Objeck Programmer's Guide

Randy Hollines
objeck@gmail.com

August 2, 2010

Abstract

Provides an introduction to the Objeck programming language and it's features. This article is intended to introduce programmers and compiler enthusiasts to the unique features and design of the Objeck programming language. Unless otherwise noted, this article covers functionality that is included in release *1.1.0*. For additional information please refer to the general and technical project websites.

Contents

1 Introduction

The Objeck program language is an object-oriented computer language with functional features. The language was designed to be an easy to use general purpose programming system. The Objeck language allows programmers to quickly create solutions by leveraging pre-existing class libraries. 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)
- Functional support (first-class functions)
- Cross platform (support for OS X, Linux and Windows)
- Concurrent runtime JIT support for Intel processors
- Multi-threaded memory management (garbage collector)
- Peephole optimizations
- Support for static libraries
- Command-line debugger

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 Hello {
    function : Main(args : String[]), Nil {
      "Hello World!"->PrintLine();
    }
  }
}
```

2.1 Compiling Source

The example below compiles the source program `hello.obs` into the target binary file `hello.obe`. The two output file types that the compilers supports are executables and shared libraries. Shared libraries are binary files that contain all of the metadata needed by the compiler to relink them into programs. Both executables and shared libraries contain 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`.

Below is an example of compiling the "Hello World" program

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

Additional compiler options are:

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; default is `exe`
`-opt`	optimization level `s0`–`s3` with `s3` being the most aggressive; default is `s0`
`-dest`	output file name
`-debug`	if set, produces debug out for use by the interactive debugger (see below)

2.2 Executing

The command-line example below executes the `hello.obe` executable. Note, for executables all required libraries are statically linked in the target output file. When compiling shared libraries, other shared libraries are not linked into the target output library file.

```
obr hello.obe
```

3 The Basics

Now lets introduce you the core features of the Objeck programming 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

As mentioned above, basic types are objects and have associated methods for each basic class type. For example:

```
13->Min(3)->PrintLine();
13->Max(3)->PrintLine();
-22->Abs()->PrintLine();
Float->Pi()->PrintLine();
```

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 made during a declaration or at any other point in a program. Variables may be declared as local, class instance or class variables. Class level variables are declared using the `static` keyword. A class that is derived from another class may access it's parents variables if the parent class is declared in one of the source programs. *If a class is derived from a class declared in a shared library then that class cannot access it's parents variables, unless an accessor method is provided.* Local variables can be declared without specifying their data type, such variables are bound to a type peceeding their first assignment. Three different declaration styles are shown below:

```
a : Int;
b : Int := 13;
c := 7;
```

Types that are not initialized at declaration time are initialized with 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 call 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 `PrintLine()` method for the literal `42`. The second prints the product of a variable and a literal.

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

The following mathematical operators are supported in the Objeck language for integers and decimal types:

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

In addition the following assignment operators are supported:

- addition-equals (`+=`)
- subtraction-equals (`-=`)
- multiplication-equals (`*=`)
- division-equals (`/=`)

The following bitwise operators are also supported for integer types:

- and (`and`)
- or (`or`)
- xor (`xor`)

The [`*`, `/`, `%`] operators have a higher precedence than the [`+`, `-`] 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 [`&`, `|`] logical operators use short-circuit logic; meaning that some expressions may not be executed if evaluation criteria is not satisfied.

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 supports single and multi-dimensional arrays. Arrays are allocated dynamically from the system 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 array of type `Int` is allocated and dereferenced.

```
array := Int->New[2,3];
array[0,2] := 13;
array[1,0] := 7;
values : Int[,] := [[2,3][4,5]];
values[1,1]->PrintLine();
```

The size of an array can be obtained by calling the array's `Size()` method. The `Size()` method will return the number of elements in a given array. For a multi-dimensional array the size method returns the number of elements in the first dimension. Character array literals are allocated as `String` objects. It should also be noted that language has a `String` class that provides support for advanced string operations.

