

Quick Guide to Dao

Limin Fu (phoolimin@gmail.com)

September 19, 2009

Contents

1	Basics	3
1.1	How to run	3
1.2	Hello World	3
1.3	Basic Data Types	4
1.4	Data Storage Types	6
1.5	Data Declaration	7
1.6	Subindexing	7
1.7	Operators	8
2	Logic and Loop Controls	11
2.1	If Else	11
2.2	While	12
2.3	For	12
2.4	Do-Until	14
2.5	Do-While	14
2.6	Switch-Case	14
2.7	Other Controls	16
3	Input & Output	17
4	Routine or Function	19
4.1	Definition	19
4.2	Named Parameter	20
4.3	Parameter Type and Default Value	20
4.4	Constant Parameter	20
4.5	Parameter Passing by Reference	21
4.6	Parameter Grouping	21
4.7	Routine Overloading	21

- 4.8 Routine As First Class Object 22
- 4.9 Generator and Coroutine 23
- 5 Class and Object-Oriented Programming 25**
 - 5.1 Class Definition 25
 - 5.2 Class Instance 27
 - 5.3 Member Variable 27
 - 5.4 Setters, Getters and Overloadable Operators 28
 - 5.5 Method Overloading 29
 - 5.6 Inheritance 30
- 6 Module Loading 31**
 - 6.1 Compiling Time Loading 31
 - 6.2 Running Time Loading 32
 - 6.3 Path Management 32
- 7 Programming Tips 34**
 - 7.1 Handling Command Line Arguments 34
 - 7.2 Convenient Type Casting 35

(For version 1.1)

This document is licensed under *GNU Free Documentation License* .

Dao is a simple yet powerful object-oriented programming language featured by, optional typing, BNF-like macro system, regular expression, multidimensional numeric array, asynchronous function call for concurrent programming etc. Supporting for multi-threaded programming is also implemented as an integrate part of Dao. Dao also provides some built-in methods for functional style programming. Network programming and concurrent programming by message passing interface are supported as standard library of Dao. Moreover, Dao can be easily extended with C/C++, through a simple and transparent interface; and easily embedded into other C/C++ programs as well.

Chapter 1

Basics

1.1 How to run

From command line, type the name of the executable of Dao interpreter followed by a Dao script file:

```
dao [options] script_source.dao
```

Use *dao -h* to list the available options.

1.2 Hello World

Since *helloworld* example is the first example one would see in almost all tutorials for programming languages, here is the Dao version, just to introduce a few basic staffs in Dao.

```
# This is a simple demo:  
io.write( "Hello World!" );
```

```
{  
Here are multi-lines comments.  
Here are multi-lines comments.  
}
```

Output

Hello World!

As commented in the example, `#` and `{ # }` can be used to comment single line or multiple lines respectively. In many cases, semicolon can be omitted in the end of a statement, as long as there is no ambiguity; two exceptions for this are the **load,return** statements.

#	single line comments
{#}	multiple line comments
;	statement ending

In the above example, **io** (with alias **stdio**) is the basic IO library in Dao, **io.write()** writes to the standard output device (normally the computer screen) the resulting values of the expressions in the parameter list. Since Dao 1.1, some basic math functions and functional methods have been added as built-in functions. All other standard functions or methods are available as part of certain built-in library or as methods of certain type.

The available built-in libraries in Dao include: **io**, **std** (with alias **stdlib**), **math**, **reflect**, **coroutine**, **network**, **mpi** (message passing interface) and **mtlib** (multi-threading library) etc. The available methods for each data type and library can be listed by **stdlib.listmeth(obj)**. See also *Dao Library Reference*.

1.3 Basic Data Types

Dao language supports the following data type: integer, float, double, string, list, associative array (map, or dictionary), class object, complex number and numeric array etc. They can be created by assignment, enumeration or function calls.

int	integer	1234
	hex	0xff88
float	decimal	12.34
	scientific	1e-10 (lower case e)
double	decimal	12.34D
	scientific	1E-10 (upper case e)
complex	\$ as imaginary part	1+3\$
long	big integer	1234L
string	multi-bytes string (MBS)	'mbs'
	wide character string (WCS)	"wcs"
list	enumeration	{ 1, 2, 3 }
	range	{ init : step : size }
map	enumeration	{ "A"=>1, "B"=>2, "C"=>3 }
array (numeric array)	vector enumeration	[1, 2, 3]
	range	[init : step : size]
	matrix enumeration	[1, 2, 3; 4, 5, 6]
	with complex elements	[1, 2\$, 3]
tuple	tuple enumeration	(1, 2, "abc")
	enumeration with field name	(x => 1.5, y => 2.0)

