An Introduction to the Objeck Programming Language

Randy Hollines
objeck@gmail.com

October 11, 2009

Abstract

Brief introduction to the Objeck programming language and it's accompanying features. This article is intended to introduce programmers and compiler designers to the unique features and language syntax of the Objeck programming language. Unless otherwise noted, this article covers functionality that will be included in the first *0.0.1 alpha* release.

Contents

1 Introduction

The Objeck program language is an object-oriented computer language that is designed to be a general purpose distributed programming system. The Objeck language allows programmers to quickly create solutions by leveraging pre-existing class libraries. Note, this initial *0.0.1 alpha* release lacks rich library support, focusing mainly on language syntax and virtual machine (VM) functionality. Please check the project home page for the latest updates. The syntax for the language was designed with symmetry in mind and enforces the notion that there should only be one way to do something. Features of this release include:

- Support for object-oriented programming (all data types are treated as objects)
- Cross platform independent (OS X, Linux and Windows)
- Runtime JIT support (for Intel based computers)
- Automatic memory management
- Support for static shared library support
- Basic block compiler optimizations

2 Getting Started

The Objeck computer language consist of a compiler and virtual machine. The compiler program is named `obc`, while the runtime virtual machine (VM) program is named `obr`.

Here is the world famous "Hello World" program written in the Objeck language:

```
bundle Default {
  class Test {
    function : Main(), Nil {
      "Hello World!"->Print();
    }
  }
}
```

2.1 Compiling Source

The example below compiles the source program `hello.obs` using the shared library `lang.obl` into the target binary file `hello.obe`. The two output file types that the compilers supports are executables and shared libraries. Shared libraries and executables are binary files that contain all of the metadata needed by the compiler to relink them into programs. They also have enough metadata to support runtime introspection (a feature that will be added in a future release). As a naming convention, executables must end in `.obe` while shared libraries must end in `.obl`.

```
obc -src src_programs/tests/hello.obs -lib lang.obl -dest hello.obe
```

Compiler options are listed below:

Option	*Description*
`-src`	path to source files, delimited by the '`,`' character
`-lib`	path to library files, delimited by the '`,`' character
`-tar`	target output `exe` for executable and `lib` for library
`-opt`	optimization level `s1` – `s3` with `s3` being the highest
`-dest`	output file destination

2.2 Executing

The above command line command executes the `hello.obe` VM executable. Note, for executables all required libraries are statically linked in the target output file. When compiling shared libraries, other shared libraries are <u>not</u> linked into the target output file.

```
obr ../compiler/hello.obe
```

3 The Basics

Now lets introduce you the core features of the Objeck language.

In Objeck, all data types are treated as objects. Basic objects provide supports for boolean, character, byte, integer and decimal types. These basic objects can be used to create complex user defined objects. The listing below defines the basic objects that are supported in the language:

Type	*Description*
`Char`	1 byte character
`Char[]`	character array
`Bool`	boolean value
`Bool[]`	boolean array
`Byte`	1 byte integer
`Byte[]`	byte array
`Int`	4 byte integer
`Int[]`	integer array
`Float`	8 byte decimal
`Float[]`	decimal array

3.1 Variable Declarations

Variables can be declared for all of the basic types described above and for user defined objects. Variables can be declared anywhere in a program and are bound to traditional block scoping rules. Variable assignments can be make during a declaration or at any other point in the program. Two different declaration styles are shown below:

```
...
value : Int;
another_value : Int := 13;
...
```

Types that are not initialized at declaration time are set to the following default values:

Type	*Initialization*
`Char`	'\0'
`Byte`	0
`Int`	0
`Float`	0.0
Array	`Nil`
Object	`Nil`

3.2 Expressions

The Objeck language supports various expression types. Some of these expression types include mathematical, logical, array and method expressions. The preceding sections describe some of the expressions that are supported in the Objeck language.

3.2.1 Mathematical and Logical Expressions

The following code example demonstrates two ways to printing the number 42. The first way invokes the `Print()` method of the literal `42`. The second prints the product of a variable and a literal.

```
bundle Default {
  class Test {
    function : Main(), Nil {
      42->Print();
      a : eight := 8;
      (eight * 7)->Print();
    }
  }
}
```

The following mathematical operators are supported in the Objeck language:

- addition (`+`)
- subtraction (`-`)
- multiplication (`*`)
- division (`/`)
- modulus – for integer values only (`%`)

The [`*`, `/`, `%`] operators have a high precedence then `+` and `-` operators. Operators of the same precedence are evaluated from left-to-right. Logical operations are of lower precedence than mathematical operations. All logical operators are of the same precedence and order is determined via left-to-right evaluation.

The following logical operators are supported in the Objeck language:

- and (`&`)
- or (`|`)
- equal (`=`)
- not equal (`<>`)
- less than (`<`)
- greater than (`>`)
- less than equal (`<=`)
- greater than equal (`>=`)

3.2.2 Arrays

The Objeck language support single and multi-dimensional arrays. Arrays are allocated dynamically from the heap. The memory that is allocated for arrays is managed automatically by the runtime garbage collector. All of the basic types described above (as well as user defined types) can be allocated as arrays. The code example below shows how a two-dimensional `Int` array is allocated and de-referenced.

```
...
array_2d := Int[,] = Int->New[2,3];
array_2d[0,2] := 13;
array_2d[1,0] := 7;
...
```

The size of an array can be obtained by calling the arrays `GetSize()` method. The `GetSize()` method will return the number of elements in a given array. For a multi-dimensional array the size method returns the number of elements across all dimensions. Character arrays can also be allocated using string literals as show below. It should also be noted that language has a `System.String` class that provides support for traditional string operations.

```
...
str : Char[] := "Hello World!";
str->GetSize()->Print();
...
```

3.3 Statements

Besides providing support for declaration statements the language has support for conditional and control statements. As with other languages, control statements can be nested in order to provide better flow of control. General control statements include `if` and `select` statements. Basic looping statements are `while` and `for` loops. Note, all statements rather decelerations or controls end with a '`;`'.

3.3.1 If Statement

An `if` statement is a control statement that executes the associated block of code if it evaluates to true. If the evaluation statement does not evaluate to true than an `else if` statement may be evaluated (if it exits), otherwise an `else` statement will be executed (if it exits). The example below demonstrates an `if` statement.

```
...
value : Int := System.ReadLine()->ToInt();
if(value <> 3) {
  "Not equal to 3"->Print();
}
else if(value < 13) {
  "Less than 13"->Print();
}
else {
  "Other number"->Print();
};
...
```

3.3.2 Select Statement

A `select` statement maps a value to 1 or more labels. Labels are associated to statement blocks. A label may either be a literal or an `enum` value. Multiple labels can be mapped to the same statement block. Below is an example of a `select` statement.

```
...
select(v) {
```

```
   label Color->Red: {
      "Red"->Print();
   }

   label 9:
   label 19: {
      v->Print();
   }

   label 27: {
      (3 * 9)->Print();
   }
};
...
```

3.3.3 While Statement

A `while` statement is control statement that will continue to execute it's main body as long as it's conditional expression evaluates to true. When it's conditional expression evaluates to false than the loop body will cease execution.

```
...
i : Int := 10;
while(i > 0) {
   i->Print();
   i := i - 1;
}
...
```

3.3.4 For Statements

The `for` statement is another common looping construct. The `for` loop consists of a pre-conditional statement followed by an evaluation expression and an update expression.

```
...
name : Char[] := "John";
for(i : Int := 0; i < name->GetSize(); i := i + 1;) {
```

```
  name[i]->Print();
}
...
```

4 User Defined Types

4.1 Enums

Enums are user defined enumerated types. The main use of an `enum` is to group a class of countable values, for example colors, into their own distinct class. Once `enum` values have been defined they may not be assigned or associated to a other `enum` groups or integer classes. The valid operations and expressions for enums are:

- assignment (`:=`)
- equals (`=`)
- not equals (`<>`) (not equals).

In addition, enum values may be used in `select` statements as conditional tests or labels.

```
enum Color {
   Red,
   Black,
   Green
}
```

4.2 Classes

Classes are user defined types that allow programmers to create specialized data types. Classes are made up of attributes (data) and operations (methods). Classes are used to encapsulate programming logic and localize information. Operations that are associated to a class may either be at the class level or instance level. Class instances are created by calling an object's `New()` function. Note, an object instance can only be created if one or more `New()` functions have been defined.

