td>83</td>
<td>Object instVarAt:put:</td>
</tr>
<tr>
<td>84</td>
<td>Object asOop</td>
</tr>
<tr>
<td>85</td>
<td>Object hash</td>
</tr>
<tr>
<td>86</td>
<td>Symbol hash</td>
</tr>
<tr>
<td>87</td>
<td>SmallInteger asObject</td>
</tr>
<tr>
<td>88</td>
<td>Behavior someInstance</td>
</tr>
<tr>
<td>89</td>
<td>Object nextInstance</td>
</tr>
<tr>
<td>90</td>
<td>BlockContext value:value:value:</td>
</tr>
<tr>
<td>91</td>
<td>BlockContext value:</td>
</tr>
<tr>
<td>92</td>
<td>BlockContext value:value:</td>
</tr>
<tr>
<td>93</td>
<td>BlockContext valueWithArguments:</td>
</tr>
<tr>
<td>94</td>
<td>Object perform:withArguments:</td>
</tr>
<tr>
<td>95</td>
<td>Behavior flushCache</td>
</tr>
<tr>
<td>96</td>
<td>Character =, Object ==</td>
</tr>
<tr>
<td>97</td>
<td></td>
</tr>
<tr>
<td>98</td>
<td></td>
</tr>
<tr>
<td>99</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
</tr>
<tr>
<td>106</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td></td>
</tr>
<tr>
<td>108</td>
<td></td>
</tr>
<tr>
<td>109</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
</tr>
</tbody>
</table>

136
Object class
SmallInteger asObjectNotFail

(two arguments) Open the named file, second argument for read or write.
(two arguments) return the next line from file.
(two arguments) write a line into a file.
return the size of the file in characters.

EditClass(one argument). Let the user make changes to classes through an editor. Exit the editor implies reparsed and inclusion of the class description.

ViewClass. Let the user make changes or view the class description. Exit the editor does not implies inclusion.
Biography

Edgardo Ortiz was born on May 11, 1964 in Manatí, Puerto Rico. His family is composed of four people. He is the youngest brother in his family. His parents are Sixto Ortiz and Maximina Torres. His older brother is Sixto Ortiz Jr.

Edgardo studied in Jaime A. Collazo Del Rio High School. After he graduated from high school in May, 1982, he studied for a B.S. in Computer Science from Inter American University of Puerto Rico and obtained his degree in May, 1987. After that he continued courses for an MBA in Industrial Management at the same institution. He graduated with his MBA in May, 1990 and during that time he was honored by The National Dean’s List 1989-90 Thirteenth Annual Edition.

Edgardo worked as computer Professor at Allied Schools in his hometown, Morovis, and then he worked as a computer systems consultant in the clothing industry. After that he worked as a MIS supervisor in a chemical company. Then he worked as an assistant manager in the retail industry.

At Present, Edgardo is pursuing graduate studies in the Electrical Engineering and Computer Science Department at Lehigh University, Bethlehem, Pennsylvania.