The dBASE Language

This is a Borland Confidential R&D document, containing programmer notes from the Bladerunner development team. It gives a technical description of the dBASE language including the object oriented extensions that will be manifested in Bladerunner.

1 Types, Operators and Expressions

1.1 Types

1.1.1 Identifier names

Identifier Names are made up of letters, digits and _. The first character of a identifier name cannot be a digit. Identifier names can be up to 64 characters in length. Identifier names are not distinguished by case.

1.1.2 Variables and Storage

Variables store values of a given type. A variable is an association between an identifier name and a *storage-location*. The stored value and the stored value's data type may change over time. A variable is created or changed in an assignment statement and is of the form:

```
<variable-name> = <expression>
```

The results of `<expression>` are stored in the storage-location refered to by `<variable-name>`.. When a variable appears in an expression, the value in the associated storage-location is retrieved.

1.1.3 Data Types

This is the section on data types.

Character

This is a section on character type, it is not different than traditional dBASE.

Dates

This is a section on date type, it is not different than traditional dBASE.

Numeric

This is a section on number type, it is not different than traditional dBASE.

Logical

This is a section on logical type, it is not different than traditional dBASE.

Memo

This is a section on memo type, it is not different than traditional dBASE.

Function-Pointer

A function pointer is the address of a dBASE function or procedure. Function-pointer constants are of the form `identifier`, where identifier is the function name.

Object-Reference

An object reference refers to an object. An object is a collection of variables. A variable inside of an object whose type is function-pointer is called a *method.* A variable inside an object of any other type is called a *property.* Thus, an object is a collection of properties and methods. More than one object reference can refer to an object.

1.2 Operators

dBASE provides several types of operators: *arithmetic, character, date, relation, logical, object, functional.*

1.2.1 Arithmetic

The arithmetic operators are `+,-,/,*, **.`

1.2.2 Character

The character operators are `+,-,$.`

1.2.3 Date

The date operators are `+,-`

1.2.4 Relational

The relational operators are `=, <> >,>=,<,<=.`

1.2.5 Logical

Logical operators are `.and., .or., .not.`

1.2.6 Object operators

There are three operators that can be used to create and manipulate object.

1.2.6.1 The NEW operator

The `NEW` operator creates a new instance of an object. The `NEW` operator is of the form:

```
new <classname>([<parameters>])
```

The `NEW` operator creates an object of the `<classname>` to which it is applied. The `NEW` operator returns an object-reference. The lifetime of the object is as long as all outstanding object references to that object. For example:

```
x = new Font("helvetica")
```

Create a new font object. Parameters are passed to the constructor of the named class.

1.2.6.2 The Index Operator

The index operator accesses the contents of an object through a value. The index operator often refered to as the array index operator is of the form:

```
<object-reference>[<expression>]
```

The index operator applies an `<expression>` to an object reference to access a variable within an object. `<expression>` usually results in a numeric value. The result of the application of the index operator is a *storage-location.* Like a variable, a storage-location resulting from applying the index operator may be created or changed with an assignment statement. Also, like a variable its value can be retrieved when used in an expression. (Note: Traditionally, the index operator was used exclusively for Arrays. Arrays in dBASE are now Objects. The `DECLARE` syntax is retained for compatibility) The following example illustrates the use of the index operator:

```
declare a[10]
a[1] = 10               && change a[1]'s value to 10
? a[1]                  && prints '10'
```

```
b = new fixedarray(10)  && semantically identical to above
b[1] = 10               &&  statements.
? b[1]
```

1.2.6.3 The Member Access Operator

The member access operator access the contest of an object through an identifier name. The member access operator `'.'` is of the form:

```
<object-reference>.<variable-name>
```

The member access operator is applied to an object-reference to access a variable within an object. The result of the member access operator is a *storage-location.* Like a variable, a storage-location resulting from applying the member access operator may be created or changed with an assignment statement. Also, like a variable its value can be retrieved when used in an expression. The following example illustrates the use of the member access operator:

```
objVar = new Object()
objVar.memberVar = 10               && creates 'memberVar'
objVar.memberVar2 = "a string"      && creates 'memberVar2'
? objVar.memberVar                  && prints '10'
```

The member access operator works for both methods and properties:

```
objVar.methodVar = MyFunction       && assigns function-pointer
objVar.propertyVar = "a string"     && assigns data value
```

1.2.7 Functional operators

There is only one functional operator, the call operator.