Note 1: in numeric array creation by range, the *init* value can be a numeric array, in this case, a multi-dimensional numeric array will be created such that, the first "row" is a slice that equals to *init*, and the second "row" is equal to *init + step*, and so on,

```
a = [ [ 0 : 3 ] : 5 ];
io.writeln( a );

a = [ [ 0 : 3 ] : [ 1 : 3 ] : 5 ];
io.writeln( a );
```

Output

```
row[0,:]: 0 1 2
row[1,:]: 1 2 3
row[2,:]: 2 3 4
row[3,:]: 3 4 5
row[4,:]: 4 5 6
```

```
row[0,:]: 0 1 2
row[1,:]: 1 3 5
row[2,:]: 2 5 8
row[3,:]: 3 7 11
row[4,:]: 4 9 14
```

Note 2: tuple enumeration has to contain at least two items when enumerating without field names, since enumeration of zero or one item will be confused with the usual grouping of arithmetic expressions. However, they can be created by casting from lists and associative arrays:

```
ls = { "abc" };
tp = (tuple<string>) ls;
```

For convenience, it is possible to use keyword **typedef** to define an alias for a type:

```
typedef tuple<x:float,y:float,z:float> Point3D;
pt : Point3D = ( 1.0, 2.0, 3.0 );
io.writeln( pt.x );
```

Output

```
1.000000
```

MBS vs WCS

String quoted with single quotation symbol is stored as Multi-Bytes String (MBS) in UTF-8, which is more efficient to represent ASCII and latin letters; while string quoted with double quotation symbol is stored as Wide Character String (WCS) in Unicode, which is more efficient to represent other types of symbols, such as CJK (Chinese, Japanese and Korean) symbols. These two types of strings can be mixed together, one can be converted to the other automatically when necessary. But it's better to use one of them consistently.

1.4 Data Storage Types

Dao data with different scopes are stored differently, there are local/global/class constants and variables, and class instance variables which are specified by the following keywords.

1. **const** : used to declare constant data, which could be local, or global in a namespace or class, depending on the context it is used. If it is used in a class body, the declared data will be a constant member of the class. Otherwise, if it is used outside a function/routine, and is not nested inside any lexical scope, the data will be a global constant. In other cases, it will declare a local constant;
2. **global** : used to declare global variable;
3. **var** : used to declare local variable, or instance variable if used inside class body;
4. **static** : used to declare static variable, similar to that of C++.

1.5 Data Declaration

Example	Meaning
<i>variable = expression</i>	declaration by assignment; type can be fixed if it is inferable.
<i>variable : type</i>	variable with fixed type.
<i>variable : type = expression</i>	declaration by assignment with fixed type.

For `=`, if it is used to declare a constant, the expression in the right side of the assignment must be evaluable at compiling time. If `=` is used in class constructor with prefix `var` and its right side is constant, it will declare the class instance variable with a fixed type and default value.

type can be one of the type names in the previous table, it can also be a class name or C data type name. For *list, map, array*, the type can also be composite, e.g., *list<int>*, *map<string, list<float> >*, *array<double>* etc.

1.6 Subindexing

Some data types such as string and array support accessing their elements by sub-indexing.

Form	Meaning	Supported Types	Notes
data[i]	single element	string, list, hash, numeric array	multidimensional numeric array is treated as a vector!
data[from:to]	elements within index/key range	string, list, hash, numeric array	<i>from</i> and/or <i>to</i> can be omitted; by default <i>from</i> is the first index/key, <i>to</i> is the last
data[list]	elements specified by indices	string, list, numeric array	
data[numarray]	elements specified by indices	numeric array	
data[d1, d2, ...]	multiple dimensional index	numeric array	index for each dimension can be one of the first three forms

1.7 Operators

Dao Language supports a set of abundant operators to facilitate the writing of more expressive scripts. Many of these operators can be used for different data types.

Arithmetic:

Operator	Name	Supported Types
+	addition	number, string, complex, numeric array
-	subtraction	number, complex, numeric array
*	multiplication	number, complex, numeric array
/	division	number, complex, numeric array
%	mod	number, numeric array
**	pow	number, complex
++	self increment	number, complex, numeric array
--	self decrement	number, complex, numeric array
-	unary minus	number, complex, numeric array
+=	addition assignment	number, string, complex, numeric array
-=	subtracting assignment	number, complex, numeric array
*=	multiplication assignment	number, complex, numeric array
/=	divide assignment (XXX, name)	number, numeric array
%=	mod assignment (XXX, name)	number, numeric array

Note1: for numeric array, the binary operators in the table perform pairwise element operation; unary operators perform on each element.

Note2: for self in/decrement operators, there is no difference between prefix and postfix. They yield the value after the increment or decrement.

