OE Basics

Introduction

OE is an object oriented language, similar (in spirit) to JavaScript, Ruby and Python. It is a fully object oriented language (unlike JavaScript) in that it supports encapsulation, inheritance and polymorphism. You can use existing classes and objects as well as create your own. OE is also designed to be extended by creating classes in C/C++.

While Python and JavaScript use the UCS-2 character set and Ruby uses Ansi (with Unicode add-ons), OE was designed from the start around UTF-16. Therefore, you can name variables and other identifiers with unicode characters.

Like these languages, OE source code is case sensitive. For example, a variable named xyz is different than a variable named Xyz (or XYZ, etc). Variables are also dynamically typed. A variable can hold any type of data, unlike C\C++ or Java which require a variable to be of a predefined type.

Unlike these languages, OE supports free form program statements using a semicolon to separate statements. You can have many statements on one line, have one statement per line or extend a statement over several lines. OE also compiles source code into object code (ECodes) for the Evm (ElmSoft virtual machine) as well as the “load and go” method that the other languages use.

- Data and methods are all properties
- Methods can act like data properties.

OE has a set of built-in standard data types: `Boolean`, `Integer`, `Number`, `String` and `Array` as well as special data types, `Null`, `Undefined`, `NaN` (not a number), `PosInfinity` and `NegInfinity`. All of these data types are objects. All data in OE are objects and each has an associated class. In fact, classes themselves are objects.

OE has a full set of operators: + - * / % etc. Each operator is (except for logical operators, && || and !) implemented as a method. This means that you can override operator behavior and use operators in your own classes.

We can go on stating information like this, but the best way to understand a language is to look at some examples.

The first example is the program that everyone uses: the Hello World example. Here it is in OE.

```
writeln("Hello World");
```

This writes out Hello World to the console (stdout).

Yes, this is the entire program. Types these lines into a text file, save it (script1.oes), then run the program as follows:

```
c:\EvmRun script1.oes
```

You could also use the alert method to display the message in a dialog box:

```
alert(“Hello World”);
```

Names/Identifiers

Names are used to identify various program elements, such as variables, methods and classes. Names can consist of alphanumeric characters such as 0-9, A-Z, a-z and '_'. It can also contain unicode characters above U+00C0. Keywords are all in lower case. There is a convention to use camelCase for variable and method names. This means that the first letter is lower case and each embedded word within the name is capitalized. Here are a few examples, "isTrue" or "isKindOf" or "toString". Class names, by convention, start with a capital letter. These naming conventions are not enforced. You may whatever conventions you wish, but, unless you have a compelling reason to be different, it is best to keep with the conventions. It will make sharing code easier.

Keywords

OE has the following keywords. These are words with special meaning to the language that you cannot use as identifiers.

```
abstract anyway break catch cdef class const continue cvar def do else elseif elseunless end
function if ivar mvar new public private protected return throw try until var while
```

Source Layout

Comments

The source code can be commented in one of two ways. All the text inside the /* and */ will be treated as comments. This a a free form construction meaning that it can extend over many lines or on one line.

```
/* This

       is a

                 long comment.
*/

var abc=/* short comment */456;
```

The other method is using two slash characters back to back (//). Any characters following this will be treated as comments until the end of the line is reached.

```
var abc=456; // Assign the variable abc to the value of 456;
```

Execution order

OE is a dynamic language. Execution starts at the top of the source file and continues until the end. The definition of classes and methods are executable code. Both classes and methods have to be defined before they can be used.

Semicolons, whitespace

White space characters (spaces and tabs) separate tokens from each other. Except as separators or within quote characters, whitespace characters are ignored in the source text. It is used to make the source code more readable.

A semicolon is used to separate statements. These are not always necessary. When keywords mark the end of a statement, then a semicolon is not strictly necessary. However, it makes the code more readable and there are cases when you might insert a statement and the keyword no longer ends the statement.

Built-in Data Types

Integer

An Integer object represents an signed integer (64-bit) value in the range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. The class for integer objects is named `Integer`. Integers can be specified in a program by typing in a literal value. For example:

```
 var a=123;
```

You can specify an integer number in hexidecimal by prefixing the number with 0x. For example:

```
var a=0x7b;
```

or binary

```
var a=0b01111011;
```

or octal

```
var a=0o173;
```

All of these examples result in the same integer value in the variable a (123_{dec}= is equal to $7B_{hex}$ is equal to 01111011_{bin} is equal to 173_{oct}).

Number

Number data types are floating point numbers with values in the range of 1.7E +/- 308 (15 digits).

Boolean

Boolean data types have only two values: `True` and `False`

String

A string value is a set of zero or more Unicode characters. These characters are UTF-16 characters when in UTF-16 mode and UCS-2 (default) characters otherwise. UCS-2 gives better performance. UTF-16 Mode allows characters outside the Basic Multilingual plane(BMP). This is only necessary in rare cases because the characters outside the BMP are used only in special cases. To set UTF-16 mode, call the `utf16Mode` method of the `Object` class with the boolean value of `True`. (e.g. `utf16Mode=True;`).

Individual characters can be accessed via the indexing operator ([]). Indexes start at zero.

String literals can be specified by enclosing the characters in single or double quotes. Characters within single quotes go in exactly as specified. This is ideal for file names (full path) because there are no escape characters. Characters within double quotes, however, allow escape characters. The escape character is the backslash character (\). It allows you to insert special characters into a string literal. The following table shows the special escape sequences allowed within a double quoted string literal. If the backslash is followed immediately by a character which is not in the table, then the character will be inserted into the string as is without any modification. For example, if two backslashes appear together(\\), then one backslash goes into the string. To insert a quote character (single or double) into a string literal, use the backslash followed by that character (e.g. \' or \").

Some examples:

```
var str1='c:\temp\myFile.txt';  // store the full path of a file name
var str2="This is a single quote\'";
var str3="This is a double quote\"";
var str4="This is an upper unicode character \u0100"; // This should be Ā
var str5="This is a non-BMP character \U10085";
```

Table 1: Special String Escape Sequences

Escape Sequence	Description
\n	Insert a line feed character (0x0A).
\r	Insert a carriage return character (0x0D).
\t	Insert a horizontal tab character(0x00)
\a	Insert a bell character (0x07).
\b	Insert a backspace character (0x08).
\f	Insert a form feed character(0x0B).
\v	Insert a vertical tab character(0x0C).
\cXX	Insert a character code represented by up to two hexidecimal characters. The number stops when a non-hexidecimal character is encountered.
\uXXXX	Insert a character code represented by up to four hexidecimal characters. The number stops when a non-hexidecimal character is encountered..This is used to insert a character code in the basic multilingual plane.
\UXXXXXXXX	Insert a character code represented by up to eight hexidecimal characters. The number stops when a non-hexidecimal character is encountered. This is used to insert a character code beyond the basic multilingual plane. The hexidecimal number should represent the character code, not the internal representation.

Array

Arrays are collections of objects. An array can contain zero or more objects of the same type or of different types.

Special Data Types

There are five fixed data objects. These are `Null`, `Undefined`, `NaN` (not a number), `PosInfinity` and `NegInfinity`. You can use `Null` in a script by specifying the name itself.

```
var nullValue=Null;
```

The others are generated from various actions and errors. `PosInfinity` and `NegInfinity` are generated when a calculation is to large. `Undefined` is generated when too few arguments are supplied when a method is called. Each unspecified argument will have the value of Undefined.

Variables/Properties

Variables are places to store data. Since all data in OE are objects, variables hold references to objects. Variables do not have types as they do in strongly typed languages (like C/C++ and Java). A variable can hold any type of object.The names of variables follow the conventions outlined above.

Variables can reside in many places. There are local variables, which are only accessible from the current method. Objects have variables as well. When variables occur inside objects, they are called properties.

You have to declare variables in OE. They are not automatically created for you. Most dynamic languages will create variables on the fly whenever a new name is referenced in the code. This sometimes leads to unfortunate results, especially when names are case sensitive. If you have a typo in a name and it causes no error, it may result in hours of debugging. OE will tell you when you have a variable/property name that does not exist.

To create a local variable, use the `var` keyword as follows:

```
var varName[=expression],[varName2 ...];
```

For example:

```
var abc=123;
```

creates a local variable called 'abc' and gives it an integer value of 123.

```
var myInt=777,myString=”My String Value”,myNumber=55.12, myVar;
```

This creates four local variables, an integer, a string, a floating point number and one where a value is not assigned. Since myVar does not have a value assigned to it, it will have a value of `Undefined`.

Expressions

An expression is a combination of operators, data and methods returning a value. Expressions in OE are similar to expressions in other languages. For example,

```
var a=b+5;
```

or

```
var a=2.0,b=7.0,c=-4.0;
var x=(-b+(b*b-4*a*c).sqrt)/(2*a);  // The positive factor in the quadratic equation
```

This is a quadratic formula example.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The equation is $2x^2+7x-4$, giving the a, b and c values to be 2,7 and -4 respectively. The value of x will be 0.5. Change the first + operator to a minus (-) to get the negative factor, which is -4.

or

```
var s="Hello"+" "+"World";
```

The plus operator will concatenate strings, so the value in s will be a string containing “Hello World”.

Operators

OE has a standard set of operators with a few concepts and implementations that might not be so standard. Many operators are implemented as methods. This means they can be used with user defined objects, if it makes sense for the object. More information on this is in the section on User Objects.

The following table shows the operators and their precedence, which means the order that they are evaluated. The higher precedence operator gets evaluated first. For example:

```
var a=2;
var b=4+6*a;
```

In the second expression, multiplication has a higher precedence than addition, so it gets evaluated first. The value of b will be 16. If it were evaluated left to right, the value would be 20. Operators of equal precedence are evaluated left to right.

Table 2: Operators by Precedence (low to high)

Operator(s)	Description

`=`
`+= -= *= /=`
`%= &= |= ^=`
`<<= >>=`
	Assignment statements. The equal sign(=) is the standard assignment operator. When one of the specified binary operators appears in front of the =, it means that the assignment target is used as the left side of the operation before the assignment.
	For example:
	a+=b+2;
	is equivalent to a=a+(b+2);

`or`	Logical or. This is used to combine other operations into a conditional. For example, if a>5 or b<9
`and`	Logical and. This is used to combine other operations into a conditional. For example, if a>5 and b<9
`| ^`	| is the bit wise 'or' operator and ^ or the bitwise 'xor' operator. These are used with integer values. The resulting integer of the 'or' operation is a new integer with bits set if the bit is set in either of the operands. The resulting integer from an 'xor' operation contains bits set from either of the operands as long as the corresponding bit is not set in both.
`&`	This is the bitwise 'and' operator.This is used with integer values. The resulting integer of the 'and' operation is a new integer with bits set if the bit is set in both of the operands.
`== <> ===`	These are tests for equality (or inequality). The result of these binary operations is a Boolean True or False. In most cases the === operator is equivalent to ==. It is placed here for special occasions when there might be different ways to specify equality.
`> >= < <=`	These are comparison tests. The result of these binary operations is a Boolean True or False

`<=>`	This a comparison test. The result of this binary operation is an Integer value. For example:
	`var r=leftValue <=> rightValue;`
	The result will be as follows:
	` 1 if leftValue > rightValue`
	` 0 if leftValue==rightValue`
	`-1 if leftValue < rightValue`

Table 2: Operators by Precedence (low to high)

Operator(s)	Description
`<< >>`	The left and right shift operators. These are used for Integer values to shift bits left or right. These are binary operations. The left operand is an integer value that is the source of the bits; the right operand is an integer value indicating the number of bits to shift.
`+ -`	The plus and minus operators. For numbers these are used to add or subtract the two values, respectively. For strings, the plus is used to concatenate the strings into a new string.
`* / %`	The multiply, divide and modulo operators. For numbers these are used to multiply, divide or get the remainder of the two values, respectively. For strings, the multiply is used to duplicate the string value an integer number of times into a new string.
`**`	The exponent operator. This is used for numbers as an exponent.
`not - ~`	The logical not, the unary minus and the bitwise not operators. These are all unary operators. The logical not is used to reverse the current comparison. The unary minus sign is to use negative numbers. The bitwise not reverses the bit values of the operand.

`++ (prefix)`
`-- (prefix)`
	The pre increment and pre decrement operators. These are used with variables that contain integer values. These should not be used with constants. The ++ will increment the variable before using it in the expression and the -- will decrement the variable before using it in the expression.
	For example:
	`var a=9;`
	`var b=++a*2;`
	The value of a will be 10 and the value of b will be 20.

`++ (suffix)`
`-- (suffix)`
	The post increment and post decrement operators. These are used with variables that contain integer values. These should not be used with constants. The ++ will increment the variable after using it in the expression and the -- will decrement the variable after using it in the expression.
	For example:
	`var a=9;`
	`var b=a++*2;`
	The value of a will be 10 and the value of b will be 18.

`[]`
`.`
`method call`
	The indexing (Get or Set) operator, the Get Property operator and the method call. The brackets are used to either retrieve a value from an array or string or set the value.

Table 2: Operators by Precedence (low to high)

Operator(s)	Description
`?`	The unary existence operator. This returns a True value if the variable name specified exists and False if it does not exist. Most operators operate on the value contained inside a variable. This operator operates on the variable name itself.

Control structures

If-else

Like any procedural language, execution proceeds from statement to statement until a control structure is encountered. One of the most common control structure is the if-else construct. The if-else has the following form:

```
if expr
   << statements1>>
[elseif expr2
   <<statements2>>
]
[else
   <<statementsN>>
]
end
```

If the `expr` resolves to a True value (True, non-zero, non-Null) then the `statements1` block is executed, otherwise it goes to the next (optional) `elseif` and evaluates that. If no `elseif` resolves to a True value, then the `else` block (`statementsN`) is executed. If there is no else block, then control is passed to after the `end` keyword. You may nest the if control structures as much as you like.

You may also use the `unless` keyword in place of the `if` and `elseunless` in place of an `elseif`. The unless versions have the opposite effect as the if versions. Control is passed to the statements if the expression resolves to a False value.

while

No language would be complete without some form of looping control. OE has the while-end construct as follows:

```
while expr
   <<statements>>
   [break;]
   [continue]
   <<statements>>
end;
```

This loop will continue as long as the expr evaluates to a True value (True, non-zero, non-Null). If a break keyword is encountered, control will pass to after the end. If an optional continue is encountered, then control is passed back to the beginning of the loop.

> **Note:** To avoid infinite loops you should make sure that the expr will eventually evaluate to a False value or that the break keyword is encountered.

Here is a short example of a while loop.

```
var idx=0,count=10;
while(idx<count)
   writeln("Loop Value="+idx.toString);
   idx++;
end
```

This loop writes out the numbers 0 through 9 to the console. (stdout).

```
var idx=0,count=10;
while(idx<count)
   idx++;
   if idx==5 continue; end;
   writeln("Loop Value="+idx.toString);
   if idx==7 break; end;
end
```

This is a contrived example of using both the continue and break keywords. It is the same loop as in the previous example, except that when the loop index reaches 5, it skips the processing and when it gets to 7, it leaves the loop. There is one other difference. We moved the incrementing of the loop variable (idx) to the front. If we had kept it near the end, then when it reached 5 the continue keyword would jump back to the beginning of the loop. In this case, the idx variable would always be 5 and hence an infinite loop.

Method iterators

Methods can be used as iterators. This is explained in the methods section.

Methods

In object oriented programming the term ‘method’ is equivalent to sub or function in non-object-oriented languages.

Calling methods

In OE, methods are called similar to most other languages. The name of the method is followed (optionally) by the arguments enclosed in parentheses. For example:

```
mySub(1,2,3);
```

calls the method named mySub with three arguments (the integers 1, 2 and 3). If there are no arguments then you do not need to use the parentheses.

```
mySub;
```

Of course, methods can return a value. In fact they always return a value. If the called method does not specifically return a value, then `Undefined` will be returned.

```
var theReturnValue=mySub(1,2,3);
```

Defining methods

You define a method using the `def` keyword, as follows.

```
def methodName[(arg1[=value], arg2[=value], ...,argN[=value])
...
end;
```

The name of the method follows the standard identifier naming rules. In OE, a method is a data item, so the name must be unique. Also, since methods are data, they must be defined before they are used. The optional arguments names are enclosed in parentheses and separated by commas. Each argument can have a default value. The default value is used if the method is called without specifying that argument. Some examples:

```
def addNums(first, second=1)
   return first+second;
end;
...
var rr=addNums(1,2);
var rr1=addNums(5);
```

This is a trivial example of defining and calling a method. This simple method just adds the two arguments. In the first case, the two arguments are 1 and 2, therefore the returned value will be 3. In the second case, since the second argument is not provided and a default value is specified (1), the default value is used. The first argument is 5, so he returned value is 6.

Variable length arguments

The number of arguments in OE can be of variable length. If fewer arguments are supplied than defined when calling a method, the `Undefined` value is used for each missing value (unless, of course, a default value is defined). If more arguments are supplied than defined, then the extra arguments can be accessed via the Args special variable. Args is a psuedo array where each array element corresponds to an argument. For example,

```
def addManyNums
   var aCount=Args.count, sum=0, curr=0;
   while(curr<aCount)
      sum=sum+Args[curr];
      curr++;
   end;
   return sum;
end;
...
var rr=addManyNums(7,3,4,5,7,8,9,1,12,45,67);
var rr1=addManyNums;
```

This method adds all the values of all the arguments together and returns the result. This method has no arguments defined. However the Args array will allow you access to each argument and the count property will indicate how many arguments there are. Also, calling the method without any arguments returns 0.

All methods support recursion, so a method may call itself repeatedly.

Method iterators

Several built-in data types have iterator methods. These can be used to loop through a block of statements in a similar way that the `while-end` control structure does. Calling an iterator method is the same as calling any other method with the exception of the syntax for the block of statements. The `do` keyword is used to define a block of statements, which is terminated by the standard `end` keyword. The best way to illustrate this is with some examples.

The Integer data type has several iterator methods. The simplest is the `times` method. The times method loops for the number of times in the integer value.

The following code loops 5 times and prints the values 0 through 4 to the console.

```
var i=0;
5.times do
   writeln("Integer Loop-"+i.toString);
   i++;
end
```

Note: The integer value does not have to be a constant. It can also be a variable containing an integer object.

As it turns out you do not have to keep a counter. The `times` method will pass a value to the loop block. when you supply an argument, using parentheses, after the `do` keyword, the `times` method will pass a value starting with zero up to the number in the integer minus 1. This means that the folowing code is equivalent to the code above. The `times` method supplies the values of 0 through 4 successively to the code block.

```
5.times do(i)
   writeln("Integer Loop-"+i.toString);
end
```

The `times` method is useful for simple cases where you wish the start value to be zero and the integer value incremented by 1 each time. There is another integer method which is more flexible. It is called `upto`. The `upto` method have one required argument (the last value loop value) and one optional argument (the increment value). It uses the Integer receiver value as the start value. Here is an example.

```
0.upto(5) do(i)
   writeln("upto Loop-"+i.toString);
end
```

This code loops six times with the loop value (i) starting at 0 and going up to 5, inclusively. Since the increment value is not supplied it is assumed to be 1. This is like the following javascript code.

```
for (x=0; i<=5; i++) {
   document.write(i);
}
```

The following example increments the loop argument by 2 instead of 1, so it will loop 3 times with the loop variable having the values 0, 2, and 4 successively.

```
0.upto(5,2) do(i)
   writeln("upto Loop-"+i.toString);
end
```

The `downto` method works the same way but in the opposite direction. It starts with the receiver value and loops down to the value in the first argument.

```
5.downto(0) do(i)
   writeln("downto Loop-"+i.toString);
end
```

Another important iterator method is the `each` method. This is used in strings and arrays. It loops through each character of a string and through each member of an array.

Here is an example that dumps information about an array and each member of that array to the console.

```
def dumpArray(arr,msg="")

   writeln("===Dump Array("+msg+")===");
   if arr.className=="Array"
      var arrType=arr.dataType;
      if arrType
         writeln("Array Type-"+arrType.name);
      end;
      var count=arr.count;
      writeln("Array count-"+count.toString);
      var aidx=0;
      arr.each do(arrItem)
         writeln("Array["+aidx.toString+"]="+arrItem.toString+" Class-"+arrItem.className);
         aidx++;
      end;
   else
      writeln("Not An Array-"+arr.className);
   end

end
```

About Objects

We use the term ‘object’ to mean an abstract representation of something. Objects have data and methods to manipulate that data. For example, in OE a string is an object (like all other data items). The data it contains is the set of characters in the string. The methods manipulate those characters. The `length` method returns the number of characters in the string; the `subStr` method returns a substring of the original string. A more complicated example is the OutputTextFile object. It has several data items, including `encoding` and `includeBom` as well as methods `open`, `write` and `close`.

The data and methods of an object are collectively called properties. As stated above the data items are objects. It turns out that the methods are objects too. They are accessed by name, which means, of course, that you cannot have a method property and a data property using the same name in the same object.

In addition to its properties, an object also has a class associated with it. A class is a special type of object which is used to create other objects. For example, a string value has an associated class called `String`. In fact, every string value will have the same associated class. The string value is calld an instance of the class String. In fact, some classes might not have any instances. Since a class is an object, it can have properties of its own (data and methods) independent of the methods defined for the instances.

Object oriented principles

The three basic principles of object oriented programming are encapsulation, inheritance and polymorphism.

Encapsulation

Encapsulation is all about data hiding and data protecting. As programs grow larger and there are more and more variables, it gets harder and harder to avoid unintended modification of values. You might create a variable such abc in one part of the program and assign it a value, then later create another by the same name. Unbeknowst to each other the variable is modified by different parts of the program. This generates some of the most difficult errors to find.

In OE, encapsulation is achieved in several ways. One is using local variables. Local variables (defined with the `var` keyword) are local to the method where they were created. No other method can modify them. In fact, even if you call a method recursively, each new call creates a different set of local variables.

Another way is to store variables inside of objects. These object variables are called properties. Properties can have limited or no access at all to methods outside the object.

Inheritance

As the old saying goes "There's no point in reinventing the wheel". This idea is the point behind inheritance. If one class has many things (properties, methods) that you need, you do not have to write it over again. You can create a new class that inherits these things from the old class, keeping the old class intact (after all someone else might need it the way it is), then adding (or replacing) the things that the old class does not have.

Polymorphism

This one is the most difficult one to explain simply. It allows you to have different (but usually similar) data types and to act upon them in the same way. In strongly typed languages (C++ and Java) this is achieved through base classes and inheritance. Java also has interfaces for this purpose. In OE (and other dynamic languages) this is achieved through duck typing. This comes from the idea "If is walks like a duck and quacks like a duck, it must be a duck.". In programming terms, if an object has a set of properties with the same names, you can use them the same way, even if they are different types.

Accessing properties

Inside a class method properties can be accessed by using the property name. Outside the class properties are accessed via the dot notation. A variable name (or constant) containing an object (called the receiver) is followed by the period which is following by the property name.

Creating Object Instances

Object instances are created from their respective classes. To create an object instance of a particular class, you call the `make` method of that class object. The `make` method is a built-in class method for all classes. The `make` method may require one or more arguments or no arguments at all, depending on the class itself. Some languages use a `new` operator to create classes, but using a method can make computations more straightforward. Of course, some built-in types (such as Integers) can be created just by using the constant value in the source code (you can also use the `make` method if you wish). Others, such as Arrays, require that you use the `make` method.

For example, to create an array instance, do the following:

```
var myArr1=Array.make; // This creates a variable length array
var myArr2=Array.make(10); // This creates an array with 10 elements
```

Deleting Object Instances

You cannot delete object instances. They are handled by the garbage collection routine when the object is not referenced by any variable. You can, however, specify a method that will be called when the garbage collection deletes it. This will be discussed later.

Creating Classes

You can write programs without creating you own classes. You can use just the built-in classes (or imported classes) to write many programs. Eventually, though, as programs get larger and more complicated, especially considering the reasons given above (See “Object oriented principles” on page 15.), it will be useful to create your own classes.

Classes can be simple or they can be very complicated. They also have many features only some of which are used in any particular class.

Let’s start with a simple example.

You can create a new class by using the class keyword followed by the new class name. The name should follow the standard naming rules and, by convention, should start with a capital letter. The name should also be unique within the context of the definition.

```
class MyClass
...
end;
```

This is the basic form of a class. Of course, you also have to describe the class properties. Most classes are going to have one or more methods that the object instances created from this class can use. Commonly the first instance method to describe is the `onInit` method.

onInit and instance properties

The `onInit` method is an instance method that is automatically called (by the `make` method) whenever an object instance is created. It is usually used to create the variables for the new instance. You do not have to create them here. Instance properties can be created by any instance method. However, if you want to be sure that they exist, its a good idea to do it here. The `onInit` method can have arguments. These come from the `make` method. Any arguments that are passed to the `make` method when the instance is created are passed in turn to the `onInit` method.

You can create instance variables using the `ivar` keyword. The ivar keyword is similar to the var keyword, except that it creates an instance variable instead of the local variable. Consider the following example.

```
1.    class ChessPiece
2.       def onInit(pieceName, pieceShortName, side='W', rankNum=-1, fileNum=-1)
3.          ivar m_PieceName=pieceName, m_PieceShortName=pieceShortName;
4.          ivar m_Rank=rankNum, m_File=fileNum, m_Side=side;
5.       end;
6.       def moveTo(rankNum, fileNum)
7.          m_Rank=rankNum;
8.          m_File=fileNum;
9.       end;
10.      def getShortName; return m_PieceShortName; end;
11.      def getName; return m_PieceName; end;
12.      def getRank; return m_Rank; end;
13.      def getFile; return m_File; end;
14.      def getSide; return m_Side; end;
15.   end;
16.
17.      var whiteRook1=ChessPiece.make('Rook', 'R', 'W', 0, 0);
18.      var whiteRook2=ChessPiece.make('Rook', 'R', 'W', 0, 7);
19.      writeln("A white rook is on rank "+(whiteRook1.getRank+1).toString);
20.      writeln("A white rook is on rank "+(whiteRook2.getRank+1).toString);
```

This class represents a chess piece in a chess game. In chess parlance, a row on a chess board is called a rank and a column is called a file. In OE parlance, the first row is 0 and the last row is 7. When the instance of the class `ChessPiece` is created in line 17, it calls the `onInit` instance method passing the five arguments identifying the chess piece from the make method. It will then create five instance variables and assigning the specified values to them. These variables will be available to any other instance method inside the class. However, they are not available to any method outside the instance.This is part of the encapsulation technique. You can hide the data from other objects. If you wish other objects to access them, you have to provide a method to return them and a method to set their values. An examples of these are shown starting with the method on line 10 and continuing through line 14. Another instance method (moveTo) is provided as a way to move the piece from one place to another.

In this example, two white rooks are created and placed on the first row at positions 0 and 7. The ranks (rows) of those two rooks are printed out (adding 1 to make it consistent with chess notation.

Class properties

The previous section showed how to create instance variables with the `ivar` keyword and how to create instance methods (methods available to instances created by this object) using the `def` keyword. But earlier we mentioned that classes are objects too and they can have properties. Class variables are variables that belong to the class object itself. While instance properties are unique to each instance (and can have different values in each instance), there is just one set of class variables per class. You can create class variables using the `cvar` keyword and we can also create class methods using the `cdef` keyword. The `cvar` keyword is allowed in the class definition and inside class methods, but not inside instance methods. Class properties can be accessed in class methods and in instance methods. Consider the following:

```
1.     class ChessPiece
2.        cvar cm_PieceCount=0;
3.        cdef getPieceCount
4.           return cm_PieceCount;
5.        end;
6.        def onInit(pieceName, pieceShortName, side='W', rankNum=-1, fileNum=-1)
7.           ivar m_PieceName=pieceName, m_PieceShortName=pieceShortName;
8.           ivar m_Rank=rankNum, m_File=fileNum, m_Side=side;
9.           cm_PieceCount++;
10.       end;
11.       def moveTo(rankNum, fileNum)
12.          m_Rank=rankNum;
13.          m_File=fileNum;
14.       end;
15.       def getShortName; return m_PieceShortName; end;
16.       def getName; return m_PieceName; end;
17.       def getRank; return m_Rank; end;
18.       def getFile; return m_File; end;
19.       def getSide; return m_Side; end;
20.    end;
21.    ...
22.       var whiteRook1=ChessPiece.make('Rook', 'R', 'W', 0, 0);
23.       var whiteRook2=ChessPiece.make('Rook', 'R', 'W', 0, 7);
24.       writeln("A white rook is on rank "+(whiteRook1.getRank+1).toString);
25.       writeln("A white rook is on rank "+(whiteRook2.getRank+1).toString);
26.
27.       writeln("Total Piece Count is "+ChessPiece.getPieceCount.toString);
28.
```

This adds a class variable (cm_PieceCount) and a class method (getPieceCount). The class method cannot access any instance properties (variables or methods), but is can access the class variable. The purpose of this variable is to keep a count of every chess piece that has been created. To make it work, we have added the following line (at line 9)

```
cm_PieceCount++;
```

to the `onInit` method. We've also added the following line

```
writeln("Total Piece Count is "+ChessPiece.getPieceCount.toString);
```

to print out the number of pieces created at the end of the program. Notice that you use the class name itself as the receiver to access the class method, since it is independent of any instance.

More example

There is no point in having chess pieces without a chess board to put them on. We have to create another class to represent the chess board and its 64 squares.

```

class ChessBoard
   cvar cm_BoardCount=0;
   def onInit;
      ivar m_RankCount=8, m_FileCount=8;
      ivar m_Board=Array.make(m_RankCount);
      m_RankCount.times do(rank)
         m_Board[rank]=Array.make(m_FileCount);
      end;
      cm_BoardCount++;
   end;
   def SetupStartGame;
       // Put a pawn on each file in the 2nd and 7th ranks
      m_FileCount.times do(currFile)
         m_Board[6][currFile]=ChessPiece.make('Pawn', 'P', 'B', 6, currFile);
         m_Board[1][currFile]=ChessPiece.make('Pawn', 'P', 'W', 1, currFile);
      end;
       // Put the other pieces on the 1st and 8th ranks
      m_Board[0][0]=ChessPiece.make('Rook', 'R', 'W', 0, 0);
      m_Board[0][7]=ChessPiece.make('Rook', 'R', 'W', 0, 7);
      m_Board[7][0]=ChessPiece.make('Rook', 'R', 'B', 7, 0);
      m_Board[7][7]=ChessPiece.make('Rook', 'R', 'B', 7, 7);
      m_Board[0][1]=ChessPiece.make('Knight', 'N', 'W', 0, 1);
      m_Board[0][6]=ChessPiece.make('Knight', 'N', 'W', 0, 6);
      m_Board[7][1]=ChessPiece.make('Knight', 'N', 'B', 7, 1);
      m_Board[7][6]=ChessPiece.make('Knight', 'N', 'B', 7, 6);
      m_Board[0][2]=ChessPiece.make('Bishop', 'B', 'W', 0, 2);
      m_Board[0][5]=ChessPiece.make('Bishop', 'B', 'W', 0, 5);
      m_Board[7][2]=ChessPiece.make('Bishop', 'B', 'B', 7, 2);
      m_Board[7][5]=ChessPiece.make('Bishop', 'B', 'B', 7, 5);
      m_Board[0][3]=ChessPiece.make('Queen', 'Q', 'W', 0, 3);
      m_Board[0][4]=ChessPiece.make('King', 'K', 'W', 0, 4);
      m_Board[7][3]=ChessPiece.make('Queen', 'Q', 'B', 7, 3);
      m_Board[7][4]=ChessPiece.make('King', 'K', 'B', 7, 4);
   end;
   def PrintBoard;
      var currRank=0, currSquare, outStr;
      writeln("             Black");
      writeln("   --------------------------");
      m_RankCount.downto(1) do(currRankNumber)
         currRank=currRankNumber-1;
         outStr=currRankNumber.toString+". !";
         m_FileCount.times do(currFile)
            currSquare=m_Board[currRank][currFile];
```

```
45.                 if isUndefined(currSquare)
46.                    outStr+=" - ";
47.                 else
48.                    if currSquare.isKindOf('ChessPiece')
49.                       outStr+=' '+currSquare.getShortName+' ';
50.                    else
51.                       outStr+=" E ";
52.                    end;
53.                 end;
54.              end;
55.              writeln(outStr+'!');
56.           end;
57.           writeln("   --A--B--C--D--E--F--G--H--");
58.           writeln("             White");
59.        end;
60.
61.     end;
62.
63.
```

Lines 6-9 create an array of arrays with each hvaing 8 members resulting in a 8x8=64 square board. The dimensions are saved in two instance variables m_RankCount and m_FileCount on line 5. The SetupStartGame method creates all the pieces and pawns and puts them in their start of game positions. The rest of the squares remain empty and therefore will have the Undefined value. The PrintBoard method scans over the board in reverse rank order and prints the current position along with headers and footers.

> **Note:** In chess terminology a pawn is not considered to be a piece. In this case however we use that terminology for any chess object as a conveniece.

You setup the board and print it out with the following.

```

var game=ChessBoard.make;
game.SetupStartGame;
game.PrintBoard;

```

Receivers and me

As mentioned earlier the target object when accessing properties is called the receiver. When accessing instance properties from outside the object, the receiver has to be an object instance. In the above example, whiteRook1 is a variable containing an instance of ChessPiece. When accessing the getRank (whiteRook1.getRank) method, the receiver is the instance object. When accessing the class property (getPieceCount) outside the object, the receiver is the class itself (ChessPiece). When inside the object, all you need is the name of the property itself.

But what is the receiver called when inside the object? Suppose one object wishes to use itself as an argument in a method? The answer is the keyword **me**. From inside an object the keyword **me** refers to itself. Some languages (C++ and Java) use the `this` keyword for this purpose. The pascal programing language uses `self`.

In our chess example, suppose we wanted to create a chess piece outside of the ChessBoard object, then have the ChessPiece object add itself to the board. You could do something like the following from inside a ChessPiece instance method.

```
cb.AddPieceToBoard(me, m_Rank, m_File);
```

where cb is a variable containing an instance of the ChessBoard class. We do not need this however, so we won't put it in.

Inheritance

So far so good for our chess example. The next thing we want to do is to have a method (movesAvailable) that tells us what moves are available for this piece. Of course, each piece has a different way of moving. We would have to put in a bunch of if statements (one for each piece type). This would make it a long method and perhaps a little difficult to follow. It would also be bad for the encapsulation principle. We could break it down into a method for each piece type (e.g. movesAvailableForQueen, etc.) and use the if statements to call the appropriate method. This would encapsulate it better and be a better overall solution. Although it does not apply here because the number of chess pieces are fixed, in other similar systems, the number of potential objects could be very long. Everytime you had to add a new object type, you would have to modify the large if statement and add a new method to handle it. This is where the object oriented solution comes in handy. The solution is to have each type of piece as a separate class. This way we can encapsulate all the information about each piece type in a separate class and if we ever had to add a new piece type all we have to do is to create a new class. We will not have to go back and modify code that is already debugged and working. As you may have guessed, this is all accomplished through inheritance.

Each piece type has some things in common with the other piece types and also some differences. We could copy the common code from the ChessPiece class and use it over again in each of the new classes. Besides bloating the code, if we had to make a change to this part of the code, we would have to make changes in every new class.

We already have code that handles positioning of the piece and on identifying to what side the piece belongs. We can keep this and just add the new stuff to the new classes. We use inheritance to add this functionality to our new classes.

The following line shows how to have one class inherit the code from another class.

```
class ChessRook < ChessPiece
...
end;
```

This code creates a new class which inherits all the methods and variables from specified class. An object instance of ChessRook will have a getRank, a getFile and a getSide method just like ChessPiece. In fact, you can use them interchangably. The terminology of this relationship varies. Sometimes ChessPiece will be called the base class of ChessRook. Sometimes it is called the parent or super class. Likewise ChessRook is sometimes called the derived class of ChessPiece. Sometimes it is called the child or subclass. We'll go with base class and derived class terminology here.

How do you initialize this new object. You use the `onInit` method the same as before with one little change.

Consider the following.

```
class ChessRook < ChessPiece
   def onInit(side='W', rankNum=-1, fileNum=-1)
      super("Rook", 'R', side, rankNum, fileNum);
   end;
end;
...
var rook=ChessRook.make('W',0, 7);
```

We do not have to supply a name or a short name anymore. A rook object type (et al) knows what it is called, so the make method only needs to provide the piece side and the position. In that case, what does the **super** do? First things first, however.

Overriding and super

ChessRook inherits all the methods from ChessPiece, including the `onInit` method. But we have just named a method in ChessRook `onInit`, the same name as a method in the base class. Which does it call? When you create a method in a derived class with the same name as in a base class, this is called overriding. In other words, you have replaced the old method with a new one. This idea is an important part of inheritance. When you derive one class from another, you do it to add to or alter (and sometimes delete) the functionality of the base class. This means that sometimes you want to completely replace a method in the base class, sometimes you want to remove it (by creating a new but empty method) and sometimes you want have the method act differently. To make it act differently usually means still calling the method in the base class, but doing other things as well.

Now we are back to **super**. From outside of a ChessRook object only the `onInit` in the ChessRook class is available. However, from inside the class it is a different situation. You cannot use the `onInit` name inside the ChessRook `onInit` method because it would just call itself. The keyword **super**, however, means to call the method by the same name as the current method, but in the base class. In the above example, the

```
super("Rook", 'R', side, rankNum, fileNum);
```

line calls the `onInit` method in the ChessPiece class. To be more precise, **super** calls the method in the base class, which as the same name as the current method. In this case, it is `onInit`.

Ordinarily, when you have a derive one class from another and call the make method on the derived class, there are arguments that are intended to initialize items in both the derived class and the base class. In our ChessRook example, that is not the case, but most of the time it will be. You might have something like the following:

```
class B
   def onInit(x,y)
      ivar myX=x; myY=y;
   end;
end
class A < B
   def onInit(x,y,z)
      super(x,y);
      ivar myZ=z;
   end;
end;
```

The **super** method is called before doing any other initalization. In most cases this is the recommended approach. In a more complicated case, where initialization involves calling other methods, you might inadvertently call a method in the base class and since, the base class has not yet been initialized, it might not work correctly. The rule of thumb during initialization is to always initialize base classes before the derived classes.

In other derived classes (those not named `onInit`), however, you can use the **super** keyword wherever it makes sense. Sometimes you want to call it right away, sometimes you want to call it at the end, and sometimes in the middle. It depends on what you are trying to achieve.

Back to our example.