4.2.1 Class Inheritance

Classes may be derived from other class using the `origin` keyword. Class inheritance allows classes to share common functionality. The Objeck language supports single class inheritance, meaning that a derived class may only have one parent. The language also supports virtual classes, which assure that derived classes have defined all required operations from the base class. Virtual classes also allow programmer to define non-virtual methods that contain program behavior.

```
class Foo {
  @lhs : Int;

  New(lhs : Int) {
    @lhs := lhs;
  }

  method : native : AddTwo(rhs : Int), Int {
    return 2 + rhs;
  }

  method : virtual : AddThree(int rhs), Int;

  method : GetLhs(), Int {
    return lhs;
  }
}

class Bar origin Foo {
  New(value : Int) {
    Parent(value);
  }

  method : native : AddThree(rhs : Int), Int {
    return 3 + rhs;
  }

  function : Main(), Nil {
    bar : b := Bar->New(31);
```

```
    b->AddThree(9)->Print();
  }
}
```

4.2.2 Class Casting and Identification

An object that is inherited from another object may be either upcasted or downcasted. Object casting can be performed using the `As()` operator. The Object languages detects upcasting and downcastng at compile time. Upcasting requires a runtime check, while down casting does not. If cross casting is detected than a compile time error will be generated.

```
method : public : Compare(right : System.Base), Int {
  if(right <> Nil) {
    if(GetClassID() = right->GetClassID()) {
      a : A := right->As(A);

      if(@value = a->GetValue()) {
        return 0;
      };
    ...
...
```

The class that given object instance belongs to can found by calling it's `GetClassID` method. This method return an enum that is associated with that instance's class type. This method is generally used to determine if two object instance are of the same class.

4.2.3 Methods and Functions

The Objeck language support both methods and functions. Functions are public static procedures that may be executed by any class. Methods are operations that may be performed on an object instance. Methods have `public` and `private` qualifiers. Method that are `private` may only be called from within the same class while `public` methods may be called from other classes. Note, methods are `private` by default. The Objeck language support polymorphic methods and functions, meaning that there can be multiple methods with the same name within the same class as long as their arguments vary.

Methods and functions can either be executed in an interrupted or JIT compiled mode. Interrupted execution mimics microprocessor execution in a platform independent manner. JIT execution takes the compiled stack code an produces native machine code. Note, that there is initial overhead involved in the JIT complication process since it occurs at runtime. In addition, some methods can not be compiled into native machine code, normally this is due to detected register spills, but this is a rare case. The `native` keyword is used JIT compile methods and function at runtime.

A function or method may be defined as `virtual` meaning that any class that originates from the class declaring the virtual method must implement that class's virtual method or function. Virtual methods are a way to ensure that certain operations are available to a family of classes. If a class declares a virtual method than the class become virtual, meaning that it cannot be directly instantiated.

```
method : virtual : public : Compare(right : System.Base), Int;
```

5 Code Examples

5.1 Prime number between 1–1,000,000

```
bundle Default {
  class FindPrime {
    function : Main(), Nil {
      Run(1000000);
    }

    function : native : Run(topCandidate : Int), Nil {
      candidate : Int := 2;
      while(candidate <= topCandidate) {
        trialDivisor : Int := 2;
        prime : Int := 1;

        found : Bool := true;
        while(trialDivisor * trialDivisor <= candidate & found) {
          if(candidate % trialDivisor = 0) {
            prime := 0;
            found := false;
```

```
          }
          else {
            trialDivisor := trialDivisor + 1;
          };
        };

        if(found) {
          candidate->Print();
        };
        candidate := candidate + 1;
      };
    }
  }
}
```

6 Appendix: VM Instructions

The appendix below lists the types of stack instructions that can be executed by the VM. The VM was designed to portable and language independent. Early development versions of the VM included an inline assembler, which may be re-added in future releases.