Numeric and String Comparison:

Operator	Name	Supported Types
==	equal	number, string, complex
!=	not equal	number, string, complex
<	less than	number, string
>	greater than	number, string
<=	less than or equal	number, string
>=	greater than or equal	number, string

Boolean Logic:

Operator	Name	Supported Types
&&	and	number
	or	number
!	not	number

Note, to make code more readable, *and*, *or* and *not* are also supported as equivalent operators. And the *and* and *or* operator behave in the same way as the Lua *and* or operators, namely, the last evaluated operand is returned as the result. This means, when the first operand can determine the result, return the first operand, otherwise return the second. When the operands have values of 0 or 1, this behaviour is exactly the same as normal boolean logic. The following lists some examples for other cases,

```

10 && 0 => 0
0 && 10 => 0
10 && 20 => 20
20 && 10 => 10

10 || 0 => 10
0 || 10 => 10
10 || 20 => 10
20 || 10 => 20

```

Properly exploiting the behaviour of the *and* or operators can simplify coding for some cases.

Type Operator

Operator	Name	Usage
?=	type equal	value1 ?= value2
?<	is type of	value ?< type

Assertion Operator

Operator `??` can be used to assert if the last operation is executed successfully, and return 1 on success, and 0 otherwise. In the case that the last operation can produce a value, this operator can also be used to specify an alternative value if that operation failed.

```
a = alist[i] ?? # check if item accessing by index is successful;
b = amap[key] ?? # check if a map has a particular key;
val = amap[key] ?? another # if amap has the key, assign its value, otherwise assign another;
```

Multiple Assignment:

```
( C, A, B, ... ) = ( A, B, C, ... )
( A, B ) = func();
```

the expression in the right side should yield a list or tuple, and each of the elements in the tuple/list is assigned accordingly to each of the variables in the left side. Extra elements are ignored.

Other Operators

	bits or
&	bits and
^	bits xor
	flip bits, unary
&=	bit and assignment
=	bit or assignment

Chapter 2

Logic and Loop Controls

To accomplish a task, a program often need to conditionally and/or repeatedly execute a block of codes according to whether a condition is fulfilled or not. This can be achieved by using logic and loop control statements. Currently *if-else* , *while* , *for* , *do-until* , *switch-case* , *break* and *skip* etc. are supported.

2.1 If Else

If a condition is true, execute a block of codes:

```
if( expr1 ){
    block1;
}elseif( expr2 ){
    block2;
}else{
    block3;
}
```

If *expr1* is true, *block1* is executed; otherwise, if *expr2* is true, *block2* is executed; otherwise, *block3* is executed; zero or more **elseif** and zero or one **else** statement can be used.

```
if( 2 > 1 ) io.writeln("2 is larger than 1.");
```

2.2 While

When a condition is true, **repeatedly** execute a block of codes:

```
while( expr ){
    block;
}
```

If *expr* is true, *block* is executed and repeated until *expr* becomes false, namely, while *expr* is true, execute *block* .

```
i = 0;
while( i < 5 ){
    io.writeln( i );
    i ++;
}
```

2.3 For

Dao supports different styles of for-looping, the most useful one is probably the following,

```
for( var = init_value : step_value : max_value ){
    block;
}
```

The looping will be started with *var = init_value* , then after each cycle, *var* is increased by *step_value* , the looping will stop when the value of *var* exceed the *max_value* . The *step_value* can be omit, in this case, value 1 is taken as the step. Please note, the initial, step and maximum values are always evaluated before the looping.

C/C++ style **for** looping is supported by Dao:

```
for( init; condition; step ){
    block;
}
```

The execution sequence of **for** statement is the following:

1. execute initial expression *init* , and goto 3;
2. execute *step* ;
3. evaluate the condition expression *condition* ;

4. check the value of *condition* : if true, goto 5; otherwise, goto 6;
5. execute *block* , and goto 2;
6. stop looping; and start to execute the statements after the loop body.

Usually, the C/C++ style **for** loop is equivalent to,

```
init;
while( condition ){
    block;
    step;
}
```

Dao also supports Python style *for-in-do* loop,

```
for( item in list ){
    block;
}
```

Multiple *in* can appear in one loop, and the items of the same indices from multiple lists are taken in each cycle. These lists should contain the same number of items, otherwise an exception will be arose, as shown in the following example.

```
for( item1 in list1; item2 in list2; ... ){
    block;
}
```

for-in can also be used for maps,

```
for( item in a_map ){
    block;
}
```

Examples,

```
for( i:=0; i<3; i++ ){
    io.writeln( i );
}

hash = { "b" => 11, "a" => 22, "e" => 33, "c" => 44 };
for( a in hash.key(); b in hash.value(); c in {1 : 1 : hash.size()-1 } ){
    #if a == "a" break
    io.writeln( a, b, c );
}
```

Output

```
ERROR( /Users/min/projects/DaoVM/tools/sdml2tex.dao : 6 ): key, member not exist, for instruction:
GETF      :      11 ,      12 ,      13 ;      6,
```

This example raises an exception, because the last list *c* contains one less element.

