# ReyScript

# Programmer’s Guide

Concept Version: February 6, 2002

## TABLE OF CONTENTS


1 INTRODUCTION 4

1.1 About this Manual 5

2 BASICS 7

2.1 Syntax 7
2.2 Blocking and Scopes 9
2.3 Static and Dynamic Variables 10
2.4 Commenting 10
2.5 Style 10

3 RESIDENT CLASSES 11

3.1 Number 11
3.2 String 13
3.3 Flag 19
3.4 Date 19
3.5 Stream 22

4 PROGRAM FLOW 23

4.1 Initialization 23
4.2 Termination 24
4.3 If-Then-Else 25
4.4 Switch 26
4.5 Looping 27
4.6 Function Calling 29
4.7 Exception Handling 31
4.8 Macros 32

5 OPERATORS 33

5.1 Precedence and Associativity 33
5.2 Assignment 34
5.3 Arithmetic 35
5.4 Relational 37
5.5 Logical 38

6 OBJECT-ORIENTED PROGRAMMING 39

6.1 Public Members 40
6.2 Constructors and Destructors 40
6.3 Deriving Classes 42
6.4 Abstraction 43

7 SHELL SCRIPTING 44

**7.1 Files and Commands** **44**
**7.2 Streams** **45**

**8 REYSCRIPT RUNNER** **50**

**8.1 Debugger** **51**

**9 ERRORS** **53**

# 1 INTRODUCTION

Software programs come in different shapes and sizes. Those of the big and tall variety are usually written in powerhouse languages such as C++ and Java. The petite ones can be done using an interpretive language such as Bourne/Korn shell or Perl. Unlike the big programs, performance isn't a critical issue with the little ones, so you can sacrifice some execution speed for development turnaround. After all, it usually takes fewer lines of code to implement something in shell than in C++.

Whereas the shell's strength lies in its I/O facility, its weakness lies in the robustness of its data structures. I wanted to find a language maintains the scripting aspect of a shell language yet improve on data storage, which can be achieved by incorporating an object-oriented methodology. I haven't heard of such a language out there, so I decided to give it a shot and create one myself -- hence ReyScript was the end result.

ReyScript is not supposed to replace the big-league languages like C++ or Java, only to supplement it. The heavy-duty applications must still be written in those languages, but small scripts or programs you considered using shell for can be written instead in ReyScript. Its simplified approach to some of the language's concept makes ReyScript easier to use than C++ or Java, but it also limits the language's flexibility and performance. I see ReyScript more as an alternative to shell and Perl – for one thing, if you're a C++ or Java programmer you would have to learn a totally different syntax if you want to use those other scripting languages. With ReyScript, the object-oriented syntax is nearly identical, making it quicker and easier to transition to.

So let's get started. I think the best way to demonstrate a new language is to jump right in with a few samples. Let's start with the customary "Hello World" program:

*Sample 1-1: Hello World*

```
/* This is Hello World */

main()
{
    write "Hello World\n";  // I'm writing to standard output
}
```

It's as simple as that. For you C++ and Java programmers, the basic syntax is the same: statements are terminated with a semi-colon, curly braces are used for denoting scope, and comments can be specified in block (/*,*/) or single-line (//) notation. In addition, every ReyScript program must define a *main* function in the *workspace* (also known as the *global class*). The *write* statement (along with its twin *read*) serves as the key I/O mechanism in the language. We'll discuss this in detail in chapter 7.

Let's add some more stuff to this program:

*Sample 1-2: Hello World Part 2*

```
number a;

main()
{
    number b = 5;

    while(a<=b)
    {
        write "Hello World " + a + "\n";
        a++;
    }
}
```

In this example, we see the use two of the language's primitive classes: *number* and *string*. The variable "a" is defined as a number, but what is not clearly seen is that the constants "Hello World" and "\n" are resolved into string objects during execution. We'll discus these classes in detail in chapter 3. Here's how the output to the program above.

```
Hello World 1
Hello World 2
Hello World 3
Hello World 4
Hello World 5
```

Our last sample in this section will show how classes can be used in conjunction with the read/write statements. This is just the tip of the iceberg. All will be revealed in chapter 7.

*Sample 1-3: Hello You*

```
class user
{
    string username;

    ~read(stream x)
    {
        write ("What is your name? ") to x;
        read (username) from x;
    }

    ~write(stream x)
    {
        write ("Hello " + username + "\n") to x;
    }
}

main()
{
    user a;

    read a;
    write a;
}
```

Here's how the output might look:

```
What is your name? Dave
Hello Dave
```

## 1.1 About this Manual

This document describes the ReyScript language from syntax to program arguments. A language by itself is merely a specification on how to code a program. It is then implemented through the *ReyScript Runner* program which is a parser, interpreter , and debugger. Chapter 8 will discuss how to use the program, but during the course of reading this manual, you will see parts of the program used often. The *parser* reads in code in a file, translates into binary data, and loads it into memory. The *interpreter* and *debugger* execute the program by processing the binary data. The difference between the two is that with the debugger you can diagnose your program and identify any errors or bugs in it.

This manual is written for an audience with C/C++/Java background. With this assumption, I can quickly describe the nuances of the languages without starting from scratch. If I did, this manual would probably be five times bigger. For those who don't have this background, I hope the samples shown here will help you get started. Hopefully, this can help you transition into C++ and Java quicker and less painful.

# 2 BASICS

If you know C++ and/or Java, then the stuff in this chapter won't be new to you. Basically, the ReyScript syntax is the same as C++ or Java, however the terms and lingo might be a little bit different (after all, I created the language so I can put my own spin on it – when you write your own language, you can do the same).

To begin with, let's look at another "Hello World" sample:

```
main()
{
    string me = "Bob";
    write me + " says Hello World\n";
}
```

Every ReyScript source file is made up of *atoms*, which is a character or group of characters that represents the smallest unit of code the parser reads in. Here is the list of atoms (there are fifteen of them) for the above code:

```
main
(
)
{
string
me
=
"Bob"
;
write
me
+
" says Hello World\n"
;
}
```

The parser then determines if a atom is a constant, operator, command, delimiter, or symbol. *Constants* are numbers or strings used as data. The "Hello World\n" is a string constant. *Operators* "=" and "+" perform special tasks -- they will be covered in Chapter 5. The "write" symbol is a *command* or reserved word, which specifies a type of *construct* or a way for the interpreter to process the following atoms. *Delimiters* are a way separate groups of symbols into elements. The symbols "(", ")", "{", "}", and ";" are all delimiters. Finally, *symbols* such as "main", "string", and "me" are used to identify these elements.

So what are *elements*? Well, there are four basic elements of any ReyScript program: statement, function, class, and variable. Every atom in the program is associated with one or more of these elements. A *variable* first and foremost stores information. A *statement* is an instruction to do something. Information stored in variables usually affects the way a statement is processed. A *function* contains a set of statements to perform a task. At this level you define variables that the statements in the function can access. Finally, a *class* is a collection of related functions and variables shared by those functions. To come full circle, a class serves as the blueprint to a variable.

In the above example, "main" is a function containing two statements (that are ended with the ";" delimiter) and one variable called "me" which is an instance of class "string". The main function is actually part of the global class, which is instantiated when the program is executed.

## 2.1 Syntax

Like any other programming language, there are rules on how to write code in ReyScript. These rules or *syntax* is similar to C++/Java. Here is a rundown of some of the fundamental syntax:

### 2.1.1 Atom Sizes

Delimiters are made up of only a single character while operators can be one or two characters in length. Commands are never more than nine characters in length, and constants and symbols can have at most 255 characters.

### 2.1.2 Constants

Numeric constants can only contain digits 0-9. You can add a "." for floating-point numbers. String constants must begin and end with double-quote delimiters. All characters within the delimiters become part of the string, except for escape sequences which will be substituted with special characters (see section 3.2).

### 2.1.3 Symbols

Symbols can contain any character not used for operators or delimiters. Unlike C++/Java, you can begin symbols with a numeric digit.

### 2.1.4 End of Statement

All statements must end with a termination delimiter (";") so that the parser can recognize the beginning and end of a statement.

### 2.1.5 Variable Declaration

A variable declaration, which is a type of statement, must begin with a class symbol followed by one or more symbols identifying the new variable instantiation of that class. Multiple variables can be declared using the "," delimiter. You can initialize a variable using the "=" operator or calling the constructor of the class with the appropriate argument. Here are some examples of variable declarations:

```
number a=1, b=0;
userclass c, d("Me");
```

### 2.1.6 Sub-References

To access a variable or function of an object, use the sub-reference operator ("."). The member you're trying to accessed must be declared with the `public` keyword (see section 6.1). Some examples of accessing members include:

```
a.x = 1;
a.func1();
```

### 2.1.7 Function Definition

The syntax for a function definition is as follows:

```
[return_type] function_name([[argtype1 arg1][, argtype2 arg2]...]) body...
```

If the function returns a value, then specify the *return_type* which is a class symbol. The *return_type* tells what kind of value to expect from the function. The *function_name* is a symbol identifying the function. It follows the *return_type* or starts the definition if the function doesn't return a value. The function can be defined with zero or more parameter variables. Each variable must be declared with the class type followed by the variable symbol. If more than one parameter is needed, use the "," delimiter to separate each parameter declaration. Here are some examples of function definitions:

```
main()
```

```
number minmax(number x, number y)
```

The body of the function can contain one or more statements. If the body contains more than one statement, you must enclose them in a function block (see section 2.2).

### 2.1.8 Main Function

Every ReyScript program must have a “main” function defined. You can only specify a number for the return type; this serves as the exit status of the program. You cannot defined parameters for this function. However, you can use program parameter or shell variables to pass data into the program (see section 4.1.1).

### 2.1.9 Class Definition

The syntax for a class definition is as follows:

```
class class_name [base baseclass_name]
```

The *class_name* serves as the symbol for the class which can be derived from another class specified by *baseclass_name*. Chapter 6 will cover classes in detail.

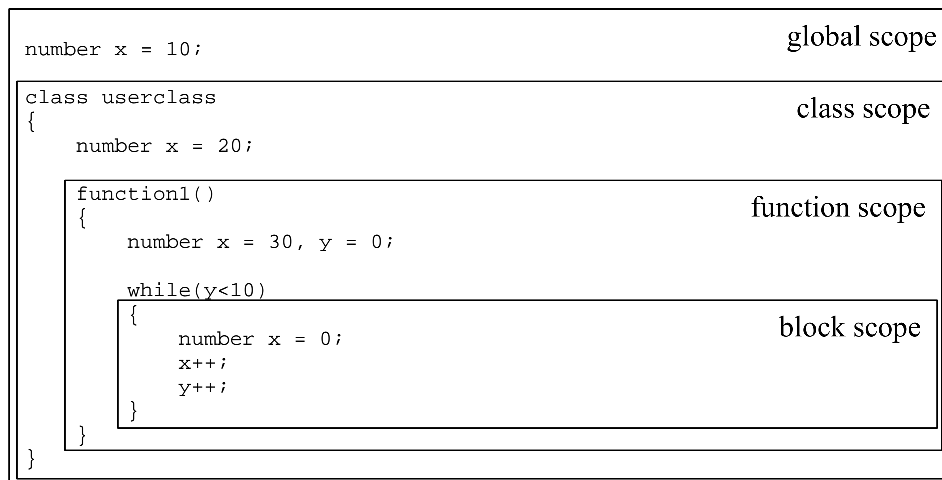
## 2.2 Blocking and Scopes

You can group elements into blocks using the begin-block (“{“) and end-block (“}”). Blocking is used to extend a construct, meaning that you can substitute one statement for a block of statements. Chapter 4 will discuss which constructs you can extend with blocks.

