

Ctalk Tutorial

Object Oriented Programming with Ctalk

Robert Kiesling

Ctalk Tutorial Version 0.0.96a.

Copyright © 2007, 2008, Robert Kiesling

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of this license is included in the chapter entitled, “GNU Free Documentation License.” See Chapter 11 [GNU Free Documentation License], page 59.

Table of Contents

1	A Brief Introduction to Objects	1
1.1	Methods, Messages, and Receivers	1
1.2	Polymorphism and Overloading	2
1.3	Class and Instance Data	2
1.4	Inheritance	2
1.5	The Class Hierarchy	3
2	Hello, world! and Other Simple Programs ...	5
3	Basic Classes and Ctalk Statements	9
3.0.1	String Class	11
3.0.2	Array Class	12
3.0.2.1	Repeating Operations for Each Element of a Collection	13
3.0.3	Compound Statements	15
3.0.4	Variable Promotion and Type Conversion	15
3.0.5	Default Methods and Instance Variable Messages	16
4	File Input and Output	17
5	self and super	23
5.1	self as a Receiver	23
5.2	The super Keyword	23
5.3	Determining the Class of self	24
5.3.1	Method Dispatchers	24
5.4	Class of a Method Return Object	27
5.5	Determining the Class of Method Arguments	28
6	Collections	31
6.1	List Class	31
6.2	AssociativeArray Class	33
6.3	Collection Elements as Receivers	34
7	Defining Classes and Objects	35
7.1	Superclasses	35
7.2	Instance Variables	35
7.3	Class Variables	36
7.4	Class Initialization	36
7.5	require Keyword	38
7.6	Where Ctalk Looks for Class Libraries	39
7.7	Instance Variables and Method Overloading	39

8	Writing Methods	41
8.1	Overview of the Method Application Programming Interface ...	42
8.1.1	Primitive Methods	43
8.1.2	Instance and Class Variable Messages	43
8.2	Method Declarations	44
8.3	Using Methods in Simple Statements	44
8.4	The <code>value</code> Message and Interfacing with C	45
8.4.1	The <code>OBJECT</code> Type	46
8.4.2	Return Values	47
9	Panes and Graphics	49
9.1	Hello, world! in a Window	49
9.2	Pane and Stream Classes	49
9.3	InputEvent Class	50
9.4	Using Queued InputEvents	51
9.5	The Widget API	52
9.5.1	Complex Widgets and Sub-Panes	53
9.6	Serial Terminals	53
9.7	X11Pane Class	54
10	Debugging	57
10.0.1	Printing Call Sequences for Exceptions	57
10.0.2	Printing Objects	57
11	GNU Free Documentation License	59
12	Index	69

1 A Brief Introduction to Objects

In object oriented programming, *objects* are the pieces of information that contain both code and data. Look at the following examples. They implement the same operations in BASIC, C, and Ctalk.

```
10 let a = 2
20 let b = a + a
30 print b
```

```
int a, b;
```

```
a = 2;
b = a + a;
printf ("%d\n", b);
```

```
Integer new a;
Integer new b;
```

```
a = 2;
b = a + a;
stdoutStream writeStream b;
```

The examples look similar, but there is a difference in the object oriented example. When a program declares new objects, in this case **a** and **b**, Ctalk creates them as members of a *class*. In this example, **a** and **b** are members of the class **Integer**.

If this were as far as object oriented programming went, these declarations would function exactly the same as their C counterpart, `int a, b;`

However, classes belong to a library of classes, called a *class hierarchy*. Class hierarchies contain all of the routines of the language, and you can create your own classes. The class hierarchy is organized so that the most general classes are at the top of the hierarchy, and proceed to more specialized *subclasses*. Each class may have more than one subclass. Ctalk classes each have one *superclass*, except for class **Object**, the topmost class in the class hierarchy. More on class hierarchies in a moment.

1.1 Methods, Messages, and Receivers

You can think of a class hierarchy as a library. Unlike run-time libraries, however, classes contain both the specification of the objects and the source code of the routines or functions that operate with each class. These routines are called *methods*.

Look at this statement from the example above.

```
a = 2;
```

In this statement, the object **a** is called the *receiver* of the method. The method is the assignment operator, `=`. The number 2 is the *argument* to the method. When the program tells **a** to use the method `=`, the program sends the *message* `=` to the receiver, **a**.

1.2 Polymorphism and Overloading

Each class can have its own implementation of `=`. The class `String`, for example, can have its own `=` method, which would do exactly what you would expect.

```
String new myString;

myString = "Hello, world!";
```

The next line from the example above,

```
b = a + a;
```

contains two methods, `=` and `+`. The receiver of `=` is `b`, and the receiver of `+` is `a`. The argument of `=` is the complete expression, `a + a`.

Programs can do the same with character strings.

```
String new myString;

myString = "Hello, " + "world!";
```

This is a simple example of a characteristic of computer languages: *polymorphism*. That means different types of data have a common interface and are treated the same by the language. When the `=` and `+` methods work with different classes of objects, we say that the methods *overload* the operators.

1.3 Class and Instance Data

In the next line from the example above,

```
stdoutStream writeStream b;
```

the receiver, `stdoutStream`, receives the message `writeStream` with the argument `b`.

The receiver `stdoutStream` is a *class variable*. There is one `stdoutStream` for its class, `WriteFileStream`. `stdoutStream` represents the program's standard output. The argument `b`, however, is an *instance* of class `String`, and it contains an *instance variable, value*, which contains the character string itself. A program can create as many *instances of a class* as it needs.

Each instance of a class contains its own data. For `Integer` and `String` objects, the data is simply the *value* of the object, either `2` or `Hello, world!`

Class `WriteFileStream` objects, however, need to contain more data, like the permissions of the file stream, the position in the stream, the type of I/O device, and the memory address of the stream itself. This information is contained in *instance variables*. Each class defines its own set of instance variables, and each instance of a class has its own copy of them.

1.4 Inheritance

The methods that create objects, which are generally called `new`, have a special status. They are called *constructors*.

The following two statements look similar, but they are not.

```
String new myString;
WriteFileStream new myOutput;
```

Recall from the previous sections that the class hierarchy precedes from the more general objects at the top of the hierarchy, to the more specific subclasses. Also, instances of class `WriteFileStream` contain more instance data than many other classes of objects.

What actually occurs in the above statements is that `myString` is created like any other object. `myOutput`, however, uses the `new` method of its class, which first creates `myOutput` like any other object and then adds the additional instance variables that we mentioned above.

So in addition to the `value` instance variable, which the method `new` obtains from the `Object` class, the object `myOutput` also contains the instance variables it needs to manage writing to a file. This is called *inheritance*. We say that instances of class `WriteFileStream` inherit the `value` instance variable from the `Object` class. Instances of class `WriteFileStream` also inherit other instance variables, as described below.

Here is another example, which shows how classes inherit methods. Suppose you wanted to create a class that represented positive integers. You would accomplish this, most likely, by creating a subclass of `Integer`.

```
Integer class PositiveInteger;
```

Objects of your new class could use the `+` method of the `Integer` class. So you could write:

```
PositiveInteger new a;
PositiveInteger new b;
```

```
a = 2;
b = a + 2;
```

Ctalk, if it doesn't find a method for `+` in the `PositiveInteger` class, then uses the `+` method of class `Integer`.

If you wanted to subtract two `PositiveInteger` objects, however, you would need to implement the method `-` to insure that the result is also positive. The code within that method might look something like the following.

```
if ((b = (a - a)) < 0)
    warning ("Negative result for PositiveInteger.");
```

The later chapters describe the details of creating your own classes and methods.

1.5 The Class Hierarchy

Let's return to the constructor example above. You should note that the following statement does not actually *define* its instance variables.

```
WriteFileStream new myOutput;
```

This is because the instance variables necessary for reading from files (with class `ReadFileStream` objects) and writing to files (with class `WriteFileStream` objects) are similar. Instead, Ctalk defines them in a *superclass*, `FileStream`.

A section of the class hierarchy, showing the instance variables that each class defines, should help explain this organization.

Class	Instance Variables Defined
-----	-----

```

Object          <-----  value
Stream
  FileStream    <-----  streamMode
                <-----  streamDev
                <-----  streamRdev
                <-----  streamSize
                <-----  streamAtime
                <-----  streamMtime
                <-----  streamCtime
                <-----  streamPos
                <-----  streamErrno
                <-----  streamPath

  ReadFileStream
  WriteFileStream

```

When constructed, instances of `ReadFileStream` and `WriteFileStream` each have their own copies of the same set of instance variables.

The *Ctalk Reference Manual* describes the Ctalk class hierarchy, and the methods and variables, that each class defines.

Ctalk's language syntax and class library organization are based very closely on the Smalltalk language, one of the earliest truly object oriented languages. In addition, Ctalk contains additional features that allow programs to use C variables and functions. You can find much information about Smalltalk and its design in on-line and printed documents.

If you want to read further about C and object oriented programming languages in general, there are many books and tutorials, also on line and in print, that describe these subjects in detail.

2 Hello, world! and Other Simple Programs

Ctalk assumes that you are familiar with the C language, and that you have an understanding of basic object oriented programming concepts, which we described in the previous chapter.

This chapter presents a few simple programs that demonstrate how the Ctalk language works. Later chapters will describe the more advanced features of the language in greater detail.

Printing "Hello, world!"

To learn how to write and compile a Ctalk application, this chapter starts with the obligatory Hello, world! program.

Here is the Hello, world! program listing.

```
int main () {  
  
    String new helloObject;  
  
    helloObject = "\"Hello, world!\"";  
  
    printf ("%s\n", helloObject value);  
  
    exit (0);  
}
```

Save this listing in a file called `hello.c`, and then build it with the `ctcc` command.

```
$ ctcc hello.c -o hello
```

Then, you can run the program with this command, and the program will print the output.

```
$ ./hello  
"Hello, world!"
```

Printing Parts of a String

Here is another simple program, `ctpath.c`. You can find the source file in the `programs` subdirectory of the Ctalk source code distribution.

This program prints the directory path that you supply as an argument, one directory to a line.

```
int main (int argc, char **argv) {  
  
    String new path;  
    Array new paths;  
    Integer new nItems;  
    Integer new i;  
    Character new separator;  
  
    if (argc != 2) {  
        printf ("Usage: ctpath <path>\n");  
    }
```

```

        exit (1);
    }

    path = argv[1];

    separator = '/';

    nItems = path split separator, paths;

    for (i = 0; i < nItems; i = i + 1)
        printf ("%s\n", paths at i);

    exit (0);
}

```

Although there is a lot of code for initializing variables and checking the command line arguments, `ctpath` does the actual work in three lines of code.

```

nItems = path split separator, paths;

for (i = 0; i < nItems; i = i + 1)
    printf ("%s\n", paths at i);

```

The method `split` (class `String`) splits the string `path` at each occurrence of `separator` and places the result in the array, `paths`. The next two lines print each element of the array `paths` with the statement, `paths at i`.

As with the previous example, you can compile the program using the following commands.

```
$ ctcc ctpath.c -o ctpath
```

When you run the program, the output should look like this, depending on the directory path you provide on the command line.

```
$ ./ctpath /home/users/joe
home
users
joe

```

If you're not certain of how the arguments to `main`, `argc` and `argv`, function (they contain the command line, split into individual strings, and the number of command line strings), then you should go back and study the C language until you are comfortable with the language.