Note: if a single string is used in the condition expression in **if**, **while**, **for** statements, it returns true, if the string has length larger than zero, otherwise, returns false.

2.4 Do-Until

do-until can be used to execute a code *block* until a *condition* is fulfilled,

```
do{
    block;
} until ( condition )
```

```
a = 10;
do{
    c = 1
    io.writeln( "here", a -- );
}until( a == 0 )
```

2.5 Do-While

```
do{
    block;
} while ( condition )
```

Execute *block* , and then repeat executing it when the *condition* is true.

2.6 Switch-Case

Switch-case control provides a convenient way to branch the code and choose a block of code to execute based on the value of a object.

```

switch( value ){
  case C_1 : block_1
  case C_2 : block_2
  case C_3 : block_3
  ...
  default: block0
}

```

If the *value* equals to *C_i*, *block_i* will be executed. Here *C_i* must be a constant, but they can be of different types, that means, you can mix numbers and strings as case values. Unlike in C/C++, no **break** statement is required to get out of the **switch**.

If you want to execute the same block of codes for different case values, you just need to organize them together in the following way:

```

switch( value ){
  case C1, C2, C3 :
    block3
  ...
  default: block0
}

```

Namely, Dao allows one case entry to have multiple values. In this way, *block3* will be executed for case values *C1*, *C2* and *C3*. As a simple example,

```

a = "a";
switch( a ){
  case 1, "a" : io.write("case 1 or a");
  default : io.write("case default");
}

```

Output

```

case 1 or a

```

Dao also allows the use of a value range represented as *start ... end* as case entry, so that the corresponding code block is executed if the value in switch falls inside the range. Please note that, the range includes the boundary values, and if the ranges of different case entries overlaps, the entry with the lowest *start* value is used when the switch value belongs to multiple overlapping ranges.

```

switch( 5 ){
  case 1 ... 5 : io.writeln( 'case 1-5' );
  case 5 ... 10 : io.writeln( 'case 5-10' );
}

```

```
case 10 ... 11 : a = 1;
}
```

Output

```
case 1-5
```

2.7 Other Controls

break can be used to exit a loop, and **skip** can be used to skip the rest part of script and start the next cycle of a loop. **skip** is equivalent to **continue** in C/C++.

```
for( i=0; i<5; i++ ){
    io.writeln( i );
    if( i == 3 ) break;
}
```

Chapter 3

Input & Output

Few programs don't need to deal with files or some kind of Input/Output. Dao language supports the basic operations for IO. IO functionalities are accessible through **io** library. **io.read()** , **io.write()** , **io.writeln** and **io.writef()** can be used to read from and write to the standard IO device. To read and write a file, **io.open()** can be used to open a file and create a file stream object, which can use the common methods such as **read()** , **print()** , **println()** and **printf()** to perform reading and writing. There are also other methods such as **eof()** , **tell()** and **seek()** to check or set stream position. **sstream()** can be used to created a string stream.

For more informations about this library, please see *The Dao Library Reference—daoweb.dao?page*

Example,

```
# Open a file for writing:
fout = io.open( "test1.txt", "w" );

# Write to the file:
fout.write( "log(10)=", math.log( 10 ) );

# Open a file for reading:
fin = io.open( "test2.txt", "r" );

# while there is something to read:
while( ! fin.eof() ){
    # Read from the file:
    line = fin.read();
    # Write to std out:
    io.write( line );
}

# Read from std in:
d = io.read();
```

```
io.writeln( d );
```

Chapter 4

Routine or Function

Routine is a block of codes, once defined, can be used in different places at different time repeatedly. It can accept parameters to changes its behaviour. It may also return results to its callee.

4.1 Definition

Dao routines are declared with keyword **routine** or **function** or **sub** (which is exactly equivalent to **routine**),

```
routine func( a, b )
{
    io.writeln( a, b );
    a = 10;
    b = "test";
    return a, b; # return more than one results.
}
```

```
r1, r2;
( r1, r2 ) = func( 111, "AAA" );
r3 = func( r1, r2 );
io.writeln( "r1 = ", r1 );
io.writeln( "r2 = ", r2 );
io.writeln( "r3 = ", r3 );
```

Output

```
111 AAA
10 test
r1 = 10
r2 = test
r3 = ( 10, test )
```

4.2 Named Parameter

In Dao the function parameters are named, and parameter values can be passed in by name:

```
func( b => 123, a => "ABC" );
```

4.3 Parameter Type and Default Value

It is also possible to specify the type or the default value of a parameter.

```
routine MyRout( name : string, index = 0 )
{
  io.writeln( "NAME = ", name )
  io.writeln( "INDEX = ", index )
}
```

Here *name* is specified as string, and *index* is specified as number with default value 0. If a routine is called with wrong type of parameters, or no value is passed to a parameter without a default value, an exception will be issued and the execution will abort.

