

JCL3 Language Guide

Release A02

Cino Group
PC Worth Int'l Co., Ltd.



Notice

Information in this document is subject to change without notice and does not represent a commitment on the part of PC Worth. PC Worth makes no warranty of any kind with regard to this publication, including, but not limited to, implied warranty of merchantability and fitness for any particular purpose. PC Worth shall not be liable for errors contained herein or for incidental consequential damages in connection with the furnishing, performance, or use of this publication. This publication contains proprietary information that is protected by copyright. All right reserved. No part of this publication may be photocopied, reproduced or translated into any language, in any forms, in an electronic retrieval system or otherwise, without prior permission of PC Worth.

No license is granted, either expressly or by implication, or otherwise under any patent right or patent, covering or relating to any combination, system, apparatus, machine, material, method, or process in which PC Worth products might be used. An implied license only exists for equipment, circuits, and subsystems contained in PC Worth products.

The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only on accordance with the terms of the agreement. It is against the law to copy and software on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

Cino Group

PC Worth Int'l Co., Ltd.

© Copyright Cino Group, 2002. All right reserved.

© Copyright PC Worth Int'l Co., Ltd., 2002. All rights reserved.

Table of Contents

1. Welcome to JCL3	1
Language Features	
▪ Data Types and Operators	
▪ Statements and Subroutines	
▪ Run-Time Library and Functions	
How this document is organized	
2. Overview of the Language.....	5
Program Structure	
Variables and Subroutines	
Strings	
Referencing and Dereferencing	
Arrays	
3. Data Types, Constants and Variables	15
Predefined Data Types	
▪ Number	
▪ String	
▪ Reference	
▪ Array	
▪ Null	
Variables	
4. Identifiers and Keywords.....	21
Identifiers	
Keywords	
5. Operators	25
Unary Operators	
Binary Operators	
▪ Arithmetic Operators	
▪ Relational Operators	
▪ Boolean Operators	
▪ Bitwise Operators	
Mixing Integer and Floating Point Arithmetic	

Table of Contents

6. Statements	31
Variable Declaration Statements	
Assignment Statements	
Conditional and Looping Statements	
▪ if	
▪ if-else	
▪ switch	
▪ while	
▪ do...while	
▪ for	
▪ break, return, continue	
7. Subroutines	41
Declaring Subroutines	
Parameter Passing Mechanism	
Referencing Variables	
Returning Values	
8. Functions	47
9. Arrays	49
Creating Arrays	
Indexing Arrays	
Arrays and Variables	

1. Welcome to JCL3

JCL is a simple, powerful, easy-to-learn, easy-to-use computer programming language with English-like statements and mathematical notations. With JCL you will be able to write both simple and complex programs to run on your terminal. You will also be able to modify existing software that is written in JCL. This enables the application to be used in ways not envisioned by the programmer, thus providing the application with much more flexibility and power.

JCL is not only a Basic-like interpreted language, but also a programmer's library that permits a programmer to develop sophisticated platform-independent software. In addition to providing the JCL language, the built-in function of library provides facilities for screen management, keymaps, peripheral control, low-level terminal I/O, database manipulation, etc.

However, this document is concerned only with the JCL3 language and does not address these other features of the JCL3 library and built-in functions. For information about the other components of the library and built-in functions, the reader is referred to the JCL3 Programmer's Reference.

Language Features

The JCL (Job Control Language) features variables, branching and looping constructs, user-defined subroutines, data types, and built-in functions.

Data Types and Operators

The language provides built-in support for null, string, number (signed and unsigned integers, long and short integers, double precision floating point), and multi-dimensional array types. To facilitate the construction of sophisticated data structures, a “reference” type was added to the language. The reference type provides much of the same flexibility as pointers in other languages.

The language provides standard arithmetic operations such as addition, subtraction, multiplication, and division. It also provides support for modulo arithmetic as well as operations at the bit level, e.g., exclusive-or. Any binary or unary operator may be extended to work with any data type. For example, the addition operator (+) has been extended to work between string types to permit string concatenation. The binary and unary operators work transparently with array types. For example, if a and b are arrays, then a + b produces an array whose elements are the result of element by element addition of a and b. This permits one to do vector operations without explicitly looping over the array indices.

Statements and Subroutines

The JCL language supports several types of looping constructs and conditional statements. The looping constructs include while, do...while, and for. The conditional statements include if and if...else.

User defined subroutines may be defined to return zero, one, or more values. subroutines that return zero values are similar to “procedures” in languages such as PASCAL. The local variables of a subroutine are always created on a stack allowing one to create recursive subroutines. Parameters to a subroutine are always passed by value and never by reference. However, the language supports a reference data type that allows one to simulate pass by reference.

Unlike many interpreted languages, JCL allows subroutines to be dynamically loaded. It also provides constructs specifically designed for error handling and recovery as well as debugging aids.

Subroutines and variables may be declared as private belonging to a namespace associated with the compilation unit that defines the subroutine or variable. The idea implementation stems from the C language and should be quite familiar to any one familiar with C.

Run-Time Library and Functions

Built-in functions that compose the JCL run-time library are called Functions. For examples, there are almost hundred JCL functions available to every JCL application, including string and database manipulation, data entry, screen management, terminal I/O and communication, and some miscellaneous functions.

How this document is organized

The JCL3 language guide is divided into 9 chapters:

Chapter 1, “ Welcome to JCL3 ”

Introduces the language features of JCL and important elements, and explain the organization of this document for user’s convenience.

Chapter 2, “ Overview of the language ”

Summarizes the basic structure and key elements of JCL program to give the reader a feel for the JCL language, its syntax, and its capabilities.

Chapter 3, “ Data Types, Constants and Variables “

Covers the predefined data types which JCL can handle, constant, and how to declare the variables.

Chapter 4, “ Identifiers and Keywords”

Describes how to give names to constants, variables and subroutines. It also mentions the restrictions upon the actual characters that make up an identifier. Furthermore, the reserved keywords are also briefing here.

Chapter 5, “ Operators ”

Describes all operators can be used in JCL and how to use operators for desired purpose.

Chapter 6, “ Statements ”

Describes the syntax of all statements, including variable declaration, assignment, conditional and looping, etc.

Chapter 7, “ Subroutines ”

Describes the subroutine declaration, parameter pass mechanism, and related topics.

Chapter 8, “ Functions ”

Surveys the coverage of built-in functions.