Printing the Time

Here is a program that uses a method to get the local time and store it in an array.

```

Array instanceMethod getTime (void) {

    Time new timeNow;
    Array new currentLocalTime;

    timeNow utcTime;
}

```

```
        currentLocalTime = timeNow localTime;

        self atPut 0, (currentLocalTime at 2);
        self atPut 1, (currentLocalTime at 1);
        self atPut 2, (currentLocalTime at 0);

        return NULL;
    }

    int main () {

        Array new clockTime;

        clockTime getTime;

        printf ("%02d:%02d:%02d\n", (clockTime at 0), (clockTime at 1), (clockTime at 2))
    }
```

In the method `getTime`, `self` refers to the method's receiver, `clockTime`, which is declared in `main`.

The `localTime` method (`Time` class), returns an `Array` that is filled in by the C library's call to `localtime(3)`. The *Ctalk Language Reference* describes how `localTime` returns time and date information.

Note that in `main`, the arguments to `printf` are enclosed in parentheses, so there is no ambiguity in evaluating the expressions in each argument.

3 Basic Classes and Ctalk Statements

Ctalk is designed to work with C. There are a number of classes that correspond directly to the basic C data types.

In Ctalk, you can use instances of the classes that are shown in the table below interchangeably with their C equivalents.

Ctalk Class	C Type
-----	-----
Array	char **, int **, ...
Character	char
Float	float, double
Integer	int, long int
LongInteger	long long int
String	char *

Creating and Working with Simple Objects

Like most object oriented languages, when you declare an object, you are actually creating it. As mentioned earlier, the methods that create objects are called *constructors*. As with most object oriented languages, you create an object with the message `new`.

```
Integer new myInt;
```

Objects of class `Integer` work exactly the same as C variables of type `int`. You can use expressions like the following, as though you were programming in C.

```
Integer new myInt;

myInt = 2;
printf ("%d\n", myInt + myInt);
```

With several exceptions, you can use all of the operators that C integers recognize. Postfix operators, like `+`, `-`, `*`, `/`, and so on, also work as methods.

However, prefix operators, like unary `!` and `~`, do not work. Instead, Ctalk provides equivalent methods for them, `invert` and `bitComp`.

This example shows how you would use the methods in C statements.

```
int i;                /* C variable. */
Integer new myInt;   /* Ctalk Object. */

i = 2;
myInt = 2;

/* These two statements are equivalent. */
printf ("%d\n", !i);
printf ("%d\n", myInt invert);

/* These two statements are also equivalent. */
printf ("%d\n", ~i);
printf ("%d\n", myInt bitComp);
```

The same is also true for instances of the `Character`, `LongInteger`, and `Float` classes. For instance, you can add and subtract two characters. This is sometimes useful for conversions, as in the following examples.

```
Character new receiver;
Character new operand;

receiver = 'A';
operand = ' ';

printf ("%c\n", receiver + operand);
```

The result is the lower case 'a'. This has the same effect as:

```
Character new receiver;
receiver = 'A';
printf ("%c\n", receiver toLower);
```

Or even:

```
printf ("%c\n", 'A' toLower);
```

To convert a character to uppercase, you can use either `-` or `toUpper`. Both are implemented by the `Character` class.

```
Character new receiver;
Character new operand;

receiver = 'a';
operand = ' ';

/* These two statements produce the same result. */
printf ("%c\n", receiver - operand);
printf ("%c\n", receiver toUpper);
```

For completeness, we should note that you can also achieve the same effect with the bit operators `&`, `|`, and `^`, although their application can be a bit more involved. If you're not certain how the operators would work, enter the ASCII value of a character in a programmer's calculator and convert it to base 2. Then notice the effect of changing different bits.

For now, we'll just show an example of changing a letter's case - from lower to upper case, or vice versa.

We do this by XOR'ing the fifth bit of the character value. (Note that 2^5 is 32, the ASCII value of a space (' ') character.)

Here is a simple example of the operation.

```
Character new upperCase;
Character new lowerCase;
Integer new toggleBit;

upperCase = 'Z';
lowerCase = 'z';
toggleBit = 100000b;
```

```
printf ("%c\n", upperCase ^ toggleBit);
printf ("%c\n", lowerCase ^ toggleBit);
```

3.0.1 String Class

As the introduction mentioned, Ctalk overloads many operators. This is especially obvious of class `String`.

This code shows how `String` class overloads some math operators.

```
String new s1;
String new s2;
String new helloString;

/* C's strcpy () would also work here. */
s1 = "Hello, ";
s2 = "world!";

/* And strcat () could perform the same task here. */
helloString = s1 + s2;

printf ("The value of \"helloString\" is \"%s\".\n", helloString);

s1 = "s1";
s2 = "s2";

/* Test whether String objects are equal without strcmp (). */
if (s1 == s2) {
    printf ("The strings s1 and s2 are equal.\n");
} else {
    printf ("The strings s1 and s2 are not equal.\n");
}
```

Briefly, the `String` instance methods `=`, `==`, and `+` function similarly to the C library functions `strcpy(3)`, `strcmp(3)`, and `strcat(3)`, respectively, although they are not identical.

Like C, however, instances of `String` and `Character` are not interchangeable. If you want to use an instance of class `Character` as a string, you must use the method `asString` (class `Magnitude`) to return a `String` version of the `Character` object.

Here is a brief example. The two `printf` statements produce the same output.

```
Character new myChar;

myChar = 'a';

printf ("%c\n", myChar);
printf ("%s\n", myChar asString);
```

`String` class implements several convenience methods. The `split` method, which we described in the previous chapter, splits a string along a separator character and puts each substring token in an array.

Here is the statement from the example in the previous chapter. See [ctpath.c], page 5.

```
nItems = path split separator, paths;
```

If the receiver, `path`, contains a string like `"/home/users/joe"`, and `separator`, a `Character` contains `'/'`, then `split` places the individual names in the `Array`, `paths`.

The operation looks something like this.

```

path    --->    "/home/users/joe"
              /   |   \
             /    |    \
split      /     |     \
          v      v      v
-----
paths    ---> |paths at 0|paths at 1|paths at 2|
              | "home"  | "users"  | "joe"   |
              -----

```

The expression, `paths at 0`, is equivalent to the `paths[0]` element of a C array, and so on, for the other elements of the array.

Another convenience method, `subString`, returns the part of the receiver string beginning at the first argument. The second argument is the length of the substring. Here is an example.

```
myString = "This is a string.";
mySubString = myString subString 5, 2;
```

3.0.2 Array Class

An `Array` in Ctalk is similar to arrays in every other programming language. Ctalk arrays are collections of objects, indexed sequentially from index 0 to the last element in the `Array`.

To add an element to an `Array` object, use the method `atPut`.

```

Array new myArray;
String new myString;

myString = "Hello, world!";

myArray atPut 0, myString;
```

The `atPut` method has two arguments, the index of the array element, and the object you want to store there.

To retrieve an `Array` element, use the method `at`. The `at` method takes one argument, the index of the array element.

Using the previous example, this statement retrieves the first element of `myArray` and prints it.

```
printf ("%s\n", myArray at 0);
```

You do not have to predeclare an array to a specific size. Objects of `Array` class can hold as many objects as necessary.

If you want to know how many elements an `Array` object contains, use the method `size`, which returns the number of elements as an `Integer` object.

```
printf ("myArray has %d elements.\n", myArray size);
```

If you add an object to an array at an index that already contains an object, `atPut` replaces the array element with the new object.

```
Array new myArray;
```

```
myArray atPut 0, "My";
myArray atPut 1, "name";
myArray atPut 2, "is";
myArray atPut 3, "Bill";
```

```
WriteFileStream classInit;
```

```
stdoutStream printOn "%s %s %s %s.\n", myArray at 0,
myArray at 1, myArray at 2, myArray at 3;
```

```
myArray atPut 3, "Joe";
```

```
stdoutStream printOn "%s %s %s %s.\n", myArray at 0, myArray at 1,
myArray at 2, myArray at 3;
```

The `WriteFileStream classInit` statement initializes the program's standard output. You need to include it (also run by the `WriteFileStream` method `new`) before printing to the console with Ctalk methods. Later chapters describe file input and output in greater detail.

3.0.2.1 Repeating Operations for Each Element of a Collection

Generally, however, if you want to repeat an operation on all members of an array, you can use the method `map`.

The `map` method can take the name of another method as its argument, and calls the method with each element of the array as its receiver.

Here is a more complete implementation of the example above, written using `map`.

```
Array instanceMethod printArrayElement (void) {
  printf ("%s ", self);
  methodReturnNULL;
}
```

```
int main () {
  Array new myArray;

  myArray atPut 0, "My";
  myArray atPut 1, "name";
  myArray atPut 2, "is";
  myArray atPut 3, "Bill";

  myArray map printArrayElement;
```

```

    printf ("\n");

    myArray atPut 3, "Joe";
    myArray map printArrayElement;
    printf ("\n");
}

```

In the `printArrayElement` method, `self` refers to each successive element of `myArray` (in `main`.) Each element of `myArray` is an instance of class `String`.

However, you should note that `Array` elements, like elements of any other collection, can be objects from any class, and C does not necessarily handle classes that correspond to complex data so easily.

In this case, it is relatively easy for the program to determine how to deal with a `String` object. Mapping over other collection types, `AssociativeArrays`, requires that methods examine receivers of unknown classes. See Chapter 6 [Collections], page 31.

If `Array` elements are instances of complex classes, or the elements represent complex C data types, then the method may need to determine the class of its receiver, which is discussed in the following chapters, and the method may need to translate objects from one class to another. See [Variable Promotion and Type Conversion], page 15.

The `map` method can also use a block of code as its argument, as in this example.

```

int main () {

    List new l;
    String new sPrefix;
    String new s;

    sPrefix = "This element is ";

    l push "l1";
    l push "l2";

    l map {
        s = sPrefix + self;
        printf ("%s\n", s);
    }
    exit(0);
}

```

You should note that the block shares the scope of the function or method where it occurs. Also, `self`, when it occurs inside a code block in this manner, refers to each successive element of the receiver collection (each member of `List l` in the example above). Argument blocks are the *only* place where `self` can occur within a C function. Otherwise, either Ctalk or the compiler will not be able to find a receiver that `self` refers to, and issues a warning or error message.

Currently, `map` (implemented in the `List`, `Array`, and `AssociativeArray` classes) is the only method that implements in-line execution of code. The methods use the `__ctalkInlineMethod` API function to actually perform the in-line call. Programs

that implement inline calls elsewhere can (and should) use this function. Refer to the `__ctalkInlineMethod` description in the *Ctalk Language Reference* for the gritty details.

3.0.3 Compound Statements

The result of evaluating one message can be the receiver of another message, as in the following example.

```
int main () {

    ReadFileStream new infileStream;
    Array new pathDirs;
    Integer new nDirs;
    SystemErrnoException new e;

    /* Substitute the path of the file here. */
    infileStream openOn "/path/name/here";
    if (e pending)
        e handle;

    nDirs = infileStream streamPath split '/', pathDirs;

    printf ("%i\n", nDirs);
    printf ("%s\n", pathDirs at 0);
    printf ("%s\n", pathDirs at 1);
    printf ("%s\n", pathDirs at 2);
}
```

The receiver of the method `split`, in the statement

```
nDirs = infileStream streamPath split '/', pathDirs;
```

is the result of sending the message `streamPath` to `infileStream`. The result is the path name of `infileStream`, a `String`.

The result of one message must be a member of the same class as the receiver of the next message, or a member of a subclass if the message refers to a method.

Note the use of the `SystemErrnoException` object, `e`. This class is used for recording system errors from methods that interact with the operating system. The later sections describe exception handling in more detail.

The *Ctalk Language Reference* describes the class and instance variables of each class, and the return classes of methods.