Default value can be defined for a parameter before another parameter which does not have default value. In this case, if you want to use the default value for that parameter with default value, you need to pass value by name to the parameter without default value:

```
routine MyRout2( i=0, j ){ io.writeln( i, " ", j ) }
MyRout2( j => 10 )
```

4.4 Constant Parameter

Constant parameter can be specified by adding "const" in front of the parameter type:

```
routine Test( a : const list<int> )
{
  a[1] = 100; # error !!!
  io.writeln( a );
}
```

```
a = { 1, 2, 3 }
Test( a );
```

4.5 Parameter Passing by Reference

Parameter passing by reference can be done by adding **&** in front of a parameter. Only local variables of primitive types can be passed as references, and references can be only created in parameter list.

```
routine Test( p : int )
{
    p += p;
}
i = 10;
Test( & i );
io.writeln( i );
```

4.6 Parameter Grouping

Dao also supports parameter grouping as in Python. Parameter grouping is defined by putting a pair of parenthesis around one or more parameters in a function prototype. When a tuple is passed as parameter to a function with parameter grouping in the corresponding position, and if the items of the tuple have types compatible to those parameters in the grouping, the tuple will be expanded with items passed as corresponding parameters.

```
routine Test( a : int, ( b : string, c = 0 ) )
{
    io.writeln( a, b, c );
}
t = ( 'abc', 123 )
Test( 0, t )
```

4.7 Routine Overloading

Routine overloading by parameter types is also supported in Dao, which means that multiple routines can be defined with the same name, but different parameters.

```

routine MyRout( index : int, name = "ABC" )
{
  io.writeln( "INDEX = ", index )
  io.writeln( "NAME = ", name )
}

MyRout( "DAO", 123 ) # invoke the first MyRout()
MyRout( 456, "script" ) # invoke the second MyRout()

```

4.8 Routine As First Class Object

Dao also support first class functions / routines. They can be created in the following way:

```

foo = routine( x, y : TYPE, z = DEFAULT )
{
  codes;
}

```

The definition of such functions is identical to the normal function definition, except the following differences:

1. there is no need for a function name, but the created function must be assigned to a variable;
2. the default value expressions for parameters do not necessary to be constant expressions, they are evaluated at running time when the function is created;
3. the function body may contain variables defined in the "upper" function that creates it; depending on the type of the "upper" variable, its copy (for simple types) or reference will be used by the created function.

Here is an example,

```

a = "ABC";

rout = routine( x, y : string, z = a+a ){
  a += "_abc";
  io.writeln( "lambda ", a )
  io.writeln( "lambda ", y )
  io.writeln( "lambda ", z )
}

rout( 1, "XXX" );

```

4.9 Generator and Coroutine

If the function name is prefixed with `@` when it is called, this call will return a generator or coroutine body, so that it will yield a value or a tuple each time the generator or coroutine is called. Inside the generator and coroutine, value(s) emitted by the **yield** statement. Their execution will be suspended after they **yield**, and when they are resumed, they will start to execute from where they are suspended. The data returned by the **yield** statement is the parameters passed to them when they are resumed. When a **return** statement is executed or the end of the function is reached, the generator or coroutine will exit and become not resumable anymore.

```
# int => tuple<int,int>
routine gen1( a = 0 )
{
    k = 0;
    while( k ++ < 3 ) a = yield( k, a );
    return 0,0;
}
routine gen2( a = 0 )
{
    return gen1( a );
}
g = @gen2( 1 );
# parameters can be omitted the first time it's called;
# the first call may use the parameters
# that are used for creating the generator:
io.writeln( 'main1: ', g() );
io.writeln( 'main2: ', g( 100 ) );
io.writeln( 'main3: ', g( 200 ) );
```

```
routine foo( a = 0, b = '' )
{
    io.writeln( 'foo:', a );
    return yield( 2 * a, 'by foo()' );
}

routine bar( a = 0, b = '' )
{
    io.writeln( 'bar:', a, b );
    ( r, s ) = foo( a + 1, b );
    io.writeln( 'bar:', r, s );
    ( r, s ) = yield( a + 100, b );
    io.writeln( 'bar:', r, s );
    return a, 'ended';
}

co = @bar( 1, "a" );
```

```
io.writeln( 'main: ', co() );
io.writeln( 'main: ', co( 1, 'x' ) );
io.writeln( 'main: ', co( 2, 'y' ) );
# coroutine has been finished, the following will rise an exception.
io.writeln( 'main: ', co( 3, 'z' ) );
```

Generators and coroutines can also be created by the standard method **stdlib.coroutine()**, but then the values must be yielded by the method **stdlib.yield()**. Besides this, there is another important difference between these two ways of using generators and coroutines regarding Dao typing system.