Chapter 9, “ Arrays ”

Describes how arrays are defined and used in the JCL language

2. Overview of Language

The purpose of this chapter is to give the reader a feel for the JCL language, its syntax, and its capabilities. The information and examples presented in this chapter should be sufficient to provide the reader with the necessary background to understand the rest of the document.

Program Structure

The JCL is a free-text format, case-insensitive style languages, the program can be developed under any generic text editor. A JCL program consists of one or more subroutines, the main program just be treated as a subroutine which must declared before calling. Furthermore, JCL executes the program from the end of the source code, to list main program subroutine at the end of source code is necessary.

A subroutine consists of one or more statements, and a statement might be a function, an expression assignment, etc. Each statement has to end with semi-colon - “;”, and all statements must be enclosed by braces - “{” and “}”. If user has to put comment in program, the double slash - “//” will be added. Anything listed after “//” will not affect program execution.

A simple program may look like following:

```

number variable 1;    // variable declaration
string variable 2;

sub main-program();  // main-program subroutine declaration
sub subroutine();    // subroutine declaration

sub main-program()   // main-program subroutine body
{
    statements;
    statements;
    subroutine();    // calling subroutine
    function();      // calling function
    .....;
}

sub subroutine()     // subroutine body
{
    statements;
    .....;
}

main-program();     // calling main program

```

Variables and Subroutines

JCL is different from many other interpreted languages in the sense that all variables and subroutines must be declared before they can be used.

Variables are declared using the variable keywords - “ null “, “ number ” , “ string ” , “ reference ” , and “ array “. The examples are listed below:

```
number x, y, z;
string a, b, c;
array m, n;
```

The first example declares three number type variables, x, y, and z. Note the semicolon at the end of the statement. All JCL statements must end in a semi-colon (;).

It is necessary to specify the data type of a JCL variable for identification purpose. The data type of a JCL variable is determined upon declaration, and variable assignment will be

```
x = 3;
y = sin(5.6);
c = "I think, therefore I am.";
```

x will be an integer, y will be a double, and c will be a string. Finally, one can combine variable declarations and assignments in the same statement:

```
number x = 3, y = sin(5.6);
string c = "I think, therefore I am.";
array m = number [10];
```

Most subroutines are declared using the keyword - “ sub ”. A simple example is

```
sub compute_average(x, y)
{
    number s = x + y;
    return s / 2.0;
}
```

which defines a subroutine that simply computes the average of two numbers and returns the result. This example shows that a subroutine consists of three parts: the subroutine name, a parameter list, and the subroutine body.

The parameter list consists of a comma separated list of variable names. It is not necessary to declare variables within a parameter list; they are implicitly declared. However, all other local variables used in the subroutine must be declared. If the subroutine takes no parameters, then the parameter list must still be present, but empty:

```
sub go_left_5()
{
    go_left(5);
}
```

The last example is a subroutine that takes no arguments and returns no value. Some languages such as PASCAL distinguish such objects from subroutines that return values by calling these objects procedures. However, JCL, like C, does not make such a distinction.

The language permits recursive subroutines, i.e., subroutines that call themselves. The way to do this in JCL is to first declare the subroutine using the form:

```
sub subroutine-name();
```

It is not necessary to declare a parameter list when declaring a function in this way.

The most famous example of a recursive subroutine is the factorial subroutine. Here is how to implement it using JCL:

```
sub factorial(); //declare it for recursion

sub factorial(n)
{
    if (n < 2) return 1;
    return n * factorial (n - 1);
}
```

This example also shows how to mix comments with code. JCL uses the “//” character to start a comment and all characters from the comment character to the end of the line are ignored.

Strings

Perhaps the most appealing feature of any interpreted language is that it frees the user from the responsibility of memory management. This is particularly evident when contrasting how JCL handles string variables with a lower level language such as C. For example, consider a subroutine that concatenates three strings in C, the simplest subroutine would look like:

```
char *concatenate_3strings(char *a, char *b, char *c)
{
    unsigned int len;
    char *result;
    len = strlen(a) + strlen(b) + strlen(c);
    if (NULL == (result = (char *) malloc(len + 1)))
        exit (1);
    strcpy(result, a);
    strcat(result, b);
    strcat(result, c);
    return result;
}
```

Even this C example is misleading since none of the issues of memory management of the strings has been dealt with. The JCL language hides all these issues from the user.

Binary operators have been defined to work with the string data type. In particular the + operator may be used to perform string concatenation for programmer's convenience. The actual example is listed below:

```
sub concatenate_3strings(a, b, c)
{
    return a + b + c;
}
```

Referencing and Dereferencing

The unary prefix operator (&), may be used to create a reference to an object, which is similar to a pointer in other languages. References are commonly used as a mechanism to pass a subroutine as an argument to another subroutine as the following example illustrates:

```

sub compute_sum(function)
{
    number i, sum;

    sum = 0;
    for (i = 0; i < 10; i++)
    {
        sum += @function(i);
    }
    return sum;
}

number total_amount = compute_sum(&amount);
number total_quantity = compute_sum(&quantity);

```

Here, the subroutine `compute_sum` applies the subroutine specified by the parameter `function` to the first 10 integers and returns the sum. The two statements following the subroutine definition show how the `amount` and `quantity` subroutines may be used.

Note the `@` operator in the definition of `compute_sum`. It is known as the dereference operator and is the inverse of the reference operator.

Finally, references may be used as an alternative to multiple return values by passing information back via the parameter list. Another example is

```

sub set_xyz(x, y, z)
{
    @x = 1;
    @y = 2;
    @z = 3;
}

number X, Y, Z;

set_xyz(&X, &Y, &Z);

```

which, after execution, results in X set to 1, Y set to 2, and Z set to 3. A C programmer will note the similarity of subroutine `set_xyz` to the following C implementation:

```
void set_xyz(int *x, int *y, int *z)
{
    *x = 1;
    *y = 2;
    *z = 3;
}
```

Arrays

The JCL language supports multi-dimensional arrays of all datatypes. For example, one can define arrays of references to functions as well as arrays of arrays. Here are a few examples of creating arrays:

```
array A = number [10];
array B = number [10, 3];
array C = number [1, 3, 5, 7, 9];
```

The first example creates an array of 10 integers and assigns it to the array variable A. The second example creates a 2-dimensional array of 30 integers arranged in 10 rows and 3 columns and assigns the result to B. In the last example, an array of 5 integers is assigned to the array variable C. However, in this case the elements of the array are initialized to the values specified. This is known as an inline-array.