Stack Operators		
Mnemonic	*Opcode(s)*	*Description*
LOAD_INT_LIT	4-byte integer	pushes integer onto stack
LOAD_FLOAT_LIT	8-byte float	pushes float onto stack
LOAD_INT_VAR	variable index	pushes integer onto stack
LOAD_FLOAT_VAR	variable index	pushes float onto stack
LOAD_SELF	n/a	pushes self integer on stack
STOR_INT_VAR	variable index	pops integer from stack and saves to index location
STOR_FLOAT_VAR	variable index	pops float from stack and saves to index location
COPY_INT_VAR	variable index	copies an integer from stack and saves to index location
COPY_FLOAT_VAR	variable index	copies a float from stack and saves to index location
LOAD_BYTE_ARY_ELM	array dimension	pushes byte onto stack; assumes array address was pushed prior
LOAD_INT_ARY_ELM	array dimension	pushes integer onto stack; assumes array address was pushed prior
LOAD_FLOAT_ARY_ELM	array dimension	pushes float onto stack; assumes array address was pushed prior
LOAD_ARY_SIZE	n/a	pushes array size as integer onto stack; assumes array address was pushed prior
STOR_BYTE_ARY_ELM	variable index	stores byte at index location; assumes array address was pushed prior
STOR_INT_ARY_ELM	variable index	stores integer at index location ; assumes array address was pushed prior
STOR_FLOAT_ARY_ELM	variable index	stores float at index location; assumes array address was pushed prior

Logical Operators		
Mnemonic	*Opcode(s)*	*Description*
EQL_INT	n/a	pops top two integer values and pushes result of equal operation
NEQL_INT	n/a	pops top two integer values and pushes result of not-equal operation
LES_INT	n/a	pops top two integer values and pushes result of less-than operation
GTR_INT	n/a	pops top two integer values and pushes result of greater-than operation
LES_EQL_INT	n/a	pops top two integer values and pushes result of less-than-equal operation
GTR_EQL_INT	n/a	pops top two integer values and pushes result of greater-than-equal operation
EQL_FLOAT	n/a	pops top two floats values and pushes result of equal operation
NEQL_FLOAT	n/a	pops top two floats values and pushes result of not-equal operation
LES_FLOAT	n/a	pops top two floats values and pushes result of less-than operation
GTR_FLOAT	n/a	pops top two floats values and pushes result of greater-than operation
LES_EQL_FLOAT	n/a	pops top two floats values and pushes result of less-than-equal operation
GTR_EQL_FLOAT	n/a	pops top two floats values and pushes result of greater-than-equal operation
AND_INT	n/a	pops top two integer values and pushes result of add operation
OR_INT	n/a	pops top two integer values and pushes result of or operation

Mathematical Operators		
Mnemonic	*Opcode(s)*	*Description*
ADD_INT	n/a	pops top two integer values and pushes result of add operation
SUB_INT	n/a	pops top two integer values and pushes result of subtract operation
MUL_INT	n/a	pops top two integer values and pushes result of multiply operation
DIV_INT	n/a	pops top two integer values and pushes result of divide operation
SHL_INT	n/a	pops top two floats values and pushes result of shift left operation
SHR_INT	n/a	pops top two floats values and pushes result of shift right operation
MOD_INT	n/a	pops top two integer values and pushes result of modulus operation
ADD_FLOAT	n/a	pops top two floats values and pushes result of greater-than-equal operation
SUB_FLOAT	n/a	pops top two floats values and pushes result of subtract operation
MUL_FLOAT	n/a	pops top two floats values and pushes result of multiply operation
DIV_FLOAT	n/a	pops top two floats values and pushes result of divide operation
I2F	n/a	pop top integer and pushes result of float cast
F2I	n/a	pop top float and pushes result of integer cast

Objects/Methods/Traps		
Mnemonic	*Opcode(s)*	*Description*
RTRN	n/a	exits existing method returning control to callee
MTHD_CALL	integer values for class id and method id	synchronous call to given method releasing control
ASYNC_MTHD_CALL	integer values for class id and method id; pushes new thread id	asynchronous call to given method
ASYNC_JOIN	thread id	waits for identified thread to end execution
LBL	label id	identifies a jump label
JMP	label id and conditional context (1=true, 0=unconditional, -1=false)	jump to label id
NEW_BYTE_ARY	array dimension	pushes address of new byte array
NEW_INT_ARY	array dimension	pushes address of new integer array
NEW_FLOAT_ARY	array dimension	pushes address of new float array
NEW_OBJ_INST	integer value for class id	pushes address of new class instance
OBJ_INST_CAST	integer values for from class and to class	performs runtime class cast check (note: only required for up casting)
TRAP	integer value for trap id	calls runtime subroutine releasing control
TRAP_RTRN	integer value for trap id and number of arguments	calls runtime subroutine releasing control and then process an integer return value