Variables may be only accessible to a certain part of the program. In other words, the variable has a *scope* in which it applies to. By creating a block in the code, you inherently define a new scope for variables as well. Therefore, if you declared a variable in a block, all classes, functions, and statements defined within that block can use it as well as any nested blocks. There are different levels of scopes, of which a variable with *global scope* is accessible anywhere in the program. Member variables of a class have *class scope* meaning that they can be accessed by any subclass or member function within that class. Variables with *function scope* can be accessed anywhere in the function. Finally, variables with *block scope* can be accessed within that block and any nested block.

Having different scopes allows you to identify different variables using the same name. The figure below shows how “x” can be used to refer to four different number variables, each defined in a different scope.

*Figure 2-1: Scopes*



As to which variable is accessed depends on the current scope. Let's say the "main" function be being executed. If "x" is not redeclared within the function, then any references to "x" will access the global variable. On the other hand, the "x" variable referenced in the while-loop accesses the variable declared in that block. All the other "x" variables are not altered from inside this block.

## 2.3 Static and Dynamic Variables

A big limitation in ReyScript is that there is no dynamic memory allocation -- all variables are declared statically with memory already allocated to them. Because of this, there are no pointers and no memory allocation commands such as `new` and `delete`; these are things you definitely need in C++/Java. As to why ReyScript doesn't have this, let's remember its place in the programming universe. When it comes to scripting or small-scale programs, I just don't see the need for dynamic variables. For programs that do need them, then they should be written in C++ or Java.

## 2.4 Commenting

There are times you need to make a note about what you just coded to yourself or to other people who wants to read your program. You can add *comments* to your program either as a one-liner or a whole block. To add just one comment line, you can insert a double slash "//" before the start of your comment. The ReyScript parser will then assume the rest of the line is a comment and ignore it, but will begin processing the following line. If you want to add a much bigger comment, one that takes up several lines, you can either insert the double slash in front of every line, or simply insert comment block delimiters "/*" and "*/" at the beginning and end of your comment body respectively. Again, the ReyScript parser will ignore everything between the delimiters.

Unlike the comment line syntax, you can write code immediately following a comment block on the same line. There's nothing wrong with this, it's just an issue of style and readability. Generally you want the comments to stand out and been seen by you and your fellow programmers, so writing code in this way is usually not done.

Here are some examples of comments:

*Sample 2-1: Comments*

```
/* This is a comment block
   so these first three lines are ignored by the parser
*/

main()
{
    number a;  // This is a comment line...
               // so is this one...

    /* this one too, but the next statement
    can be processed */  write a;
}
```

## 2.5 Style

All programmers have their own way of writing code in terms of how much spacing to put between statements, what kind of naming convention to use on classes, functions, and variables, and where to put commenting. There's no right or wrong way to these issues, but most companies develop a coding standard that their programmers must adhere to, so in a way, they follow a corporate style.

I have my own style which I adopted from working for previous companies and applied a little bit in this manual. You don't have to write code in this way. Just remember that when developing your own style of coding, readability is a key factor in the code's aesthetics.

# 3 RESIDENT CLASSES

There are several classes built into the language known as *resident classes*. There are two flavors of resident classes: primitive and compound. *Primitive classes* are used for storing a single and most basic form of data. On the other hand, *compound classes* can either store more than one piece of information or perform more complex duties. There are four primitive classes (number, string, flag, and date) and a compound one (stream).

A big difference to remember between C++/Java and ReyScript is that there are no so-called "simple data types" in ReyScript. Every variable is an instance of some class, whereas in the other languages, you can simply allocate memory to store data. Try not to confuse "simple data types" for primitive classes -- although both are used to store data, primitive classes have added functionality as detailed in the following sections.

## 3.1 Number

Number object stores (what else) a number. But what you may not know is that it stores either an integer or a floating-point value. For you C++/Java users, this is equivalent to an "int" and "float" type all rolled into one. The class determines which type to use based on the operations performed on the variable. You can simply force a variable to store a float by using another float in an operation. To force a value from float to integer, simply use the *round()* function. For example:

```
number a, b, c, d;    // all variables are initialized to 0

a = 10;    // this is an integer
b = 20.0;   // this is a float

c = 10 * 2;      // result is an integer = 20
c = 10 * 2.0;    // result is a float = 20.0

c.round();       // result is an integer = 20

if (c == b)      // result is true
    d = 1;
else
    d = -1;
```

At the end of this sample code, two numbers are being compared: one with a value of 20, the other with 20.0. In this case, the two values are equal and the variable d will be set to 1. However if the value of c is set to 19.9, then the result would be false and d would be set to -1. Also, a number evaluates to false if the value is 0 or 0.0, and true otherwise.

### 3.1.1 Number Functions

#### number.abs

| | |
|---|---|
| Syntax | `number abs()` |
| Parameters | None |
| Return | The variable itself with the new value |
| Description | Sets the variable to the absolute value. |

**number.cos**

| | |
|---|---|
| Syntax | `number cos()` |
| Parameters | None |
| Return | The variable itself with the new floating-point value |
| Description | Sets the variable to the cosine of its value which is in radians. |

**number.exp**

| | |
|---|---|
| Syntax | `number exp()` |
| Parameters | None |
| Return | The variable itself with the new floating-point value |
| Description | Sets the variable to the exponential(*e*) of the value. |

**number.log**

| | |
|---|---|
| Syntax | `number log()` |
| Parameters | None |
| Return | The variable itself with the new floating-point value |
| Description | Sets the variable to to the natural logarithm(*ln*) of the value. |

**number.pow**

| | |
|---|---|
| Syntax | `number pow(number exponent)` |
| Parameters | *exponent* - number to raise the value with |
| Return | The variable itself with the new floating-point value |
| Description | Raises the value of the number to the power specified by *exponent*. |

**number.round**

| | |
|---|---|
| Syntax | `number round()` |
| Parameters | None |
| Return | The variable itself with the new value |
| Description | Sets the variable to an integer type. If the value was a float, it is first rounded down. |

### number.sin

Syntax `number sin()`

Parameters None

Return The variable itself with the new floating-point value

Description Sets the variable to the sine of the value which is in radians.

### number.tan

Syntax `number tan()`

Parameters None

Return The variable itself with the new floating-point value

Description Sets the variable to the tangent of the value which is in radians.

## 3.2 String

This class is similar to the standard string class of C++ and Java. The class manages its memory use, so you don't have to worry about allocating and freeing space for the characters. Space for characters is increased when the size of the string increases, however it is not decreased if the string becomes smaller. Only when the string goes out of scope is memory freed entirely. Strings evaluate to false if they are empty, and true otherwise.

String can contain any one of 255 characters define by ASCII. Most of the characters you will need can be represented by a keystroke. Others, such as the line feed has to be entered using an escape sequence (those starting with an "\"). See the chart below for a list of possible escape sequences defined by ANSI.

*Table 3-1: Escape Sequences*

| Escape Sequences | Name | ASCII Hex Code | Description |
|---|---|---|---|
| `\a` | bell | 07 | rings a bell |
| `\b` | backspace | 08 | deletes a character |
| `\f` | form feed | 0C | printer page break |
| `\n` | newline | 0A | line break |
| `\r` | carriage return | 0D | DOS line break, use in front of line feed |
| `\t` | tab | 09 | same effect as 8 spaces |
| `\'` | single quote | 27 | |
| `\"` | double quote | 22 | |
| `\\` | backslash | 5C | |
| `\xx` | hexadecimal | 00-FF | generate any ASCII character |

Of the four primitives, strings are most likely to be used for scripting or interfacing with other programs. With that in mind, ReyScript provides some facilities to simplify some of those tasks. The next sections will describe some things you can do with them.

### 3.2.1 Formatting

If you want to generate a string using information from other variables, you can use the *format()* function. This is borrowed from the *printf* function of C where you define a format string containing *specifications*, which are

command sequences starting with a "%". Each specification instructs the function how to display the data from another variable. A format specification which consists of optional and required fields, has the following syntax:

```
%[flags][width][.precision]type
```

*Flags* are one or more characters that control justification of output and printing of signs, blanks, decimal points, and octal and hexadecimal prefixes. *Width* specifies the minimum number of characters to display, whereas *precision* specifies the maximum number of characters or the minimum number of digits printed for integer values to display. Finally, the required field *type* determines whether the associated argument is interpreted as a string, or a number. The following tables list the possible values for each field.

*Table 3-2: String Format Types*

| Type | Class | Description |
|---|---|---|
| d | number | decimal integer |
| o | number | octal integer. |
| x | number | hexadecimal integer, using "abcdef." |
| X | number | hexadecimal integer, using "ABCDEF." |
| e | number | Has the form [ – ]d.dddd e [sign]ddd where d is a single decimal digit, dddd is one or more decimal digits, ddd is exactly three decimal digits, and sign is + or –. |
| E | number | Identical to the "e" format except that E rather than e introduces the exponent. |
| f | number | Has the form [ – ]dddd.dddd, where dddd is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision. |
| g | number | Printed in "f "or "e" format, whichever is more compact for the given value and precision. The e format is used only when the exponent of the value is less than –4 or greater than or equal to the precision argument. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it. |
| G | number | Identical to the "g" format, except that E, rather than e, introduces the exponent where appropriate. |
| s | string<br>flag<br>date | Characters are printed up to the first null character or until the precision value is reached. |

*Table 3-3: String Format Flags*

| Flag | Description | Default |
|---|---|---|
| – | Left align the result within the given field width. | Right align. |
| + | Prefix the output value with a sign (+ or –) if the output value is of a signed type. | Sign appears only for negative signed values (–). |
| 0 | If width is prefixed with 0, zeros are added until the minimum width is reached. If 0 and – appear, the 0 is ignored. If 0 is specified with an integer format (d, o, x, X) the 0 is ignored. | No padding. |
| (space) | Prefix the output value with a space if the output value is signed and positive; the blank is ignored if both the blank and + flags appear. | No space appears. |
| # | When used with the o, x, or X format, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively. | No space appears. |

| Flag | Description | Default |
|---|---|---|
| # (cont) | When used with the e, E, or f format, the # flag forces the output value to contain a decimal point in all cases. | Decimal point appears only if digits follow it. |
| | When used with the g or G format, the # flag forces the output value to contain a decimal point in all cases and prevents the truncation of trailing zeros.<br>Ignored when used with d or s. | Decimal point appears only if digits follow it. Trailing zeros are truncated. |

*Table 3-4: String Format Precision*

| For types | Description | Default |
|---|---|---|
| d, o, x, X | The precision specifies the minimum number of digits to be printed. If the number of digits in the argument is less than precision, the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds precision. | 1 |
| e, E | The precision specifies the number of digits to be printed after the decimal point. The last printed digit is rounded. | Default precision is 6; if precision is 0 or the period (.) appears without a number following it, no decimal point is printed. |
| f | The precision value specifies the number of digits after the decimal point. If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits. | Default precision is 6; if precision is 0, or if the period (.) appears without a number following it, no decimal point is printed. |
| g, G | The precision specifies the maximum number of significant digits printed. | Six significant digits are printed, with any trailing zeros truncated. |
| s | The precision specifies the maximum number of characters to be printed. Characters in excess of precision are not printed. | Characters are printed until a null character (end of string) is encountered |

The code below shows how to create a string from with two numbers. The format function comes in handy when generating reports allowing you to concatenate data together much quicker than say writing each piece of the report one at a time.