```
str := "Hello World!";
str->Size()->PrintLine();
strs := ["Hello","World!"];
strs->Size()->PrintLine();
```

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 finer grain logical control. General control statements include `if` and `select` statements. Basic looping statements include `while`, `do/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 exists), otherwise an `else` statement will be executed (if it exists). The example below demonstrates an `if` statement.

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

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"->PrintLine();
   }
```

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

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

3.3.3 While Statement

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

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

3.3.4 Do/While Statement

A `do/while` statement is a control statement that will execute its main body at least once and continue to execute its main body as long as its conditional expression evaluates to `true`. When its conditional expression evaluates to `false` than the loop body will cease to execute.

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

3.3.5 For Statements

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

```
name : Char[] := "John"->ToCharArray();
for(i : Int := 0; i < name->Size(); i := i + 1;) {
  name[i]->PrintLine();
}
```

3.3.6 Each Statements

The `each` statement is a specialized version of a `for` statement. The `each` loop consists of a counter variable and a data structure that has a `Size` method, such as arrays and `Vector` classes. The statement iterates thru all elements in the data structure.

```
values := Int->New[3];
values[0] := 3;
values[1] := 9;
values[2] := 1;

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

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 a 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 for enums are as follows:

- assignment (`:=`)

- equal (`=`)
- not–equal (`<>`)

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 classes using the `from` 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 assures that derived classes have been defined for all required operations declared in the base class. Virtual classes also allow the programmer to define non-virtual methods that contain program behavior. Virtual classes are dynamically bound to implementation classes at runtime.

```
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 from 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)->PrintLine();
  }
}
```

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 language detects upcasting and downcastng at compile time. Upcasting requires a runtime check, while down casting does not. If cross casting is detected then a compile time error will be generated.

```
method : public : Compare(right : Base), Int {
  if(right <> Nil) {
```

```
if(GetClassID() = right->GetClassID()) {
  a : A := right->As(A);

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

The class that a given object instance belongs to can found by calling its `GetClassID` method. This method returns an enum that is associated with that instance's class type. This method is generally used to determine if two object instances are of the same or different classes.

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. Methods 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 supports polymorphic methods and functions, meaning that there can be multiple methods with the same name within the same class as long as their declaration arguments vary.

Methods and functions can either be executed in an interpreted or JIT compiled mode. Interpreted execution mimics microprocessor functions 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 compilation process since it occurs at runtime. In addition, some methods can not be compiled into native machine code but this is a rare case. The keyword `native` is used to JIT compile methods and function at runtime.

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

Below is an example of declaring a virtual method:

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

4.3 First-Class Functions

The Objeck language supports the notion of first-class functions such that a given function may be bound to a variable at runtime. Variables are assigned based upon functional prototypes. Prototypes enforce strong type checking by ensuring that a function parameters and return type are consistent between assignments. Once a variable is bound, it may be assigned to other variables, passed to other functions/methods, returned from other functions/method or dynamically evoked. Please note, methods are not treated as first-class constructs only functions.

4.3.1 Assigning and Passing Functions

The following example shows how a function is defined and assigned to a variable:

```
class Foo {
  function : GetSize(s : String) ~ Int {
      return s->Size();
  }
}
....
s1 : (String) ~ Int := Foo-> GetSize(String) ~ Int;
s2 := Foo-> GetSize(String) ~ Int;
size := EnvokeSize("Hello", s2);
```

4.3.2 Envoking Functions

The following example shows how a function variable is envoked:

```
...
method : public : EnvokeSize(s : String, f : (String) ~ Int) ~ Int {
  return f(s);
}
...
```

5 Debugger

The Objeck compiler toolset contains a simple interactive read-only debugger, which allows programmers to inspect values within their programs. The debugger allows programmers to set breakpoints within methods based upon source line numbers. The debugger can also calculate simple arithmetic expressions involving variables and constants. The following commands are currently supported:

5.1 Starting the Debugger

The source program must be compiled with the `-debug` set. The command line debugger is started up running the `odb` executable. The `-exe` option must be present and specify the path to the executable. The The `-src` option is optional and specifies the path to the program source. Also note, to print instance level variables the path must start with `@self`→.