With giving each piece type its own class, is it necessary for the ChessPiece class to know about the name? After all, the ChessRook object knows it own name. The answer is, of course, no. We want to put everything that is unique to a particular piece in the class that describes it. The ChessPiece class can have everything that is common to all pieces. In this case, we can make the following changes to ChessPiece and ChessRook (along with all the other pieces).

```
class ChessPiece
   cvar cm_PieceCount=0;
   cdef getPieceCount
      return cm_PieceCount;
   end;
   def onInit(side='W', rankNum=-1, fileNum=-1)
      ivar m_Rank=rankNum, m_File=fileNum, m_Side=side;
      cm_PieceCount++;
   end;
   def moveTo(rankNum, fileNum)
      m_Rank=rankNum;
      m_File=fileNum;
   end;
   def getRank; return m_Rank; end;
   def getFile; return m_File; end;
   def getSide; return m_Side; end;
end;

class ChessRook < ChessPiece
   def onInit(side='W', rankNum=-1, fileNum=-1)
      super(side, rankNum, fileNum);
   end;
   def getShortName; return "R"; end;
   def getName; return "Rook"; end;
end;

```

We take the getShortName and the getName out of ChessPiece and put it in each of the classes for individual pieces. We also remove the properties from ChessPiece and just keep the positioning properties. We also have to change the way we create these objects in the ChessBoard class. Instead of creating ChessPiece objects, we must create ChessRook objects and the rest as well. Consider the following.

```
        m_Board[0][0]=ChessRook.make('W', 0, 0);
        m_Board[0][7]=ChessRook.make('W', 0, 7);
        m_Board[7][0]=ChessRook.make('B', 7, 0);
        m_Board[7][7]=ChessRook.make('B', 7, 7);
        m_Board[0][6]=ChessKnight.make('W', 0, 6);
        m_Board[0][6]=ChessKnight.make('W', 0, 6);
... and so on
```

Polymorphism and duck typing

Before we add the movesAvailable method to each chess piece class, we need to add two more things to make it a little more object oriented. At the moment, the empty spaces on the chess board are undefined. This will work, but it would be better to have a real chess object there instead of an undefined value. We can then use it in certain cases to make the code easier (fewer if statements). Consider the following.

```

class ChessEmptySpace
   def getShortName; return "-"; end;
   def getName; return "EmptySpace"; end;
   def isSpace; return true; end;
end;


```

We will create one (and only one) ChessEmptySpace object instance. The ChessBoard class will create and save this as a class variable. Since we only need one of these, there is no point in duplicating it in every instance. Why do we only need one? Because it has no data. It is used as a place holder. If there were different kinds of empty spaces, then we might need more than one.

When the ChessBoard class creates a new board, every square will have a value. It will either be a piece object or the ChessEmptySpace object. Our printBoard method no longer has to check the value in the square. It only has to call the getShortName method, which will return the piece letter (K,Q,B,N,R,P) or the '-'. This is another example of duck typing. The ChessEmptySpace class and the chess piece classes are unrelated, but since each has a getShortName method, they can be used in the same way. Notice also the isSpace method. It returns true here. A similar one is added to ChessPiece which returns false. This will come in handy later.

The second thing to change is the position information. The rank and file values always go together. It takes two numbers to determine a position on a chess board. So we will create a new class to handle the position. Consider the following.

```
1.
2.    class ChessPos
3.       def onInit(rankNum=-1, fileNum=-1)
4.          ivar m_Rank=rankNum, m_File=fileNum;
5.       end;
6.       def getRank; return m_Rank; end;
7.       def getFile; return m_File; end;
8.       def setRank(newRank);  m_Rank=newRank; end;
9.       def setFile(newFile);  m_File=newFile; end;
10.   end;
11.
12.   class ChessPiece
13.      cvar cm_PieceCount=0;
14.      cdef getPieceCount
15.          return cm_PieceCount;
16.      end;
17.      def onInit(side='W', rankNum=-1, fileNum=-1)
18.         ivar m_Pos=ChessPos.make(rankNum, fileNum), m_Side=side;
19.         cm_PieceCount++;
20.      end;
21.      def moveTo(rankNum, fileNum)
22.         m_Pos.setRank(rankNum);
23.         m_Pos.setFile(fileNum);
24.      end;
25.      def getPos; return m_Pos; end;
26.      def getSide; return m_Side; end;
27.      def isSpace; return False; end;
28.   end;
29.
```

We have a new class called ChessPos which combines the rank and file values into one class. We have to change the onInit method to create a ChessPos instance(see line 18) instead of using individual values.

```

class ChessBoard
   cvar cm_BoardCount=0;
   cvar cm_EmptySpace=ChessEmptySpace.make;
   def onInit;
      ivar m_RankCount=8, m_FileCount=8;
      ivar m_Board=Array.make( cm_EmptySpace, cm_EmptySpace, cm_EmptySpace,
          cm_EmptySpace, cm_EmptySpace, cm_EmptySpace,
          cm_EmptySpace, cm_EmptySpace);
      m_RankCount.times do(rank)
         m_Board[rank]=Array.make(
                cm_EmptySpace, cm_EmptySpace, cm_EmptySpace,
                cm_EmptySpace, cm_EmptySpace, cm_EmptySpace,
                cm_EmptySpace, cm_EmptySpace);
      end;
      cm_BoardCount++;
   end;
   def getSquareValue(pos)
      var sqr=null;
      if pos.isInstanceOf('ChessPos')
         if pos.getRank>=0 and pos.getRank<8 and pos.getFile>=0 and pos.getFile<8
            sqr=m_Board[pos.getRank][pos.getFile];
         end;
      elseif Args.count>1
         var tempRank=Args[0];
         var tempFile=Args[1];
         if tempRank>=0 and tempRank<8 and tempFile>=0 and tempFile<8
            sqr=m_Board[tempRank][tempFile];
         end;
      end;
      return sqr;
   end;
...
```

Here is the first part of the new ChessBoard class. The cmEmptySpace class variable contains the one value for the empty space object. When the board matrix is created, it will initally have an empty square in each place. We have also added a new method to retrieve the value of any square given the ChessPos or rank and file values. It uses the isInstanceOf standard method to determine what type the argument is. If the position is out of range, null is returned.

Now we are ready to write the movesAvailable method for each piece. This is left for an exercise. We’ve demonstrated the object oriented principles and basic techniques for writing OE programs.

Has A/Is A

When designing classes and class structure sometimes decisions have to be made about whether to extend an class or to use an instance of the class as a instance variable. For example, in the chess program described earlier, we used a ChessKing class which was extended from a base class (ChessPiece) because the ChessPiece class had data and methods that the ChessKing class needed. However, another option would have been to not extend the class but instead use the ChessPiece instance. Consider the following.

```

class ChessRook
   def onInit(side='W', rankNum=-1, fileNum=-1)
      ivar m_Piece=ChessPiece.make(side, rankNum, fileNum);
   end;
    def getShortName; return "R"; end;
    def getName; return "Rook"; end;
    def getPos; return m_Piece.getPos; end;
    def getSide; return m_Piece.getSide; end;
    def isSpace; return False; end;
end;

```

This is a case of using ChessPiece instead of deriving from it. It can sometimes be a difficult decision what to do. You can only have one base class, but you can use many different classes.

The standard rule of thumb is the Has a/Is a rule. If the potential derived class is a base class (for example, if a ChessRook is a ChessPiece) then you can derive the class from it. However, if it satisfies a Has a question (for example, if a ChessPiece has a position) then use it instead of deriving from it.

Advanced features

Garbage Collection

There is no delete method or keyword. All data is automatically garbage collected for you. When an instance no longer has any path to reach it, it will be deleted. Adding an onDelete method (described below) to an instance will cause it to be run just before the instance is deleted.

onDelete

Like the `onInit` method, which is called when an instance is created. There is also an `onDelete` method which will be called when an object is deleted. This method has no arguments.

Objects are only deleted by the garbage collector. You cannot delete anything directly via programming. The garbage collector deletes an object when the object is not referenced by any variable. The garbage collector runs periodically in the background. There is usually very little reason to use the `onDelete` method. Any data stored in instance variables will also be automatically deleted by the garbage collector. The only reason to include this method is when the object needs to release some external resources (such as open files). The built-in file classes (`InputTextFile` and `OutputTextFile`) already have an onDelete method, so you do not need to have another.

Using an onDelete will have an effect on performance. The garbage collector will have to call an extra method when it deletes these objects and it needs to handle it with the special cases.

In summary, use this if you absolutely need to, but not otherwise. In our chess program from the last section, there are class variables that count the number of chess boards created and the number of pieces created. It is reasonable that you could use the onDelete to decrement that counters to keep an accurate count. But you have ask yourself how important is this before taking the performance hit.

Operator methods

In OE, the action of operators (plus, minus, etc.) are performed by methods. For example, to evaluate the expression 1+2, the Plus operator method is called using the integer 1 as the receiver and passing 2 as an argument (1.plus(2)).

Left/Right operators

Access Control

When you create an instance variable inside a class, you cannot access this variable from outside the class. On the other hand, methods *can* be accessed from outside the class. The default access for variables is protected and the default access for methods is public. It is assumed that (due to encapsulation) that you want to keep data hidden in the class, but methods need to be accessed othwerwise the class would be of no use.