*Sample 3-1: Product Cost*

```
main()
{
    string priceStr;  // value initialized to empty string ""
    number quantity, unit_price;

    quantity = 5;
    unit_price = 3.75;

    priceStr.format("For %d items at $%.2f each, the total price is
$%.2f.\n",
        quantity, unit_price, quantity * unit_price);

    write priceStr;
```

```
}
```

Output:

```
For 5 items at $3.75 each, the total price is $18.75.
```

### 3.2.2 Parsing

Strings are mainly used when reading in lines from a file or another process. Sometimes there are several pieces of information that must be extracted from the string and copied to other variables. If you know the location of the data you want to extract, use the *substr()* function which will create a another string from the source string based on the starting position and size that you give it.

If the source string contains variable-width fields, use the *token1()* and *token2()* functions. Typically, a line with multiple fields varying in width uses a delimiter to distinguish data of one field from the next. The token1() function is similar to the C *strtok()* which breaks the string at the delimiters and returns each field as *tokens* one at a time. You know that you're out of tokens when the function returns an empty string. The token2() function is similar to token1() except that it can handle empty fields. When two adjacent delimiters are encountered, token1() treats it as a single delimiter as skip over it; however, token2() treats it as a field with no value. When using token2() you must also use the *eol()* function to detect when the end-of-line is reached since an empty string return by token2() can be actual field data.

To tokenize the string, both functions replace the delimiters with NULL values so that each token is terminated properly. Once the first token is retrieve, the string will only display at most the first token, even though there may data following it. Also, an internal marker is kept within each string object. The marker is reset if string has been assigned or concatenated, so you will restart the parsing from the beginning of the string again. To be safe, do not alter the string until you finish reading all of the tokens.

The following sample code shows the difference between the two token functions:

*Sample 3-2: Tokenizer*

```
main()
{
    string record;
    string field;

// this will use the token1 function
    record = "abc-def-g-hi--klm";
    while(field = record.token1("-"))
        write "The next token1 field is " + field + "\n";

    write "\n";

// this will use the token2 function
    record = "abc-def-g-hi--klm";
    while(!record.eol())
    {
        field = record.token2("-");
        write "The next token2 field is " + field + "\n";
    }
}
```

Output:

```
The next token1 field is abc
```

```
The next token1 field is def
The next token1 field is g
The next token1 field is hi
The next token1 field is klm

The next token2 field is abc
The next token2 field is def
The next token2 field is g
The next token2 field is hi
The next token2 field is
The next token2 field is klm
```

### 3.2.3 String Functions

#### string.eol

| | |
|---|---|
| Syntax | `flag eol()` |
| Parameters | None |
| Return | true if end-of-line is reached, false otherwise |
| Description | Use this function with *token2()* to see if anymore tokens can be retrieved from the string. |

#### string.find

| | |
|---|---|
| Syntax | `number find(string` *`searchStr,`* `number` *`pos`*`)` |
| Parameters | *searchStr* - string to search for<br>*pos* - starting position for the search |
| Return | number >= 0 if found, -1otherwise |
| Description | Returns the position of the first occurrence of the search string. Search is initiated at the given position. First character in the string is considered position 0. If no match was found, the function returns -1. |

#### string.format

| | |
|---|---|
| Syntax | `string format(string` *`formatStr`*`, ?)` |
| Parameters | *formatStr* - string containing specifications to create string<br>? - 0 or more primitive variables containing values inserted into the string |
| Return | The variable itself containing the newly formed string |
| Description | Generates string based on the format string and optional data. formatStr can contain format specifications in the following syntax:<br><br>%[flags][width][.precision]type<br><br>For descriptions of each of the field specifiers, refer to tables 3-2 for flags, 3-3 for precision, and 3-4 for types. |

### string.length

Syntax `number length()`

Parameters None

Return A 0 for an empty string or non-zero otherwise

Description Returns length of the string.

### string.reserve

Syntax `string reserve(number size)`

Parameters size - amount of space to reserve

Return The variable itself

Description Reserves memory for string storage. Use this function to allocate large amount of memory prior to using the string for major operations. This prevents the string from having to allocate memory as you are building the string.

You can also free the memory used by specifying a size of 0. The string is then set to an empty string.

### string.sar

Syntax `string sar(string searchStr, string replStr)`

Parameters *searchStr* - string to search for
*replStr* - string to replace if searchStr was found

Return The variable itself

Description Replaces all occurrences of searchStr with replStr in the given string. If replStr was an empty string, then all occurrences of searchStr are deleted.

### string.substr

Syntax `string substr(number pos, number size)`

Parameters *pos* - starting position of string
*size* - size of new string

Return A newly created string

Description Copies string from the given position into a new string. First character in the string is considered position 0. If either argument is a non-positive number, then the function returns an empty string. If pos + size exceeds the length of the string, then only characters up to the terminating NULL are copied.

### string.token1

Syntax `string token1(string delimitStr)`

Parameters *delimitStr* - string containing delimiters to search for

Return The next token parsed from the source string

Description Parses and retrieves the source string for the next token. If no more tokens are found, an empty string will be returned.

### string.token2

Syntax `string token2(string delimitStr)`

Parameters *delimitStr* - string containing delimiters to search for

Return The next token parsed from the source string

Description Parses and retrieves the source string for the next token. Use *eol()* to determine if no more tokens are found.

## 3.3 Flag

Flags can only store one of two possible values. This is similar to the "bool" type found in C++/Java. An added feature to the ReyScript class is that it can keep track of the string enumerator used set the flag. With this *enumerator mode*, the class can determine how to write out the value of the flag. For example, if you assign the string "true" to the flag, then if you write the flag or assign it to another string, the flag generates the string "TRUE" or its complement "FALSE". Other enumeration pairs include t/f, yes/no, y/n, and the default 0/1. Flags are not case sensitive and will write to output in upper-case.

The enumerator mode is set when flag is assigned a string or read in from a stream. The mode is also passed to the resulting flag of an operation if the left or unary operand was a flag. The sample code below will clear this up a little more:

```
flag a, b, c, d;  // values initialized to 0

a = "true";  // set to "TRUE"
write "negation is " + !a + "\n"; // writes out "FALSE"

b = "no";    // set to "NO"
write "a == b is " + (a==b) + "\n";   // writes out "FALSE"
write "b == a is " + (b==a) + "\n";   // writes out "NO"

c = a;       // set to 0 since a is not a string;
d = a != b;  // set to 1 since result of != is not a string;
```

## 3.4 Date

Use this class to store date and time values. When writing the value to the stream or assigning to a string, the date class formats the output according to your hardware settings or *locale*. To change the format of the output, use the *format()* function. It is similar to the string's format() because the format string contains specifications. Unlike the string format(), there are no width and precision fields and only one optional flag is allowed. The following gives you a rundown of the date specifications.

*Table 3-5: Date Format Types*

| Type | Description |
|---|---|
| a | Abbreviated weekday name |
| A | Full weekday name |
| b | Abbreviated month name |
| B | Full month name |
| c | Date and time representation appropriate for locale |
| d | Day of month as decimal number (01 – 31) |
| H | Hour in 24-hour format (00 – 23) |
| I | Hour in 12-hour format (01 – 12) |
| j | Day of year as decimal number (001 – 366) |
| m | Month as decimal number (01 – 12) |
| M | Minute as decimal number (00 – 59) |
| p | Current locale's A.M./P.M. indicator for 12-hour clock |
| S | Second as decimal number (00 – 59) |
| U | Week of year as decimal number, with Sunday as first day of week (00 – 51) |
| w | Weekday as decimal number (0 – 6; Sunday is 0) |
| W | Week of year as decimal number, with Monday as first day of week (00 – 51) |
| x | Date representation for current locale |
| X | Time representation for current locale |
| y | Year without century, as decimal number (00 – 99) |
| Y | Year with century, as decimal number |
| z | Time-zone abbreviation; no characters if time zone is unknown |
| Z | Time-zone name; no characters if time zone is unknown |
| % | Percent sign |

*Table 3-6: Date Format # Flag*

| For types | Remarks |
|---|---|
| a, A, b, B, p, X, z, Z, % | # flag is ignored. |
| c | Long date and time representation, appropriate for current locale. For example: "Tuesday, March 14, 1995, 12:41:29". |
| x | Long date representation, appropriate to current locale. For example: "Tuesday, March 14, 1995". |
| d, H, I, j, m, M, S, U, w, W, y, Y | Remove leading zeros (if any). |

There are three ways to set a date variable other than with another date variable. The first is to call the *mark()* function to get the current date and time from the system. Secondly, you can assign a number which represents the number of seconds since January 1, 1970 12:00 am. Third, you can assign a string to it. The string must be in this format:

YYYY:MM:DD:HH:MM:SS

YYYY - 4 digit year
MM - month (01-12)
DD - day of month (01-31)
HH - hour (0-23)
MM - minute (0-59)
SS - second (0-59)

The date class is designed only for absolute time. It's best keep relative or delta time in a number instead, but you must be responsible for formatting the output. The code below shows how to use date variables to keep track of how long the program took to execute.

*Sample 3-3: Stopwatch*

```
main()
{
    date start, end;  // values initialized to undefined
    number diff;

    start.format("%b %d, %Y %#I:%M:%S %p\n");
    end = start;     // copies the format string

    start.mark();    // get the start time

/*
    ...
    perform some task here
    ...
*/

    end.mark();     // get the end time
    diff = end - start;

    write "The program started on " + start;
    write "The program ended on " + end;
    write "The program ran for " + diff + " seconds\n";
}
```

Output:

```
The program started on Feb 2, 1998 2:30:45 PM
The program end on Feb 2, 1998 2:30:57 PM
The program ran for 12 seconds
```

Keep in mind that dates evaluate to true if they are set and false otherwise.

### 3.4.1 Date Functions

#### date.format

| | |
|---|---|
| Syntax | `date format(string formatStr)` |
| Parameters | *formatStr* - string containing specifications to create date string |
| Return | The variable itself |
| Remarks | Stores the given format string so that it can be used when writing out the date value. For descriptions of each of the field specifiers, refer to tables 3-5 for types and 3-6 for flag restrictions. |

### date.mark

Syntax `date mark()`

Parameters None

Return The variable itself

Remarks Gets the current date and time from the system and stores them in the variable.

## 3.5 Stream

A stream represents an device that can send and receive data. In ReyScript, there are three types of streams: terminal, file, and command. A *terminal stream* is used to receive user input from the keyboard or send user output to the screen. This is synonymous to the standard input and output in other languages. ReyScript provides to default terminal stream, so you don't have to define one unless you wish to write to standard error. A *file stream* allows you to access data in a file. Finally a *command stream* allows you to pass data to and from another program that can read and write to standard input and output.