The variables passed around by generators or coroutines created by prefixing the function with **@** are **type-checked** by the typing system, while generators and coroutines created by the **stdlib.coroutine()** method will not be type-checked.

Chapter 5

Class and Object-Oriented Programming

Dao supports object-oriented programming (OOP) by allowing user to define classes using keyword **class** and create instances of them by calling them in the same way as calling routines. A class is simply a set of member variables and functions, which define the properties and possible behaviours of the class. An instance (which is often called object) is the realization of a class with some properties being initialized to certain values. The functions of a class are usually used to operate on the class or its instance. Permission modifiers **private**, **protected** and **public** can be used to set permissions on the variables and functions. Optionally, permission modifiers can be followed by a colon. Class members are public by default.

5.1 Class Definition

Starting from Dao v1.1, the syntax for class definition has been changed slightly. Now parameters and codes are not allowed in class body.

```
class MyNumber
{
    private

    var value = 0;
    var name : string;

    public

    routine MyNumber( value = 0, s = "NoName" ){
        value = value;
    }
}
```

```

    name = s;
}

routine setValue( v ){ value = v }
routine getValue(){ return value }

routine setValue( v : float );
}

routine MyNumber::setValue( v : float )
{
    value = v;
}

```

Whenever an instance/object of a class is created, a specific routine is invoked to initialize the object. This kind of routine is called constructor for the class. Unlike other languages where a constructor should be explicitly defined as a member function of a class, the class body is the constructor for Dao class. Like in Python, the constructors are not used to create class instances, instead, an instance is created before, and then the constructor is called after to initialize the instance.

In the class body, whenever a variable is specified with keyword **var** , it is regarded as an instance variable. The data type of instance variable can be set by in the following way,

```
var variable : type;
```

Where *type* must be a valid type name. And *variable* will have to have the same type as *type* , or have to be an instance of *type* if it is a class.

The default value of instance variable can also be specified,

```
var variable = init_value;
```

Where *init_value* must also be a constant.

Within class methods, the special variable **self** represents the current class instance. Class methods may be declared inside class body and defined outside in the same way as in C++, but in Dao, one should make sure that, the parameter list must be exactly the same in the places for declaration and definition. If no instance variable is used in a method, this method can be invoked with class by *class.method(...)* .

Like in C++, virtual method can be declared by adding keyword **virtual** before the **routine** or **function** keyword.

5.2 Class Instance

Class instance can be created by invoking the constructor of the class in the same way as a function call,

```
obj1 = MyNumber(1);
```

Class instance may also be created by enumerating the members of a class,

```
obj2 = MyNumber{ 2, "enumerated" };
```

The names of instance variables may also be specified in enumeration,

```
obj3 = MyNumber{
    name => "enumerated";
    value => 3;
};
```

When you create a class instance using enumeration, the instance is created, and filled with the values in the enumeration. Instance creation by enumeration is much faster than creation by invoking class constructor, since no class constructor is called and there is no overhead associated with function call (parameter passing, running time context preparation for the call etc.). So such instance creation is very desirable for creating many instances for simple classes, in which there are no complicated initialization operations.

5.3 Member Variable

As mentioned above, instance variables are declared in class constructor using **var** keyword. Class constant can be declared using **const** keyword, and static member can be declared using **static** keyword as in C++:

```
class Klass
{
    const aClassConst = "KlassConst";
    static aClassStatic;
}
```

Here *aClassConst* will be constant belonging to a *Klass* . While *aClassStatic* will be a static variable in the class scope.

5.4 Setters, Getters and Overloadable Operators

Instead of defining `setXYZ()` methods, one can define `.XYZ=()` method as setter operator, so that modifying class member `XYZ` by `obj.XYZ=abc` will be allowed; similarly, if `.XYZ()` is defined, get the value by `obj.XYZ` will also be allowed:

```
class MyNumber0
{
    private

    var value = 0;

    public

    routine MyNumber0( v = 0 ){
        value = v;
    }

    operator .value=( v ){ value = v; io.writeln( "value is set" ) }
    operator .value(){ return value }
}

num = MyNumber0( 123 )
num.value = 456
io.writeln( num.value )
```

Output

```
value is set
456
```

As you may guess, accessing instance variable through getters and setters are much more expensive than using them as public variables! They should be used only when they make things more convenient (for example, when you want them to do extra work when a variable is accessed).

Other supported operators for overloaing include:

1. **operator** `=(...)` for assignment;
2. **operator** `()(...)` for function call;
3. **operator** `[](...)` for getting item(s);
4. **operator** `[]=(...)` for setting item(s);

Other operators will be supported in the future versions.