3.0.4 Variable Promotion and Type Conversion

This section could also be titled *When to use become, copy, or =*.

Because objects can be much more complex than C data types, variable promotion and type conversion become much more complex operations in object oriented languages than in C. This will become apparent later on.

For now, because we're discussing basic object classes, it should be sufficient to look at the methods of class `Magnitude`, which is the superclass of these basic classes.

You need to use the **Magnitude** methods `asCharacter`, `asInteger`, and `asLongInteger` to perform type conversions explicitly. The *Ctalk Language Reference* describes the type conversion methods for each class.

When making assignments, you can use `=` for receivers and arguments that a function or method declares explicitly, or with constant receivers and objects. The classes `Integer`, `Character`, and `String` each implement a `=` method, so a program would use `=` with objects that are instances of these classes, or with constants like `1`, `a`, or `"Hello, world!"`

In other cases, if a statement needs to make an assignment and the statement contains receiver or instance variables, or an operand that the program cannot determine until run time, or if the receiver or operand are of completely different classes, then `become` (class `Object`) can in most cases perform the type conversion and assignment without affecting the objects that they are instance variables of. This is often the case with classes that represent complex data, like files or windows.

The `become` method takes into account situations where the type of a receiver or local object may change during the course of a program's execution.

When a statement needs to duplicate objects of the same class, then it should use the `copy` (class `Object`) method.

In all of these cases, if a class contains data that requires a special protocol to duplicate, then it implement its own versions of these methods, or any other way it needs to assign objects.

3.0.5 Default Methods and Instance Variable Messages

There is one other point to note about methods and messages, and it is related to some of the examples in the previous section. Ctalk objects also respond to messages that name instance variables.

All Ctalk objects contain the instance variable, `value`, which is inherited from class `Object`. Because of this, if you use an object without a message, Ctalk uses the `value` message by default.

In this simple example, the following two statements are equivalent.

```
printf ("%d\n", myInt);
```

```
printf ("%d\n", myInt value);
```

With more complex objects, for example like those in the `Stream` and `Pane` subclasses, statements that contain instance variable messages can quickly become complex. The later chapters and code snippets in *The Ctalk Language Reference* describe how to use instance and class variable messages with complex objects.

4 File Input and Output

This chapter describes how Ctalk performs basic file operations - opening, closing, reading, and writing files.

There are three classes that provide file input and output: `ReadFileStream`, which implements objects and methods for reading from files, `WriteFileStream`, which provides methods and objects for writing to files, and their superclass, `FileStream`, which provides methods and class definitions that are common to both reading and writing files. There is also a `DirectoryStream` class, which this chapter describes below.

Ctalk provides many of the same facilities that C's `stdio.h` library functions provide, although the language implements them as objects.

Here is an example that shows how to use the `openOn` method (class `ReadFileStream`).

```
ReadFileStream new myInput;

myInput openOn "myFileName";
```

If the program was able to open the file, you can then read input from the `myInput` stream.

We should note here that Ctalk also provides classes and methods that handle file I/O errors. We will deal with them later on. If you need to see how to catch file I/O errors right now, look at some of the example programs in the Ctalk package, especially `ctwc.c`.

Opening a file for output is similar to opening a file for input, except that the stream is an instance of class `WriteFileStream`.

```
WriteFileStream new myOutput;

myOutput openOn "outputFileName";
```

After a program has opened the file streams successfully, you can read and write from them with the following methods.

Operation	Reading	Writing
-----	-----	-----
Read or write one character.	<code>readChar</code>	<code>writeChar</code>
Read or write one line.	<code>readLine</code>	-
Read or write all the data.	<code>readAll</code>	<code>writeStream</code>
Read or write formatted data.	<code>readFormat</code>	<code>printOn</code>
Read fixed length data.	<code>readRec</code>	

Here is a simple example that shows opening a file, and reading each character.

```
ReadFileStream new inputChars;
Character new c;

inputChars openOn "inputFileName";

while ((c = inputChars readChar) != EOF)
    printf ("%c", c);
```

When `readChar` reaches the end of a file, it returns an EOF character, which is '-1' on most systems.

Using the method `streamEof` (class `FileStream`) to check for the end of the input is slightly more reliable than looking for an EOF character, because it distinguishes the end of the input caused by an error from the end of the input caused by reaching the end of a file.

```
ReadFileStream new inputChars;
Character new c;

inputChars openOn "inputFileName";

while (TRUE) {                               /* Loop until the end of the input. */
    c = inputChars readChar;
    if (inputChars streamEof)
        break;
}
```

If you don't need to examine each character as it is read, however, then you can simply use `readAll` (class `ReadFileStream`), which provides the complete input as a `String` object.

```
ReadFileStream new inputStream;
String new inputString;

inputStream openOn "myInput";
inputString = inputStream readAll;
printf ("%s", inputString);
```

Simple applications can use the methods `readLine` (class `ReadFileStream`) and `writeStream` (class `WriteFileStream`) together to process input.

Here is a portion of the program `ctrep.c`. You find the program in the Ctalk package. It checks the input for occurrences of a character string, and replaces the string before writing the output.

```
/*
 * Loop until the end of input.
 */
while (TRUE) {
    line = stdinStream readLine;
    if (stdinStream streamEof)
        break;
    inputLineLength = line length;

    word = "";

    for (i = 0; i < inputLineLength; i = i + 1) {
        inputChar = line at i;
        if (inputChar isSpace) {
            if (word == pattern) {
                stdoutStream writeStream replacement;
            } else {
                stdoutStream writeStream word;
            }
        }
    }
}
```

```

        stdoutStream writeStream inputChar;
        word = "";
    } else {
        word = word + inputChar asString;
    }
}
}
}

```

Standard Input and Standard Output

Notice that the previous example used `stdinStream` and `stdoutStream` as its input and output streams.

These two objects are class variables of `ReadFileStream` and `WriteFileStream`, respectively. They represent the program's standard input (`stdin`) and standard output (`stdout`) file streams.

Because `ctrep.c` is a filter program, it uses these two streams instead of streams that work with normal files.

Class `WriteFileStream` also implements the `stderrStream` class variable, which represents the application's standard error (`stderr`) stream.

You should notice, also, that all of the examples so far have used `printf` to print output. You can accomplish the same task with the following Ctalk statement.

```

    stdoutStream writeStream "Hello, world!\n";

```

This is equivalent to:

```

    printf ("Hello, world!\n");

```

Ctalk initializes `stdinStream` automatically when the program creates the first `ReadFileStream` object, and it initializes `stdoutStream` and `stderrStream` when the program creates the first `WriteFileStream` object.

If the program simply needs to use `stdinStream`, `stdoutStream`, or `stderrStream` without opening files, then it can call the class method `classInit` with either the class `ReadFileStream` or `WriteFileStream` as the receiver.

```

    ReadFileStream classInit;    /* Initialize stdinStream. */
    WriteFileStream classInit;   /* Initialize stdoutStream and stderrStream. */

```

Here is the example from the previous chapter that printed elements of arrays. This version uses `stdoutStream` to output the array elements.

```

Array instanceMethod printArrayElement (void) {

    WriteFileStream classInit;

    stdoutStream writeStream self;

    return NULL;
}

int main () {

```

```

    Array new myArray;

    myArray atPut 0, "My";
    myArray atPut 1, "name";
    myArray atPut 2, "is";
    myArray atPut 3, "Bill";

    myArray map printArrayElement;

    printf ("\n");

    myArray atPut 3, "Joe";

    myArray map printArrayElement;

    printf ("\n");
}

```

The `printArrayElement` method initializes class `WriteFileStream`. You need to perform the class initialization before the class is first used in the program. There is no problem with calling `classInit` multiple times, however, because the method checks to determine if the class is already initialized.

Directories

The class `DirectoryStream` provides the methods `mkDir` and `rmDir` which create and delete directories. The methods function exactly as their C library counterparts, except that they raise a `SystemErrnoException` on error and return an `Integer` with the value `-1`.

Ctalk uses `0755` (`'drwxr-xr-x'`) as the default mode for new directories. Programs can change that value by redefining the macro `CTALK_DIRECTORY_MODE`.

Here is a program that creates a new directory.

```

/*
 * Define more restrictive permissions for
 * new directories. Undefine the macro first
 * to avoid a warning message.
 */
#undef CTALK_DIRECTORY_MODE
#define CTALK_DIRECTORY_MODE 0700

int main () {

    DirectoryStream new thisDir;

    thisDir mkDir "testDir";

}

```

The `rmDir` method works similarly to the `mkDir` method.

```
int main () {  
  
    DirectoryStream new thisDir;  
  
    thisDir rmdir "testDir";  
  
}
```

Standard Input and Standard Output Implementations

The C99 standard requires that `stdin`, `stdout`, and `stderr` should be implemented as macros, which on some systems (notably Solaris) causes problems with C-to-object translation.

If Ctalk cannot register the macros as C variables, then you must call C functions like `sscanf(3)` and `fscanf(3)` with only C variables, or use a method with `stdoutStream` or `stderrStream` (`WriteFileStream` class), or `stdinStream` (`ReadFileStream` class).

5 `self` and `super`

The `self` and `super` keywords refer to receivers and superclass methods at run time. They allow you to write methods without knowing beforehand what objects called them.

5.1 `self` as a Receiver

Within a method, `self` refers to the receiver of the method that `self` appears in. Within the method, you can use `self` interchangeably with other receivers.

When used as a receiver, Ctalk's `self` works much like the `self` keyword in Smalltalk and the `this` keyword in C++.

In this example, `self` in the method `add2` refers to the method's receiver, `myInt`, which is declared in `main`.

```
Integer instanceMethod add2 (Integer arg) {  
  
    methodReturnInteger(self + 2)  
  
}  
  
int main () {  
  
    Integer new myInt;  
    Integer new myTotal;  
  
    myInt = 2;  
  
    myTotal = myInt add2;  
  
}
```

You need to be careful of using `self` within a constructor (a `new` method), because the method's receiver is the class object of the instance you are creating, and not the new object itself, and that is probably not what you want. Ctalk prints a warning message in this case if you enable verbose warnings with the `-v` command line option.

5.2 The `super` Keyword

When used before a message, `super` tells Ctalk that the message refers to a method in the superclass of the receiver.

The most common use of `super` is in constructors of subclass objects. The keyword allows constructor methods, which are almost always called `new`, to refer to the constructor of a superclass, which allows subclass objects to inherit the instance variables of its superclasses.

The example in the first chapter, of how `FileStream` and its subclasses inherit instance variables, describes how constructors can call class constructors. See [Class Hierarchy], page 3.

To show how `super` is used, here is the `new` method from class `WriteFileStream`.

```

WriteFileStream instanceMethod new (char *name) {

    WriteFileStream super new name;

    __ctalkInstanceVarsFromClassObject (name);

    WriteFileStream classInit;

    methodReturnObject(name)
}

```

5.3 Determining the Class of self

Generally, the receiver object that `self` refers to belongs to the same class as the method where it appears. But there are many occasions when a method cannot assume that it knows in advance the class of `self`, or the result of an expression that contains `self`. This section describes a few of these situations and how to deal with them.

A method might be declared in a superclass of the receiver, or the method might need to work objects that are members of a collection, as in this example. A method might belong to the class of a collection like `List` or `AssociativeArray`, but the collection's elements may be another class entirely.

In these cases, methods need to account for the fact that the receiver object is actually an instance of `Key` class, which is the actual class of the collection's instance variables, and that `Key` class contains references to the collection's member objects.

This method is from the `ctcheckquery.c` example program, the parameters and values of a CGI query.