JCL also supports something called an range-array. An example of such an array is

```
array C = number [1:9:2];
```

This will produce an array of 5 integers running from 1 through 9 in increments of 2.

Arrays are passed by reference to subroutines and never by value. This permits one to write subroutines which can initialize arrays. For example,

```
sub initiate_array(a)
{
    number i, imax;

    imax = length(a);
    for (i = 0; i < imax; i++)
    {
        a[i] = 7;
    }
}

array A = number [10];

initiate_array(A);
```

creates an array of 10 integers and initializes all its elements to 7.

There are more concise ways of accomplishing the result of the previous example. These include:

```
array A = [7, 7, 7, 7, 7, 7, 7, 7, 7, 7];
array A = number [10]; A[[0:9]] = 7;
```

The second and third methods use an array of indices to index the array A. In the second, the range of indices has been explicitly specified, whereas the third example uses a wildcard form.

Although the examples have pertained to integer arrays, the fact is that JCL arrays can be of any type, e.g.,

```
array C = string [10];
array D = reference [10];
```

create 10 element arrays of string, and reference types, respectively.

The language also defines unary, binary, and mathematical operations on arrays. For example, if X and Y are integer arrays, then X + Y is an array whose elements are the sum of the elements of X and Y. A trivial example that illustrates the power of this capability is

```
array X, Y;
X = [0:2*PI:0.01];
Y = 20 * sin(X);
```

which is equivalent to the highly simplified C code:

```
double *X, *Y;
unsigned int i, n;

n = (2 * PI) / 0.01 + 1;
X = (double *) malloc(n * sizeof(double));
Y = (double *) malloc(n * sizeof(double));
for (i = 0; i < n; i++)
{
    X[i] = i * 0.01;
    Y[i] = 20 * sin(X[i]);
}
```


3. Data Types, Constants and Variables

The current implementation of the JCL language permits several predefined data types, including null, number, string, array, and reference.

Constants are objects such as the number 3 or the string "hello". The actual data type given to a constant depends upon the syntax of the constant. Variables should be used after declaring, the actual data types given to variable depends upon the declaration. The following chapter describe the syntax of constants and variables of specific data types.

Predefined Data Types

The current version of JCL defines null, number, string, array, and reference types. The purpose of data types are discussed below.

Number

For user's convenience, the JCL language consolidates all number relative data types together including both signed and unsigned characters, short integer, long integer, plain integer, single and double precision floating points.

Integers

Generally speaking, on a 16 bit system, plain integers are 16 bit quantities with a range of -32767 to 32767. An plain integer constant can be specified in one of several ways:

- As a decimal (base 10) integer consisting of the characters 0 through 9, e.g., 127. An integer specified this way cannot begin with a leading 0. That is, 0127 is not the same as 127.
- Using hexadecimal (base 16) notation consisting of the characters 0 to 9 and A through F. The hexadecimal number must be preceded by the characters 0x. For example, 0x7F specifies an integer using hexadecimal notation and has the same value as decimal 127.
- In Octal notation using characters 0 through 7. The Octal number must begin with a leading 0. For example, 0177 and 127 represent the same integer.

Finally, a character constant may be specified using a notation containing a character enclosed in single quotes as 'a'. The value of the character specified this way will lie in the range 0 to 255 and will be determined by the ASCII value of the character in quotes. For example,

```
i = '0';
```

assigns to i the character 48 since the '0' character has an ASCII value of 48.

Any integer may be preceded by a minus sign to indicate that it is a negative integer.

Floating Point Numbers

Single and double precision floating point literals must contain either a decimal point or an exponent (or both). Here are examples of specifying the same double precision point number:

```
18.    18.0    18e0    1.8e1    180e-1    .18e2    0.18e2
```

Note that 18 is not a floating point number since it contains neither a decimal point nor an exponent. In fact, 18 is an integer. One may append the f character to the end of the number to indicate that the number is a single precision constant.

String

A string constant must be enclosed in double quotes as in:

```
"This is a string".
```

Although there is no imposed limit on the length of a string, string constants must be less than 256 characters in length. It is possible to go beyond this limit by string concatenation, e.g.,

```
"This is the first part of a long string"  
+ "and this is the second half"
```

Any character except a newline (ASCII 10) or the null character (ASCII 0) may appear explicitly in a string constant. However, these characters may be used implicitly using the mechanism described below.

The backslash character is a special character and is used to include other special characters (such as a new-line character) in the string. The special characters recognized are:

```
\ "      - double quote  
\ '      - single quote  
\ \      - backslash  
\ a      - bell character (ASCII 7)  
\ t      - tab character (ASCII 9)  
\ n      - new-line character (ASCII 10)  
\ e      - escape character (ASCII 27)  
\ xhhh   - character expressed in HEXADECIMAL notation  
\ ooo    - character expressed in OCTAL notation  
\ dnnn   - character expressed in DECIMAL
```

For example, to include the double quote character as part of the string, it must be preceded by a backslash character, e.g.,

```
"This is a \"quote\""
```

Reference

Objects of reference are created using the unary reference operator “&”. Such objects may be dereferenced using the dereference operator “@”. For example,

```
number table_index = &index;
number y = @table_index(1.0);
```

creates a reference to the `index` subroutine and assigns it to `table_index`. The second statement uses the dereference operator to call the subroutine that `table_index` references.

The reference type is useful for passing subroutines as arguments to other subroutines, or for returning information from a subroutine via its parameter list. The dereference operator is also used to create an instance of a structure.

Array

Variable of array type is known as container object. It is much more complicated than the simple data types discussed so far and each obeys a special syntax. For these reasons, it is discussed in a separated chapter later.

Null

Objects of null type can have only one value: `NULL`. About the only thing that you can do with this data type is to assign it to variables and test for equality with other objects. Nevertheless, null is an important and extremely useful data type. Its main use stems from the fact that since it can be compared for equality with any other data type, it is ideal to represent the value of an object which does not yet have a value, or has an illegal value.

As a trivial example of its use, consider

```
sub add_numbers(a, b)
{
    if (a == NULL) a = 0;
    if (b == NULL) b = 0;
    return a + b;
}

number c = add_numbers(1, 2);
number d = add_numbers(1, NULL);
number e = add_numbers(1,);
number f = add_numbers(,);
```

It should be clear that after these statements have been executed, c will have a value of 3. It should also be clear that d will have a value of 1 because NULL has been passed as the second parameter. One feature of the language is that if a parameter has been omitted from a subroutine call, the variable associated with that parameter will be set to NULL. Hence, e and f will be set to 1 and 0, respectively.