5.5 Method Overloading

Class methods can be overloaded in the same way as normal functions. Class constructor may also be overloaded by simply adding a method with the same name as the class. For example, class *MyNumber* can be modified to hold numeric value only:

```
class MyNumber
{
    private

    var value : int = 0;

    public

    routine MyNumber( value = 0 ){ # accept integer as parameter
        self.value = value;
    }

    # overloaded constructor to accept MyNumber as parameter:
    routine MyNumber( value : MyNumber ){ self.value = value.value }

    operator .value=( v : int ){ value = v }
    operator .value=( v : MyNumber ){ value = v.value }
    operator .value(){ return value }
}

num1 = MyNumber( 123 )
num1.value = 456
io.writeln( num1.value )

num2 = MyNumber( num1 )
io.writeln( num2.value )

num2.value = 789
io.writeln( num2.value )

num2.value = num1
io.writeln( num2.value )
```

Output

```
456
456
789
456
```

5.6 Inheritance

```

class ColorRGB
{
    var Red = 0;
    var Green = 0;
    var Blue = 0;

    routine ColorRGB( r, g, b ){
        Red = r;
        Green = g;
        Blue = b;
    }

    routine setRed( r ){ Red = r; }
    routine setGreen( g ){ Green = g; }
    routine setBlue( b ){ Blue = b; }

    routine getRed(){ return Red; }
    routine getGreen(){ return Green; }
    routine getBlue(){ return Blue; }
}

yellow = ColorRGB( 255, 255, 0 ); # create an instance.

```

The following will define a derived class of *ColorRGB*,

```

class ColorQuad : ColorRGB
{
    var alpha = 0; # alpha component for tranparency.

    routine ColorQuad( r, g, b, a ) : ColorRGB( r, g, b ){
        alpha = a;
    }
}

yellow2 = ColorQuad( 255, 255, 0, 0 ); # not tranparent.
yellow2.alpha = 127; # change to half tranparency.

```

In the definition of derived class, the parent class *ColorRGB* should be put after the derived class and be separated with `:`. If there are more than one parent classes, separate them with `,`. The parameters for derived class can be passed to parent classes in the way as shown in the example.

Derived class will automatically inherit constructors from its parent class, if it has only one parent class, and there is no redefined constructors with signatures the same as those to be inherited.

Chapter 6

Module Loading

Sometimes, it is important to modularize an application for easier maintenance. Dao provides several mechanism to load modules from files. Such modules can be written in Dao language, or in C/C++ language using C interfaces provided by Dao. Modules can be loaded at compiling time (static loading) or at running time (dynamic loading).

If the module is written in Dao, its scripts will be executed immediately after loading. When the module is loaded for the second time, the content of the script file will be examined to see if it is changed. If not, the previous loaded representation of the module will be use, and the compiled scripts in the module will be executed again. If yes, the module will be re-compiled.

6.1 Compiling Time Loading

The simplest example to load a module is:

```
load MyModule; # the same as: load "MyModule";
```

In this case, *MyModule.dao* will be searched in the current path and the library paths, if not found, *MyModule.so* or *MyModule.dll* will be searched.

Additional relative path can also be specified in the *load* statement:

```
load MyPath.MyModule; # the same as: load "MyPath/MyModule";
```

And in this case, the current path and the library paths will be searched for sub-directory *MyPatch* and a file named *MyModule.dao* , or *MyModule.so* , or *MyModule.dll* in such sub-directory.

In both cases, all the global variables, functions, classes and registered C++ types are loaded into the current name space. To restrict which objects should be loaded, one can use *import* :

```
load MyModule import name1, name2;
```

If you do not want them to be imported into the current namespace, do

```
load MyModule as MyNS1;
load MyModule import name1, name2, name3 as MyNS2;
```

then new namespace *MyNS1* and *MyNS2* will be created, and can be used as *MyNS1.xyz* or *MyNS1.xyz(...)* .

In some cases, a C/C++ module may depend on other C/C++ modules, which can be specified with *require* ,

```
load MyModule require AnotherMod1, AnotherMod2;
```

Here the module names after *require* must be the file names (excluding path and suffix) of the required modules. If there are multiple modules with the same name loaded from different paths, only the last loaded one will be considered.

6.2 Running Time Loading

To load modules dynamically, use function *stdlib.load(mod)* , where *mod* must be the path and file name to a module. The global variables, routines and classes defined in that module will be loaded in to the current name space.

