ObjectScript Programming Language Reference

Robin D. Clark

Dec 29, 2000

Abstract

This document describes ObjectScript which is a weakly typed object oriented programming language that is suitable for use as a scripting language. The language is designed to be easy to use, yet powerful. It provides private fields and methods, inheritance, exceptions, synchronization and threading, nested functions and classes, and operator overloading.

Contents

1 Introduction

ObjectScript is a weakly typed object oriented programming language. Being weakly typed means that values have a type, which may be a built-in or user defined type, but variables do not have a type.

Every operation, such as built-in operations (`+`, `-`, `%`, etc.), including array operations (`[]`), are translated into a method call. This is similar to smalltalk, except that a C-like grammar is retained to preserve the familiar syntax and expected order of operations (ie. `*` has precedence over `+`).

An ObjectScript program does not have a `main` function or method, like C or Java, but instead the source file is evaluated in order from beginning to end. The source file(s) may define functions that are called at a later time.

In order to make ObjectScript easier to learn for programmers familiar with C/Java/JavaScript it uses similar conditional and looping constructs (see example 1.1), as well as synchronization and exception constructs that are similar to Java.

Example 1.1 Sample Program

```
// calculate the n'th fibonacci number
function fib( n )
{
  if( n == 0 )
    return 0;
  else if( n == 1 )
    return 1;
  else if( n > 1 )
    return fib(n-1) + fib(n-2);
  else
    throw new Exception("bad input");
}

try
{
  for( var i=0; i<10; i++ )
    writeln("fib(" + i + ") is " + fib(i));

  writeln("fib(1.5) is " + fib(1.5));
}
catch(e)
{
  writeln("error: " + e);
}
```

All references (variables or functions) are resolved as the reference is evaluated. This means that it is possible for a reference to resolve to different functions or variables at different times (see example 1.2).

Example 1.2 References are evaluated at runtime

```
const var a = 1;

{
  function foo()
  {
    return a;
  }

  writeln("a: " + a);               // prints: ‘‘a: 1’’
  writeln("foo(): " + foo());       // prints: ‘‘foo(): 1’’

  var a = 2;

  writeln("a: " + a);               // prints: ‘‘a: 2’’
  writeln("foo(): " + foo());       // prints: ‘‘foo(): 2’’
}
```

2 Language Overview

This section gives an overview of the ObjectScript language, including description of the language's operators, how references are resolved, flow control, functions, and how functions can be used to define new types.

2.1 Operators

The operators used by ObjectScript are similar in function and precedence to those used in C, Java, JavaScript, etc. (see table 2.1). The main difference is that, with the exception of the `new`, `()`, and assignment operators, all the operators are converted into method calls, which means the exact behavior of an operator will depend on the values being operated upon.

Table 2.1 Operator Precedence

Name	Operators
Postfix	`() [] ++ --`
Unary	`++ -- + -   !`
Type	`new`
Multiplicative	`* / %`
Additive	`+ -`
Shift	`<< >> >>>`
Relational	`< > >= <= instanceof`
Equality	`== !=`
Bitwise And	`&`
Bitwise Xor	`^`
Bitwise Or	`|`
Logical And	`&&`
Logical Or	`||`
Conditional	`?:`
Assignment	`= += -= *= /= %= >>= <<= >>>= &= ^= |=`

The following subsections will illustrate how the evaluator translates the built-in operator to a method call. In these examples, the values `a`, `b`, and `c` may infact be other expressions of greater or equal precedence than the operator in question. Unless otherwise specified, the order in which `a`, `b`, and `c` are evaluated is undefined.

Note that just because a programming language provides a mechanism for operator overloading does not mean that it is a good idea to do so. Operator overloading is a feature that is easily abused by the programmer, and can lead to difficult to understand programs. This feature is provided in order that the language can be extended by add-on libraries.

2.1.1 Postfix Operators

Operator	Translatest To
`a()`	n/a (call `a` as a function)
`a[b]`	`a.elementAt(b)`
`a++`	`a = a.uopIncrement()` (evaluates to the initial value of `a`)
`a--`	`a = a.uopDecrement()` (evaluates to the initial value of `a`)

The methods called to evaluate the `++` and `--` operators don't actually change the value of the operand, but instead the evaluator also performs an assignment operation to change the value of the operand.

2.1.2 Unary Operators

Operator	Translatest To
`++a`	`a = a.uopIncrement()`
`--a`	`a = a.uopDecrement()`
`+a`	`a = a.uopPlus()`
`-a`	`a = a.uopMinus()`
`a`	`a = a.uopBitwiseNot()`
`!a`	`a = a.uopLogicalNot()`

Like with the *Postfix* (see section **??**) operators, the methods called to evaluate these operators don't actually chaneg the value of the operand, but instead the evaluator also performs an assignment operation to change the value of the operand.

2.1.3 Type Operator

Operator	Translatest To
`new a()`	n/a (call `a` as a constructor)

2.1.4 Multiplicative Operators

Operator	Translatest To
`a * b`	`a.bopMultiply(b)`
`a / b`	`a.bopDivide(b)`
`a % b`	`a.bopRemainder(b)`

2.1.5 Additive Operators

Operator	Translatest To
`a + b`	`a.bopPlus(b)`
`a - b`	`a.bopMinus(b)`

2.1.6 Shift Operators

Operator	Translatest To
`a << b`	`a.bopLeftShift(b)`
`a >> b`	`a.bopSignedRightShift(b)`
`a >>> b`	`a.bopUnsignedRightShift(b)`

2.1.7 Relational Operators

Operator	Translatest To
`a < b`	`a.bopLessThan(b)`
`a > b`	`a.bopGreaterThan(b)`
`a <= b`	`a.bopLessThanOrEquals(b)`
`a >= b`	`a.bopGreaterThanOrEquals(b)`
`a instanceof b`	`a.bopInstanceOf(b)`

2.1.8 Equality Operators

Operator	Translatest To
`a == b`	`a.bopEquals(b)`
`a != b`	`a.bopNotEquals(b)`

2.1.9 Bitwise And Operator

Operator	Translatest To
`a & b`	`a.bopBitwiseAnd(b)`

2.1.10 Bitwise Xor Operator

Operator	Translatest To
`a b̂`	`a.bopBitwiseXor(b)`

2.1.11 Bitwise Or Operator

Operator	Translatest To
`a | b`	`a.bopBitwiseOr(b)`

2.1.12 Logical And Operator

Operator	Translatest To
`a && b`	`a.bopLogicalAnd(b)`

To evaluate the `&&` operator, first `a` is evaluated, and if it evaluates to `true` then `b` is evaluated.

2.1.13 Logical Or Operator

Operator	Translatest To
`a || b`	`a.bopLogicalOr(b)`

To evaluate the `||` operator, first `a` is evaluated, and if it evaluates to `false` then `b` is evaluated.

2.1.14 Conditional Operator

Operator	Translatest To
`a ? b : c`	n/a

To evaluate the `?:` operator, first `a` is evaluated, and if it evaluates to `true` then `b` is evaluated, otherwise `c` is evaluated. The result of evaluating this operator is the result of evaluating `b` or `c`, which ever one is actually evaluated.

2.1.15 Assignment Operators

Operator	Translatest To
`a = b`	n/a
`a += b`	`a = a.bopPlus(b)`
`a -= b`	`a = a.bopMinus(b)`
`a *= b`	`a = a.bopMultiply(b)`
`a /= b`	`a = a.bopDivide(b)`
`a %= b`	`a = a.bopRemainder(b)`
`a <<= b`	`a = a.bopLeftShift(b)`
`a >>= b`	`a = a.bopSignedRightShift(b)`
`a >>>= b`	`a = a.bopUnsignedRightShift(b)`
`a &= b`	`a = a.bopBitwiseAnd(b)`
`a ≙ b`	`a = a.bopBitwiseXor(b)`
`a |= b`	`a = a.bopBitwiseOr(b)`

2.2 Casting

The evaluator, or any script code which needs to ensure that a value is of a particular type, uses the casting methods (see table 2.2). For example, when evaluating an `if` (see section ??) statement, the evaluator must convert the conditional expression to a boolean value.

If the value to be cast is already of the specified type, then the casting method simply returns the object, otherwise it returns a new object.