However, there are cases when you want to allow external access to variables and there are also cases where you want to keep some methods from being called outside the class. This is achieved using attributes when you create the variable or method. The **`public`** keyword indicates that the variables or method can be accessed from outside the class. The **`protected`** keyword indicates that the variables or method can be access anywhere inside the class structure and the **`private`** keyword indicates that the variables or method can only be access within the same class. For instance variables **`protected`** and **`private`** are the same since they occur in the same place.

Consider the following:

```
class B
    public cvar pubVar=111;  // Can be accessed outside of class B
    protected cvar protVar=222; // Can be accessed anywhere inside class B and in
                              // classes derived from B
    private cvar privVar=333; // Can be access only from with class B
    cvar defAccessVar=444; // This has protected access by default
    ...
    cdef pubSub(x,y)  // This has public access by default. It can be accessed
                        // from within class B and outside class B
    ...
    end;
    protected cdef protSub(x,y)  // Can be accessed from inside class B and in
                                  // classes derived from B
    ...
    end;
    private cdef privSub(x,y)  // Can be accessed only from inside class B
        ...
    end;
    def pubInstSub(x,y)  // This has public access by default. It can be accessed
                        // by instances of class B and from outside class B
    ...
    end;
    protected def protInstSub(x,y)  // Can be accessed by instances of class B and
                                  // any instance derived from class B
    ...
    end;
    private def privInstSub(x,y)  // Can be accessed only by instances of class B
        ...
    end;
end
class A < B
    // From here pubVar, protVar, pubSub, and protSub can be accessed.
    def mySub
    // From here pubVar, protVar, pubSub, and protSub can be accessed.
    // Also From here pubInstSub and protInstSub can be accessed.
    end
end;
```

get/set

Single argument methods can be called in two ways. The first is the standard way as follows:

```
myMethod(myArg);
```

There is anyway to do this. You can do the following:

```
myMethod=myArg;
```

OE knows that myMethod is a property that contains a method data type and calls it instead replacing its value. This is the same reason that using myMethod (without parentheses) on the right side of an expression causes it to be called instead of returning a method data type.

The concept is useful if you decide to make an instance variable in a class public, then discover later that you need to make it a method in order to do validation. You can the public variable to a method and the code already written will still work.

MethodRef

In OE, when you use a method name in an expression, the method is evaluated (whether or not you use parentheses for a argument list). However, there are times when you might want to treat a method as data. For example, you might want tell some object to call a specified method, when an event occurs. To do this, you need to create a MethodRef object. The easiest way to do this is to use the unary ampersand (&) operator. When you use the & operator in front of a method name, instead of evaluating the method, it will create a MethodRef object, which is a data item. The example below illustrates a common use of the MethodRef object. Two functions are created (X and Y). X takes one argument, Y takes no arguments but returns a 1. When you call X using Y as the argument, the Y method is evaluated first (reeturns 1), so the value passed to X is 1. However, when you call X using &Y as the argument, it creates a MethodRef object and the MethodRef is passed to X.

```
def  X(pArg)
        writeln("Arg Value is "+pArg.toS);
        if pArg.isKindOf(MethodRef)
            var vRet=pArg.run;  // If it is a method, call it
            writeln("In X-Method Return Value is "+vRet.toS);
        end;
end;
def Y
        return 1;
end;
...
...
X(Y);  // call the method X with the parameter Y (Y is evaluated first)
        //  The result will be:
        //  Arg Value is 1
X(&Y);  // call the method X with the MethodRef of Y
        //  The result will be:
        //  Arg Value is Instance Class(MethodRef)
        //  In X-Method Return Value is 1
```

Iterators/Closures/functions

A closure is a function that can be passed as an argument to a function call. It keeps its state until it is no longer used. By state, it means that the local and argument variables keep their values, between calls. This makes it ideal for developing iterators, methods similar to loops.

Some standard objects (Integers, Strings and Arrays) have built-in iterators (each, times, upTo, etc.). These iterators are implemented as closures. Closures are created using the do keyword following a method call. For example, the way to iterate through a list of integers is to use the times Integer method as follows:

```
5.times do(vIdx)
       writeln(" Loop index="+vIdx.toS);
end
```

This uses the integer value of 5 as the receiver and calls the times method of the integer object. Since the method call is followed by the **do** keyword, the code following (until the matching end keyword) will be a closure method. This method will be passed to the times method as the last parameter (in the case, the only parameter).

You can also do this directly in OE code. For example, to create an iterator similiar to the Integer times method, you could do the following:

```
def mytimes(finalVal,callIt)
   var currVal=0;
   while (currVal<finalVal);
      callIt(currVal);   //   Call the closure method
      currVal++;
   end;
end
```

You would call this method as follows:

```
myTimes(5) do(vIdx)
       writeln(" Loop index="+vIdx.toS);
end;
```

The first parameter will be the integer value and the second parameter will be the closure method (callIt).

Exceptions

During the execution of program unexpected things happen. A user enters invalid data, a file which should be present is missing, or any number of other things might happen. Good programming should anticipate when things might go wrong. Also when developing or debugging a program many errors occur. It is good to be able to trace back to where the error occurred.

One way to do this is to have methods return error codes. The calling program then checks the error code before proceeding. This means managing these error codes through sometimes many layers of method calls. A more modern approach is to use exceptions.

Exceptions use the following structure:

```
try
       // Some code that might produce an error.
catch Type1[(e)]
...// code to run if a 'Type' error condition occurs
catch Type2[(e)]
...// code to run if a 'Type2' error condition occurs
else
// Code to run if no error occurs
anyway
// Code to run whether an error occurs or not
end;
```

The keyword **try** starts a block of code subject to the error checking. An error means that somewhere in the code between the **try** and the first **catch** an exception has been thrown. This includes the statements here as well as in any method that is called. Exceptions can be thrown in the ensuing code or they can be thrown by OE/evm itself. A **catch** block describes what to do if an certain type of exception is thrown. You can have as many of these as you wish each for a different type of exception. The 'TypeN' are class objects, which are used to identify the class of the exception. They are usually derived from the base class `Exception`. The optional parameter on the catch clause specifies the name of the variable to hold the exception instance object. From this you can get the information about the exception including a stack trace.

The **else** clause is run only if the **try** code is successful and no exceptions were thrown. The **anyway** block runs whether there is an exception or not. If an exception occurs and there is no matching catch block, the search will continue at the previous try block. It will keep on going back until a catch block is found or control reaches the beginning of the program. Then the program will stop and the exception information will be sent to the console.

To throw an exception, use the **throw** keyword, followed by an instance (usually of one of the Exception derived classes). The Exception class has two instance variables (message and code) that are useful for most any exception. For example:

```
throw TypeError.make("User entered the wrong type of data",0);
```

If you had a catch clause with TypeError specified, you could catch any error of this type. You can also create your own exception derived classes and use them if you wish to return more information.