Variables

A variable must be declared before it can be used, otherwise an undefined name error will be generated. A variable is declared using the variable keyword, such as number, string, array and reference.

```
number x, y, z;
```

declares three variables, x, y, and z. This is an example of a variable declaration statement, and like all statements, it must end in a semi-colon. The actual type checking is performed at run-time. For example,

```
x = "This is a string";  
x = 1.2;  
x = 3;
```

results in x being set successively to a string and a number. Any attempt to use a variable before it has acquired a type will result in an un-initialized variable error.

It is legal to put executable code in a variable declaration list. That is,

```
variable x = 1, y = index(x);
```

are legal variable declarations. This also provides a convenient way of initializing a variable.

4. Identifiers and Keywords

This chapter describes how to give names to constants, variables and subroutines. It also mentions the restrictions upon the actual characters that make up an identifier. Furthermore, the reserved keywords are also briefing here.

Identifiers

The names given to constant, variables, subroutines and data types are called identifiers. There are some restrictions upon the actual characters that make up an identifier. An identifier name must start with a letter ([A-Z] or [a-z]), or a dollar sign. The rest of the characters in the name can be any combination of letters, digits, dollar signs, or underscore characters. However, all identifiers whose name begins with single or two underscore characters are reserved for internal use only and declarations of objects with such names should be avoided.

Examples of valid identifiers include:

```
Johnson
is_living_here
b9a6
$128
```

However, the following are not legal:

```
7123456
6abc
2rfgxx
+abc
.e12
_ $128
```

Although the maximum length of identifiers is unspecified by the language, the length should be kept below 64 characters.

Furthermore, the JCL is a case-insensitive oriented language. For example, following identifiers will be treated as the same one:

```
Johnson
JOHNSON
johnson
JOHNson
```

To avoid mistake, please refer to following section to understand all reserved keywords.

Keywords

The following identifiers are reserved by the language of current version and future release for using as keywords:

abs	and	andelse	array			
break						
case	chs	continue				
define	do	do...while				
else	error	exch	exit			
for	foreach	forever				
if						
loop						
mod	mod2					
next	not	null	number			
or	orelse					
pop	private	public				
reference	return					
shl	shr	sign	sqr	static	string	
struct	sub	switch				
typecast	typedef					
use	using					
variable						
where	while					
xor						

Furthermore, all function names of run-time library are reserved as keywords. Please refer to the JCL3 Programmer's Reference for details.

5. Operators

JCL supports a variety of operators that are grouped into three classes: assignment operators, binary operators, and unary operators.

An assignment operator is used to assign a value to a variable. They will be discussed more fully in the chapter of “ Statements “.

An unary operator acts only upon a single quantity while a binary operation is an operation between two quantities. The boolean operator not is an example of an unary operator. Examples of binary operators include the usual arithmetic operators +, -, *, and /. The operator given by - can be either an unary operator (negation) or a binary operator (subtraction); the actual operation is determined from the context in which it is used.

Binary operators are used in algebraic forms, e.g., $a + b$. Unary operators fall in one of two classes: postfix-unary or prefix-unary. For example, in the expression $-x$, the minus sign is a prefix-unary operator.

Not all data types have binary or unary operations defined. For example, while string data type objects support the + operator, they do not admit the * operator.

Unary Operators

The unary operators operate only upon a single operand. They include: not, `~`, `-`, `@`, `&`, as well as the increment and decrement operators `++` and `--`, respectively.

The boolean operator not acts only upon integers and produces 0 if its operand is non-zero, otherwise it produces 1.

The bit-level not operator (`~`) performs a similar function, except that it operates on the individual bits of its integer operand.

The arithmetic negation operator (`-`) is the most well-known unary operator. It simply reverses the sign of its operand.

The reference (`&`) and dereference (`@`) operators will be discussed later on. Similarly, the increment (`++`) and decrement (`--`) operators will be discussed in the context of the assignment operator.

Binary Operators

The binary operators may be grouped according to several classes: arithmetic operators, relational operators, boolean operators, and bitwise operators.

All binary and unary operators may be overloaded. For example, the arithmetic plus operator has been overloaded by the string data type to permit concatenation between strings.

Arithmetic Operators

The arithmetic operators include `+`, `-`, `*`, `/`, which perform addition, subtraction, multiplication, and division, respectively. In addition to these, JCL supports the mod operator `%`.

The data type of the result produced by the use of one of these operators depends upon the data types of the binary participants. If they are both integers, the result will be an integer. However, if the operands are not of the same type, they will be converted to a common type before the operation is performed. For example, if one is a floating point value and the other is an integer, the integer will be converted to a float. In general, the promotion from one type to another is such that no information is lost, if possible. As an example, consider the expression `8/5` which indicates division of the integer 8 by the integer 5. The result will be the integer 1 and not the floating point value 1.6. However, `8/5.0` will produce 1.6 because 5.0 is a floating point number.

Relational Operators

The relational operators are `>`, `>=`, `<`, `<=`, `==`, and `!=`. These perform the comparisons greater than, greater than or equal, less than, less than or equal, equal, and not equal, respectively. The result of one of these comparisons is the integer 1 if the comparison is true, or 0 if the comparison is false. For example, `6 >= 5` returns 1, but `6 == 5` produces 0.

Boolean Operators

There are only two boolean binary operators: `or` and `and`. These operators are defined only for integers and produce an integer result. The `or` operator returns 1 if either of its operands are non-zero, otherwise it produces 0. The `and` operator produces 1 if and only if both its operands are non-zero, otherwise it produces 0.

Neither of these operators perform the so-called boolean short-circuit evaluation.

For example, consider the expression:

```
(x != 0) and (1/x > 10)
```

Here, if x were to have a value of zero, a division by zero error would occur because even though $x \neq 0$ evaluates to zero, the and operator is not short-circuited and the $1/x$ expression would still be evaluated.

Bitwise Operators

The bitwise binary operators are defined only with integer operands and are used for bit-level operations. Operators that fall in this class include `&`, `|`, `shl`, `shr`, and `xor`. The `&` operator performs a boolean AND operation between the corresponding bits of the operands. Similarly, the `|` operator performs the boolean OR operation on the bits. The bit-shifting operators `shl` and `shr` shift the bits of the first operand by the number given by the second operand to the left or right, respectively. Finally, the `xor` performs an EXCLUSIVE-OR operation.