This method is relatively simple because it can assume beforehand that the values it is going to print are instances of `String` class. The actual receiver is an instance of `Key` class, so `self` in this method understands both the `name` and `getValue` methods, which are defined in `Key` class.

```

AssociativeArray instanceMethod printQueryParamPair (void) {
    printf ("<tt>%s=", self name);
    printf ("\\"%s\\"</tt><br>\n", self getValue);
    methodReturnNULL
}

```

5.3.1 Method Dispatchers

If you must use a complete, original receiver object from a collection class, then the method cannot alias the original receiver to a local object.

In this case, the program can use another method to dispatch the receiver's message to the correct at run time. For this, the dispatch method can use the `eval` keyword.

This situation is frequently the case with complex objects. Here is an example from `ANSIYesNoBoxPane` class that selects the highlight of an already selected input pane.

The receiver of the `focusButton` method's `map` message is the widget's list of child windows (a `List`), and this method checks the list element's class and dispatches the program to the `highlightButton` method of class `ANSIButtonPane`.

```

ANSIButtonPane instanceMethod highlightButton (void) {
    if (self hasFocus) {
        self focusHighlightOnOff;
    } else {
        self resetGraphics;
    }
    methodReturnNULL;
}

List instanceMethod highlightButton (void) {
    Exception new e;
    if (self value is ANSIButtonPane) {
        eval self highlightButton;
    } else {
        e raiseCriticalException INVALID_RECEIVER_X,
            "Receiver of \"highlightButton\" is not an ANSIButtonPane object";
    }
    methodReturnNULL;
}

ANSIYesNoBoxPane instanceMethod focusButton (void) {
    self children map highlightButton;
    methodReturnNULL;
}

```

Note that when checking the class, the program uses

```
if (self value is ANSIButtonPane) { ...
```

This is because the receiver, a button widget, is an instance variable of an `ANSIYesNoBoxPane` object, so its member class is also `ANSIYesNoBoxPane`, but its *value* is `ANSIButtonPane`.

You can also write most of a method in C, if necessary.

Here is the `asInteger` method from class `Magnitude`. Generally, only the `Object` class is completely evaluated before this method, so `asInteger` needs to perform many of its operations in C. However, it can use the method `is` from `Object` class to determine the class of the receiver.

```

Magnitude instanceMethod asInteger (void) {
    OBJECT *self_value;
    int i;
    long long int l;

    returnObjectClass Integer;

    self_value = self value;

    if (self is Integer) {
        methodReturnSelf;
    }
}

```

```

} else {
    if (self is LongInteger) {
        sscanf (self_value -> __o_value, "%lld", &l);
        if (l > MAX_UINT)
_warning ("Overflow in type conversion.\n");
        i = (int)l;
    }
    /*
     * Character to int conversion, plus escape sequences.
     */
    if (self is Character) {
        if (*self_value -> __o_value == '\\') {
if (self_value -> __o_value[1] == '\\') {
switch (self_value -> __o_value[2])
{
case 'a':
    i = 1;
    break;
case 'b':
    i = 2;
    break;
case 'f':
    i = 6;
    break;
case 'n':
    i = 10;
    break;
case 'r':
    i = 13;
    break;
case 't':
    i = 9;
    break;
case 'v':
    i = 11;
    break;
case '0':
    i = 0;
    break;
default:
    i = (int) self_value -> __o_value[2];
    break;
}
} else {
    i = (int) self_value -> __o_value[1];
}
        } else {

```

```

    i = (int) *self_value -> __o_value;
    }
    } /* if (self is Character) { */

    if (self is Float) {
        /*
         * In case Exception class isn't loaded yet, simply
         * print a warning.
         */
        _warning ("asInteger (class Magnitude): Receiver truncated to Integer.\n");
        i = (int)atof (self);
    }
    }
    methodReturnInteger (i);
}

```

5.4 Class of a Method Return Object

Often a program can determine the class of a method's return object from the class of the receiver. Sometimes, however, the `returnObjectClass` statement is necessary. Consider the following example from `X11TerminalStream` class.

```

X11TerminalStream instanceMethod parentPane (void) {
    OBJECT *receiver_alias,
        *parent_alias;
    returnObjectClass X11Pane;
    receiver_alias = self;
    if (receiver_alias -> __o_p_obj) {
        parent_alias = receiver_alias -> __o_p_obj;
    } else {
        parent_alias = NULL;
    }
    return parent_alias;
}
...
self parentPane origin x = event_data1;

```

If Ctalk couldn't determine that `parentPane` returns a `X11Pane` object, it would not be able to determine that the class of `origin` (an instance variable of `X11Pane`) is a `Point`, and that `x` is an instance variable of `origin`.

Defining the object return class also helps in checking the class of arguments in complex statements. This helps Ctalk determine what is and isn't a method argument, as in this example.

```

Integer new intA;
Integer new intB;

if (intA + intB == 4)
    do something;

```

Here, Ctalk can determine that `==` isn't part of the argument to `+`, because it returns a `Boolean`, while `+` expects an `Integer` or an expression that returns an `Integer` as its argument.

5.5 Determining the Class of Method Arguments

While we're on the subject, we should mention that methods do not always know the class of their arguments, either. If the class of a method argument can vary, that makes it even more difficult to handle complex objects that contain numerous instance variables.

One way to deal with arguments of different classes is to duplicate the argument into the class the method needs, as in this example, the `=` method from `Integer` class.

```
Integer instanceMethod = set_value (int __i) {
    Integer new localInt;
    localInt become __i asInteger;
    __ctalkAddInstanceVariable (self, "value", localInt value);
    methodReturnSelf
}
```

Remember, a method can specify the class or type of an argument, but that doesn't mean, when the program runs, that the argument actually *is* the class that the method expects.

Another issue is trying to print objects. If you were to write a method like the following.

```
MyClass instanceMethod myMethod (Object myArg) {
    printf ("%s\n", myArg myInstanceVariable);
    methodReturnNULL;
}
```

Then you would need to make sure that `myInstanceVariable` is a `String`. If possible, Ctalk tries to match the class of an expression with its context; in the example above, if Ctalk cannot determine the class of `myInstanceVariable` from the context of the method, it will try to translate it into a `char *` argument for `printf`.

There are a few ways that a method can determine the class of an expression. One is to use the `Object` method `is`, as in the `asInteger` method in the previous section.

Another solution is to "cast" the argument to a local object using, for example, the `Object` method `become`, as in the `= (class Integer)` example above.

Yet another way to deal with an unknown argument class is to use a `printOn` method. The method is better than `printf` at determining an object's class at run time.

```
MyClass instanceMethod myMethod (Object myArg) {
    WriteFileStream classInit;
    stdoutStream printOn "%s\n", myArg myInstanceVariable;
    methodReturnNULL;
}
```

Many classes implement a `printOn` method, and it is relatively easy for a class that you write to define a `printOn` method also. The Ctalk libraries take care of much of the work of translating objects of different classes and formatting them correctly.

The `printOn` methods can also use receivers other than stream classes, so a method could use `printOn` to format a `String`, as in this example.

```
String instanceMethod myIntConv (int myArg) {  
    self printOn "%s\n", myArg myIntVariable;  
    methodReturnSelf;  
}
```


6 Collections

Collections are groups of objects. Ctalk has the following main classes of collections.

- **Array** class, which orders elements of a group by a numerical index.
- **AssociativeArray** class, which orders and retrieves elements using a key value.
- **List** class objects, which contain groups of objects that you can treat sequentially.
- **Stream** class objects, which are generally groups of characters similar to **String** class objects, that allow reading and writing to the collection.

These classes are subclasses of **Collection** class. Here is the section of the class library that shows the organization of **Collection** and its subclasses.

```
Collection
  Array
  List
  AssociativeArray
  Stream
  FileStream
  ReadFileStream
  WriteFileStream
```

The previous chapters described **Array** and **Stream** objects (the classes **FileStream**, **ReadFileStream**, and **WriteFileStream**), so this chapter simply provides a brief description of the remaining collection classes, **AssociativeArray** and **List**.

6.1 List Class

List class objects contain doubly linked lists of object references.

If you want to add an item to a **List** object's contents, use the method **push**.

```
List new l;
Integer new i;
Integer new i2;

l push i;
l push i2;
```

To remove items from the end of a list and retrieve the items, use the method **pop**.

```
List new l;
Integer new i;
Integer new i2;
Integer new i3;
Integer new i4;

l push i;
l push i2;

i3 = l pop;
i4 = l pop;
```

The methods `shift` and `unshift` are similar to `push` and `pop`, respectively, but they add and remove items from the front of the list.

Here is a simple, if slightly unwieldy, example of how these methods work. You can find this program in `test/basiclist.c`.

```
int main () {

    List new l;
    Integer new i1;
    Integer new i2;
    Integer new i3;
    Integer new i4;
    Integer new i5;
    Integer new i6;

    i1 = 1;
    printf ("%d ", i1);
    i2 = 2;
    printf ("%d ", i2);
    i3 = 3;
    printf ("%d ", i3);
    printf ("\n");

    l push i1;
    l push i2;
    l push i3;

    i4 = l pop;
    printf ("%d ", i4);
    i5 = l pop;
    printf ("%d ", i5);
    i6 = l pop;
    printf ("%d ", i6);
    printf ("\n");

    l shift i1;
    printf ("%d ", i1);
    l shift i2;
    printf ("%d ", i2);
    l shift i3;
    printf ("%d ", i3);
    printf ("\n");

    i4 = l unshift;
    printf ("%d ", i4);
    i5 = l unshift;
    printf ("%d ", i5);
```

```

        i6 = l unshift;
        printf ("%d ", i6);
        printf ("\n");

        return 0;
    }

```

If you want to work with every item in a list, use the method `map`.

```

List instanceMethod printItem (void) {
    Integer new element;
    element = self;
    printf ("%d ", element);
    methodReturnNULL;
}

int main () {

    List new l;
    Integer new i1;
    Integer new i2;
    Integer new i3;

    i1 = 1;
    i2 = 2;
    i3 = 3;

    l push i1;
    l push i2;
    l push i3;

    l map printItem;
    printf ("\n");
}

```

Remember that the argument to `map`, in this example `printItem`, must be an instance method of class `List`, even though the receiver of each call to `printItem`, each successive member of `List l` (in `main()`), can have its own class.

This example shows one way to handle method receivers when the program can make assumptions about what the class of `self` is. That is not always the case, however. See [self Class Resolution], page 24, for a detailed discussion.

Within `printItem`, you have to use the library functions `__ctalk_self_internal()` and `__ctalk_to_c_int()` to refer to the items of `l`, because `printItem` does not know until run time what class `self` (each member of `l`) is going to be, and Ctalk cannot perform the object-to-C translation for `printf` automatically.

6.2 AssociativeArray Class

`AssociativeArray` objects store their elements using a `Key` object.

Programs use name of the `Key` object, normally a `String`, to store and retrieve elements of the array. The value of the `Key` object is the object that you want to store in the array. The `AssociativeArray` class does the work of assigning keys and storing objects in the arrays for you, using the methods `at` and `atPut`.

For example, to store an object, use a statement similar to the following.

```
myAssociativeArray atPut "keyName", myObject;
```

To retrieve an object, use the method `at`.

```
myObjectFromBefore = myAssociativeArray at "keyName";
```

You can also treat `AssociativeArray` elements sequentially, using the method `map`.

The *Ctalk Language Reference* describes the details of storing and retrieving `Key` object names and values.