Streams will be covered in detailed later in chapter 7. For now, the one thing you need to know about streams are that they are used conjunction with the read and write commands. The examples you've seen so far in these first two chapters only write output to the terminal stream.

# 4 PROGRAM FLOW

This chapter will discuss and introduce the rest of the constructs available in ReyScript. Again if you're familiar with C++/Java most of this stuff won't be new to you, except for this following section.

## 4.1 Initialization

The first thing that gets initialized in a ReyScript program is the program's *workspace*. Think of the workspace as the global class, so all the classes, functions, and variables for a program are defined within this workspace. In essence, the workspace, like any other class, is instantiated and its member variables (which in this case are the global variables) are initialized. These variables are defined outside of any class or main function, but within the workspace. After the global variables are created, the main function is called and off we go.

### 4.1.1 Parameter and Shell Variables

Data can be exchange into and out of ReyScript program in one of two ways. The first is to pass them in as command-line arguments. When you run the program, you include these *program parameters* as part of the interpreter options -- see chapter 8 for more information on how to define a parameter on the command-line. From within the ReyScript program, you must define a global variable with the keyword `parameter` in front. Here's an example of a parameter variable:

```
parameter number emplID;
```

This will tell the interpreter to look for data for the variable "emplID" on the command-line. If the parameter was not given, the interpreter gives an error and does not execute the program. If you define a parameter on the command-line but not in the program, it will simply be ignored.

The other method is to access the data from shell environment variables. You typically run the ReyScript program from a shell, therefore it inherits the environment variables defined for that shell. To access one of these variable, you must global variable with the same name as the environment variable and add the keyword `shell` in front. Here's an example of a shell variable:

```
shell string PWD;
```

This will tell the interpreter to initialize the global variable PATH with the value found in the environment variable PATH (which contains directories to locate other programs). If the environment variable does not exist, the interpreter will just continue executing the program and initialize the global variables to default values. If you alter the value of the shell variable in the program, it will not be saved to the environment variable upon termination.

### 4.1.2 Parameter and Shell Blocking

If you have several parameter and shell variables to declare, you can list them in a functional block and insert either the `parameter` or `shell` keyword in front. This is just a short-hand to the other format so that you don't have to re-type the keywords before every variable declaration. For example:

```
parameter
{
    string username;
    string password;
}

shell
{
    string LIBPATH;
```

```
        string PATH;
    }
```

Both username and password will be read from the command-line, and LIBPATH and PATH will be accessed from the environment variables.

One final note on parameter and shell variables is that you can only declare them in the workspace. If you define them anywhere else such as within a class or function, the parser will generate an error.

### 4.1.3 Libraries

Sometimes you want to break up your program into several files either to make the program more maintainable or to share classes and functions with other programs. To tie the files together, use the `library` command to specify the external source files or *libraries* you want to include in the program. This is similar to the `#include` directive of C/C++ where source code from other files is pasted onto the current one. All global variables declared in the libraries become global variables of the program. Similarly, all classes and macros defined in the libraries become part of the program as well. One side effect is that if declare a global variable with the same name in several libraries, then the parser will generate an error since they are declared in the same global scope -- the same can be said with classes.

You can specify a library anywhere in the source code the parser reads it in first before the rest of the source code. The syntax is listed below:

```
library filename;
```

Where *filename* is the name of the library you want to include. This field must be a string constant since it is processed before any variables are initialized..

### 4.1.4 Constants

When the interpreter comes upon a constant in a statement, a number or string temporary object is created to store the value, allowing you to operate on that constant as if it was an variable. As a result, you can change the value of the object holding the constant, but it is pointless since you cannot refer to it anywhere else in the program. Let's look at this example:

```
5 = 8;
```

The interpreter creates two number objects: one to store the value 5 and the other to store the value 8. The above statement then resets the first object to store the value of 8, but since it's not a variable, it can't be used anywhere else. For more on temporary objects, see section 5.2.1.

## 4.2 Termination

The interpreter keeps running the program reading, statement after statement, until it reaches the end of the main function. However, you can break the flow of the program or a function prematurely in one of several ways. To end the program immediately, use the `exit` command in the following form:

```
exit [expression];
```

The result of the expression serves as the exit status for the program so it must evaluate into a number. If no expression is given, then the default exit status is 0.

If you just want to return control from a function to the calling function, just use the `return` command in the following form:

```
return [expression];
```

The expression must evaluate the return type specified for the function. Therefore, if no return type was specified, do not use an expression.

## 4.3 If-Then-Else

Sometimes you need to make a decision as to what statements to execute based on a criteria. The "If-Then-Else" construct is the most basic form of conditional branching in ReyScript. The construct has the following syntax:

```
if (expression) then-statement; else else-statement;
```

If *expression* is evaluated to true, the *then-statement* is executed, otherwise the *else-statement* is executed. The above format is not very readable, so the more accepted style is this:

```
if (expression)
    then-statement;
else
    else-statement;
```

You can combine several statements into a functional block which you can use as part of the then or else clause, like this:

```
if (expression)
{
    then-statement1;
    then-statement2
    ...;
}
else
{
    else-statement1;
    else-statement2;
    ...
}
```

As you can see, there's no need to have a `then` keyword since it's required just like the `if` expression. However, it's not necessary to have an `else` clause, so the keyword is needed to inform the interpreter that the next statement is conditional and part of the `if` statement.

### 4.3.1 Nested Branching

You can have "if" statements within another "if" statement, making a *nested-if* statement. Be careful when coding the "else" clause as it may end up as part of the wrong "if" statement. Let's look at this example:

```
if (a>100)
    if (b == 10)
        write "Do This\n";
else
    write "Do that\n";
```

I want to display the message "Do That" if a <= 100. However, with the code above, that message is not printed because the "else" statement actually belongs with the inner "if" statement which gets executed when a > 100 and b != 10. To make it work the way I want to, I should place the inner statement inside a functional block, like this:

```
if (a>100)
```

```
{
    if (b == 10)
        write "Do This\n";
}
else
    write "Do that\n";
```

### 4.4 Switch

Suppose you want to execute more than two different sets of statements based on the value of a single variable. You can construct a nested-if statement like this:

```
if (a == 1)
    write "First choice\n";
else if (a == 2)
    write "Second choice\n";
else if (a == 3)
    write "Third choice\n";
else
    write "No choice\n";
```

However, if you have quite a number of "if" expressions, then this construct get pretty hairy and hard to read. Instead, you can use the `switch` statement to perform the same task. The syntax for the switch statement looks like this:

```
switch(expression)
{
case value1:
     statement1;
     ...

case value2:
     ...
default:
    ...
}
```

The *expression* must evaluate to either and number or string. The `case` keyword denotes a label use for determining the location of the next statement to execute. Every label must include a value, either a number or string constant. It is necessary to enclose the labels and statements of the "switch" construct within a function block., which in this case is called a *switch block*.

Once the "switch" expression is processed, the interpreter searches each label in order to check if the result of the expression equals the value of the label. If a match was found, the program execution continues with the statement following the label. If no match was found, execution continues after the `default` label if one exist, otherwise nothing is the switch block is executed. Because the entry point for the functional block is not determined until the labels are checked, you cannot declare variables within the scope of the switch block; you can, however, declare variables within functional blocks for any other constructs nested within the switch statement.

The "if" sample listed at the beginning of this section can now be replaced by this "switch" statement:

```
switch(a)
{
case 1:
    write "First choice\n";
```

```
        break;

    case 2:
        write "Second choice\n";
        break;

    case 3:
        write "Third choice\n";
        break;

    default:
        write "No choice\n";
    }
```

Use the `break` keyword, to exit the switch block. Without the "break", the interpreter will processed the next non-label statement it sees. I think the "switch" statement looks cleaner than the "if" statement, although in this example, you have to write more code to perform the same task. The "switch" statement does become more efficient with larger switch blocks that contains more labels. However, it is not as flexible as the "if" statement since you can check for equality against a single value. For range and multiple variable checking, you still have to use the "if" statement.

### 4.5 Looping

If you want to repeat executing a set of statements, you can place those statements within a loop. A loop contains a body of statements and a *trigger* which is an expression that determines if the body is executed, in other words, if an *iteration* of the loop is performed. There are two types of loops which differs only when the trigger is checked*: pre-trigger* and *post-trigger*. For a pre-triggered loop, the trigger is evaluated before any iteration whereas for a post-triggered loop, the trigger is evaluated after an iteration, which means the body of the loop is executed at least once.

To implement a pre-triggered loop, use the "while" statement in this format:

```
while (trigger-expression) body-statement;
```

Or for larger loop bodies, try placing the body in a functional block like this:

```
while (trigger-expression)
{
    body-statement1;
    body-statement2;
    ...
}
```

If the *trigger-expression* is evaluated to true, then the body of loop is executed. When execution of the body is complete, the trigger-expression is re-evaluated. The interpreter will continue to iterate through the loop until the trigger-expression evaluates to false. Now, let's look at this example:

```
number i=1;
string output;

while(i<=5)
{
    output.format("This is iteration #%d\n",i);
    write output;
    i++;
}
```

The interpreter will repeat the loop as long as i <= 5. The variable "i" serves as the *iterator*, or the basis for the iteration. Usually, the value of the iterator changes for every iteration of the loop and is evaluated as part of the trigger-expression. The output for the above sample looks like this:

```
This is iteration #1
This is iteration #2
This is iteration #3
This is iteration #4
This is iteration #5
```

To implement a post-triggered loop, use the "do-while" statement in this format:

```
do body-statement; while (trigger-expression);
```

Or for larger loop bodies, try placing the body in a functional block like this:

```
do
{
    body-statement1;
    body-statement2;
    ...
}
while (trigger-expression);
```

Here, the body of the loop is executed at least once. Subsequent iterations are called only when the trigger expression is true.