These operators are commonly used to manipulate variables whose individual bits have distinct meanings. In particular, `&` is usually used to test bits, `|` can be used to set bits, and `xor` may be used to flip a bit.

Mixing Integer and Floating Point Arithmetic

If a binary operation (+, -, * , /) is performed on two integers, the result is an integer. If at least one of the operands is a float, the other is converted to float and the result is float. However, JCL consolidates all number related objects into number type for convenience. User may just review following examples as notice.

```
11 / 2      --> 5 (integer)
11 / 2.0    --> 5.5 (float)
11.0 / 2    --> 5.5 (float)
11.0 / 2.0  --> 5.5 (float)
```

Finally note that only integers may be used as array indices, loop control variables, and bit operations.

6. Statements

Loosely speaking, a statement is composed of expressions that are grouped according to the syntax or grammar of the language to express a complete computation. Statements are analogous to sentences in a human language and expressions are like phrases. All statements in the JCL language must end in a semi-colon (;).

A statement that occurs within a subroutine is executed only during execution of the subroutine. However, statements that occur outside the context of a subroutine are evaluated immediately.

The language supports several different types of statements such as assignment statements, conditional statements, and so forth. These are described in detail in this chapter.

Variable Declaration Statements

Variable declarations were already discussed in chapter 3 “Data Types, Constants and Variables”.

For the sake of completeness, a variable declaration is a statement of the form

```
number variable-declaration-list;  
string variable-declaration-list;  
array variable-declaration-list;
```

where the variable-declaration-list is a comma separated list of one or more variable names with optional initializations, e.g.,

```
variable x, y = 2, z;
```

Assignment Statements

Perhaps the most well known form of statement is the assignment statement. Statements of this type consist of a left-hand side, an assignment operator, and a right-hand side. The left-hand side must be something to which an assignment can be performed. Such an object is called an value.

The most common assignment operator is the simple assignment operator (=). Simple of its use include

```
x = 3;
x = subroutine(10);
x = 34 + 27/y + subroutine(z);
x = x + 3;
```

In addition to the simple assignment operator, JCL also supports the assignment operators (+=) and (-=). Internally, JCL transforms

```
a += b;
```

to

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

Similarly, a -= b is transformed to a = a - b. It is extremely important to realize that, in general, a+b is not equal to b+a. This means that a+=b is not the same as a=b+a. As an example consider

```
a = "hello"; a += "world";
```

After execution of these two statements, a will have the value "helloworld" and not "worldhello".

Since adding or subtracting 1 from a variable is quite common, JCL also supports the unary increment and decrement operators ++, and --, respectively. That is, for numeric data types,

```
x = x + 1;
x += 1;
x++;
```

are all equivalent. Similarly,

```
x = x - 1;  
x -= 1;  
x--;
```

are also equivalent.

Strictly speaking, ++ and -- are unary operators. When used as x++, the ++ operator is said to be a postfix-unary operator. However, when used as ++x it is said to be a prefix-unary operator. The current implementation does not distinguish between the two forms, thus x++ and ++x are equivalent. The reason for this equivalence is that assignment expressions do not return a value in the JCL language as they do in C. Thus one should exercise care and not try to write C-like code such as

```
x = 10;  
while (--x) do_job(x); // Ok in C, but not in JCL
```

The closest valid JCL form involves a comma-expression:

```
x = 10;  
while (x--, x) do_job(x); // Ok in JCL and in C
```

Conditional and Looping Statements

JCL supports a wide variety of conditional and looping statements. These constructs operate on statements grouped together in blocks. A block is a sequence of JCL statements enclosed in braces and may contain other blocks. However, a block cannot include subroutine declarations. In the following, *statement-or-block* refers to either a single JCL statement or to a block of statements, and *integer expression* is an integer-valued expression, *next-statement* represents the statement following the form under discussion.

if

The simplest condition statement is the if statement. It follows the syntax

```
if (integer-expression) statement-or-block next-statement
```

If *integer-expression* evaluates to a non-zero result, then the statement or group of statements implied *statement-or-block* will get executed. Otherwise, control will proceed to *next-statement*.

An example of the use of this type of conditional statement is

```
if (x != 0)
{
    y = 1.0 / x;
    if (x > 0) z = 0;
}
```

This example illustrates two if statements where the second if statement is part of the block of statements that belong to the first.

if...else

Another form of if statement is the if...else statement. It follows the syntax:

```
if (integer-expression) statement-or-block-1 else
statement-or-block-2 next-statement
```

Here, if expression returns non-zero, *statement-or-block-1* will get executed and control will pass on to *next-statement*. However, if expression returns zero, *statement-or-block-2* will get executed before continuing with *next-statement*. A simple example of this form is

```
if (x > 0) z == 0; else y = " This is an error. ";
```

Consider the more complex example:

```
if (country == "America")
    if (city == "New York") found = 1;
else if (country == "Taiwan")
    if (city == "Taipei") found = 1;
else found = 0;
```

This example illustrates a problem that beginners have with if-else statements. The grammar presented above shows that the this example is equivalent to

```
if (country == "America")
{
    if (city == "New York") found = 1;
    else if (country == "Taiwan")
    {
        if (city == "Taipei") found = 1;
        else found = 0;
    }
}
```

It is important to understand the grammar and not be seduced by the indentation!

switch

The switch statement deviates the most from its C counterpart. The syntax is:

```
switch (x)
{ ...:... }
.
.
{ ...:... }
```

The ‘:’ operator is a special symbol which means to test the top item on the stack, and if it is non-zero, the rest of the block will get executed and control will pass out of the switch statement. Otherwise, the execution of the block will be terminated and the process will be repeated for the next block. If a block contains no : operator, the entire block is executed and control will pass onto the next statement following the switch statement. Such a block is known as the default case.