6.3 Collection Elements as Receivers

When using collection elements, programs often don't know until run time exactly what the class of a receiver might be - elements of object. In addition methods like `map`, use each element of a collection as a receiver.

Collections all use `Key` objects to store the actual data. In methods that use collection elements, programs can either declare a `Key` object explicitly, or it can rely on a method in a superclass to determine whether the receivers are valid. The *Ctalk Web Utilities* manual provides examples of both techniques.

Here is `getValue` in `Collection` class, which determines that the actual receiver of a `getValue` message at run time is a `Key` object and generates an exception otherwise.

```
Collection instanceMethod getValue (void) {
    Exception new e;

    if (self is Key) {
        return self getValue;
    }
    e raiseCriticalException INVALID_RECEIVER_X,
        "Message \"getValue\" sent to a Collection object, not a Key object.";
    methodReturnNULL
}
```

7 Defining Classes and Objects

The previous chapters described some of the methods that the Ctalk classes define. This chapter discusses in further detail how to define Classes and variables that belong to the classes, and discusses some of the ways that Ctalk initializes instances of a class.

Many of the methods used in this chapter are *primitive* methods. The primitive method definitions are in the `Object` class, and any class can use them.

7.1 Superclasses

At a minimum, a class definition defines the name of the new class and its superclass. The primitive method `class` (`Object` class) takes the name of a new class as its argument. The new class is defined in the class library as a subclass of the receiver's class.

This statement defines a class with the name `WriteFileStream` as a subclass of the receiver, `FileStream`.

```
FileStream class WriteFileStream;
```

Or, more generically,

```
superclassname class newclassname;
```

The superclass indicates where the new class fits into the class library, and that determines which methods and instance data the new class inherits. The only class that does not have a superclass is `Object`, the topmost class in the class library.

When defining new classes, it is important that the superclasses of the new class already be defined. That statement may seem obvious, but you need to take care when defining complex objects that all of the definitions you need are already present.

Another reason is that, when you define an instance or class variable, the variable *is a member of your new class*, unless you specify that the variable belongs to another class. If your class's instance and class variables can inherit methods from the class's superclasses, then you don't need to write new methods to manage the variables.

However, you can direct Ctalk to initialize other classes if your class needs them, by using, for example, statements like `require`, as described below.

The previous chapters have described some of the ways that you can work with instance data using C. Many of the Ctalk library's API functions also check for the type and value instance data. The method API is described later on.

Ctalk also contains several programs that define their own classes. Although the example programs are too long to list here, this chapter mentions them when necessary.

7.2 Instance Variables

We mentioned already that all objects have a default instance variable named `value`. However, many classes define other instance variables. Classes define instance variables with the `instanceVariable` method.

Here is an example of an instance variable definition.

```
MyNewClass instanceVariable textMessage String NULL;
```

This example declares that all instances of `MyNewClass` will have an instance variable named `textMessage`, and that it is an instance of class `String`, and it has the initial value `NULL`.

The second and third arguments of `instanceVariable` are optional. Here is the syntax of the `instanceVariable` method.

```
classname instanceVariable varname [varclass|typecast_expr] [initialvalue]
```

Remember that `instanceVariable` only *defines* variables. The method does not actually create them. When you create an object that is an instance of your class, the object gets its own copy of the variable.

If the program declares a variable with a type cast instead of a native class for the variable, Ctalk matches the type cast to a class. If the type cast of the variable is a pointer to some data type that is not commonly used in Ctalk or C programs, then Ctalk simply translates the native type to `Symbol`. This is because the program needs to implement its own methods for handling the data, and `Symbol` is the most generic class that Ctalk can use for referring to C data in memory.

7.3 Class Variables

On the other hand, the `classVariable` method creates the single copy of a class variable that the instances of a class (and most other classes) can use. Creating a class variable is similar to creating an instance variable.

```
MyNewClass classVariable systemError Integer 0;
```

For completeness, here is the `classVariable` method's syntax.

```
classname classVariable varname [varclass|typecast_expr] [initialvalue]
```

Remember that if you define a variable as a different class than its member class, then the variable responds to messages of its own class. In these cases, you need to take care that you are working with the variable, and not the class that it belongs to.

If the variable's native type is declared with a type cast, then Ctalk tries to translate the type cast into a Ctalk class, as described in the previous section.

7.4 Class Initialization

Although you can define an initial value for a class variable, in many cases you need to wait until the program is run before setting the actual value of a class variable. An example of this is `stdoutStream` (class `WriteFileStream`).

In many cases, a constructor method (which in Ctalk are named `new`), can call a class initialization method.

Here is a hypothetical class with `classInit` a method. This example simply sets the the name of the system in a class variable called `hostName`.

```
MySuperClass class MyClass;

MyClass classVariable myClassVar String NULL;

...
Other class and instance variable definitions.
...
```

```

MyClass classMethod classInit (void) {

    OBJECT *classObject_object,
           *classInitVar_object,
           *myClassVar_object;
    char buf[MAXLABEL];

    /*
     * Return if we've already initialize the class.
     */
    if (self classInitDone)
        methodReturnNULL

    /*
     * Retrieve the class object, because we'll need it below.
     */

    classObject_object = __ctalkGetClass("MyClass");

    /*
     * Look for myClassVar.
     * The third argument of __ctalkGetClassVariable instructs the
     * function not to print a warning if it doesn't find the
     * variable. If the variable is not present, then the class
     * is not completely initialized, so return.
     */

    if ((myClassVar_object =
        __ctalkGetClassVariable (self, "myClassVar", FALSE)) == NULL) {
        methodReturnNULL
    }

    ...
    Initialize myClassVar here.
    ...

    classInitVar =
        __ctalkCreateObjectInit ("classInitDone", "Integer",
                                "Magnitude", GLOBAL_VAR, "1");
    __ctalkAddClassVariable (classObject, "classInitDone", classInitVar);

    methodReturnNULL
}

...
Other methods.

```

...

When writing classes, you need to remember that Ctalk initializes the class and its methods and data in the following order.

1. The class itself and its superclass.
2. The class' instance and class methods.
3. Instance variables.
4. Class variables.

Remember that a class definition only requires a superclass. All of the other elements are optional.

It is possible, when initializing complex classes, to construct objects before the class is completely defined. That's why `classInit` above checks for the class variable `myClassVar` and returns if it is not present.

However, Ctalk checks the class initialization every time it creates an instance of `MyClass`, so a constructor method for `MyClass` might look like the following.

```
MyClass instanceMethod new (char *myObjectName) {

    MyClass super new myObjectName;

    __ctalkInstanceVarsFromClassObject (myObjectName);

    MyClass classInit;

    methodReturnObjectName(myObjectName)
}
```

The program then initializes the class variables when the class is completely defined. If `classInitDone` is `TRUE`, then the program won't initialize the class again.

Note: Ctalk does not automatically create a copy of a class's instance variables when creating objects. Your program must call `__ctalkInstanceVarsFromClassObject` to create the instance variables in the new object.

If you simply want to initialize a class without creating a new object, you can add a statement like this to your program.

```
MyClass classInit;
```

Another example of a method that performs class initialization is the method `initDigits` (class `ClockDigit`) in the program `ltime.c`. The method is too long to list here, but you can find `ltime.c` in the Ctalk distribution's `programs` subdirectory.

The later chapters describe methods and library functions that perform class initialization in more detail.

7.5 require Keyword

Ctalk programs initialize the basic classes before starting to initialize the application's class and method definitions. However, if your definition needs a class that Ctalk does not include automatically, you can initialize the class and include its method and variable definitions before defining your own class by using the keyword `require`.

The use of `require` is simple. Include it with the name of the class that your class needs, and Ctalk inserts the its class and method definitions at that point.

```
require List;
```

7.6 Where Ctalk Looks for Class Libraries

The Ctalk configuration determines where the class libraries are located when you build and install Ctalk. In a normal UNIX system, that location is the directory `/usr/local/include/ctalk`.

However, if you want your own classes to reside in the same directory as the program, or any other directory, you can include the directory in Ctalk's library search path with the `-I` option.

On many systems the shorthand for the current directory is `.`, so if your class is located in the same directory as the program, add the directory to the library search path as in this example.

```
$ ctalk -I. myprog.c -o myprog.i
```

7.7 Instance Variables and Method Overloading

One consequence of all this convenient instance data and overloaded methods is that it is easy to confuse which method a message actually translates into. That is because the member class of an instance or class variable might not be the class of the variable's own instance data.

Here is an example. The `openOn` message in the first statement translates to the `openOn` method in the `ANSITerminalPane` class, while the `openOn` message in the second statement translates to the `openOn` method in the `ReadStream` class.

```
ANSITerminalPane new mainWindow;

mainWindow paneStream openOn "/dev/ttyS0";           /* Correct. */

mainWindow paneStream inputHandle openOn "/dev/ttyS0"; /* Incorrect. */
```

This is because `inputHandle` in the second statement is actually a `ReadStream`, while the program actually wants to call is the `openOn` method from `ANSITerminalStream`, which is the effect of the first statement. The `Stream` classes have many classes in common. In most cases this makes things simpler, but not always.

There are several ways to work around this issue. This first is to use stack walkbacks and exception handlers whenever a program opens a file. That allows you to see what methods the program's statements actually translate into. See Chapter 10 [Debugging], page 57.

If you can't avoid an incorrect message translation in a program, you can write your own method that only one of your program's classes responds to. For example, instead of yet another `openOn` method, your program might define a method like, for example, `openOnInputData`.

Finally, if the program cannot determine in advance which class is going to receive a message, you can write a method that dispatches the receiver to a method of the correct class. Here is an example.

```
Stream instanceMethod openOnTerminal (String pathName) {
    Exception new e;
    if (self is ANSITerminalStream) {
        if (pathName isATty) {
            self openOn pathName;
        } else {
            e raiseException NOT_A_TTY_X, pathName;
        }
    } else {
        e raiseException INVALID_RECEIVER_X, pathName;
    }
    methodReturnSelf;
}
```

8 Writing Methods

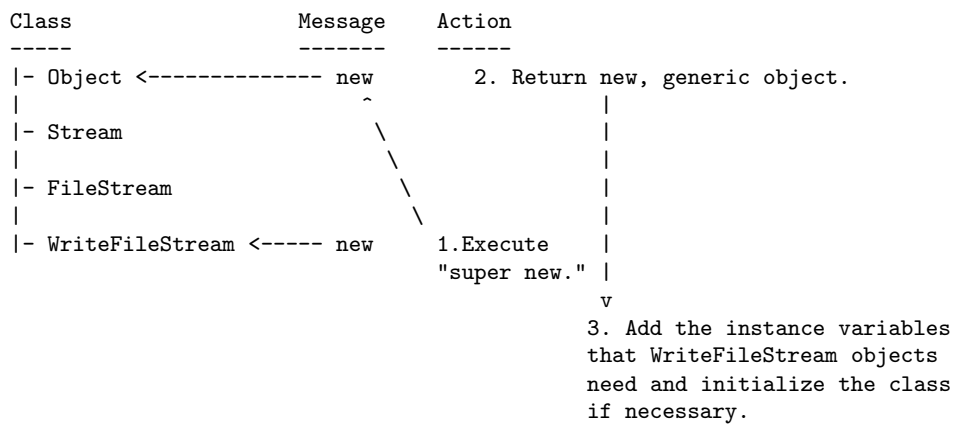
Languages like C identify and call functions by matching a label in the source code against an identifier in an object module, and the compiler (or more properly, the linker or interpreter) inserts a call to the function in the program.

In Ctalk, like many other object oriented languages, objects respond to *messages* that refer to methods defined by the object's class or the object's superclasses.

Here is an example. If you wanted to create a new `WriteFileStream` object, you would add a statement like this to the program.

```
WriteFileStream new myOutputStream;
```

When the receiver, `WriteFileStream`, receives the message, `new`, the process of executing the methods that create a `WriteFileStream` object works in the following manner.



Here is a typical method. This is the instance method `parseQueryString` (class `CGIApp`.) The `CGIApp` class is in the Ctalk `programs/cgi` subdirectory and is used by the Common Gateway Interface programs in that directory.

```
CGIApp instanceMethod parseQueryString (void) {
    String new queryString;
    String new paramString;
    Array new queryParams;
    Array new paramArray;
    Integer new nParams;
    String new paramKey;
    String new paramValue;
    Integer new i;
    Integer new j;

    queryString = self serverEnvironment at "QUERY_STRING";

    if ((queryString length == 0) || (queryString isNull)) {
        methodReturnNULL;
    }

    nParams = queryString split '&', queryParams;
```

```

    for (i = 0; i < nParams; i = i + 1) {
        paramString = queryParams at i;
        j = paramString split '=', paramArray;
        if (j == 1) {
            self queryValues atPut (paramArray at 0), "1";
        } else {
            if (j == 2) {
                self queryValues atPut (paramArray at 0), (paramArray at 1);
            }
        }
    }
    methodReturnNULL;
}

```

The `parseQueryString` method parses the parameters of a CGI query and places them in the application's `queryValues` instance variable, an `AssociativeArray`. The `queryValues` instance variable then contains the key-value pairs of each of CGI application's parameters.

If you call a CGI application with a URL like the following example

```
http://my-web-server/cgi-bin/mysearchprog?term=ctalk&case=ignore&showresults=10
```

Then `parseQueryString` sets the application's `queryValues` `AssociativeArray` to the following.

	Key	Value
<code>queryValues</code> at	<code>term</code>	<code>ctalk</code>
<code>queryValues</code> at	<code>case</code>	<code>ignore</code>
<code>queryValues</code> at	<code>showresults</code>	<code>10</code>

The `parseQueryString` method uses the `split` method (class `String`) several times to split the query string at the `'&'` delimiters, and then each parameter and its value at the `'='` character. If a parameter is defined but does not have a value associated with it, `parseQueryString` assigns it the value of `'1'`.

You should also note that the method checks for both a zero-length input value, and an *uninitialized* input value with the method `isNull`, which might occur, in this example, if the environment variable `'QUERY_STRING'` is not defined.

8.1 Overview of the Method Application Programming Interface

You might recall from the earlier chapters that if a program uses an operator like `'+'` or `'-'` with a C variable, then Ctalk treats the operator as a C math operator. However, if you use a math operator with an object as the receiver, then Ctalk uses a method to perform the operation.

Although there are several exceptions, most methods are written in Ctalk and are located in Ctalk's class library. On UNIX systems, the class library normally resides in `/usr/local/include/ctalk`, although the location may vary depending on the operating system.

8.1.1 Primitive Methods

One exception is the way Ctalk implements *primitive methods*. Ctalk implements these methods in C, and you can call them without worrying if a program defines a class.

These methods are available to all classes, so they effectively belong to the `Object` class. You can also subclass these methods, as is often the case with the constructor `new`. The previous chapters have described in general how to subclass methods, and this manual describes the process in detail later on.

Here are the primitive methods that Ctalk defines.

`class classname`

Defines a new class. The receiver is the name of the superclass, and the argument is the name of the class that you want to create.

`classMethod alias selector (args)`

Declares a new class method. The next section describes the syntax of method declarations. See [Method Declarations], page 44.

`classVariable varname varclass initial-value`

Declare a class variable. The receiver is the class of the variable. You must provide a *name* for the class variable. The statement also allows you to define a class for the class variable and an initial value. See [Class Variables], page 36.

`instanceMethod alias selector (args)`

Declares a new instance method. The next section describes the syntax of method declarations. See [Method Declarations], page 44.

`instanceVariable varname varclass initial-value`

Define an instance variable. The receiver is the class of the variable. Unlike class variables, `instanceVariable` only defines a variable's declaration. A constructor like `new` creates the actual instance variables when it creates an object. See [Instance Variables], page 35.

`new newobjectname`

Create a new object. The receiver of `new` is the object's class.

8.1.2 Instance and Class Variable Messages

For completeness, we should mention that the other exception in the method API is the use of instance and class variable names as messages.

The `value` message is an example of a message that refers to an instance variable. Remember that a statement like the following example returns only the instance variable, and not the complete object.

```
return self value;
```

Instance variable messages are useful mainly when creating C references to objects. Methods, on the other hand, generally work with complete objects. An example is the `at` method in class `Array`.

```
myElement = myArray at 0;
```

The message `at` returns the *0*th array element.

When you use objects with C, you must also check to determine that the object references are really the objects that the program expects.

The following sections describe the Ctalk interface to C, and the many of the examples in this manual describe Ctalk's C API.

8.2 Method Declarations

As you probably noticed from the example above, you can declare methods in much the same manner as C functions.

However, method declarations have a different meaning, and the declarations allow for multiple methods with the same name in different classes.

The following example is the basic syntax of an instance method declaration.

```
class instanceMethod [alias] selector (args) {
... Method body
}
```

The third term, *alias*, is optional. It is often used to alias a method to an operator like '+', '-', or another operator that you want to overload in *class*.

The syntax of a class method declaration is similar, but it uses the `classMethod` method.

```
class classMethod [alias] selector (args) {
... Method body
}
```

8.3 Using Methods in Simple Statements

Methods can be either labels or C operators. Although Ctalk's syntax is similar to C, it allows more flexibility in the construction of statements.

As we noted earlier, you can change characters from upper case to lower case and vice versa by toggling the fifth bit of the character. Here is that example again.

```
Character new upperCase;
Character new lowerCase;
Integer new toggleBit;

upperCase = 'Z';
lowerCase = 'z';
toggleBit = 32;

printf ("%c\n", upperCase ^ toggleBit);
printf ("%c\n", lowerCase ^ toggleBit);
```

The ^ operator in these statements is actually a method in class `Character`.

Ctalk also provides two methods in `Character` class, `toUpper` and `toLower`, that provide the same function, and the methods also check that the receiver is a letter of the appropriate case before changing the case.

Here is the `toLower` method.

```
Character instanceMethod toLower (void) {
Character new result;
```

```

    if (self isAlpha && self isUpper) {
        result = self asInteger ^ 32;
    } else {
        result = self;
    }
    return result;
}

```

We could omit the `asInteger` message, but using `asInteger`, which is implemented in `Magnitude` class, allows us to reliably check that the receiver of `^` is actually a valid number, even if the receiver of `toLower` is the result of another statement that translates its value from a completely different receiver elsewhere in the program.

The result of `self asInteger` is an `Integer` object, so it uses `^` from `Integer` class to perform the translation.

The `asCharacter` method, like `asInteger`, is defined in `Magnitude` class.

The `value` message in the example above is optional. Ctalk sends method arguments as complete objects.

Ctalk statements can appear in many of the places that a C expression can appear. Here is another simple example.

```

int main () {
    String new ten;
    ten = "10";
    printf ("%d\n", ten asInteger);
}

```

As a general rule, keep in mind that receiver objects and other methods that are part of complex statements work with objects.

There are many cases, however, when you need to use a message like `value` in a statement to return an instance variable - generally when you are using objects with C statements, which the next section describes.

8.4 The value Message and Interfacing with C

We have already described how refer to objects with C variables in many places. This section provides a slightly more formal definition of how and when to use `value`.

If you are assigning an object to a C variable, then in most cases you want to use the `value` message. This message, you might remember from previous chapters, returns the receiver's `value` instance variable.

So if you want to assign an object's value to a C variable, you might use a set of statements like the following.

```

OBJECT *value_object;
value_object = self value;

```

You should *not*, however, simply use an object like `self` on its own, because Ctalk cannot determine exactly how you plan to use the result. The following example will provide the complete object, which is not what you want.

```
OBJECT *value_object;
value_object = self;
```

It is okay, however, to use an object name on its own when used with other objects.

```
Object new receiverObjectCopy;
receiverObjectCopy = self;
```

Once you have assigned an object to a C variable, you can treat it as a C `struct`. The next section describes the `OBJECT C typedef`.

8.4.1 The OBJECT Type

Ctalk represents objects internally as an `OBJECT typedef`, so when you need to use an object in a C statement, you can create a C alias for the object by declaring and initializing a C `OBJECT *`.

Here is the `OBJECT` type declaration. It is located in the file `classes/ctalklib`, and several other places.

```
typedef struct _object OBJECT;
...
struct _object {
    char sig[16];
    char __o_name[MAXLABEL];
    char __o_classname[MAXLABEL];
    OBJECT *__o_class;
    char __o_superclassname[MAXLABEL];
    OBJECT *__o_superclass;
    OBJECT *__o_p_obj;
    VARENTRY *__o_vareentry;
    char *__o_value;
    METHOD *instance_methods,
        *class_methods;
    int scope;
    int nrefs;
    struct _object *classvars;
    struct _object *instancevars;
    struct _object *next;
    struct _object *prev;
};
```

So when a program uses statements like those in the following example, the program is actually creating a C reference to an object.

```
OBJECT *rcvr_value;
rcvr_value = self value;
```

Here, `=` is a C operator. Ctalk evaluates the `value` message, and then determines that it doesn't need to perform further translation on the objects because the receiver simply needs the `value` instance variable as C `struct`.

Generally, C programs are mainly interested in the `__o_value` field of an object, because this is where objects store their data as C `char *`'s.

There are also API C functions, like `__ctalk_to_c_char_ptr ()` and `__ctalk_to_c_int ()`, that translate objects to C. The *Ctalk Language Reference* describes these functions.

8.4.2 Return Values

You can also return objects from methods, but you need to be careful to delete any objects that you create. If your method creates an object as the left-hand side of an equation, then Ctalk can generally keep track of the object.

For example, if you create a method called `addTwoAndTwo` and return the result, you might write the method as in the following example.

```
Integer instanceMethod addTwoAndTwo (void) {
    char buf[2];
    sprintf (buf, "%d", 2 + 2);
    return __ctalkCreateObjectInit ("result",
                                    "Integer",
                                    "Magnitude",
                                    LOCAL_VAR,
                                    buf);
}
```

Then, however, you need to make sure that the newly created result is assigned in the expression that calls the method, as in this example.

```
Integer new mySum;

...

mySum = myInteger addTwoAndTwo;
```

However, if the method uses the `methodReturnInteger` macro, it will keep track of the object for you.

```
Integer instanceMethod addTwoAndTwo (void) {
    methodReturnInteger(2+2);
}
```

When the function or method that contains `mySum` exits, it takes care of deleting the value you created, but, again, deleting objects that you create using C is not always automatic.

Another way to write the method would be to assign the value to `self`, as in this example.

```
Integer instanceMethod addTwoAndTwo (void) {
    char buf[2];
    sprintf (buf, "%d", 2 + 2);
    __ctalkSetObjectValueVar (self, buf);
    methodReturnSelf
}
```

Then you can use the `addTwoAndTwo` method in an expression like the following.

```
mySum addTwoAndTwo;
```

The `methodReturnSelf` statement is actually a macro. There are several macros that allow you to return common object types. The *Ctalk Language Reference* contains a complete list of them.