Sometimes, you need to stop the iteration from within the body of the loop. Use the `continue` statement to end the current iteration of the loop and re-evaluate the trigger-expression to decide whether to proceed to the next iteration. If you just want stop processing the loop altogether, use the `break` statement just like in a switch block. Keep in mind that if you have nested-loops (loops within loops), both the `break` and `continue` affects only the iteration it is specified in. The structure of a nested-loop using both termination statements might look like this:

```
while(expr1)
{
    if (expr2)
    {
        while(expr3)
        {
            if (expr4)
                break;   // ends inner-while loop
        }

        continue;  // ends current outer-while iteration
    }

    ...
}
```

### 4.5.1 Infinite Loops

A common pitfall with loops is when the trigger-expression always evaluates to true. If you have no termination statements in the body such as a `break` or `continue`, then the interpreter will keep repeating the loop, causing an *infinite loop.* Here's an example of an infinite loop:

```
while(1)  // always evaluates to true
```

```
    write "I'm running...\n";
```

This code will cause the message "I'm running..." to display on the standard output continuously. If you don't want to use an iterator in the trigger-expression, then you must use it to break out of the loop, like this:

```
number i=0;

while(1)  // always evaluates to true
{
    write "I'm running...\n";

    if (i>100)
    {
        write "I'm tired.\n";
        break;
    }

    i++;
}
```

### 4.5.2 Where's the For-Loop?

C++/Java users should be familiar with another kind of loop -- the "for" loop. Well, guess what? ReyScript does NOT provide a "for" loop. After contemplating this for a while, I came to the conclusion that the syntax for the "for" loop is awkward and inconsistent with the rest of the constructs in the language. The "for" statement is basically a short-cut to the "while" statement, although I don't think you gain too much with the previous. Besides, the "while" loop is more flexible, so every potential "for" loop can be implemented as a "while" loop. Therefore, a "for" loop looking like this:

```
for(i=0;i<10;i++)
{
    body-statements
    ...
}
```

Can be rewritten like this:

```
i=0;
while(i<10)
{
    body-statements
    ...
    i++;
}
```

## 4.6 Function Calling

One way of modularizing the code is to separate them into functions. To execute code located in one function from another is to *call* the function by coding the name of the function into the statement or expression along with its appropriate arguments. A*rguments* are constants or variables passed into the function and are determined by the *functional parameters* (which are different from the program parameters discussed in section 4.1.1) declared for that function definition. For example, if you have this function definition:

```
number max(number a, number b)
{
```

```
    ...
}
```

The first is a number referred to as parameter “a” and the second is another number referred to as parameter “b”. When calling this function, you must pass two numbers like this:

```
number x = 6, y = 18;
z = max(x, y);
```

From within the “max” function, the parameter “a” takes on the value of “x” while “b” takes on the value of “y”. The values aren’t copied into the function, instead both parameters are references to the outside arguments. This is referred to as *call by reference* where the parameters accesses the storage area of the arguments. Because of this, if you alter the a parameter within the function, you change the value of its corresponding argument like in this example:

```
main()
{
    number x = 6;
    funct1(x);
    write x;      // x is set to 7
    funct1(78);   // the object that stores to constant is pointlessly
                  // set to 79
}

funct1(number a)
{
    a++;
}
```

### 4.6.1 Overloading

Unlike classes and variables, you can define several functions within the same scope using the same identifier. This is referred to as *function overloading*, where the interpreter determines which function to execute based on the argument types. For example, you can define these two functions in the same class:

```
funct1(number a)
{
    a++;
}

funct1(number a, number b)
{
    a += b;
}
```

Then you can call either function simply by changing the arguments.

```
main()
{
    number x=5;
    funct1(x);    // calls the first funct1
    funct1(x,6); // calls the second funct1
}
```

For readability sake, you shouldn’t use overloaded functions too often unless the functions are nearly identical. It would be very hard to read code whose functions all use the same name.

## 4.7 Exception Handling

When you want to bail out of several levels of function calls because you get data you don't want, you can "throw" a program error or *exception* . The syntax is similar to C++/Java; you first "try" some functions calls, and if an exception is thrown, you "catch" the exception and perform a special routine for it.

Let's start with the throwing aspect. To generate or *throw* an exception, use the `throw` statement in the following form:

```
throw expression;
```

The *expression* represents the value of the exception and can be evaluated to any type, typically a number or string. However, you have to be able to catch that type of exception if you don't want the interpreter to abort the program, which occurs when an exception is not caught.

The catch the exception, use the try-catch construct like this:

```
try try-statement;
catch(argument) catch-statement;
```

This tells the interpreter that an exception could be throw in the *try-statement*. If the exception type matches the *argument* type then the catch-statement is executed. If you want to code several statements for either trying or catching, just place you statements in functional blocks like this:

```
try
{ // this is a try block
    try-statement1;
    try-statement2;
    ...
}
catch(argument)
{
    catch-statement1;
    catch-statement2;
    ...
}
```

Think of the catch statement as special function and the thrown exception is an argument to the function. If you plan to throw different types within the try block, you may want to specify multiple catch functions, each one defining a different argument type. For example, if you want to throw either a number or string exception, you can define a catch function for each one like this:

```
try
{
    try-statement1;
    try-statement2;
    ...
}
catch(number excpNum)
{
    catch1-statement1;
    catch1-statement2;
    ...
}
catch(string excpStr)
{
```

```
        catch2-statement1;
        catch2-statement2;
        ...
    }
```

### 4.8 Macros

Macros are used to substitute a single word with a sequence of words in the source code. This is similar to the `#define` directive in C/C++. You use macros typically to define constants, but you can use it to substitute anything, even class and function definitions. Macros are read by the parser at the same time as libraries, which is before any source code. This means you can define macros anywhere in the source code and use anywhere in the workspace. However for readability, macros should be defined near the beginning of the source file. You can also define macros on the command line. If your code supposed to have macros in it, but you did not define it in the code or the command line, you could get a "symbol not found" error.

As the parser begins reading source code, it replaces any word identified as a macro with the corresponding *macro body* containing the replacement words. To define a macro, use the following:

```
macro macro-name macro-body;
```

For example, you can have a macro defined for constants like this:

```
macro MAX_NUM 999999;
```

So wherever the source code contains the word MAX_NUM, the parser replaces it and processes the source code with the value 999999.

# 5 OPERATORS

An *operation* basically is a special type of function call. Rather than calling a function by name and passing it arguments in the form of a list, an *operator* is usually nested between *operands* which serves as the arguments to the operator. In reality, operators are resident functions defined for each of the primitive classes, so they are only available to those classes. To imitate an operator for user-defined classes, you will just have to define a regular function for it. Unlike C++, ReyScript does not provide operator overloading.

This chapter discusses the operators that are available to the primitive class. Some operators can be even used on different types of operands, so which implementation of the operator to call is determined by the type of the left operand or *primer*. The right operand or *augmentor* then serves as the argument to the operator function of the primer. For *unary* operators, i.e. those operators with only one operand, that single operand is by default the primer.

## 5.1 Precedence and Associativity

Statements usually contain at least one operation. When two or more operations are involve, the interpreter must determine in which order the operations will be processed. Therefore, an operator with higher *precedence* will be processed first. Let's look at this example:

```
7 + 4 * 5
```

If we perform the operations from left to right, the final result would be 55 since we add 7 and 4 first then multiply the result of that (which is 11) by 5. However, since multiplication has a higher precedence than addition, we should multiply 4 by 5 first, resulting in 20 then add 7 to that which comes to a final answer of 27. If you do want to add 7 and 4 first, you can place the operation inside parentheses like this:

```
(7 + 4) * 5
```

Think of parentheses as operators as well, but they have the highest precedence so anything inside of it will be processed before those outside of it. If in doubt of the precedence of an operator, using parentheses will remove the guesswork and make the code more readable anyway.

If two adjacent operators have the same precedence, then associativity determines which direction the operators are processed: either left-to-right or right-to-left. In most cases, associativity doesn't matter as in this case:

```
1 + 2 + 3
```

The result is 6 either way you process this statement. However, take a look at this statement:

```
a = b = 1;
```

If we go left to right, then a is first assigned to b, then b is assigned to 1. Depending on what the value of b was before is was reassigned, then a may not be set to 1. However, going from right to left, then both variables are definitely set to 1, so associativity is important in this case.

The table below list all the operators available in ReyScript is order of precedence. The functional operators have been covered in the previous chapter, so the following sections will cover each of the remaining operators in detail.

*Table 5-1: Operator Chart*

| Precedence | Associativity | Operator | Description | Category |
|---|---|---|---|---|
| 1 | left to right | `()`<br>`.` | function call, sub-expression<br>class member | functional |
| | | `++`<br>`--` | postfix increment<br>postfix decrement | arithmetic |

<table>
<tr><th>Precedence</th><th>Associativity</th><th>Operator</th><th>Description</th><th>Category</th></tr>
<tr><td>2</td><td>right to left</td><td>!</td><td>NOT</td><td>logical</td></tr>
<tr><td></td><td></td><td>++<br>--<br>-</td><td>prefix increment<br>prefix decrement<br>unary negation</td><td>arithmetic</td></tr>
<tr><td>3</td><td>left to right</td><td>*<br>/<br>%</td><td>multiplication<br>division<br>modulus</td><td>arithmetic</td></tr>
<tr><td>4</td><td>left to right</td><td>+<br>-</td><td>addition<br>subtraction</td><td>arithmetic</td></tr>
<tr><td>5</td><td>left to right</td><td><<br><=<br>><br>>=</td><td>less than<br>less than or equal<br>greater than<br>greater than or equal</td><td>relational</td></tr>
<tr><td>6</td><td>left to right</td><td>==<br>!=</td><td>equal<br>not equal</td><td>relational</td></tr>
<tr><td>7</td><td>left to right</td><td>&&</td><td>AND</td><td>logical</td></tr>
<tr><td>8</td><td>left to right</td><td>||</td><td>OR</td><td>logical</td></tr>
<tr><td>9</td><td>right to left</td><td>=<br>+=<br>-=<br>*=<br>/=<br>%=</td><td>assignment<br>add<br>subtract<br>multiply<br>divide<br>modulus</td><td>assignment</td></tr>
</table>

## 5.2 Assignment

Assignment operators set the value of the primer based on the value of the augmentor. If the both are not of the same type, then the value of the augmentor is converted before setting the primer. There are six assignment operators, of which the most basic is the "=". All the other operators also perform arithmetic functions so they will be discussed in the next section instead.

The table below describes the what conversion is used for the given primer and augmentor types. All assignment operators return the primer.

*Table 5-2: Conversion for = Operator*

| Primer | Augmentor | | | |
|---|---|---|---|---|
| | number | string | flag | date |
| number | no conversion, just copies value | converts to number or 0 if conversion invalid | converts to 0 if false or 1 if true | converts to number of seconds since January 1, 1970, or -1 if not set |
| string | converts to string | no conversion, just copies value | converts according to the enumerator mode | converts according to date format, or "(DATE ERROR)" if not set |
| flag | converts to false if 0/0.0, or true otherwise | converts an enumerator to true or false (see section 3.3) | no conversion, copies value and enumerator mode | converts to true if date is set, false otherwise |
| date | converts to number of seconds since January 1, 1970 | see section 3.4 on how to convert string to date | N/A | no conversion, copies value and date format |

Here are some examples using the basic assignment operator:

```
number a;
string b;
flag c;
date d;

a = 10;        // no conversion
a = "78.4";    // string constant converted to 78.4 before assignment
a = c;         // flag converted to 0
a = d;         // set to -1

b = 99;        // convert to string "99"
b = "Hello";   // no conversion
b = c;         // set to "0"
b = d;         // set to "(DATE ERROR)"

c = 1;         // set to true
c = "YES"      // set to true, set enumerator mode to YES/NO
c = d;         // set to false

d = 0;         // set to January 1, 1970 12:00 am
d = "2001:10:15:10:30:00"  // set to October 15, 2001 10:20 am
```

### 5.2.1 Temporary Objects

The assignment operators are the only operators that are guaranteed to return the primer since the new value is stored in there. Therefore if you want to access that value, make sure the primer for these operations is a variable. All other operations, save the new values into *temporary objects*. Think of them as variables, but not accessible anywhere in program. Constants are loaded into these objects and any operation that does not store a value into a variable will be stored in a temporary object instead. Keep this in mind when passing arguments into a function or reading data from a stream.

```
number a = 2;

a + 5;   // result stored in a temporary object
a += 6;  // result is a
```

## 5.3 Arithmetic

*Arithmetic* operators perform algebraic calculations for numeric or date operands. Flags and strings generally do not have arithmetic operators. Although they both have one defined for addition, they actually don't perform addition in the normal sense. Normally, addition suggest adding two numbers together, but strings and flags aren't numbers. Instead, concatenations are done strings, while addition for flags is substituted for the "!=" operation.