There are many built-in exception classes that you can use. The following

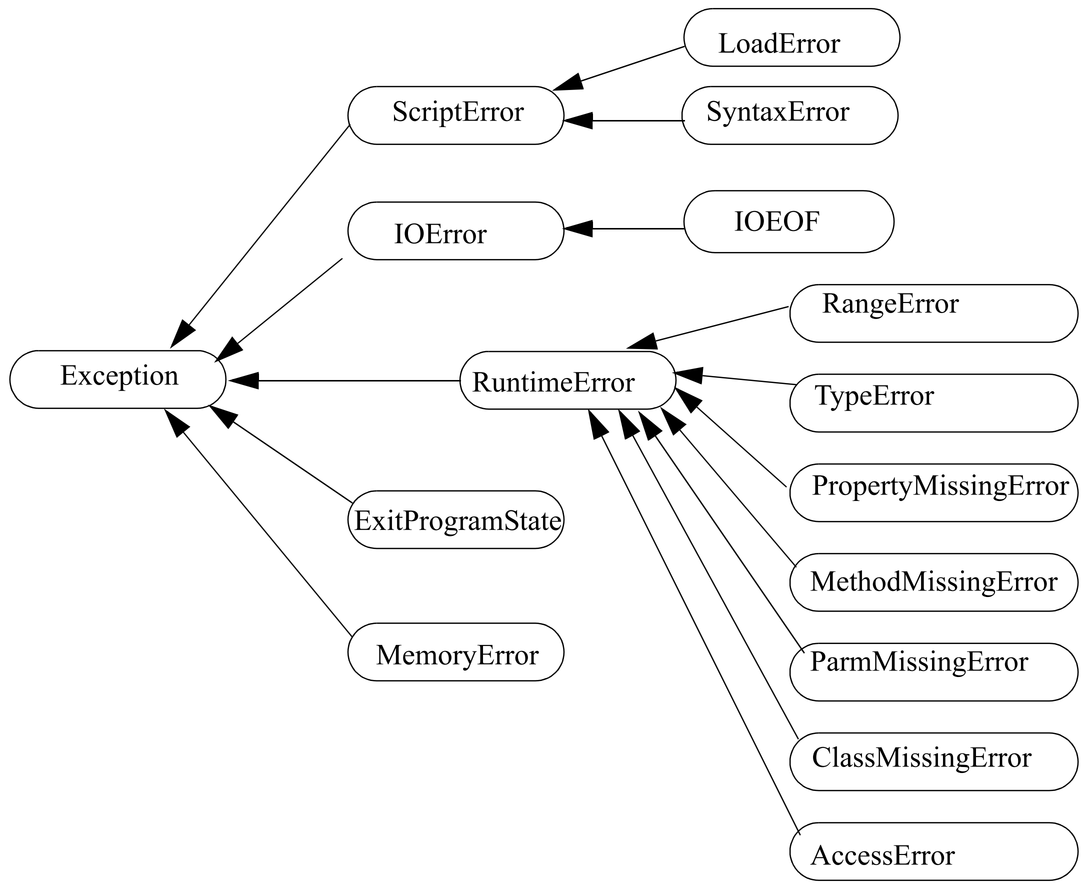
Table 3:

Class Name	Description
Exception	The base class of all exception classes
ScriptError	An error occured while compiling a script.
LoadError	A general error occured during the compilation process
SyntaxError	A syntax error occured during a compile.
RuntimeError	Any of a series of run time errors
RangeError	A value is out of range.

Table 3:

Class Name	**Description**
TypeError	The wrong data type was used.
MethodMissingError	A method is missing.
ClassMissingError	A class is missing.
ParmMissingError	A needed argument is missing.
PropertyMissingError	A property is missing
AccessError	Trying to access an inaccessible property.
IOError	An error occured during input or output
MissingFileError	A file is missing.
IOEOF	An end of file is reached. This is not used but can be used by a program if desired
MemoryError	A fatal memory error.
ExitProgramState	An exit program condition occured. The exitProgram method was called or the user aborted the program.

These exception classes have a structure to them. For example, `AccessError` is derived from `RuntimeError`. The following figure shows the relationship of the various exception classes. It is important to know the structure because a **`catch`** clause specifying `RuntimeError` will catch any `RuntimeError` as well as any subclass of `RuntimeError` (e.g. `AccessError`).


LoadError
ScriptError
SyntaxError
IOError
IOEOF
RangeError
Exception
RuntimeError
TypeError
PropertyMissingError
ExitProgramState
MethodMissingError
MemoryError
ParmMissingError
ClassMissingError
AccessError

Consider the following:

```
class MyException<TypeError
   def onInit(message,code, code2)
      super(message,code);
      ivar myCode2=code2;
   end;
   def GetCode2() return myCode2; end;
end;

def ThrowSub
   throw MyException.make("Test Error Message", 555, 888);
   //throw "My Throw String";
   writeln("Leaving ThrowSub");
end;

def TestSub()
   writeln("In TestSub");
   try
      writeln("In TestSub Try block");
      ThrowSub;
   catch RangeError(e)
      writeln("In TestSub Try catch RangeError");
      writeln("   It should NOT come here");
   else
      writeln("In TestSub Try Else block");
   anyway
      writeln("In TestSub Try anyway block");
   end;
   writeln("In TestSub Leaving");
   writeln("   It should NOT come here");
   return;
end;
// Start HERE
try
   writeln("Before calling TestSub");
   TestSub;
   writeln("After calling TestSub");
catch MyException(e)
   writeln("In MAIN Catch block MyException");
   writeln("Error Message="+e.message);
   writeln("Error Code="+e.code.toString);
   writeln("Error Code2="+e.GetCode2.toString);
   e.dumpStackTrace;
catch String(e)
   writeln("In Catch block String-"+e.toS);
else
   writeln("In MAIN Try else Block");
anyway
   writeln("In MAIN Try anyway Block");
end;

```

Here is a short contrived program to illustrate the try-catch-else-anyway features. A new exception class (MyException) is created derived from the TypeError exception, followed by two methods. The program starts at line 32 with the beginning of the try block setting up the area concerned with exceptions. The following is the console output from this run.

```
Before calling TestSub
In TestSub
In TestSub Try block
In TestSub Try anyway block
In MAIN Catch block MyException
Error Message=Test Error Message
Error Code=555
Error Code2=888
==Stack Trace==
Object$ThrowSub#10(75)
Object$TestSub#19(105)
Object$main#35(1098)
ObjectBase$startRun#0(0)
In MAIN Try anyway Block
```

Some text lines are written showing the progress of the program. It calls the TestSub method which has its own try block. It writes a console message, then calls a method which throws an exception of type MyException (it creates an instance of MyException, supplying the values for message and code along with a second kind of code, unique to MyException instances. When the exception is created (with the throw keyword), the evm starts looking for a catch block to handle the condition. There is no try block in the ThrowSub method, so it jumps right back to the calling method. Notice that is does not execute the writeln statement after the throw line. It is skipping back through try blocks only. It gets back to the TestSub method and examines the catch blocks there. It does not find one that matches the MyException type. It skips the else on line 23 since there was an error, however, it does perform the anyway block on line 25. It jumps back to the calling method, skipping the two writeln statments on lines 28 and 29. It gets to the main try block and finds a matching catch block on line 37. It then writes the exception information from the exception object (that we created back on line 10). It also dumps a stack trace showing where the exception was thrown and the path to it. The else clause on line 45 is skipped, but the anyway on line 47 is performed.

As a second example using (mostly) the same code, look at lines 43 and 44. This catch block is using String as the catch class object. You do have to use a class based on the Exception class. It is very convenient to do so, but not absolutely necessary. In fact, any class object can be used. In this case, we used the String class. On line 11 we have commented out a throw statement. Uncomment this line (remove the double slashes) and comment line 10 (add doub le slashes to the beginning of the line). When it is run this time, a String type exception thrown. A similar process to the above will occur, but this time the MyException catch block on line 37 will not match, but the catch block on line 45 will. It writes the message from the exception object to the console. Since this is not an instance of an object based on the Exception class, the stack trace, message and code are not available. The following is the output.

```
Before calling TestSub
In TestSub
In TestSub Try block
In TestSub Try anyway block
In Catch block String-My Throw String
In MAIN Try anyway Block
```

Importing

As programs gets larger, it gets more and more impractical to keep all the code in one source file. A large program will consist of many classes each doing their part of the overall function. It may also need to use extensions written in C/C++. have the main source module contain part of the main program. These classes will be put into separate source files where they can be used by more than one program. If the classes have general enough functionality they may exist in a library for use by many programs. Other source modules might derive new classes from some of these.

All in all this points to a system with many source modules. A source module might contain one class or many related classes. To assemble a program, we need a way to put all the pieces together. This is done with the import method.

The import method has the following form.

```
import(fileName);
```

If a complete file path is provided, it will use that to locate the file to import. If only a file name is provided (no path), then it searches the search path for the file.

The imported file can also import other files. The usual practice is to use the import method at the top of a source to import any file that it depends on. For example, assume that there are three class files(File1, File2 and File3) and File2 has classes derived from the classes in File3 and, in turn, File1 has classes derived from the classes in File2. File2 would import File3 before defining the new derived classes. File1 would only have to import File2 (since File3 would already be imported by File2.

Files are imported only once. If a file is already imported and an attempt is made to import it again, it will do nothing and just use the information from the first importation.