You need to remember that the return value macros are code blocks. Generally they work in any statement, but there can be exceptions, as in this example.

```
Object instanceMethod isNull (void) {
    if (self name == "(null)")
        methodReturnTrue;
    else
        methodReturnFalse;
}
```

This method will cause a syntax error, because `methodReturnTrue` and `methodReturnFalse` expand in this manner.

```
Object instanceMethod isNull (void) {
    if (self name == "(null)")
        {methodReturnTrue statements};
    else
        {methodReturnFalse statements};
}
```

To avoid this, you can write the method in either of the following ways.

```
Object instanceMethod isNull (void) {
    if (self name == "(null)") {
        methodReturnTrue;
    } else {
        methodReturnFalse;
    }
}
```

```
Object instanceMethod isNull (void) {
    if (self name == "(null)")
        methodReturnTrue
    else
        methodReturnFalse
}
```

You should use these statements whenever possible. Unless a method needs to free dynamically allocated buffers before exiting (the section in the language reference about method return values discusses this), these statements will be portable even if the Ctalk method API changes in later versions of the language, and they help guarantee that your methods follow a common call and return protocol.

9 Panes and Graphics

So far, this tutorial has used either C's `printf()` function or `printOn` and other methods of various classes to display information on a terminal.

This chapter describes how to use `Pane` and its subclasses to display information in windows, handle user input, and retrieve information provide by Graphical User Interfaces.

There are `Pane` subclasses the following display types.

ANSI Compatible, Text-mode Displays

Most UNIX consoles, `xterms`, VT-100 many other serial RS-232 terminals, and in some cases, Win32 `cmd.exe` boxes.

X Window System

GUIs that use a X server, window manager, and Xlib libraries.

Closely related to these `Pane` classes are several `Stream` subclasses that handle the chores of processing user input. Graphical User Interfaces use more sophisticated input devices than simple text prompts, and the following sections also describe how to process and respond to different types of input.

We'll begin with a simple example program and then briefly describe the inner workings of the text-mode pane classes.

9.1 Hello, world! in a Window

For the sake of portability and (relative) simplicity, we'll start with an example that uses the `Pane` subclass, `ANSIMessageBoxPane`. This class is a widget that pops up a message window on a text-mode screen, then waits for the user's input, and then closes the window.

The program itself is relatively simple. Here is the listing.

```
int main () {
    ANSIMessageBoxPane new messageBox;
    messageBox withText "Hello, world!";
    messageBox show 10, 10;
    messageBox cleanup;
}
```

The program displays the message until the user presses *Enter* to close the window. Pressing *Esc* also closes windows, but, for most widget classes, it doesn't return any input.

As you might expect, what happens behind the scenes is a little more involved.

9.2 Pane and Stream Classes

When a program creates a new instance of a widget (that is, a subclass of `ANSIWidgetPane`), the new object initializes the instance variables it inherits from its superclasses, initializes the storage to hold the information that is to be displayed, and a `Stream` object that handles the window's input and output. In the case of `ANSIWidgetPane` subclasses, the input and output is handled by an `ANSITerminalStream` object, which is also an instance variable of the widget.

Here are the classes that handle `Pane` graphics, and, for widget classes, their corresponding `Stream` classes. This organization makes it relatively easy to add widgets and support for other text-mode and GUI interfaces.

Pane	Stream
ANSITerminalPane	TerminalStream
ANSIWidgetPane	ANSITerminalStream
ANSIButtonPane	
ANSIMessageBoxPane	
ANSITextBoxPane	
ANSITextEntryPane	
ANSIYesNoBoxPane	
X11Pane	X11TerminalStream

Now that you've seen a little of what the `ANSITerminalPane` subclasses can do, the next sections describe how to write a simple application for text-mode terminals.

9.3 InputEvent Class

Ctalk uses `InputEvent` objects to communicate between terminal input—from keyboards, for example, and mice, and serial tty devices—and the pane objects that you see on the screen. Ctalk uses `InputEvent` objects extensively, and you need to know about them to write a significant graphical app.

Text-mode displays use instances of `ANSITerminalStream` class to handle user input. The input can come from either a console that uses standard input and standard output, or from a tty device in the case of serial terminals.

When a program creates an `ANSITerminalStream` object, Ctalk initializes its input and output to the application's standard input and output channels. For the sake of simplicity, the following examples use standard input and output.

Text-mode displays use the `getCh` method (class `ANSITerminalStream` to receive characters and return them to the application. `ANSITerminalStream` also provides some basic terminal capabilities, and you can also use the class independently of a window.

Here is a brief example that echoes characters to the display.

```
int main () {
    ANSITerminalStream new term; /* Initializes input and output */
                                /* to stdin and stdout.          */

    Character new c;

    term rawMode;
    term clear;
    term gotoXY 1, 1;

    while ((c = term getCh) != EOF)
        term printOn "%c", c;

    term restoreTerm;
}
```

There are a few things to note about this example. The first is that the program sets the terminal to *raw* mode. That means the characters you type at the keyboard are received directly by the application, without any buffering or echoing by the terminal device itself. All terminal output is handled by the methods `clear`, `gotoXY`, and `printOn`.

It also means that the program *must* restore the terminal to its original state before exiting. That is the purpose of the `restoreTerm` method. If the application did not restore the terminal before exiting, the display would likely be unusable, so be sure to use `restoreTerm` whenever you use `rawMode` in your applications.

The `ANSITerminalStream` version of `printOn` notes what mode the terminal is in, so the program can display output regardless of whether the terminal is raw mode or not.

You should note that while `getCh` recognizes an EOF (`C-z`) character, in practice it is often difficult or impossible to enter an EOF from the keyboard.

The example above would have difficulty handling many control characters, and it doesn't recognize terminal escape sequences; for example, the type generated when you press a cursor key.

That is why applications that use panes use `InputEvent` objects. `InputEvent` objects contain information about what type (or *class*) of input is coming from the terminal, and the actual *data*.

In the case of terminal input, there are two event classes: `KBDCUR`, for terminal escape sequences, and `KBDKEY`, for alphanumeric keypresses.

The terminal stream queues these events, and the pane can retrieve them and process them in whatever manner it needs to.

Programs that use GUI displays don't actually need to use input events, but you will find that handling input beyond a simple press of the `Return` key can become quite involved. For an example, look at the `ANSITerminalPane` section of the language reference.

9.4 Using Queued InputEvents

If you're wondering how to go about adding all of this to a program, don't worry. All of the `ANSIWidgetPane` subclasses implement a method `handleInput`, which directs the pane's `ANSITerminalStream` to queue the events, and handles the events as it receives them.

In addition, all instances of `ANSITerminalPane` and its subclasses contain an instance variable (`paneStream`) that is an `ANSITerminalStream` object. You might recall from the previous sections that `ANSITerminalStream` initializes its input and output to `stdin` and `stdout`. Most applications need not handle that chore themselves.

If you plan to write widgets, you might need to see to the I/O initialization yourself. However, if the widget is a subclass of `ANSITerminalPane`, then the constructor in that class takes care of I/O initialization anyway.

For now, we'll just describe `ANSIWidgetPane`'s implementation of the `handleInput` method. Here it is, minus a few comments omitted for space reasons.

```
ANSIWidgetPane instanceMethod handleInput (void) {

    Integer new c;
    InputEvent new iEvent;

    while ((c = self paneStream getCh) != EOF) {
        iEvent become self paneStream nextInputEvent;
        switch (iEvent eventClass)
        {
```

```

    case KBDCHAR:
    case KBDCUR:
        switch (iEvent eventData)
        {
            case ESC:
                self withdraw;
                self paneStream restoreTerm;
                methodReturnNULL;
                break;
        }
        break;
    }
}
methodReturnNULL;
}

```

This method is very basic—it handles only one key sequence: *ESC*, which withdraws the pane from the display. This method also uses the `getCh` and `restoreTerm` methods that we mentioned earlier, with the pane’s `paneStream` instance variable as the receiver.

More involved implementations of `handleInput` also use the `openInputQueue` method from `ANSITerminalPane` to enable event queuing. You might want to look at the `handleInput` methods of other classes to see how this is accomplished.

9.5 The Widget API

Ctalk widget classes generally implement a set of methods that handle the common chores of displaying information and handling input. The methods listed here are a minimum set of methods. Widget classes can implement additional methods as needed.

cleanup Clean up any additional data used by the widgets. This includes the `paneBuffer` and `paneBackingStore` storage, and any additional storage. Deletion of the widget objects themselves is handled by the normal object cleanup mechanisms.

handleInput

Receive `InputEvent` objects from the receiver’s `paneStream`, and take action based on the keyboard input.

new Initializes the pane and its sub-panes, and sets whatever parameters are necessary.

parent Attach a sub-widget to its parent widget.

refresh Update the pane and its child panes on the display.

It’s also likely that a widget class needs to implement its own `map` and `unmap` methods, similar to those in `ANSITerminalStream` class.

Many `ANSITerminalPane` subclasses also implement a `show` method, which pops up the window at a specified `x`, `y` position on the terminal. Strictly speaking, however, `show` is a convenience method.

This section has not discussed adding shadows, borders, titles, and other graphical effects to pane objects. There are a lot of them, and they are described in each class's section of the language reference.

There is one point that we should note about graphics effects here, however. In order to keep the widget classes and their libraries as fast as possible, when refreshing subpanes, the terminal's graphics attributes (the graphical effects other than borders and shadows; i.e., bold, reverse, or blinking text, graphics character sets, and so forth) are *not* automatically updated when a parent window is refreshed. The program needs to update the subpanes individually. As in this example, from the `ANSITextBoxPane` `show` method.

```

self refresh;                /* Refresh the main widget. */
self dismissButton          /* Toggle the button's focus */
                             /* highlight. */
                             /*
    focusHighlightOnOff;
self dismissButton refresh; /* Refreshes the button widget */
                             /* independently. */

```

If the method did not independently refresh the receiver's `dismissButton` subwidget, the change to the button's focus highlight would not be visible.

Complex widgets that have more than one button or text input box need to keep track of the input focus and its display. There is a bit more about these issues in the next section.

9.5.1 Complex Widgets and Sub-Panes

Probably the easiest way to create a widget class is to modify an existing class, because these classes already contain implementations of the API methods described previously. These classes also demonstrate how the various panes of a complex widget interact.

The C libraries themselves do not prescribe a protocol for propagating events between widget objects. (That is not the case with other `Pane` subclasses.) So the text-mode widgets use a simple protocol for sharing events among widgets and subwidgets.

That is the task of the `parent` method. Depending on the needs of a particular class, the `parent` method causes subwidgets to inherit the I/O streams of the main widget, adds subwidgets to the parent's `children` list, and can add a `parentPane` reference to the subwidget if necessary. This is often the case when a widget needs to shift input focus among subwidgets when the user presses the `Tab` key.

Once again, probably the easiest way to implement a new widget class is to modify an existing class. Also look at the language reference, which contains program examples specific to each class.

9.6 Serial Terminals

Applications that handle input and output for serial terminals must also use the `openOn` and `setTty` methods of `ANSITerminalStream` class. Their use is relatively simple.

```

ANSIYesNoBoxPane new myWidget; /* Initialize stdin and stdout. */

myWidget paneStream openOn "/dev/ttyS0"; /* A Linux tty device. */
myWidget paneStream setTty 9600, 8, 'n', 1;

```

The `ANSIWidgetPane` subclasses can, but don't always, handle all of sequences defined by the ANSI, VT-100, or `xterm` standards. It's up to each class to determine how to handle the escape sequences it needs.

9.7 X11Pane Class

The `X11Pane` class allows you to display text in a X window and handle a set of X input events, both from the X server and the window manager.