5.2 Debugging Commands

Command	*Description*	*Example*
`[b]reak`	sets a breakpoint	`b hello.obs:10`
`breaks`	shows all breakpoints	
`[d]elete`	deletes a breakpoint	`d hello.obs:10`
`clear`	clears all breakpoints	
`[n]ext`	moves to the next line with debug information	
`[o]ut`	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 @self→value`
`[l]ist`	lists a range of lines in a source file or the lines near the current breakpoint	`l hello.obs:10`
`[i]nfo`	displays the variables for a class or method/function	`i class=Foo method=New`
`stack`	displays the method/function call stack	
`exe`	loads a new executable	`exe "./../test.obe"`
`src`	specifies a new source path	`src "../../"`
`[q]uit`	exits a given debugging session	

6 Class Libraries

Objeck includes class libraries that provides access to system resources, such as files and sockets, while also providing support for basic data structures like lists and vectors. As new class libraries are added they will be documented in this section.

6.1 Core Libraries

6.1.1 Base

Base class for all objects.

- `GetClassID` - returns the class ID
  - method : public : native : GetClassID(), ClassID
- `GetInstanceID` - returns a unique instance ID
  - method : public : native : GetInstanceID(), Int

6.1.2 Bool

- `Print` - prints the current value
  - method : native : Print(), Nil
- `PrintLine` - prints the current value along with a line return
  - method : native : PrintLine(), Nil
- `ToString` - converts the current value to a `String` object instance
  - method : native : ToString(), String

6.1.3 Char

- `IsDigit` - determines if the character is a digit (in the range of `0-9`)
  - method : native : IsDigit(), Bool
- `IsChar` - determines if the character is a alpha (in the range of `A-Z` or `a-z`)

- method : native : IsChar(), Bool

- `Min` - returns the smallest of the two numbers; returns the same number if they are equal
  - method : native : Min(r : Byte), Byte
- `Max` - returns the largest of the two numbers; returns the same number if they are equal
  - method : native : Max(r : Byte), Byte
- `Print` - prints the current value
  - method : native : Print(), Nil
- `PrintLine` - prints the current value along with a line return
  - method : native : PrintLine(), Nil
- `ToString` - converts the current value to a `String` object instance
  - method : native : ToString(), String

6.1.4 Byte/Int

- `Min` - returns the smallest of the two numbers; returns the same number if they are equal
  - method : native : Min(r : Byte), Byte
- `Max` - returns the largest of the two numbers; returns the same number if they are equal
  - method : native : Max(r : Byte), Byte
- `Abs` - returns the absolute value of the current number
  - method : native : Abs(), Byte
- `Print` - prints the current value
  - method : native : Print(), Nil

- `PrintLine` - prints the current value along with a line return
  - method : native : PrintLine(), Nil
- `ToString` - converts the current value to a `String` object instance
  - method : native : ToString(), String

6.1.5 Float

- `Min` - returns the smallest of the two numbers; returns the same number if they are equal
  - method : native : Min(r : Byte), Float
- `Max` - returns the largest of the two numbers; returns the same number if they are equal
  - method : native : Max(r : Byte), Float
- `Abs` - returns the absolute value of the current number
  - method : native : Abs(), Float
- `Floor` - returns the floor of the current number
  - method : native : Floor(), Float
- `Ceiling` - returns the ceiling of the current number
  - method : native : Ceiling(), Float
- `Sin` - returns the sine of the radian value
  - method : native : Sin(), Float
- `Cos` - returns the cosine of the radian value
  - method : native : Cos(), Float
- `Tan` - returns the tangent of the radian value
  - method : native : Tan(), Float

- `Log` - returns the natural log of the radian value
  - method : native : Long(), Float
- `Pi` - returns the value of Pi
  - function : native : Pi(), Float
- `Print` - prints the current value
  - method : native : Print(), Nil
- `PrintLine` - prints the current value along with a line return
  - method : native : PrintLine(), Nil
- `ToString` - converts the current value to a `String` object instance
  - method : native : ToString(), String

6.1.6 String

- `New`
  - New()
  - New(s : String)
  - New(a : Char[])
  - New(a : Byte[])
- `Append` - Appends a `String`, `Char[]`, `Char`, `Int` or `Float` to the current String instance
  - method : public : native : Append(s : String), