February 20, 2015

Objeck Programming Language

A BREIF INTRODUCTION

RANDY HOLLINES

Contents

Introduction

The Objeck language was designed to be an easy to use strongly typed object-oriented programming language. Back in 2008, when the language was under initial development, existing object-oriented and functional programming languages were examined and their features simplified for inclusion in Objeck.

Major features include:

- Rich and intuitive class libraries
- Support for functional programming
- Concurrency (threads and mutexes)
- Automatic memory management
- Native JIT execution (byte to machine code)
- Unicode with I/O support for UTF-8
- Cross-platform libraries
- Full support for Windows, Linux and OS X

Getting Started

To obtain binaries (or source code) please visit the project website at objeck.org. Interactive installers can be downloaded for Windows and 64-bit Ubuntu. Binary archive files (i.e. *.zip and *.tgz) can be downloaded for all other supported platforms. The distribution consists of a compiler, virtual machine, debugger and library inspection tool. The compiler is named **obc** while the virtual machine (VM) is named **obr**.

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

```
class Hello {
  function : Main(args : String[]) ~ Nil {
    "Hello World"->PrintLine();
    "Καλημέρα κόσμε"->PrintLine();
    "こんにちは 世界"->PrintLine();
  }
}
```

Save the above program in a UTF-8 text file called **hello.obs**. The preceding commands will create an executable named **hello.obe** and run it.

To compile the code type the following:

```
obc -src hello.obs -dest hello.obe
```

To run the program type the following:

```
obr hello.obe
```

To better view the results on Windows try redirecting the output to a text file and opening it in Notepad:

```
obr hello.obe > hello.txt
```

Compiling and Running Code

The Objeck compiler produces two types of binaries. The first type is an executable and the second type is a shared library. Shared libraries can be linked into executables by passing the names of libraries to the compiler. As a naming convention executables end with ***.obe** while shared libraries end with ***.obl**.

Here are a few more examples. The first example compiles and runs a program that processes XML. For this program we link in the collections and XML parsing libraries.

```
obc -src examples/xml_path.obs -lib collect.obl,xml.obl -dest xml_path.obe
obr xml_path.obe
```

The next example compiles and runs program that uses the encryption library.

```
obc -src examples/encryption.obs -lib encrypt.obl -dest encryption.obe
obr encryption.obe
```

To learn more about other libraries please check out the API documentation.

TABLE 1 – COMPILER OPTIONS

Option	Description
-src	path to source files delimited by ‘,’
-lib	path to library files delimited by ‘,’
-tar	target output, options are **exe** for executable and **lib** for library; default is **exe**
-opt	optimization level, **s0** thru **s3** being most aggressive; default is s0
-dest	output filename
-alt	compile code that is written using a C-like syntax
-debug	if set, produces debug out for use by the interactive debugger

The Basics

Let's first look at literals, variables and control flow.

Literals and variables

Literals are defined as they are in most programming languages. In Objeck literals are treated as objects and may have methods associated with them.

```
'\u00BD'->PrintLine();
13->Min(3)->PrintLine();
3.89->Sin()->PrintLine();
"Hello World"->Size()->PrintLine();
```

Here are a few examples of variable declarations and assignments. Variable types can be explicitly defined or implicitly inferred through assignments or casts. If a variable's type is inferred it cannot be redefined later in the program however it can be cast.

```
a : Int;
b : Float := 13.5;
c := 7.25; # type inferred as Float
d := (b * 2)->As(Int); # type inferred as Int
```

TABLE 2 – DATA TYPES

Type	Description
Char	Unicode character value
Char[]	Unicode character array
Bool	Boolean value
Bool[]	Boolean array
Byte	1-byte integer value
Byte[]	1-byte integer array
Int	4-byte integer value
Int[]	4-byte integer array

Float	8-byte decimal value
Float[]	8-byte decimal array
Object	Reference to an abstract datatype
Object[]	Array of abstract datatypes
Function	Functional reference

Comments

Comments may span a single line or multiple lines. In addition, comments for bundles, classes, interfaces and functions/methods may be used to produce code documentation. Please refer to the project website for additional information about generating documentation from your code.

```
flag := false; # single line comment
#~
multiline comment. flag above
may be set to true or false
~#
```

Logic and control flow

As with most languages Objeck supports conditional expressions and control flow logic. One nuance is that conditional statements end with semi-colons.

If/else

An "if/else" statement is a basic control statement.

```
number := Console->ReadLine()->ToInt();
if(number <> 3) {
  "Not equal to 3"->PrintLine();
} else if(number < 13) {
  "Less than 13"->PrintLine();
} else {
  "Some other number"->PrintLine();
};
```

Select

Select statements can be used to efficiently map integer and enum values to blocks of code.

```
select(c) {
  label Color->Red: { "Red"->PrintLine(); }
  label Color->Green: { "Green"->PrintLine(); }
  label Color->Purple: { "Purple"->PrintLine(); }
  other: { "Another color"->PrintLine(); }
};

select(n) {
  label 9:
  label 19: { n->PrintLine(); }
  label 27: { (3 * 9 = n)->PrintLine(); }
```

```
};
```

The language supports for the following looping statements

Do

A "do" loop is a simple pre-test loop.

```
i := 10;
while(i > 0) {
   i->PrintLine();
   i -= 1;
};
```

Do/while

A "do/while" loop is a basic post-test loop.

```
i := 0;
do {
   i->PrintLine();
   i += 1;
} while(i <> 10);
```

For

A "for" loop is a controlled loop with an explicated control expression.

```
location := "East Bay";
for(i := 0; i < location->Size(); i += 1;) {
  location->Get(i)->PrintLine();
};
```

Each

An "each" loop is a controlled loop that iterates though all elements in an array or collection.

```
area_code := Int->New[3];
area_code[0] := 5;
area_code[1] := 1;
area_code[2] := 0;

each(i : values) {
  area_code[i]->PrintLine();
};
```

Operators

There's support for logical, mathematical and bitwise operators. Operator precedence from weakest to strongest is: **logical, [+, -]** and **[*, /, %, <<, >>, and, or, xor]**. Operators of the same precedence are evaluated from left-to-right.

TABLE 3 - LOGICAL

Operator	Description
&	And
|	Or
=	Equal
<>	Not equal and unary not
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

TABLE 4 - MATHEMATICAL

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulus

TABLE 5 – BITWISE

Operator	Description
<<	Shift left
>>	Shift right
and	Bitwise and
or	Bitwise or
xor	Bitwise xor

FIGURE 1 - SIMPLE CREDIT CARD VALIDATION

```
class Luhn {
  function : IsValid(cc : String) ~ Bool {
    isOdd := true; oddSum := 0; evenSum := 0;
    for(i := cc->Size() - 1; i >= 0; i -= 1;) {
      digit : Int := cc->Get(i) - '0';
      if(isOdd) {
        oddSum += digit;
      } else {
        evenSum += digit / 5 + (2 * digit) % 10;
      };
      isOdd := isOdd <> true;
    };
    return (oddSum + evenSum) % 10 = 0;
  }

  function : Main(args : String[]) ~ Nil {
    IsValid("49927398716")->PrintLine();
```

```
    IsValid("49927398717")->PrintLine();
    IsValid("1234567812345678")->PrintLine();
    IsValid("1234567812345670")->PrintLine();
  }
}
```

Code fragment from the Base64 encoding class

```
# Primary encoding loop
r := ""; i : Int; a := 0;
for(i := 0; i < end; i += 3;) {
  a := (data[i] << 16) or (data[i+1] << 8) or (data[i+2]);
  r->Append( lut[0x3F and (a >> 18)] );
  r->Append( lut[0x3F and (a >> 12)] );
  r->Append( lut[0x3F and (a >> 6)] );
  r->Append( lut[0x3F and a] );
};
```

Arrays, strings and collections

The language has support for dynamically allocated arrays, Unicode strings and various containers.

Arrays

Arrays can hold an indexed list of like types. Arrays are dynamically allocated from the heap and their memory managed by the garbage collector. The runtime system supports bounds checking and will cease execution (generating a stack trace) if array bounds are violated.

Allocating and indexing arrays

```
# allocate Int array
boxes := Int->New[2,3];
boxes[0,0] := 2;
boxes[0,1] := 4;
boxes[0,2] := 8;
boxes[1,0] := 1;
boxes[1,1] := 2;
boxes[1,2] := 3;
dims := boxes->Size(); # get the dimensions
dims[0]->PrintLine(); # dimension 1
dims[1]->PrintLine(); # dimension 2

# create some strings an iterate over them
directions := String->New[4];
directions[0] := "North";
directions[1] := "South";
directions[2] := "East";
directions[3] := "West";
each(i : directions) {
  directions[i]->PrintLine();
};
```

Strings

Character strings are a collection of Unicode characters backed by the **String** class. The string class supports a number of operations such as insert, find, substring, type parsing (i.e. **String** to **Float**), etc. Variable values can also be inlined into string literals. Strings can be converted into character arrays and UTF-8 byte arrays.

```
name := "DJ";
name += ' ';
name += "Premier";
name->SubString(2)->PrintLine();
name->Size()->PrintLine();
```

Code fragment from a sundial program

```
"Hour\t\tsun hour angle\t\tdial hour line angle from 6am to 6pm"->PrintLine();
for(h := -6; h <= 6; h+=1;) {
  hra := 15.0 * h;
  hra -= lng - ref;
  hla := (slat* (hra*2*Float->Pi()/360.0)->Tan())
    ->ArcTan() * 360.0 / (2*Float->Pi());
  "HR={$h}\t\tHRA={$hra}\t\tHLA={$hla}"->PrintLine();
};
```

Collections

In addition, to arrays the language supports various collections such as Vectors, Lists and Maps. To learn more about the collections classes please check out the API documentation. In order to use these classes you must reference the collections bundle using the following line of code:

```
use Collection;
```

When compiling a program that uses collections you must link in the required library, for example:

```
obc -src genres.obs -lib collect.obl -dest genres.obe
```

Vectors are arrays that can be dynamically resized. They support fast indexing, iterating and appending of values. In order to improve performance memory for vectors is pre-allocated.

```
genres := Vector->New();
genres->AddBack("Hip hop");
genres->AddBack("Classical");
genres->AddBack("Jazz");
genres->AddBack("Rock");
```

```
genres->AddBack("Folk");
each(i : genres) {
  genres->Get(i)->As(String)->PrintLine();
};
```

Lists are a collection of linear linked nodes. They support the fast insertion and removal of values. Memory for nodes is allocated on-demand however nodes may not be directly indexed as with **Vectors**.

```
artists := List->New();
artists->AddBack("Hendrix");
artists->AddFront("Beck");
# move cursor back for middle insertion
artists->Back();
artists->Insert("Common");
# move cursor to start of list
artists->Rewind();
# iterate over values
while(artists->More()) {
  artists->Get()->As(String)->PrintLine();
  artists->Next();
};
```

Map and **Hash** classes manage key/value pairs. **Maps** manage values in tree and allocate memory on demand. Maps are slower than hashes however manage memory better. **Hashes** use keys as indices into arrays and support fast insertion and deletion at the cost of memory.

```
area_codes := IntMap->New();
area_codes->Insert(510, "Oakland");
area_codes->Insert(415, "San Francisco");
area_codes->Insert(650, "Palo Alto");
area_codes->Insert(408, "San Jose");
area_codes->Find(510)->As(String)->PrintLine();
```

Objects and Functions

As mentioned in the introduction, Objeck supports object-oriented and functional programming concepts. To put class and functions into context the overall programing scope is as follows:

Bundles ➔ Classes ➔ Methods/Functions ➔ Local blocks

Classes and interfaces

As in other languages, a **class** is an abstract collection of data with related operations. An **interface** is a set of operations (a contract) that implementing classes must honor. In Objeck, all classes and interfaces are public. Member variables of classes are protected from the outside world. However, classes that are inherited from the source

of other classes may access their parent's member variables. Outside calling classes must use "getters" and "setters" for which the compiler produces optimized code.

Classes may contain public and private **methods** as well as static public **functions**. An **interface** may define any type of method/function (i.e. **public** or **private**) however it cannot define implementation. Lastly, Objeck supports **reflection** letting programmers dynamically introspect instances at runtime.

Below is a code example of that demonstrates many of these concepts. Please refer to the API documentation to learn more about reflection and other features such as object serialization.

FIGURE 2 – CLASSES, INTERFACES AND REFLECTION

```
interface Registration {
  method : virtual : public : GetColor() ~ String;
  method : virtual : public : GetMake() ~ String;
  method : virtual : public : GetModel() ~ String;
}

enum EngineType {
  Gas := 200,
  Hybrid,
  Electric,
  Warp
}

class Vehicle {
  @wheels : Int;
  @color : String;
  @engine_type : EngineType;

  New(wheels : Int, color : String, engine_type : EngineType) {
    @wheels := wheels;
    @color := color;
    @engine_type := engine_type;
  }

  method : public : GetColor() ~ String {
    return @color;
  }

  method : public : GetEngine() ~ EngineType {
    return @engine_type;
  }
}

class StarShip from Vehicle implements Registration {
  New() {
    Parent(13, "Metal Fuschia", EngineType->Warp);
  }

  method : public : GetMake() ~ String {
    return "Excelsior";
```

```
  }

  method : public : GetModel() ~ String {
    return "NX-2000";
  }

  method : public : EchoDescription() ~ Nil {
    "Partying with the Borg, they brought drinks!"->PrintLine();
  }
}

class Pinto from Vehicle implements Registration {
  New() {
    Parent();
  }

  method : public : GetMake() ~ String {
    return "Ford";
  }

  method : public : GetModel() ~ String {
    return "Pinto";
  }
}

class VehicleTest {
  function : Main(args : String[]) ~ Nil {
    pinto := Pinto->New();
    star_ship := StarShip->New();

    type_of := pinto->TypeOf(Vehicle);
    type_of->PrintLine();

    type_of := star_ship->TypeOf(Vehicle);
    type_of->PrintLine();

    type_of := pinto->TypeOf(StarShip);
    type_of->PrintLine();

    registration := star_ship->As(Registration);
    registration->GetMake()->PrintLine();
    registration->GetColor()->PrintLine();
    enterprise := registration->As(StarShip);
    enterprise->EchoDescription();
  }
}
```

Anonymous classes

Anonymous classes can be created that define "inline" required interface methods/functions. External variables may be referenced within an anonymous class if they're passed as references to the class constructor.

```
interface Greetings {
  method : virtual : public : SayHi() ~ Nil;
}

class Hello {
  function : Main(args : String[]) ~ Nil {
    hey := Base->New() implements Greetings {
      New() {}
      method : public : SayHi() ~ Nil {
        "Hey..."->PrintLine();
      }
    };

    howdy := Base->New() implements Greetings {
      New() {}
      method : public : SayHi() ~ Nil {
        "Howdy!"->PrintLine();
      }
    };
  }
}
```

Serialization

Object serialization is a mechanism that transforms object instances into bytes and vice versa. Serialization can be used to persist an instance's state (i.e. to disk) or pass a copy over a network.

FIGURE 3 – OBJECK SERIALIZATION AND INHERITANCE

```
use Collection;

class Thingy {
  @id : Int;

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

  method : public : Print() ~ Nil {
    @id->PrintLine();
  }
}

class Person from Thingy {
  @name : String;
  @values : StringMap;

  New(id : Int, name : String) {
    Parent(id);
    @name := name;
    @values := StringMap->New();
    @values->Insert("Jason", IntHolder->New(101));
```

```
    @values->Insert("Mark", IntHolder->New(9));
  }

  method : public : Print() ~ Nil {
    @id->PrintLine();
    @name->PrintLine();
    @values->Find("Jason")->As(IntHolder)->Get()->PrintLine();
    @values->Find("Mark")->As(IntHolder)->Get()->PrintLine();
  }
}

class Serial {
  function : Main(args : String[]) ~ Nil {
    t := Thingy->New(7);
    p := Person->New(13, "Bush");

    s := System.IO.Serializer->New();
    s->Write(t);
    s->Write(p);

    writer := IO.File.FileWriter->New("objects.dat");
    writer->WriteBuffer(s->Serialize());
    writer->Close();

    buffer := IO.File.FileReader->ReadBinaryFile("objects.dat");
    d := System.IO.Deserializer->New(buffer);

    t2 := d->ReadObject()->As(Thingy);
    t2->Print();
    p2 := d->ReadObject()->As(Person);
    p2->Print();
  }
}
```

Higher-order functions

Higher order functions allow programmers to pass functions into methods/functions and have methods/functions return functions. The language has support for strongly typed functional references. Due to the compositional nature of this feature functional definitions may be nesting.

FIGURE 4 – FUNCTIONAL COMPOSITION

```
class FofG {
  @f : static : (Int) ~ Int;
  @g : static : (Int) ~ Int;

  function : Main(args : String[]) ~ Nil {
    compose := Composer(F(Int) ~ Int, G(Int) ~ Int);
    compose(13)->PrintLine();
  }

  function : F(a : Int) ~ Int {
    return a + 14;
```

```
  }

  function : G(a : Int) ~ Int {
    return a + 15;
  }

  function : native : Compose(x : Int) ~ Int {
    return @f(@g(x));
  }

  function : Composer(f : (Int) ~ Int, g : (Int) ~ Int) ~ (Int) ~ Int {
    @f := f;
    @g := g;
    return Compose(Int) ~ Int;
  }
}
```

More concretely, functions can be used by collections to perform operations such as applying the result of a given function to all the elements within a vector.

```
function : native : Run() ~ Nil {
  "Print roots..."->PrintLine();
  values := IntVector->New([1, 2, 3, 4, 5, 100]);
  squares := values->Apply(Square(Int) ~ Int);
  each(i : squares) {
    squares->Get(i)->PrintLine();
  };
}

function : Square(value : Int) ~ Int {
  return value * value;
}
```

JIT compiling methods and functions

In order to speed up program execution byte code for method/functions may be JIT compiled into machine code. Methods/functions are compiled the first time they are called and subsequent calls execute the per-compiled machine code.

To direct the runtime to JIT compile code for a given function or method use the **native** keyword. Candidates for JIT compilation are frequently called methods/functions that are computationally expensive or a method/function with lots of long loops.

Observe the execution time of this prime number program with and without the use of the **native** keyword.

FIGURE 5 – PRIME NUMBERS USING JIT COMPILATION

```
class FindPrime {
  function : Main(args : System.String[]) ~ Nil {
    Run(100000);
  }
```

```
  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 += 1;
        };
      };

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

Creating libraries

Creating class libraries is pretty straightforward. Put one or more classes into one or more files and compile the code with the "**-tar lib**" option. Code for a class library cannot contain a "main" function.

```
class Pair {
  @key : Compare;
  @value : Base;

  New(key : Compare, value : Base) {
    @key := key;
    @value := value;
  }

  method : public : GetKey() ~ Compare {
    return @key;
  }

  method : public : Get() ~ Base {
    return @value;
  }
}
```

To compile the code type the following:

```
obc -src pair.obs –tar lib -dest pair.obl
```

To use the library in program type the following:

```
obc -src points.obs –lib pair.obl -dest point.obe
```

Using the Debugger

The command line debugger allows a programmer to inspect the behavior of a program at runtime. In order to use the debugger a program must first be compiled with debug symbols by passing the "-**debug**" option to the compiler.

For a working example let's use the Figure 1 - Simple credit card validation program in the "Operators" section of this document. We'll first save the program to a UTF-8 (or ASCII) text file call **luhn.obs**. The following commands will create a debug executable called **luhn.obe**.

To compile the code type the following:

```
obc -src luhn.obs -dest luhn.obe -debug
```

For this example let's assume the source file is in the same location as the executable. To start the debugger type the following:

```
obd -exe luhn.obe -src .
```

Let's first set a breakpoint on line 17 and run the program.

```
> b luhn.obs:17
added breakpoint: file='luhn.obs:17'
> r
break: file='luhn.obs:17', method='Luhn->Main(..)'
```

Next let's list the code around the breakpoint.

```
> l
     12:   };
     13:   return (oddSum + evenSum) % 10 = 0;
     14: }
     15:
     16: function : Main(args : String[]) ~ Nil {
=>   17:   IsValid("49927398716")->PrintLine();
     18:   IsValid("49927398717")->PrintLine();
     19:   IsValid("1234567812345678")->PrintLine();
     20:   IsValid("1234567812345670")->PrintLine();
     21: }
     22: }
```

Now let's step into the "IsValid" function.

```
> s
> break: file='luhn.obs:2', method='Luhn->IsValid(..)'
> l
      1: class Luhn {
=>    2: function : IsValid(cc : String) ~ Bool {
      3:   isOdd := true; oddSum := 0; evenSum := 0;
```

```
      4:   for(i := cc->Size() - 1; i >= 0; i -= 1;) {
      5:     digit : Int := cc->Get(i) - '0';
      6:     if(isOdd) {
      7:       oddSum += digit;
      8:     } else {
      9:       evenSum += digit / 5 + (2 * digit) % 10
     10:     };
> n
```

Let's print out the value for "cc".

```
> p cc
print: type=System.String, value="49927398716"
```

Now let's break on line 9 and print the value for "evenSum".

```
> b 9
added breakpoint: file='luhn.obs:9'
> c
> break: file='luhn.obs:9', method='Luhn->IsValid(..)'
> n
> break: file='luhn.obs:11', method='Luhn->IsValid(..)'
> p evenSum
print: type=Int, value=2
```

Lastly, let's print out the call stack before exiting.

```
> stack
stack:
  frame: pos=2, class=Luhn, method=IsValid(o.System.String), file=luhn.obs:5
  frame: pos=1, class=Luhn, method=Main(o.System.String*), file=luhn.obs:17
> q
breakpoints cleared.
goodbye.
```

TABLE 6 – DEBUGGER COMMANDS

Command	Description	Example
[**b**]reak	sets a breakpoint	b luhn.obs:17 b
breaks	shows all breakpoints	
[**d**]elete	deletes a breakpoint	d luhn.obs:17
clear	clears all breakpoints	
[**n**]ext	moves to the next line within the same method/function with debug information	
[**s**]tep	moves to the next line with debug information	
[**j**]ump	jumps out of an existing method/function and moves to the next line with debug information	
args	specifies program arguments	args "Hello World"
[**r**]un	runs a loaded program	
[**p**]rint	prints the value of an expression, along with metadata	p cc
[**l**]ist	lists a range of lines in a source file or the lines near	L

	the current breakpoint	
[**i**]nfo	displays the variables for a class	i
stack	displays the method/function call stack	
exe	loads a new executable	exe " luhn.obe "
src	specifies a new source path	src "."
[**q**]uit	exits a given debugging session	