Addition is the first of the four basic types of arithmetic operators. The add operator returns a constant with a type according to Table 5-3. The add-assignment and all of the other arithmetic-assignment operators return the primer.

*Table 5-3: Return Type for +, += Operators*

| Primer | Augmentor | | | |
|---|---|---|---|---|
| | number | string | flag | date |
| number | number | number | number | number |
| string | string | string | string | string |
| flag | flag | flag | flag | flag |
| date | date | date | date | date |

Here are some examples using the addition operators:

```
number a = 5;
string b = "Hello World ";
flag c;
date d;

a + 10;        // result is 15
b + 20;        // result is "Hello World 20"
c + b;         // result is true
d += a;        // d is set to January 1, 1970 12:00:04 am
```

Subtraction operators are available only to numbers and dates. Since dates are stored in numeric format, you can subtract seconds without difficulty.

*Table 5-4: Return Type for -, -=, Operators*

| | Augmentor | | | |
|---|---|---|---|---|
| Primer | number | string | flag | date |
| number | number | - | - | number |
| string | - | - | - | - |
| flag | - | - | - | - |
| date | date | - | - | date |

Here are some examples using the subtraction operators:

```
number a = 100;
date d;

d.mark();      // gets current date and time;

a - 10;        // result is 90
d -= a;        // subtracts 100 seconds from current date and time;
```

Multiplication, division, and modulus operators are available only to numbers. It just doesn't make sense for any other type. Here are some examples using the those operators:

```
number a = 100;

a *= 5;     // a is set to 500
a /= 56.7   // a is set to 8.8124
a % 3       // result is 2, a is rounded to 8 first
```

### 5.3.1 Incrementation and Decrementation

Numbers and dates have two extra operators (++ and --) that add or subtract 1 respectively with less code. The both *increment* and *decrement* operators can be used like this:

```
a++;
b--;
```

The above code does the same thing as these:

```
a = a + 1     or     a += 1;
b = b - 1     or     b -= 1;
```

These operators can be placed either in front of or behind a variable. When placed in front (*prefix*), this operator is perform first before applying the result to the surrounding expression. When placed behind the variable (postfix), the value of the variable is applied to the surrounding expression before that variable is incremented or decremented. The following shows the subtle difference between the prefix and postfix form.

```
number a, b;

a = 1;
b = a++;    // b is set to 1, a is set to 2

a = 1;
b = ++a;    // a and b are set to 2
```

## 5.4 Relational

*Relational* operators compare two values to see how they relate to each other. All relational operators return a flag denoting if a specific relationship exist between the values. For the equality operators ("==" and "!="), the augmentor is converted as shown in Table 5-5 before comparing the result to the primer.

*Table 5-5: Conversion for ==, != Operators*

| Primer | Augmentor | | | |
|---|---|---|---|---|
| | number | string | flag | date |
| number | no conversion, just compares values | converts to number or 0 if conversion invalid | converts to 0 if false or 1 if true | converts to number of seconds since January 1, 1970, or -1 if not set |
| string | converts to string | no conversion, just compares values | converts according to the enumerator mode | converts according to date format, or "(DATE ERROR)" if not set |
| flag | converts to false if 0/0.0, or true otherwise | converts an enumerator to true or false (see section 0) | no conversion, just compares values | converts to true if date is set, false otherwise |
| date | converts to number of seconds since January 1, 1970 | see section 3.4 on how to convert string to date | result always true for "!=" and false for "==" | no conversion, just compares values |

Here are some examples using the equality operators:

```
number a = 100;
string b = "100";
flag c;
date d;

c = a == 89;      // c is set to false
c = a != 89;      // c is set to true
c = a == b;       // c is set to true
c = a == d;       // c is set to false

c = b == "Hello"; // c is set to false
c = b == "Hello"; // c is set to false
c = "" == 0;       // c is set to true
```

Other relational operators check to see if one value is greater than or less than the other. Both operands must be of the same type and can only be used on numbers, strings, and dates (see Table 5-6 below).

*Table 5-6: Return Type for <, <=, >, >= Operators*

| Primer | Augmentor | | | |
|---|---|---|---|---|
| | number | string | flag | date |
| number | flag | - | - | - |
| string | - | flag | - | - |
| flag | - | - | - | - |
| date | - | - | - | flag |

Numerical comparisons are done for numbers and dates. Strings, on the other hand, are compare lexicographically. Here are some examples.

```
number a = 100;
string b = "abcde";
flag c;
date d;

c = a < 100;      // c is set to false
c = a <= 100;      // c is set to true
c = a >= 100.0;    // c is set to true
c = a > 100;       // c is set to false

c = b < "bcdef"; // c is set to true
c = b < "ab";    // c is set to false
```

## 5.5 Logical

*Logical* operators can be used to perform boolean arithmetic. Like relational operators, these return flags containing the result of the operation. However, they can be used with any combination of primitive variables -- they're just evaluated to a flag before the logical operator is applied. The negation operator simply return the compliment if the expression -- in other words, if the expression evaluates to true, then the negation of that is false. A summary of the logical operators with the results based from the value of the operands are shown below:

| NOT<br>! | |
|---|---|
| false | true |
| true | false |

| AND<br>&& | false | true |
|---|---|---|
| false | false | false |
| true | false | true |

| OR<br>\|\| | false | true |
|---|---|---|
| false | false | true |
| true | true | true |

Here are some examples using the logical operators:

```
number a = 1;
string b = "";
flag c;
date d;

c = !a;    // c is set to false
c = !b;    // c is set to true

c = a && b; // c is set to false
c = a || b; // c is set to true
c = d || b; // c is set to false since d is not set
```

# 6 OBJECT-ORIENTED PROGRAMMING

A typical object-oriented language employs the use of classes. In chapter 3, you learned about the resident classes of ReyScript. In this chapter you will see how to define your own classes. Let's start with the syntax for a class definition:

```
class class_name [base baseclass_name]
{
    [member_variable declaration]
    ...

    [member_function definition]
    ...
}
```

A class should contain at least one *member* either a variable or a function. You can define members in any order, but the usual practice is to declare the variables before the function definitions. You create a class for a specific role or responsibility in a program. The members of the class help carry out its responsibilities -- in other words, the class *encapsulates* data and functionality pertinent to its class' duties.

During the course of this chapter, we will use a accounting-type program to demonstrate object-oriented programming. Let's first define an employee class responsible for holding and processing data relevant to an employee:

```
class employee
{
    number id;
    string first_name;
    string last_name;

    number hours;
    number rate;

    printWage()
    {
        string wageStr;

        wageStr.format("ID:%d,  Name: %s %s,  Wage: %.2f\n",
            id, first_name, last_name, hours * rate);

        write wageStr;
    }
}
```

There are five member variables and one member function. All member variables have class scope so the "printWage" function can access them. However, the variable "wageStr" has only function scope, so if there are any other functions defined for this class, they cannot accessed that variable (see section 2.2 for more on scopes).

Now you can declare a variable that is an instance of the employee class like this:

```
main()
{
    employee a;
}
```

## 6.1 Public Members

By default, all members of a class are hidden from external functions. Continuing from the example above, you will receive a parser error to you tried to access any of the member variables like so:

```
write a.first_name;
```

In order for this statement to be valid, you have to go back to the class definition and declare any member you want to access from the outside with the `public` keyword before the variable declaration or function definition. This tells the parser that it's alright for another function to access the member. So, if we change the declaration of the "first_name" variable to this:

```
public string first_name;
```

Then the "write" statement would be valid. If you want to declare several member variables to be public, you can use a *public block* as a short hand to using the `public` keyword on every declaration. Note that you can only use public blocks for variables and not functions.

```
public
{
    number id;
    string first_name;
    string last_name;
}
```

### 6.1.1 Why Hide Things?

The big question that comes to mind is why should members be hidden in the first place. Data hiding is a way to control or limit the use of a class, which thereby adds to the modularity or portability of the code. With small programs though, this is not a big issue. However, I think it's easier to develop and code classes that are not intertwined or strongly dependent on one another., so keeping public variables to a minimum helps accomplish that.

## 6.2 Constructors and Destructors

When a class gets instantiated, all member variables are initialized to their default values. However, if you want to set the variables your own way or perform some task, you can do so in a constructor. A *constructor* is a special function executed by the interpreter right after the class has been instantiated. To define a constructor, you give it the same name as the class itself, similar to C++/Java. For example, if you want to initialize an employee object with an ID other than 0, you can use a constructor to do so like this:

```
class employee
{
    public number id;
    ...

    employee()
    {
        id = 1000;
    }

    ...
}
```

Therefore if you declare an employee variable, it would automatically set the "id" variable to 1000 as in this example:

```
main()
{
    employee a;
    write a.id;    // result is 1000
}
```

The above example shows how to create a default constructor. If you want to initialize an object in different ways, you can declare different constructors with each one handling a different set of arguments. For example, if you want to initialize the first and last name of the employee, you define a constructor with two string parameters like this:

```
class employee
{
    string first_name;
    string last_name;
    ...

    employee(string a, string b)
    {
        first_name = a;
        last_name = b;
    }

    ...
}
```

When you declare the variable, pass two string arguments representing the first and last name into the variable.

```
main()
{
    employee a("John", "Smith");
}
```

You can also tie several constructors together by just calling them from one another. For the employee class, you can also set the "id" variable when you initialized the name by calling the default constructor. However, calling functions is slower during execution than duplicating code, so if you only have a few lines in the default constructor like in this example, it might be better to just duplicate the code.

```
class employee
{
    public number id;
    string first_name;
    string last_name;
    ...

    employee()
    {
        id = 1000;
    }

    employee(string a, string b)
    {
        first_name = a;
        last_name = b;

        employee();   // sets the id
```

```
    }

    ...
}
```

The examples in this section show how to use constructors to initialize member variables. Other uses for constructors include initializing a file or process with header or preliminary data. We'll see this more on the next chapter, but for now if you do find yourself initializing a file or process, chances are you'll have to include terminating data when you're finish. You can do such tasks in a *destructor*, a special function executed by the interpreter right before the variable goes out of scope and is destroyed. You define a destructor similar in C++, by giving it the same name as the class itself except that begin the name with a "~". Unlike constructors, you can only have one destructor that has no parameters. The example below defines a destructor for the employee class:

```
class employee
{
    ...

    ~employee()
    {
        write "End of record\n";
    }

    ...
}
```

### 6.3 Deriving Classes

Suppose the employee class is not good enough to represent all employees. There are managers, salespeople, and technicians all having common information but needing special processing. You can derive new classes based from an existing one, resulting in the *derived classes inheriting* members from the *base class*. You then add extra members in the derived class specific to its responsibility and cannot be accessed from the base class. Here, we define a manager class based from the employee class:

```
class manager base employee
{
    string department;
    string employees;

    PrintDepartment();
}
```

The manager class defines the three members listed above along with all the members of the employee class. To declare a variable of this class, just do this:

```
main()
{
    manager a("John", "Smith");
}
```

This is valid since constructors and destructors are also inherited from the base class. In the above example, the second employee constructor is being called.

### 6.3.1 Inheriting Public Members

All functions of a derived class can access any member of the base class. In C++/Java jargon, the members of the base class behave as if they were "protected", so members that were not declared with the `public` keyword can still be accessed by any derived class.