Table 2.2 Casting to built-in types

Type	Method
Boolean	`castToBoolean`
String	`castToString`
ExactNumber	`castToExactNumber`
InexactNumber	`castToInexactNumber`

2.3 Scope

ObjectScript uses nested scope, meaning that a variable or function being referenced is resolved by first looking in the current scope, and then if not found recursively look in the previous scopes (see example 2.1). All references to to variables or functions are resolved at run-time, rather than as the source code is parsed.

Example 2.1 Scope Example

```
var foo = 1;
var bar = 1;

{
  var foo = 2;
  writeln("foo: " + foo);  // prints ‘‘foo: 2’’
  writeln("bar: " + bar);  // prints ‘‘bar: 1’’
}
```

2.4 Flow Control

ObjectScript features program flow control constructs that are similar to those provided by C, Java, and JavaScript. Unlike C and Java, `switch` statements are not supported. Unlike C, exceptions are supported.

2.4.1 If-Else

```
if ( Expression )
  EvaluationUnit_1
( else EvaluationUnit_2 )?
```

The *Expression* must evaluate to a boolean value. If the expression evaluates to `true`, then *EvaluationUnit*$_1$ is evaluated. Otherwise *EvaluationUnit*$_2$, if it is present, is evaluated.

2.4.2 While-Loop

```
while ( Expression ) EvaluationUnit
```

For each time through the loop *Expression*, which must evaluate to a boolean value, is evaluted. If not `false`, the *EvaluationUnit* is evaluated, otherwise the loop is finished being evaluated.

2.4.3 For-Loop

```
for ( (PreLoopStatement)? ; (Expression_1)? ; (Expression_2)? )
  EvaluationUnit
```

is equivalent to:

```
PreLoopStatement;
while ( Expression_1 )
{
  EvaluationUnit
  Expression_2;
}
```

If present, the *PreLoopStatement* is evaluated. Then for each time through the loop *Expression*$_1$, which must evaluate to a boolean value if present, is evaluated. If *Expression*$_1$ is not present, it defaults to `true`. If not `false` the *EvaluationUnit* is evaluated, followed by *Expression*$_2$, if present. Otherwise the loop is finished being evaluated.

2.4.4 Break and Continue

```
break;
```

or:

```
continue;
```

The `break` and `continue` statements provide a means for exiting a loop. The `break` statement causes the innermost enclosing loop to exit. The `continue` statement causes the flow of execution to jump to the beginning of the next iteration of the loop. It is an error for a `break` or `continue` to not be enclosed in a loop.

2.4.5 Try-Catch-Finally

```
try EvaluationUnit_1
( catch ( IDENTIFIER ) EvaluationUnit_2 )?
( finally EvaluationUnit_3 )?
```

To provide a means of catching exceptions, a Java-like `try- catch-finally` statement is provided. If an exception is thrown while evaluating *EvaluationUnit*$_1$, then the optional `catch` block is evaluated. The thrown object is stored in variable with name specified by *IDENTIFIER*, which is in scope while evaluating *EvaluationUnit*$_2$. Finally, the optional `finally` block (ie. *EvaluationUnit*$_3$) is evaluated.

2.4.6 Throw

```
throw Expression ;
```

The `throw` statement provides a way to throw exceptions. The *Expression* should evaluate to an object that is an instance of type `Exception`.

2.4.7 Synchronisation

```
synchronized ( Expression ) EvaluationUnit
```

ObjectScript provides a means of serialising execution of a piece of code by means of the `synchronized` statement. The *Expression* should evaluate to an object, whose monitor is acquired before evaluating *EvaluationUnit*, and released after.

3 Variables and Functions

In ObjectScript, variables and functions are closely related. Functions are, in fact, variables.

3.1 Permissions

```
( const | public )*
```

A variable of a function can be declared to be `const` and/or `public`. A `const` variable (or function) cannot be modified once it has been assigned an initial value. A `public` variable or function can be accessed as a public attribute of an object. If a variable or function that is not public is accessed as an attribute of the object (with the exception of `this`) will cause an exception to be thrown.

3.2 Variables

```
(Permissions)? var IDENTIFIER ( = Expression)? ;
```

A new variable is declared with the `var` keyword, and can optionally have an initializer to assign the variable an intial value (see example 3.1).

Example 3.1 Variables

```
var a = 1 + 2;
const var b;
b = 3;

a = b;                              // ok;
b = a;                              // error!
```

3.3 Functions

```
(Permissions)? function IDENTIFIER ( (Arglist)? )
( extends PrimaryExpression FunctionCallExpressionList )?
{ Program }
```

A new function is declared with the `function` keyword. A function can optionally be declared with permissions. If the function is not declared `const`, then it can be replaced (see example 3.2). In this regard, functions behave exactly the same as variables.

Example 3.2 Function Permissions

```
const function foo() {}             // a no-op function

function bar() {}                   // a no-op function

bar = foo;                          // ok
foo = bar;                          // error!
```

Functions can be used to implement classes and methods interchangeably. When a function is called, a new scope is allocated and pushed onto the top of the scope stack. As the function is evaluated, local variables or inner-functions are defined within the newly allocated scope. When the function returns, the newly allocated scope is popped from the scope stack. If the function is called as a constructor, the popped scope is returned as the newly constructed object, otherwise the scope is discarded. If the function is called as a constructor, the type of the newly allocated scope (ie. the object being constructed) is the function.

When a function is defined, it records the scope that it is defined within. In this way, functions defined within an outer function that is called as a constructor can have access to other functions or variables defined within the scope of the outer function (see example 3.3).

Example 3.3 Function Example

```
function ExampleFunction( a, b, c )
{
  var aVar = a + b + c;

  /* Note: ‘‘public’’ has no meaning if this is called as a function,
   *       but determines whether this member can be accessed as a
   *       member of the constructed object if this is called as a
   *       method.
   */
  public function aFunction()
  {
    return aVar;
  }

  return aFunction();
}

// Call as function:
ExampleFunction( 1, 2, 3 );  // => 6

// Call as constructor:
var foo = new ExampleFunction( 1, 2, 3 );
foo.aFunction();             // => 6

// Note: functions don’t forget the scope they are defined in:
var bar = foo.aFunction;
bar();                       // => 6

// The type is the function that was the constructor:
foo instanceof ExampleFunction;   // => true
```

A function may `extend` another function. If this is the case, that function may only be called as a constructor. When the function is called, arguments to the function being extended are first evaluated, then the function being extended is called with the newly constructed scope, finally the derived function is evaluated. This is used to implement inheritance. The parent class, ie. the function being extended, may define variables and functions within the scope of the derived class, which may then be overriden if needed by the derived class (see example 3.4).

Example 3.4 Inheritance Example

```
function A( a )
{
  var aa = a;

  public function getA()
  {
    return a;
  }
}

function B( b ) extends A( b/2 )
{
  public function getB()
  {
    return b;
  }

  public function getAplusB()
  {
    // Note: ‘‘a’’ is not accessible here, but ‘‘aa’’ is
    return aa + b;
  }
}
```

3.4 Anonymous Functions

function (*(Arglist)?*)
(extends *PrimaryExpression FunctionCallExpressionList)?*
{ *Program* }

ObjectScript also provides a way to define anonymous functions (see example 3.5). As you can see, the anonymous function syntax simply creates a new function object without binding it to a particular variable.

The anonymous function syntax is part of the *Expression* portion of the grammer, so it can be used, for example, to define functions inline with a method call or object instantiation.

Example 3.5 Anonymous Function

```
// the following two function declarations are equivalent:
var plus = function( a, b ) { return a + b; };
function plus( a, b ) { return a + b; }

// using an anonymous function to implement an action listener:
button.addActionListener( new (function() extends java.awt.event.ActionListen() {
  public function actionPerformed( evt )
  {
    ... handle button press here ...
  }
})() );
```

4 Built-in Types

TODO: object diagram showing class hierarchy, and which methods are implemented by what types.

4.1 Object

The `Object` type is the base class for all other types, built-in or user-defined.

4.2 Exact Number

The `ExactNumber` type is the built-in type representing an exact number. An exact number is a signed number represented by up to 64 bits. An exact number is immutable.

4.3 Inexact Number

The `ExactNumber` type is the built-in type representing an inexact, ie. floating point, number. A inexact number is ???. An inexact number is immutable.

4.4 String

The `String` type is the built-in type representing strings. A string is immutable.

4.5 Exception

The `Exception` type is the base class for all exceptions.

4.6 Thread

The `Thread` type is the base class used for creating new threads. The subclass should implement the `run` method.

5 Grammar

```
⟨program-file⟩                     ::== ⟨progam⟩ EOF

⟨program⟩                          ::== ⟨evaluation-unit⟩*

⟨evaluation-unit⟩                  ::== ⟨scope-block⟩
                                      |  ⟨variable-declaration⟩ ;
                                      |  ⟨function-declaration⟩
                                      |  ⟨try-statement⟩
                                      |  ⟨for-loop-statement⟩
                                      |  ⟨while-loop-statement⟩
                                      |  ⟨conditional-statement⟩
                                      |  ⟨synchronized-statement⟩
                                      |  ⟨return-statement⟩
                                      |  ⟨break-statement⟩
                                      |  ⟨continue-statement⟩
                                      |  ⟨throw-statement⟩
                                      |  ⟨import-statement⟩
                                      |  ⟨eval-statement⟩
                                      |  ⟨expression⟩ ;

⟨scope-block⟩                      ::== { ⟨program⟩ }

⟨variable-declaration⟩             ::== ⟨permission⟩* var IDENTIFIER ( = ⟨expression⟩ )?

⟨function-declaration⟩             ::== ⟨permission⟩* function IDENTIFIER ( ⟨arg-list⟩? ) ( extends
                                        ⟨primary-expression⟩ ⟨function-call-expression-list⟩ )? { ⟨program⟩
                                        }

⟨permission⟩                       ::== const
                                      |  public

⟨function-call-expression-list⟩    ::== ( ⟨expression⟩? )

⟨arg-list⟩                         ::== IDENTIFIER ( , IDENTIFER )*

⟨try-statement⟩                    ::== try ⟨evaluation-unit⟩ ( catch ( IDENTIFIER ) ⟨evaluation-unit⟩
                                        )? ( finally ⟨evaluation-unit⟩ )?

⟨for-loop-statement⟩               ::== for ( ⟨pre-loop-statement⟩? ; ⟨expression⟩? ; ⟨expression⟩? )
                                        ⟨evaluation-unit⟩

⟨pre-loop-statement⟩               ::== ⟨variable-declaration⟩
                                      |  ⟨expression⟩

⟨while-loop-statement⟩             ::== while ( ⟨expression⟩ ) ⟨evaluation-unit⟩

⟨conditional-statement⟩            ::== if ( ⟨expression⟩ ) ⟨evaluation-unit⟩ ( else ⟨evaluation-unit⟩ )?

⟨synchronized-statement⟩           ::== synchronized ( ⟨expression⟩ ) ⟨evaluation-unit⟩

⟨return-statement⟩                 ::== return ⟨expression⟩? ;

⟨break-statement⟩                  ::== break ;
```