Call operator

The call operator is used to call functions. The call operator is of the form:

```
<function-pointer>([<parameters>])
```

The call operator calls the function associated with the function pointer passing any specified parameters to the function. The operator returns the value that the associated function returns. For example:

```
? foo()                 && 'foo' is a constant of type
                        &&  function-pointer

goo = foo               && creates a variable of type
                        &&  function-pointer
? goo()                 &&  calls function 'foo'

decl a[10]
a[1] = foo              && stores function-pointer to
                        &&  a[1].
? a[1]()                && calls function 'foo'

o = new object()        && creates object
o.goo = foo             && creates variable 'goo' in o
                        &&  of type function-pointer
? o.goo()               && calls function 'foo'
```

```
function foo
      return 10
```

1.3 Expressions

The order of precedence for operators is as follows:

```
() [] -> . NEW
**
* /
+ -
= <> < <= > >=
.not.
.and. .or.
```

*note: Operators at the same level of precedence are performed left to right.

2 Statements

See the Online Help Language section for a list of language elements in the dBASE language.

3 Control Structures

The following are the flow-of-control structures in the dBASE language:
if - [else] - [elseif] - endif
do while -endo
do case
do until
for next
exit
loop

4 Functions

The following are the language elements related to the implementation of functions in the dBASE language:
FUNCTION
LOCAL
PARAMETERS
PRIVATE
PROCEDURE
PUBLIC
RETURN
STATIC

5 Objects / Instance programming

Instance programming is the *dynamic manipulation* of objects. Objects are incrementally added to and changed through assignment. What the object contains is not static and therefor can change over the lifetime of the object.

5.1 Dynamic manipulation of objects

To begin instance programming first an empty object is created by using the new operator with the class object.

```
o = new object()
```

This creates an object to which variable o refers. Variables may be dynamically added to the object through assignment. For example:

```
o.x = 10          && add a numeric variable to the object
o.s = "hello"     && add a character variable
```

More than one variable can refer to the same object. Variables contain object-references, not objects themselves. Assignment between object-references copies references, not the objects themselves. For example:

```
O = new Object()
O.X = 10
O.Y = 20
Y = O             && 'O' and 'Y' refer to the same object
? Y.X             && prints '10'
Y.Y = 30
? O.Y             && prints 30
```

(NOTE: Object references, like all other variable types, can be passed as parameters, returned as a function result, stored in arrays and generally be used like any other variables. There are only two valid operators on object-references, the member access operator ("."), and the index operator ("`[]`").)

5.2 Member Function pointer

A member function pointer is a function pointer that is stored in an object. A member function pointer has two parts: a function pointer and a reference to the object in which it is stored. In the simple case, only the function-pointer part is needed. For example:

```
o = new Object()
o.x = 10
o.f = meth        && create member function pointer 'f'
? o.f()           && prints '10'

function meth
      return "hello"
```

Member function pointers are only different from function pointer in how they relate to `this`.

5.3 This

When a member function pointer is retrieved, a local variable `this` in the procedure or function called is bound to the object refered to in the member function pointer. The binding of `this` provides the procedure or function a way to access to the object on which the call is being made. For example:

```
o = new Object()
o.x = 10
o.f = meth
? o.f()

function meth
      return this.x + 1       && 'this' and 'o' refer to same
                              &&  same object
```

Use of `this` gives functions a degree of reusability. A procedure or function need not know the object on which it is operating. By referring to the calling object as `'this'` the procedure can manipulate an object by addressing its contents. For example, the following two objects have different members but can share the same function:

```
o = new Object()
o.a = "hello"
o.x = 10
o.f = meth
? o.f()                         && prints '12'

q = new Object()
q.z = 30
q.x = 50
q.m = meth
? q.m()                         && prints '52'

function meth
      this.x = this.x + 1
      return this.x + 1         && 'this' and 'o' refer to same
                                &&  same object
```

5.4 The Member Statement

The member statement binds an identifier to a variable of the same name with in an object. The object bound is always refered to by `this.` The member statement has the form:

```
MEMBER <var>[,<var>...]
```

So instead of:

```
function DoSomething
      this.x = 10
      this.func()
      this.y = this.y +1
      return this.x+this.y
```

A more readable form can be written using member:

```
function DoSomething
      Member x,y,func
      x = 10
      func()
      y = y + 1
      return x + y
```

Using member give compilers a chance to make optimizations.