As a simple example, consider the following:

```
switch (x)
{x == 1 : Display(1,1,"Number is one.",0,21);}
{x == 2 : Display(1,1,"Number is two.",0,21);}
{x == 3 : Display(1,1,"Number is three.",0,21);}
{Display(1,1,"Number is greater than four.",0,21);}
```

Suppose *x* has an integer value of 3. The first two blocks will terminate at the `:` character because each of the comparisons with *x* will produce zero. However, the third block will execute to completion. Similarly, if *x* is 7, only the last block will execute in full. A more familiar way to write the previous example used the case keyword:

```
switch (x)
{case 1 : Display(1,1,"Number is one.",0,21);}
{case 2 : Display(1,1,"Number is two.",0,21);}
{case 3 : Display(1,1,"Number is three.",0,21);}
{Display(1,1,"Number is greater than four.",0,21);}
```

The case keyword is a more useful comparison operator because it can perform a comparison between different data types while using `==` may result in a type-mismatch error. For example,

```
switch (x)
{(x == 1) or (x == "a") : Display(1,1,"Number is one.",0,21);}
{(x == 2) or (x == "b") : Display(1,1,"Number is two.",0,21);}
{(x == 3) or (x == "c") : Display(1,1,"Number is three.",0,21);}
{Display(1,1,"Number is greater than four.",0,21);}
```

will fail because the `==` operation is not defined between strings and integers. The correct way to write this to use the case keyword:

```
switch (x)
{ case 1 or case "a" : Display(1,1,"Number is one.",0,21);}
{ case 2 or case "b" : Display(1,1,"Number is two.",0,21);}
{ case 3 or case "c" : Display(1,1,"Number is three.",0,21);}
{ Display(1,1,"Number is greater than four.",0,21);}
```

while

The while statement follows the syntax

```
while (integer-expression) statement-or-block next-statement
```

It simply causes statement-or-block to get executed as long as integer-expression evaluates to a non-zero result. For example,

```
i = 10;
while (i)
{
    i--;
    beep(10,10);
}
```

will cause the built-in function beep to get called 10 times. However,

```
i = -10;
while (i)
{
    i--;
    beep(10,10);
}
```

would loop forever (or until i wraps from the most negative integer value to the most positive and then decrements to zero).

If you are a C programmer, do not let the syntax of the language seduce you into writing this example as you would in C:

```
i = 10;
while (i--) beep(10,10);
```

The fact is that expressions such as i-- do not return a value in JCL as they do in C. If you must write this way, use the comma operator as in

```
i = 10;
while (i, i--) beep(10,10);
```

do...while

The do...while statement follows the syntax

```
do statement-or-block while (integer-expression);
```

The main difference between this statement and the while statement is that the do...while form performs the test involving integer-expression after each execution of statement-or-block rather than before. This guarantees that statement-or-block will get executed at least once.

A simple example is listed below:

```
i = 10
do
{
    beep(10,10);
    i = i - 1
}
while (i == 0);
```

for

Perhaps the most complex looping statement is the for statement; nevertheless, it is a favorite of many programmers. This statement obeys the syntax

```
for (init-expression; integer-expression; end-expression)
statement-or-block next-statement
```

In addition to statement-or-block, its specification requires three other expressions. When executed, the for statement evaluates init-expression, then it tests integer-expression. If integer-expression returns zero, control passes to next-statement. Otherwise, it executes statement-or-block as long as integer-expression evaluates to a non-zero result. After every execution of statement-or-block, end-expression will get evaluated.

This statement is almost equivalent to

```
init-expression;
while (integer-expression) { statement-or-block end-expression; }
```

The reason that they are not fully equivalent involves what happens when statement-or-block contains a continue statement.

Despite the apparent complexity of the for statement, it is very easy to use. To compute the sum of the first 10 integers, an example is listed below:

```
sum = 0;
for (i = 1; i <= 10; i++) sum += i;
```

break, return, continue

JCL also includes the non-local transfer subroutines return, break, and continue. The return statement causes control to return to the calling subroutine while the break and continue statements are used in the context of loop structures. Consider:

```
sub forever_loop()
{
    i = 1;
    while (i)    // forever loop
    {
        s1;
        s2;
        ..
        if (condition_1) break;
        if (condition_2) return;
        if (condition_3) continue;
        ..
        s3;
    }
    s4;
    ..
}
```

Here, a subroutine forever_loop has been defined that contains a forever loop consisting of statements s1, s2,...,s3, and three if statements. As long as the expressions condition_1, condition_2, and condition_3 evaluate to zero, the statements s1, s2,...,s3 will be repeatedly executed. However, if condition_1 returns a non-zero value, the break statement will get executed, and control will pass out of the forever loop to the statement immediately following the loop which in this case is s4. Similarly, if condition_2 returns a non-zero number, the return statement will cause control to pass back to the caller of forever_loop. Finally, the continue statement will cause control to pass back to the start of the loop, skipping the statement s3 altogether.

7. Subroutines

A user defined subroutine may be thought of as a group of statements that work together to perform a computation. While there are no imposed limits upon the number statements that may occur within a subroutine, it is considered poor programming practice if a subroutine contains many statements. This notion stems from the belief that a subroutine should have a simple, well defined purpose.

Declaring Subroutines

Like variables, subroutines must be declared before they can be used. The `sub` keyword is used for this purpose. For example,

```
sub factorial();
```

is sufficient to declare a subroutine named `factorial`. Unlike the `variable` keyword used for declaring variables, the `sub` keyword does not accept a list of names.

Usually, the above form is used only for recursive subroutines. In most cases, the subroutine name is almost always followed by a parameter list and the body of the subroutine:

```
sub subroutine-name (parameter-list) { statement-list }
```

The `subroutine-name` is an identifier and must conform to the naming scheme for identifiers discussed in chapter 3. The `parameter-list` is a comma-separated list of variable names that represent parameters passed to the subroutine, and may be empty if no parameters are to be passed. The body of the subroutine is enclosed in braces and consists of zero or more statements (`statement-list`).

The variables in the `parameter-list` are implicitly declared, thus, there is no need to declare them via a variable declaration statement. In fact any attempt to do so will result in a syntax error.

Parameter Passing Mechanism

Parameters to a subroutine are always passed by value and never by reference. To see what this means, consider

```
sub add_10(a)
{
    a = a + 10;
}

number b = 0;
add_10(b);
```

Here a subroutine `add_10` has been defined, which when executed, adds 10 to its parameter. A variable `b` has also been declared and initialized to zero before it is passed to `add_10`. What will be the value of `b` after the call to `add_10`? If JCL were a language that passed parameters by reference, the value of `b` would be changed to 10. However, JCL always passes by value, which means that `b` would retain its value of zero after the subroutine call.

JCL does provide a mechanism for simulating pass by reference via the reference operator. See the next section for more details.