⟨*continue-statement*⟩	::==	`continue ;`
⟨*throw-statement*⟩	::==	`throw` ⟨*expression*⟩ `;`
⟨*import-statement*⟩	::==	`import` STRING_LITERAL `;`
⟨*eval-statement*⟩	::==	`eval` ⟨*expression*⟩ `;`
⟨*expression*⟩	::==	⟨*assignment-expression*⟩ (`,` ⟨*assignment-expression*⟩)*
⟨*assignment-expression*⟩	::==	⟨*conditional-expression*⟩ (⟨*assignment-operator*⟩ ⟨*conditional-expression*⟩)*
⟨*assignment-operator*⟩	::==	`=`
	|	`+=`
	|	`-=`
	|	`*=`
	|	`/=`
	|	`%=`
	|	`<<=`
	|	`>>=`
	|	`>>>=`
	|	`&=`
	|	`^=`
	|	`|=`
⟨*conditional-expression*⟩	::==	⟨*logical-or-expression*⟩ (`?` ⟨*logical-or-expression*⟩ `:` ⟨*logical-or-expression*⟩)?
⟨*logical-or-expression*⟩	::==	⟨*logical-and-expression*⟩ (`||` ⟨*logical-and-expression*⟩)*
⟨*logical-and-expression*⟩	::==	⟨*bitwise-or-expression*⟩ (`&&` ⟨*bitwise-or-expression*⟩)*
⟨*bitwise-or-expression*⟩	::==	⟨*bitwise-xor-expression*⟩ (`|` ⟨*bitwise-xor-expression*⟩)*
⟨*bitwise-xor-expression*⟩	::==	⟨*bitwise-and-expression*⟩ (`^` ⟨*bitwise-and-expression*⟩)*
⟨*bitwise-and-expression*⟩	::==	⟨*equality-expression*⟩ (`&` ⟨*equality-expression*⟩)*
⟨*equality-expression*⟩	::==	⟨*relational-expression*⟩ (⟨*equality-operator*⟩ ⟨*relational-expression*⟩)*
⟨*equality-operator*⟩	::==	`==`
	|	`!=`
⟨*relational-expression*⟩	::==	⟨*shift-expression*⟩ (⟨*relational-operator*⟩ ⟨*shift-expression*⟩)*
⟨*relational-operator*⟩	::==	`<`
	|	`>`
	|	`>=`
	|	`<=`
	|	`instanceof`
⟨*shift-expression*⟩	::==	⟨*additive-expression*⟩ (⟨*shift-operator*⟩ ⟨*additive-expression*⟩)*

⟨*shift-operator*⟩ ::== `<<`
| `>>`
| `>>>`

⟨*additive-expression*⟩ ::== ⟨*multiplicative-expression*⟩ (⟨*additive-operator*⟩ ⟨*multiplicative-expression*⟩)*

⟨*additive-operator*⟩ ::== `+`
| `-`

⟨*multiplicative-expression*⟩ ::== ⟨*unary-expression*⟩ (⟨*multiplicative-operator*⟩ ⟨*unary-expression*⟩)*

⟨*multiplicative-operator*⟩ ::== `*`
| `/`
| `%`

⟨*unary-expression*⟩ ::== ⟨*unary-operator*⟩? ⟨*postfix-expression*⟩

⟨*unary-operator*⟩ ::== `++`
| `--`
| `+`
| `-`
| `~`
| `!`

⟨*postfix-expression*⟩ ::== ⟨*type-expression*⟩ ⟨*postfix-operator*⟩?

⟨*postfix-operator*⟩ ::== `++`
| `--`

⟨*type-expression*⟩ ::== ⟨*allocation-expression*⟩
| ⟨*primary-expression*⟩

⟨*allocation-expression*⟩ ::== `new` ⟨*primary-expression*⟩ ⟨*function-call-expression-list*⟩

⟨*primary-expression*⟩ ::== ⟨*primary-prefix*⟩ ⟨*primary-postfix*⟩*

⟨*primary-prefix*⟩ ::== ⟨*this-primary-prefix*⟩
| ⟨*identifier-primary-prefix*⟩
| ⟨*paren-primary-prefix*⟩
| ⟨*function-primary-prefix*⟩
| ⟨*literal*⟩

⟨*this-primary-prefix*⟩ ::== `this`

⟨*identifier-primary-prefix*⟩ ::== IDENTIFIER

⟨*paren-primary-prefix*⟩ ::== `(` ⟨*expression*⟩ `)`

⟨*function-primary-prefix*⟩ ::== `function` `(` ⟨*arglist*⟩? `)` (`extends` ⟨*primary-expression*⟩ ⟨*function-call-expression-lis*
)? `{` ⟨*program*⟩ `}`

⟨*primary-postfix*⟩ ::== ⟨*function-call-primary-postfix*⟩
| ⟨*array-subscript-primary-postfix*⟩
| ⟨*property-identifier-primary-postfix*⟩

⟨*function-call-primary-postfix*⟩	::==	⟨*function-call-expression-list*⟩
⟨*array-subscript-primary-postfix*⟩	::==	`[` ⟨*expression*⟩ `]`
⟨*property-identifier-primary-postfix*⟩	::==	`.` IDENTIFIER
⟨*literal*⟩	::==	INTEGER_LITERAL
	|	FLOATING_POINT_LITERAL
	|	STRING_LITERAL
	|	`true`
	|	`false`
	|	`null`