6 Classes

Classes are a mechanism for creating objects that have an identical set of variables. Variables are added to objects during the call to the *constructor*. A constructor is a special function that returns an object. The class statement is the constructors declaration and there is not explicit return statement. During construction `this` refers to the object being constructed. A constructor can only be called with the `NEW` operator. The syntax for a class definition is as follows:

```
class <classname>[(parameters)] [of <parentclass>[(<parameters>)]]
      <constructor-code>
      [function <funcname>
            <member function code>]
      [function <funcname2>
            <member function code>]
      ...
```

```
endclass
```

For example, the following class definition defines a class that has two member variables, `ten` and `twenty`.

```
class numbers
      this.ten = 10     && constructor code
      this.twenty = 20  && constructor code
endclass
```

To create on object of class `numbers`, the new operator can be called.

```
n = new numbers()
? n.ten                       && prints '10'
? n.twenty                    && prints '20'
```

6.1 Parameters to constructors

Just like functions, there are two ways to receive parameters in constructors. Parameters may be declared using the parameters statement or declared by enclosing them in parentheses following the class name. Parameters using the later method have local scope and are the preferred method. For example, the following code creates an object square that uses a parameter during construction:

```
s = new square(10)
? s.square              && prints '100'

class square(n)
      this.num = n
      this.value = n*n
endclass
```

6.2 Member Functions

Functions can be added to classes simply by declaring them inside the class body. By declaring a function inside a class body, an object of the class will have a variable of type function pointer bound to the declared function. Because this binding is through a variable, the function pointer can be subsequently overridden. The actual name of the function declared in a class body is formulated from the class name and is of the form: `<class-name>::<function-name>`. For example:

```
s = new square2()
s.num = 5
? s.value()             && prints '25'

class square2
      this.num = 0
      function value
            return this.num * this.num
endclass
```

In the above class definition, an object of class square2 has a property `value`, whose type is `<function-pointer>` and whose value is `Square2::value`.

6.3 Member Statement in Class Definitions

Member statements inside class definition are very similar to member statements inside functions. Member statements inside class definitions differ from member statements inside functions only in that they have scope of the entire class body. For example:

```
class square2
      member num
      num = 0
      function value
            return num * num
endclass
```

6.4 Subclassing / Inheritance

Inheritance, or subclassing, is essentially programming by difference. To subclass from an existing class implementation, the `OF` option is used in the class definition. The OF keyword is immediately follow by the parent class name, followed optionally, by parameters. During the constructor call, the parent constructor is called **first**, followed by the constructor for the defined class. For example:

```
b = new base()          && prints 'first '
? b.x                   && prints '10'
? b.y                   && prints '20'

d = new derived()       && prints 'first second'
? d.x                   && prints '10'
? d.y                   && prints '50'
? d.z                   && prints '100'

class base
      ? "first "
      this.x = 10
      this.y = 20
endclass

class derived of base
      ? "second"
      this.y = 50       && override value for y
      this.z = 100
endclass
```

6.5 Overriding member functions/ Polymorphism

Member functions can be overridden by simply redeclaring the functions in the derived class. For example:

```
b = new base()
d = new derived()
? b.two()               && prints 'base 2'
? d.two()               && prints 'derived 2'

class base
      function one
            return "base 1"
      function two
            return "base 2"
endclass

class derived of base
      function two
            return "derived 2"
endclass
```

This mechanism provides for **polymorphism**. We might have code that deals with graphic objects. At

some point in time we might want to have one of these graphic object draw themselves. The code to tell the objects to do the drawing would be the same regardless of the implementation.

```
function telldraw
      parameters o

      o.move(10,10)           && move to position 10,10
      o.draw()                && render the object
```

This code could be used on circles, squares, etc., provided they implemented a method `draw` and a method `move`. The implementations for these methods does not have to be similar.

6.6 Passing parameters down to base classes

Parameters can be passed to base classes by providing them to the parent class. The parameters passed to the parent class can be arbitrary expressions. For example:

```
class cube(n) of square(n)
      this.cubevalue = this.num*this.num*this.num
endclass

class square(n)
      this.num = n
      this.square = n*n
endclass
```

**Note: For alpha the only supported syntax for parameters passed to a constructor is of the form:

```
class foo
      parameter x,y,z
endclass
```