6.3 Path Management

If a relative path is given for a module, the current directory where the interpreter is invoked will be searched first, then the library paths are searched until the module is found. For the library paths, the last path added into library paths is search first. The current path and library paths can be changed by *@@PATH(+path)* at compiling time:

* Set current path,

```
@@PATH( "/home/guest" )
```

* Add path (in front),

```
@@PATH( + "./mypath/" )
```

If a relative path is provided, then it is supposed to be relative to the current path or one of the library paths. If there is a file named *addpath.dao* under *./mypath*, this file will be loaded and interpreted. So one can put the paths for the sub-directories of *./mypath* in *addpath.dao*, so that they can also be added by *@@PATH()*, an example of such file:

```
# Sample addpath.dao file:
@@PATH( + "subdir1" )
@@PATH( + "subdir2" )
@@PATH( + "subdir1/subsub" )
...
```

* Remove path,

```
@@PATH( - "./mypath/" )
```

Similarly, If there is a file named *delpath.dao* under *./mypath*, this file will be loaded and interpreted.

The following paths are added into the library paths by default: (1) a system wide path: */usr/lib/dao* (or: *C:dao* for Windows); (2) a user specific path under the user's home directory: */dao*; (3) a path defined in the environment variable *DAO_DIR*, if it exists. As mentioned before, the paths are searched in the reverse order they are added.

Chapter 7

Programming Tips

7.1 Handling Command Line Arguments

In Dao, the command line arguments are stored in two global variables: *CMDARG* and *ARGV*. *CMDARG* is a map that maps the argument names and positions to argument values; and *ARGV* is a list containing the argument values. For example, if *dao cmdarg.dao -arg1 -arg2 abc -arg3=def arg4=10* is executed from a shell, *CMDARG* and *ARGV* will contain the following elements,

```
CMDARG = { arg1 => arg1, arg2 => abc, arg3 => def, arg4 => 10, 1 => cmdarg.dao, 2 =>
arg1, 3 => abc, 4 => def, 5 => 10 }
ARGV = { cmdarg.dao, arg1, abc, def, 10 }
```

The rules of parsing command line arguments are the following,

Form	Argument Name	Argument Value
-arg	arg	arg
-arg value	arg	value
-arg=value	arg	value
arg=value	arg	value
- -arg	arg	arg
- -arg value	arg	value
- -arg=value	arg	value

Though the user can interpret the meaning of the arguments using *CMDARG* and *ARGV*, there is a much simpler automatic way to interpret the arguments. In fact, one can define a *main()* function with the argument names as parameter names, then when the script is executed, the *main()* function is invoked with the argument values as parameters.

If the arguments do not match to the parameter requirements of the *main()* function, the execution will be aborted, and the documentation of *main()* will be printed. A default value can be set as the default parameter for optional arguments.

```
# the main function for script.dao:
routine main( name : string, index=0 )

#{
  Documentation of this function, it may contain something like:
  Usage: dao script.dao name=abc [index=0]
#}
{
  ...
}
```

For this example, the script must be invoked with a string as the first argument, and an optional number as the second. So it must be invoked as the following,

```
dao script.dao xyz
dao script.dao xyz 10
dao script.dao name=xyz index=10
dao script.dao index=10 name=xyz
...
```

7.2 Convenient Type Casting

Most of the Dao data types can be converted into another type by simple type casting, just as an example,

```
num = 123456789;
str = (string) num;
ls = (list<int>) str;
ar = (array<int>) ls;
tup = (tuple<float,float>) ls[0:1];
io.writeln( num );
io.writeln( str );
io.writeln( ls );
io.writeln( ar );
io.writeln( tup );

ar += 'a'[0] - '1'[0];
ls = (list<int>) ar;
str = (string) ls;
io.writeln( ar );
io.writeln( ls );
```

```

io.writeln( str );

ar2 = [ [ 65 : 3 ] : 5 ];
ls2 = (list<list<int> >) ar2;
ls3 = (list<string>) ar2;
ls4 = (list<string>) ls2;
io.writeln( ar2 );
io.writeln( ls2 );
io.writeln( ls3 );
io.writeln( ls4 );

```

Output

```

123456789
123456789
{ 49, 50, 51, 52, 53, 54, 55, 56, 57 }
[ 49, 50, 51, 52, 53, 54, 55, 56, 57 ]
( 49.000000, 50.000000 )
[ 97, 98, 99, 100, 101, 102, 103, 104, 105 ]
{ 97, 98, 99, 100, 101, 102, 103, 104, 105 }
abcdefghi
row[0,:]: 65 66 67
row[1,:]: 66 67 68
row[2,:]: 67 68 69
row[3,:]: 68 69 70
row[4,:]: 69 70 71

{ { 65, 66, 67 }, { 66, 67, 68 }, { 67, 68, 69 }, { 68, 69, 70 }, { 69, 70, 71 } }
{ ABC, BCD, CDE, DEF, EFG }
{ ABC, BCD, CDE, DEF, EFG }

```