`X11Pane` objects use the X window's default visual, but they also maintain a copy of the window's graphics context, so you can create your own visuals if necessary. The class is flexible enough that you should be able to create complex windows using the Xlib API if the class doesn't provide a specific feature that the application needs.

The `xhello.c` example program and the `X11Pane` section of the language reference describe the features that the `X11Pane` class implements right now. For completeness, here is the listing of `xhello.c`.

```
#define FIXED_CHAR_WIDTH 8
#define FIXED_CHAR_HEIGHT 12

X11Pane instanceMethod putCenteredText (String text) {
    Integer new sizeX;
    Integer new sizeY;
    Integer new textCharWidth;
    Integer new textXSize;
    Integer new textYSize;

    sizeX = self size x;
    sizeY = self size y;
    textCharWidth = text length;

    textXSize = textCharWidth * FIXED_CHAR_WIDTH;
    textYSize = FIXED_CHAR_HEIGHT;

    self putStrXY (sizeX / 2) - (textXSize / 2),
        (sizeY / 2) - (textYSize / 2),
        text;
    methodReturnNULL;
}

int main () {
    X11Pane new xPane;
    InputEvent new e;
    Integer new nEvents;
    xPane initialize 25, 30, 200, 100;
    xPane map;
    xPane raise;
    xPane openEventStream;
}
```

```

WriteFileStream classInit;

xPane setWMTitle "xhello";
xPane putCenteredText "Hello, world!";

while (TRUE) {
    xPane inputStream queueInput;
    if (xPane inputStream eventPending) {
        e become xPane inputStream inputQueue unshift;
        switch (e eventClass value)
        {
            case CONFIGURENOTIFY:
                xPane putCenteredText "Hello, world!";
                stdoutStream printOn "ConfigureNotify\t%d\t%d\t%d\t%d\n",
                    e eventData1,
                    e eventData2,
                    e eventData3,
                    e eventData4;
                stdoutStream printOn "Window\t\t%d\t%d\t%d\t%d\n",
                    xPane origin x,
                    xPane origin y,
                    xPane size x,
                    xPane size y;
                break;
            case EXPOSE:
                xPane putCenteredText "Hello, world!";
                stdoutStream printOn "Expose\t\t%d\t%d\t%d\t%d\t%d\n",
                    e eventData1,
                    e eventData2,
                    e eventData3,
                    e eventData4,
                    e eventData5;
                break;
            case WINDELETE:
                xPane deleteAndClose;
                exit (0);
                break;
            default:
                break;
        }
    }
}
}
}
}

```


10 Debugging

Ctalk is still in development, and does not yet have all of the debugging facilities of other object oriented languages.

Apart from debugging the internals of the Ctalk front end and libraries, which is described in the *Ctalk Language Reference*, there are a few classes and methods that programs can use to display diagnostic information when they are run.

10.0.1 Printing Call Sequences for Exceptions

The method `enableExceptionTrace` in class `Object` enables the printing of the program's call stack to the console's standard error channel when a program deals with an exception using `handle` (class `Exception`).

A program can turn off the display of the call stack with the `disableExceptionTrace` method, also in class `Object`. The method `traceEnabled` in class `Object` returns `TRUE` or `FALSE` depending on whether call stack displays are enabled.

A program can print an exception's call stack using `printExceptionTrace` in class `Exception`. Here is the code from the `handle` method that calls `printExceptionTrace`.

```
    if (self traceEnabled) {
        self printExceptionTrace;
    }
```

10.0.2 Printing Objects

The method `printSelf` (class `Object`) prints its receiver to the program's standard error channel. For example, if a program contains the following lines:

```
    ReadFileStream new inputStream;
    ...
    inputStream openOn "myFile";
    ...
    inputStream printSelf;
```

Then it will print the class and value of `inputStream` and all of its instance variables. Remember that `openOn` (class `ReadFileStream`) calls `statStream` (in class `FileStream`) to fill in the stream object's instance variables.

If a program needs to print an object that is not within the scope of a method or function, it can call the library function `__ctalkPrintObjectByName` directly. The argument is a string containing the name of the object you want to print. `__ctalkPrintObjectByName` looks up the object before printing it.

11 GNU Free Documentation License

Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute example copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you."

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied example, or with

modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque."

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For

works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. EXAMPLE COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as example copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy

of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant

Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page.

If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements." Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end

of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications." You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for example copying of each of the

documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding example copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other

attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) YEAR YOUR NAME.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License."

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we

recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

12 Index

&

& method (**Character** class) 10

+

+ (**Character** class) 10

+ method (**String** class) 11

-

- (**Character** class) 10

=

= method (**Character** class) 16

= method (**Integer** class) 16

= method (**String** class) 11, 16

== method (**String** class) 11

^

^ method (**Character** class) 10, 44

_

__ctalk_to_c_char_ptr() function 46

__ctalk_to_c_int() function 46

__ctalkInlineMethod function 14

__ctalkInstanceVarsFromClassObject function
..... 38

|

| method (**Character** class) 10

A

ANSIMessageBoxPane 49

argc C variable 6

Arguments, class of 28

argv C variable 6

Array class 9, 12

Array Class 31

asCharacter method (**Magnitude** class) 15, 45

asInteger method (**Magnitude** class) 15, 45

asLongInteger method (**Magnitude** class) 15

AssociativeArray Class 31, 33

asString method (**Magnitude** class) 11

at method (**Array** class) 6

at method (**Array** class) 12, 43

at method (**AssociativeArray** class) 34

atPut method (**Array** class) 12

atPut method (**AssociativeArray** class) 34

B

become method (**Object** class) 16

bitComp method (**Integer** class) 9

C

CGIApp class 41

char C data type 9

Character class 9

Class 1

Class hierarchy 1, 3

class method (**Object** class) 35, 43

Class variable 2

classInit method (**ReadFileStream** class) 19

classInit method (**WriteFileStream** class) 19

classMethod keyword 44

classMethod method (**Object** class) 43

classVariable method (**Object** class) 36, 43

cleanup instance method (**ANSIWidgetPane** class)

..... 52

clear instance method (**ANSITerminalStream**

class) 50

Collection Class 31

Compound statements 15

Constructor 2

Constructors 9

copy method (**Object** class) 16

CTALK_DIRECTORY_MODE macro 20

ctcc command 5, 6

ctpath example program 5

D

Debugging 57

Default method 16

Directories 20

DirectoryStream class 17, 20

Dispatch methods 24

double C data type 9

E

eval keyword 24

F

FileStream class 17

FileStream Class 31

float C data type 9

Float class 9

G

<code>getCh</code> instance method (<code>ANSITerminalStream</code> class)	50, 52
<code>getValue</code> method (<code>Collection</code> class)	34
GNU Free Documentation License	59
<code>gotoXY</code> instance method (<code>ANSITerminalStream</code> class)	50
Graphics	49

H

<code>handleInput</code> instance method (<code>ANSITerminalPane</code> class)	51
<code>handleInput</code> instance method (<code>ANSIWidgetPane</code> class)	51, 52
hello example program	5

I

Inheritance	2
Instance variable	2
<code>instanceMethod</code> keyword	44
<code>instanceMethod</code> method (<code>Object</code> class)	43
<code>instanceVariable</code> method (<code>Object</code> class)	35
<code>instanceVariable</code> method (<code>Object</code> class)	43
<code>Integer</code> class	9
<code>invert</code> method (<code>Integer</code> class)	9
<code>isNull</code> method (<code>String</code> class)	42

K

Key class	33, 34
-----------------	--------

L

Library search path	39
<code>List</code> Class	31
<code>localTime</code> method (<code>Time</code> class)	6
<code>localtime(3)</code> C function	6
long int C data type	9
long long int C data type	9
<code>LongInteger</code> class	9

M

<code>Magnitude</code> class	15
<code>map</code> (<code>Array</code> class)	13, 14
<code>map</code> (<code>AssociativeArray</code> class)	14
<code>map</code> instance method (<code>ANSITerminalPane</code> class)	52
<code>map</code> (<code>List</code> class)	14
<code>map</code> method (<code>AssociativeArray</code> class)	34
<code>map</code> method (<code>List</code> class)	33
Message	1
Method	1
Method declarations	44
<code>methodReturnFalse</code> macro	48

<code>methodReturnInteger</code> macro	47
<code>methodReturnTrue</code> macro	48
Methods	41
<code>mkdir</code> method (<code>DirectoryStream</code> class)	20

N

<code>new</code> instance method (<code>ANSIWidgetPane</code> class)	52
<code>new</code> method (<code>Object</code> class)	2, 9, 43
<code>new</code> method (<code>WriteFileStream</code> class)	2, 23

O

<code>OBJECT</code> typedef	46
<code>openInputQueue</code> instance method (<code>ANSITerminalStream</code> class)	52
<code>openOn</code> instance method (<code>ANSITerminalStream</code> class)	53
<code>openOn</code> method (<code>ReadFileStream</code> class)	17
Operator overloading	2

P

<code>Pane</code> class	49
Panels	49
<code>parent</code> instance method (<code>ANSIWidgetPane</code> class)	52
<code>parseQueryString</code> method (<code>CGIApp</code> class)	41
Polymorphism	2
<code>pop</code> method (<code>List</code> class)	31
Primitive methods	43
<code>printf(3)</code> C function	19
<code>printOn</code> instance method (<code>ANSITerminalStream</code> class)	50
<code>push</code> method (<code>List</code> class)	31

R

<code>readMode</code> instance method (<code>ANSITerminalStream</code> class)	50
<code>readAll</code> method (<code>ReadFileStream</code> class)	17, 18
<code>readChar</code> method (<code>ReadFileStream</code> class)	17
<code>ReadFileStream</code> class	17
<code>ReadFileStream</code> Class	31
<code>readLine</code> method (<code>ReadFileStream</code> class)	17, 18
<code>readRec</code> method (<code>ReadFileStream</code> class)	17
Receiver	1
<code>refresh</code> instance method (<code>ANSIWidgetPane</code> class)	52
<code>require</code> keyword	35, 38
<code>restoreTerm</code> instance method (<code>ANSITerminalStream</code> class)	50, 52
<code>rmDir</code> method (<code>DirectoryStream</code> class)	20

S

<code>self</code>	23
-------------------------	----

self, class of 24
setTty instance method (**ANSITerminalStream** class) 53
shift method (**List** class) 31
size method (**Array** class) 12
split method (**String** class) 6
split method (**String** class) 11, 42
stderr 21
stderrStream class variable (**WriteFileStream** class) 19
stderrStream (**WriteFileStream** class) 21
stdin 21
stdinStream class variable (**ReadFileStream** class) 19
stdinStream (**ReadFileStream** class) 21
stdout 21
stdoutStream class variable (**WriteFileStream** class) 19
stdoutStream (**WriteFileStream** class) 21
strcat(3) C function 11
strcmp(3) C function 11
strcpy(3) C function 11
Stream class 49
Stream Class 31
streamEof method (**FileStream** class) 17
String class 9, 11
Subclass 1
substring method (**String** class) 12
super 23
super keyword 23
Superclass 1, 3

T

toLowerCase (**Character** class) 10
toLowerCase method (**Character** class) 44
toUpperCase (**Character** class) 10
toUpperCase method (**Character** class) 44
Type conversion 15

U

unmap instance method (**ANSITerminalPane** class) 52
unshift method (**List** class) 31

V

value instance variable 35
value message 45
value method (**Object** class) 16
Variable promotion 15

W

writeChar method (**WriteFileStream** class) ... 17
WriteFileStream class 17
WriteFileStream Class 31
writeStream method (**WriteFileStream** class) 17, 18

X

X11Pane class 54