## 6.4 Abstraction

A key characteristic of an object-oriented language is *polymorphism*, which is the ability of an object to behave in different ways, i.e. the ability to represent more than one class. Since ReyScript doesn't have pointers and dynamic allocation, it doesn't have the need for polymorphic elements such as abstract classes and virtual functions. However, we can still use these concepts only to help develop and control the use of classes, similar to the `public` keyword is used for data hiding. What that said, here goes.

Think of an *abstract class* as a template that you use to derive other classes. You will not be able to instantiate from an abstract class. To define an abstract class, you must have at least one virtual function. A *virtual function* creates the specification for a function but must be defined in any derived classes. In defining a virtual function, you specify the return type and any parameters the function needs; the difference is that you don't define a body for the function. If we change the employee class into an abstract class, it would look something like this:

```
class employee
{
    number id;
    string first_name, last_name;

    public PrintInfo(string a);  // this is a virtual function
}
```

The "manager" class must then define the "PrintInfo" function like this:

```
class manager base employee
{
    public PrintInfo(string x)
    {
        write "Manager's Name " + last_name;
    }
}
```

In this above example a different parameter name is used, which is alright as long as the type of the parameter matches. Now you can declare a "manager" variable but you cannot declare one for "employee".

```
main()
{
    employee e;  // this is an error
    manager m;   // this is ok
}
```

# 7 SHELL SCRIPTING

The key to the ReyScript language is in its ability to script commands, similar to a shell language. This chapter will cover the tools you'll need to perform those tasks that are normally associated with shell scripting, such as accessing files and executing commands.

## 7.1 Files and Commands

A common task in shell scripts is to perform file maintenance. So, if you want to check the status of a file, i.e. if it's readable and writeable, you can use the FILE global function in this way:

```
if (file("out.txt","rw"))
    write "Out.txt is not read/writeable\n";
```

The second string to the "FILE" function, contains characters representing properties of the file. A list of properties is shown below in Table 7-1. You can test for multiple properties and if the file has all those properties, then the function returns a true. Therefore in the above example, if the file "out.txt" was readable but not writeable, then the function returns a false.

*Table 7-1: File Properties*

| Flag | Condition |
|---|---|
| d | directory |
| l | symbolic link (UNIX only) |
| r | readable |
| w | writable |
| s | file size greater than 0 |
| x | executable |

When you're reading and writing files, you want to make sure you are accessing the correct directory. If you don't specify absolute paths in your filenames, the interpreter checks the files relative to your current *working directory*. The default working directory of the ReyScript program is the same as the working directory of the shell at the time you executed the program (in UNIX, this path is also stored in the "PWD" environment variable). To change the working directory, use the "cd" function. This is not a shell command, so you cannot use shortcuts in the path such as the home directory symbol "~".

```
cd("/home/usr/");
```

The "exec" function can be used for any command you can normally execute from the shell. You can include any arguments with the command, just make sure to use escape sequences for characters normally parse for the ReyScript program. The function returns the exit status, which you can use in this way:

```
if (exec("mv out.txt in.txt"))
    write "Could not move out.txt\n";
```

The standard input and output that the command produces goes to the terminal, only the exit status is return to the program. If you want to redirect input or output of the command, use this form:

```
if (exec("mv out.txt in.txt", null, null))
    write "Could not move out.txt\n";
```

The second argument to the function is the stream you want the command to read data from; the third argument is the stream you want the command to write data to. In the above example, a "null stream" is used for both input and

output, meaning that the don't want the command to read in or write out to anything, even the terminal. To capture the standard I/O and use it in the program, you have to use a stream object which is discussed in section 7.2.

### 7.1.1 Global Functions

#### global.cd

Syntax `flag cd(string pathname)`

Parameters *pathname* - path to directory

Return true if working directory was changed, false otherwise

Remarks Changes the working directory. The directory can have either a relative or absolute path. If the directory doesn't exist or is not executable, then the function returns a false.

#### global.file

Syntax `flag file(string filename, string mode)`

Parameters *filename* - path to file
*mode* - file mode to test

Return true if file passes test, false otherwise

Remarks Check the status of the file according to the given mode. The mode contains a series of characters representing the properties of the file. The file is then tested for that property. If more than one property is listed, then the result of the function is the ANDing of the individual tests. Refer to Table 7-1 for possible mode values.

#### global.exec

Syntax `number exec(string cmd)`
`number exec(string cmd, stream inS, stream outS)`

Parameters *cmd* - command to execute
*inS* - stream used as input to the command
*outS* -stream used as output to the command

Return exit status of command

Remarks Executes command in the shell. The second form of the function allows you to redirect input and output.

## 7.2 Streams

Use stream to pass data between the program and an external device such as a terminal, file, or another program. To create a stream, declare a variable of the stream class, then call either the "error", "file", or "command" function to set the type of stream. By default, the stream is initialized to the terminal type (standard input/output) which you don't need because a default terminal stream is already provided. The example below shows how to create both a file and shell streams:

```
stream f, s;

f.file(out.txt);
s.exec("ls -al");
```

Files are not opened and commands are not executed until you begin passing data. To pass data to and from the stream, you must use the `read` and `write` constructs. The `read` construct receives data from the stream and stores it in an object; on the other hand, the `write` construct send data stored in a variable to the stream.

```
read expression [from stream];
write expression [to stream];
```

If you don't specify the stream for either construct, the default terminal stream is used. If you do specify the stream, be sure to include the expression in parentheses. So, if you're just writing to the terminal, you can use this form:

```
write "Hello World\n";
```

However, if you write to a different stream other than the terminal, you must use this form:

```
write ("Hello World\n") to f;
```

### 7.2.1 Reading Restrictions

The expression can contain any number of atoms and evaluate to anything. However when reading from a stream, keep in mind that the data read in will be stored in the result of the expression. If the result is an temporary object that stores a constant, then the data read in will be lost since it's not accessible from anywhere else in the program. This is not a problem for writing to a stream, but for reading, be sure your expression results in a variable.

```
number a;

read (a + 5) from f;  // data is lost because result is a temporary object
read (a += 5) from f; // data is stored in a

write (a + 5) to f;   // data in temporary object is written to f
```

### 7.2.2 Callbacks

The resident classes have defined built-in functions to handle data to and from a stream. However if you want to use your own class to read and write stream data, you must define callback functions for that class. The interpreter executes a *callback function* when it performs some internal task. An example of a callback function is the destructor of a class. The program doesn't specifically call the class, it just happens as a result of the interpreter deleting the variables when a scope is exited.

For a class, you can define two callback functions, one each for reading and writing. Both functions should not have a return type and must define one stream parameter. The stream that gets passed in to these functions is the stream you specified in the `read` and `write` constructs. The function definitions should look like this:

```
~read(stream var)     // this is for reading
    body...

~write(stream var)    // this is for writing
    body...
```

The purpose of the callback functions is to break up the data into smaller chunks that you can read and write them into primitive objects. Let's use the employee class from the previous chapter to demonstrate how to use streams.

Suppose we want to read employee information from a file called “employee.txt”. Let’s assume that this is the contents of that file:

```
100;Steve;Smith
200;Judy;Cummings
300;Robert;Harris
400;Melissa;Barnes
```

The first field is the employee ID followed by the first and last name. The objective is to read in the data and generate a report to the terminal. To do this, you can first alter the employee class by adding the callback functions:

```
class employee
{
    number id;
    string first_name, last_name;

    ~read(stream x)
    {
        string inLine;

        read (inLine) from x;

        id = inLine.token1(";");
        first_name = inLine.token1(";");
        last_name = inLine.token1(";");
    }

    ~write(stream x)
    {
        string outLine;

        outLine.format("Name: %s %s, ID: %d\n",
            first_name, last_name, id);

        write(outLine) to x;
    }
}
```

In this example, a string is used to actually handle data to and from the stream. When reading data, the string is parsed, and the fields are copied into the member variables. When writing data, the values of the member variables are copied into a string. The string is most likely the object of choice for stream data since it can handle basically anything.

Now you can use this class to generate the report of the file. The file is read one line at a time, so repeat the statement until in end-of-stream is reached to process all the lines in the file.

```
main()
{
    employee e;
    stream f;

    f.file("employee.txt");

    while(!f.eos())
    {
        read (e) from f;
```

```
            write e;    // writes to standard output
        }
    }
```

The terminal output should look like this:

```
Name: Steve Smith, ID: 100
Name: Judy Cummings, ID: 200
Name: Robert Harris, ID: 300
Name: Melissa Barnes, ID: 400
```

### 7.2.3 Stream Functions

**stream.eos**

Syntax `flag eos()`

Parameters None

Return True if end-of-stream is reached, false otherwise

Remarks Checks if the end-of -stream has been reached. For files, this also known as end-of-file.

**stream.error**

Syntax `stream error()`

Parameters None

Return The variable itself

Remarks Sets the stream to standard input and error.

**stream.file**

Syntax `flag file(string` *`filename`*`, number` *`mode`*`)`

Parameters *filename* - path of the file
mode - mode to open the file: 1 - readonly, 2-writeonly, 3 - read/write

Return True if file was open, false otherwise.

Remarks Sets the stream to a file

**stream.command**

Syntax `flag command(string` *`command`*`)`

Parameters *command* - command to execute

Return True if command was executed, false otherwise.

| | |
|---|---|
| Remarks | Sets the stream to a shell command. |

**stream.reset**

| | |
|---|---|
| Syntax | `flag reset()` |
| Parameters | None |
| Return | True if the stream was reset, false if error occurred |
| Remarks | Resets the stream so you can begin reading or writing at the beginning of the stream. |

# 8 REYSCRIPT RUNNER

The ReyScript Runner is a parser, interpreter, and debugger all rolled into one. Like any other program, you can run this from a UNIX shell, DOS command prompt, or even Windows' Start Menu.. In Windows, a command window will open so you can interact with the program. Execute the program using the following syntax:

```
ReyScript [-?] [-d] [[-l library-path]...] [[-m macro_name=
      macro_body]...]  [-s shell] [-w title] program_file
      [[parameter_name= parameter_ value]...]
```

| Argument | Description |
|---|---|
| -? | Displays the version info for the ReyScript program and short description of the arguments |
| -d | Enables the debugger. If combined with the -c option, then this flag is ignored. |
| -i | Displays program stats including lines of code. Program is not executed. |
| -l *library_path* | Specifies the directory to search for library source code. You can specify multiple paths by adding another -l argument for each additional path. |
| -m *macro_name=macro_body* | Defines a macro to be use for the given program and supporting libraries. You can specify multiple macros by adding another -m argument for each additional macro definition. Macro body must be enclosed in double quotes (""). |
| -s *shell* | Specifies the shell to execute commands in. The default in UNIX is /bin/sh. For Windows, there's only one cmd.exe. |
| -w *title* | Specifies the name of the program if different from the name of the primary source file. |
| *program_file* | name of primary source file (either text or ReyScript binary) |
| *parameter_name= parameter_value* | Defines a parameter to be use in the given program and supporting libraries. You can specify multiple parameters by adding them after the program_name and to the end of the argument list. Parameter values do not have to be in quotes unless it is a string with spaces in it-- conversion will be done by the interpreter. |

To simply execute a program called "sample1.m", type in the following command at the shell prompt:

```
ReyScript sample1.m
```

You can specify the full path in the `library` command of the program, or on the command line like this:

```
ReyScript -l /usr/local/lib sample1.m
```

If you need several libraries located in several places, you can also specify them. If the library is not in the current directory, the parser with search the paths in order as they are listed on the command line.

```
ReyScript -l /usr/lib -l /usr/local/lib sample1.m
```

Finally, if the program has macros and parameters defined and you want to debug the program, you can use this form:

```
ReyScript -d -m ZERO="0" sample1.m username=max password=abc123
```

## 8.1 Debugger

Use the debugger to identify bugs in your program. To start the debugger, specify the "-d" option on the command-line along with the program name and any parameters. The parser reads in the program into memory but will not execute it. Instead you will be given a command prompt so you can begin debugging your program. There are several commands you can enter which are listed in the following table:

*Table 8-1: Debugger Commands*

| Command | Description |
|---|---|
| ? | Displays list of commands |
| b | List all current breakpoints |
| b *position* | Adds breakpoint. *Position* must be given in one of the following forms and can be either a function name or line number in a source file.<br>in [*classname.*]*functionname*<br>at [*filename*:]*line_number*<br>For a function, you must specify the full classpath for the member function. If no class is given, the function is assumed to be a global function. For a line number, if no filename is given the line is assumed to be in the current file being debugged. Each breakpoint is given a unique ID. |
| bd *id* | Deletes breakpoint. If the ID of the breakpoint is not given, then all breakpoints are deleted. |
| g | Runs or continues program |
| k | Displays stack |
| l | List source code |
| n | Step over next statement |
| o | Step out of current function |
| r | Terminate and reset program |
| s | Step into next statement |
| v | Displays all local variables |
| v *variable* | Displays specific variable. If the variable is a member of another variable, specify the parent in the following form:<br>*parent.member*<br>If no parent was given, the variable is assumed to be declared in the current block. For global variables, use "global" for the parent. |
| vg | Displays all global variables |
| w *variable* | Adds watch variable. If the variable is a member of another variable, specify the parent in the following form:<br>*parent.member*<br>If no parent was given, the variable is assumed to be declared in the current block. For global variables, use "global" for the parent. Each watchpoint is given a unique ID. |
| wd *id* | Deletes watch variable. If the ID of the watchpoint is not given, then all watchpoints are deleted. |
| x | Exit debugger |

### 8.1.1 Program Stepping

The debugger allows you to execute one statement or set of statements then pause and wait for your input. By *stepping* through your program, you can detect whether your program is operating the way you wanted to. The

alternative would be to add write statements inside your program to display debugging messages, but that's not as effective identifying bugs.

There are three ways of stepping in a program. First, you can *step into* ("s") a statement; if there is a function call in the statement, then the debugger will stop at the first statement inside that function. Second, you can *step over* ("n") the statement; this tells the debugger to execute any function calls within the statement without stopping in them. Third, you can *step out* ("o") of the current function; this tells the debugger to finish executing the rest of the function and stop in the calling function.

If you don't want to step through the program, you can enter the "go" command ("g") which continues the program. The debugger won't stop unless it hits a breakpoint. Breakpoints also causes the debugger to stop immediately even if you were stepping through the program. To add a breakpoint in a function called "getmax" defined in class "employee", enter this command:

```
mdbg> b in employee.getmax
```

The debugger will stop at the first statement of the function. If that line was number 76 in a file called "employee.m", using this command achieves the same results:

```
mdbg> b at employee.m:76
Breakpoint #1 at file employee.m, line 76
```

Each breakpoint is given a unique ID. In the above example, the ID for the breakpoint is 1. Use this ID if you want to remove the breakpoint. For example:

```
mdbg> bd 1
```

One final note on stepping -- the debugger remembers the last stepping command ("s", "n", "o", or "g") you executed, so pressing enter at the prompt will execute that last command.

### 8.1.2 Getting Info

An crucial factor in debugging is to check the contents of you variables to make sure they contain the right data. You can display variables once or you can *watch* them meaning that you can see them every time the debugger stops. Use the "v" to just view a variable. Just by itself, the "v" command displays all local variables for the current function. If you want to see a variable declared somewhere else, you must specify the variable with the command. For example, if you want to view the variable "name"of the instance "emp", enter this command:

```
mdbg> v emp.name
```

To watch the variable instead, enter this command:

```
mdbg> w emp.name
Watchpoint #5 emp.name
```

Like breakpoints, each watchpoint is given a unique ID. In the above example, the ID for the watchpoint is 5. Use this ID if you want to remove the watchpoint. For example:

```
mdbg> wd 5
```

Watchpoints are resolved every time the program is halted. This means if you specified a local variable to watch such as "x" and the program changes scope and "x" is defined in that scope as well, then you will see its value. If your watchpoints cannot be resolved in the current scope, an "undefined" error message will appear but watchpoint will still remain active until you specifically delete it.

# 9 ERRORS

Here are the list of errors generated by the parser and interpreter along with possible causes of the error and solutions. The errors are listed in numeric order, so refer to the error number generated by the program.

### #0 - syntax error at or near '*atom*'

Some possible causes for this error include:

- You forgot to terminate the previous statement with a ";".
- A reserve word such as a command or operator was used incorrectly.

### #1 - word '*atom* too long

Symbols and string constants can only be at most 255 characters long. Try renaming your symbol using fewer characters. For string constants, you may have forgotten to end the string with a double quote.

### #3 - parameter/shell only allowed in workspace

Program parameters and shell variables can only be defined in the workspace scope.

### #4 - macro name '*atom*' not valid

You used a reserve word such as a command or operator for the name of the macro.

### #5 - illegal class definition inside function

You can define a class within another class but not within a function.

### #6 -'public' only allowed in class definition

The `public` command can only be used to define public members of a class; it cannot be used in the workspace or functions.

### #7 - file '*filename*' not found

The library file does not exist or cannot be opened for reading.

### #9 - only primitive classes allowed

Program parameters and shell variables can be declared only with primitive types.

### #11 - incomplete class or function definition

You are missing a '}' somewhere in the program. Check your class or function definitions near the end of the program to make sure they are complete.

### #12 - missing 'then' clause

You tried to define an `else` clause without a `then` clause. Negate the `if` expression so that you can define a `then` clause rather than as `else` clause.

### #13 - parameter cannot be initialized

You tried to initialize a program parameter. The values for the program parameters are retrieve instead from the command-line options.

### #14 - labels can be used only in a 'switch' block

You tried to use `case` statements outside a switch block.

### #15 - 'switch' block not complete

You must define a block for a `switch` construct. A single statement for the body of the `switch` construct is not acceptable.

### #16 - only one parameter is allowed in 'catch' statement

You defined a `catch` function with more than one parameter. Since you can only throw one value at a time, this function can never be called.

### #17 - missing 'try' statement

You tried to define a `catch` function without a corresponding `try` statement.

### #18 - missing 'catch' statement

A `try` statement has no corresponding `catch` statement.

### #19 - symbol *'name'* not defined within scope

You referenced a symbol that is not defined within the current scope. Make sure the symbol is defined locally in the function or is a member of the current class.

### #20 - invalid use of symbol *'name'*

Some possible causes for this error include:

- You used a resident class as a variable name.
- You declared a variable from an element other than a class such as a function or another variable.
- You derived a class from one of the resident classes or from an element other than a class such as a function or another variable.
- You referred to a class element in one of your statements. You can only reference a function or variable.

### #21 - symbol *'name'* not member of *'class'*

The variable you reference is not a member of the class.

### #22 - missing ';' before *'atom'*

You may have forgotten to end the previous statement with a ';'.

### #23 - illegal use of operator *'atom'*

You tried to perform operations on variables other than that of primitive classes or you tried to perform an operation that does not result in a value (for example, arithmetic division on strings).

### #24 - cannot access private member *'atom'*

You tried to access a private member from outside the class. Declare the variable using the `public` keyword.

### #26 - cannot resolve function call *'function'*

You called a function that does not exists or does not have the appropriate number of arguments. If you tried to access a private function from outside the class, you will also get this error.

### #27 - category block already defined

You used `parameter`, `shell`, or `public` keyword within another one of those blocks.

```
parameter
{
    number a;
    parameter string b;  // error #27
}
```

### #28 - main function cannot have arguments

You cannot define function parameters to the main function. Use the `parameter` or `shell` keyword instead.

### #29 - expression must result into a *'type'*,

You coded an expression which does not result in the proper type for the given construct. For example, `if` and `while` constructs require that their expression result in a flag.

### #30 - should not call default constructor

Do not call the default constructor when declaring a variable. It is automatically called when the program is executed and the scope is entered. This is really not a critical error, but by indicating it as such, helps reduce code clutter.

### #31 - only primitive classes allowed for return type

Functions can only return a number, string, flag, or date.

### #32 - function *'name'* can only return *'type'*

Callback functions such as "main", "read/write", constructors, and destructors can only be defined to return a specific type. Destructors, for example, cannot return a value (void).

### #33 - expecting *'type'* for a return value

The function was defined with a given return type, but a `return` statement has a value of a different type so there is a mismatch.

### #34 - invalid 'break/continue' statement

You used a `break` or `continue` statement outside of a `do-while`, or `switch` construct.

### #35 - only numbers allowed for 'exit' value

Numeric values are only allowed for `exit` statements.

### #37 - stream must be argument to '~read/~write' function

The callback functions "~read" and "~write" must be defined with one stream parameter.

### #38 - function must return a value

You defined a function with a return type, but did not have any `return` statements in the function.

### #39 - duplicate symbol *'name'*

A symbol was used to define more than one variable or class in the same scope.

### #40 - macro text cannot be same as macro name

You used the macro name in the body of the macro.

```
macro count count * 10;  // error #40
```

### #42 - recursive definition of symbol *'name'*

A variable defined with a class where the variable is also defined, or a subclass is derived from the parent class.

```
class test1
{
    test1 a;  // error #42

    class test2 base test1 // error #42
    {
    }
}
```

### #43 - virtual function *'name'* defined in workspace

You defined a virtual function in the global class/workspace which is illegal since the global class is instantiated when the program is executed.

### #44 - cannot declare *'variable'* from abstract class

You declared a variable from an abstract class (i.e. a class that has at least one virtual function).

```
class test
{
    func1();
}

test a; // error #44
```

### #45 - class *'classname'* missing virtual function *'funcname'*

You derived a class from an abstract class, but you did not redefine the virtual function in your derived class.  You must either define the body of the functions or redefine it as a virtual function.

### #46 - pointless use of null stream

You should not `read` from or `write` to a null stream.  Null streams should only be used with the "exec" function.

### #47 - invalid command-line argument *'option'*

An unknown argument was found on the command-line.

### #48 - missing program name

The ReyScript Runner requires the name of the first library file.

### #49 - missing value for parameter/macro *'name'*

You did not supply a value for the parameter or macro on the command-line.

### #50 - shell program *'path'* invalid

The shell program you specified do not exist, or is not readable.

### #51 - cannot call constructor for primitive types

You tried to initialize a primitive variable by calling its constructor.  Use the assignment operator instead.

```
number a(5);    // error #51
number b = 6;   // ok
```

### #52 - duplicate macro definition for *'name'*

You define at least two macros with the same name.

### #53 - missing value for program parameter *'name'*

You defined a program parameter in the workspace, but you did not supply a value on the command-line.