If a subroutine is called with a parameter in the parameter list omitted, the corresponding variable in the subroutine will be set to `NULL`. To make this clear, consider the subroutine

```
sub add_two_numbers(a, b)
{
    if (a == NULL) a = 0;
    if (b == NULL) b = 0;
    return a + b;
}
```

This subroutine must be called with two parameters which is not recommended. However, we can omit one or both of the parameters by calling it in one of the following ways:

```
number s = add_two_numbers(2, 3);
number s = add_two_numbers(2, );
number s = add_two_numbers(, 3);
number s = add_two_numbers(, );
```

The first example calls the subroutine using both parameters; however, at least one of the parameters was omitted in the other examples. The interpreter will implicitly convert the last three examples to

```
number s = add_two_numbers(2, NULL);  
number s = add_two_numbers(NULL, 3);  
number s = add_two_numbers(NULL, NULL);
```

It is important to note that this mechanism is available only for subroutine calls that specify more than one parameter. That is,

```
number s = add_10();
```

is not equivalent to `add_10(NULL)`. The reason for this is simple: the parser can only tell whether or not `NULL` should be substituted by looking at the position of the comma character in the parameter list, and only subroutine calls that indicate more than one parameter will use a comma. A mechanism for handling single parameter subroutine calls is described in the next section.

Referencing Variables

One can achieve the effect of passing by reference by using the reference (&) and dereference (@) operators. Consider again the subroutine `add_10` presented in the previous section. This time we write it as

```
sub add_10(a)
{
    @a = @a + 10;
}

number b = 0;
add_10(&b);
```

The expression `&b` creates a reference to the variable `b` and it is the reference that gets passed to `add_10`. When the subroutine `add_10` is called, the value of `a` will be a reference to `b`. It is only by dereferencing this value that `b` can be accessed and changed. So, the statement `@a=@a+10;` should be read 'add 10' to the value of the object that `a` references and assign the result to the object that `a` references.

The reader familiar with C will note the similarity between references in JCL and pointers in C.

Returning Values

As stated earlier, the usual way to return values from a subroutine is via the return statement. This statement has the simple syntax

```
return expression-list ;
```

where expression-list is a comma separated list of expressions. If the subroutine does not return any values, the expression list will be empty. As an example of a subroutine that can return a value, consider

```
sub sum(x, y)
{
    number sum;

    sum = x + y;
    return sum;
}
```

It is extremely important to note that the calling routine must explicitly handle all values returned by a subroutine. Although some languages such as C do not have this restriction, JCL does and it is a direct result of a JCL subroutine's ability to return value and accept a parameters. Examples of properly handling the above subroutine include

```
number sum;

sum = sum(5, 4);
```

8. Functions

Built-in functions that compose the JCL run-time library are called Functions. For examples, there are almost hundred JCL functions available to every JCL application, including string and database manipulation, data entry functions, screen management, terminal I/O and communication, and some miscellaneous functions.

However, this document is concerned only with JCL language does not address other feature of JCL built-in function. For information about other component of the library, the reader is referred to the JCL Programmer's Reference.

9. Arrays

An array is a container object that can contain many values of one data type. Arrays are very useful objects and are indispensable for certain types of programming. The purpose of this chapter is to describe how arrays are defined and used in the JCL language.

Creating Arrays

The JCL language supports multi-dimensional arrays of all data types. Since the Array is a data type, one can even have arrays of arrays. To create a multi-dimensional array of desired-type use the syntax

```
Desired-type [dim0, dim1, ..., dimN]
```

Here dim0, dim1, ... dimN specify the size of the individual dimensions of the array. The current implementation permits arrays consist of up to 5 dimensions. When a numeric array is created, all its elements are initialized to zero. The initialization of other array types depend upon the data type, e.g., string type array is initialized to NULL.

As a concrete example, consider

```
a = number [10];
```

which creates a one-dimensional array of 10 numbers and assigns it to a. Similarly,

```
b = number [10, 3];
```

creates a 30 element array of numbers arranged in 10 rows and 3 columns, and assigns it to b.

Range Arrays

There is a more convenient syntax for creating and initializing a 1-dimensional arrays. For example, to create an array of ten integers whose elements run from 1 through 10, one may simply use:

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
```

Similarly,

```
b = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0];
```

specifies an array of ten doubles.

An even more compact way of specifying a numeric array is to use a range-array. For example,

```
a = [0:9];
```

specifies an array of 10 integers whose elements range from 0 through 9. The most general form of a range array is

```
[first-value : last-value : increment]
```

where the increment is optional and defaults to 1. This creates an array whose first element is first-value and whose successive values differ by increment. The last-value sets an upper limit upon the last value of the array as described below.

If the range array [a:b:c] is integer valued, then the interval specified by a and b is closed. That is, the kth element of the array x_k is given by $x_k = a + ck$ and must satisfy $a \leq x_k \leq b$. Hence, the number of elements in an integer range array is given by the expression $1 + (b-a)/c$.

The situation is somewhat more complicated for floating point range arrays. The interval specified by a floating point range array [a:b:c] is semi-open such that b is not contained in the interval. In particular, the kth element of [a:b:c] is given by $x_k = a + kc$ such that $a \leq x_k < b$ when $c \geq 0$, and $b < x_k \leq a$ otherwise. The number of elements in the array is one greater than the largest k that satisfies the open interval constraint.

Here are a few examples that illustrate the above comments:

```
[1:5:1]          ==> [1,2,3,4,5]
[1.0:5.0:1.0]    ==> [1.0, 2.0, 3.0, 4.0]
[5:1:-1]         ==> [5,4,3,2,1]
[5.0:1.0:-1.0]  ==> [5.0, 4.0, 3.0, 2.0];
[1:1]           ==> [1]
[1.0:1.0]       ==> []
[1:-3]          ==> []
```

Indexing Arrays

An individual element of an array may be referred to by its index. For example, `a[0]` specifies the zeroth element of the one dimensional array `a`, and `b[3,2]` specifies the element in the third row and second column of the two dimensional array `b`. As in C array indices are numbered from 0. Thus if `a` is a one-dimensional array of ten integers, the last element of the array is given by `a[9]`. Using `a[10]` would result in a range error.

A negative index may be used to index from the end of the array, with `a[-1]` referring to the last element of `a`, `a[-2]` referring to the next to the last element, and so on.

One may use the indexed value like any other variable. For example, to set the third element of an integer array to 6, use

```
a[2] = 6;
```

Similarly, that element may be used in an expression, such as

```
y = a[2] + 7;
```

Unlike other JCL variables which inherit a type upon assignment, array elements already have a type. For example, an attempt to assign a string value to an element of an integer array will result in a type-mismatch error.

One may use any integer expression to index an array. A simple example that computes the sum of the elements of 10 element 1-dimensional array is

```
number i, sum;

sum = 0;
for (i = 0; i < 10; i++) sum += a[i];
```

Unlike many other languages, JCL permits arrays to be indexed by other integer arrays. Suppose that `a` is a 1-dimensional array of 10 doubles. Now consider:

```
i = [6:8];
b = a[i];
```

Here, `i` is a 1-dimensional range array of three integers with `i[0]` equal to 6, `i[1]` equal to 7, and `i[2]` equal to 8. The statement `b = a[i];` will create a 1-d array of three doubles and assign it to `b`. The zeroth element of `b`, `b[0]` will be set to the sixth element of `a`, or `a[6]`, and so on. In fact, these two simple statements are equivalent to

```
b = number [3];
b[0] = a[6];
b[1] = a[7];
b[2] = a[8];
```

except that using an array of indices is not only much more convenient, but executes much faster.

More generally, one may use an index array to specify which elements are to participate in a calculation. For example, consider

```
a = number [1000];
i = [0:499];
j = [500:999];
a[i] = -1.0;
a[j] = 1.0;
```

This creates an array of 1000 doubles and sets the first 500 elements to -1.0 and the last 500 to 1.0. Actually, one may do away with the `i` and `j` variables altogether and use

```
a = number [1000];
a [[0:499]] = -1.0;
a [[500:999]] = 1.0;
```

It is important to understand the syntax used and, in particular, to note that `a[[0:499]]` is not the same as `a[0:499]`. In fact, the latter will generate a syntax error.

Often, it is convenient to use a rubber range to specify indices. For example, `a[[500:]]` specifies all elements of `a` whose index is greater than or equal to 500. Similarly, `a[[:499]]` specifies the first 500 elements of `a`. Finally, `a[[:]]` specifies all the elements of `a`; however, using `a[*]` is more convenient.

Now consider a multi-dimensional array. For simplicity, suppose that `a` is a 100 by 100 array of doubles. Then the expression `a[0, *]` specifies all elements in the zeroth row. Similarly, `a[* , 7]` specifies all elements in the seventh column. Finally, `a[[3:5]][6:12]` specifies the 3 by 7 region consisting of rows 3, 4, and 5, and columns 6 through 12 of `a`.

We conclude this section with a few examples.

Here is a function that computes the trace (sum of the diagonal elements) of a square 2 dimensional n by n array:

```
sub array_trace(a, n)
{
    number sum = 0, i;

    for (i = 0; i < n; i++) sum += a[i, i];
    return sum;
}
```

This fragment creates a 10 by 10 integer array, sets its diagonal elements to 5, and then computes the trace of the array:

```
a = number [10, 10];
for (j = 0; j < 10; j++) a[j, j] = 5;
the_trace = array_trace(a, 10);
```

We can get rid of the for loop as follows:

```
j = number [10, 2];
j[*,0] = [0:9];
j[*,1] = [0:9];
a[j] = 5;
```

Here, the goal was to construct a 2-dimensional array of indices that correspond to the diagonal elements of a, and then use that array to index a. To understand how this works, consider the middle statements. They are equivalent to the following for loops:

```
number i;
for (i = 0; i < 10; i++) j[i, 0] = i;
for (i = 0; i < 10; i++) j[i, 1] = i;
```

Thus, row n of j will have the value (n,n), which is precisely what was sought.

Another example of this technique is the subroutine:

```
sub unit_matrix(n)
{
    array a = number [n, n];
    array j = number [n, 2];
    j[* , 0] = [0:n - 1];
    j[* , 1] = [0:n - 1];

    a[j] = 1;
    return a;
}
```

This function creates an n by n unit matrix, that is a 2-dimensional n by n array whose elements are all zero except on the diagonal where they have a value of 1.

Arrays and Variables

When an array is created and assigned to a variable, the interpreter allocates the proper amount of space for the array, initializes it, and then assigns to the variable a reference to the array. So, a variable that represents an array has a value that is really a reference to the array. This has several consequences, some good and some bad. It is believed that the advantages of this representation outweigh the disadvantages. First, we shall look at the positive aspects.

When a variable is passed to a subroutine, it is always the value of the variable that gets passed. Since the value of a variable representing an array is a reference, a reference to the array gets passed. One major advantage of this is rather obvious: it is a fast and efficient way to pass the array. This also has another consequence that is illustrated by the function

```
sub initiate_array(a, n)
{
    number i;

    for (i = 0; i < n; i++) a[i] = some_job(i);
}
```

where `some_job` is a subroutine that generates a scalar value to initialize the *i*th element. This subroutine can be used in the following way:

```
array X = number [100000];
initiate_array(X, 100000);
```

Since the array is passed to the subroutine by reference, there is no need to make a separate copy of the 100000 element array. As pointed out above, this saves both execution time and memory. The other salient feature to note is that any changes made to the elements of the array within the function will be manifested in the array outside the subroutine. Of course, in this case, this is a desirable side-effect.

To see the downside of this representation, consider:

```
array a, b;
a = number [10];
b = a;
a[0] = 7;
```

What will be the value of `b[0]`? Since the value of `a` is really a reference to the array of ten doubles, and that reference was assigned to `b`, `b` also refers to the same array. Thus any changes made to the elements of `a`, will also be made implicitly to `b`.

This begs the question: If the assignment of one variable which represents an array, to another variable results in the assignment of a reference to the array, then how does one make separate copies of the array? There are several answers including using an index array, e.g., `b = a[*]`; however, the most natural method is to use the dereference operator:

```
array a, b;  
a = number [10];  
b = @a;  
a[0] = 7;
```

In this example, a separate copy of `a` will be created and assigned to `b`. It is very important to note that JCL never implicitly dereferences an object. So, one must explicitly use the dereference operator. This means that the elements of a dereferenced array are not themselves dereferenced. For example, consider dereferencing an array of arrays, e.g.,

```
array a, b;  
a = array [2];  
a[0] = number [10];  
a[1] = number [10];  
b = @a;
```

In this example, `b[0]` will be a reference to the array that `a[0]` references because `a[0]` was not explicitly dereferenced.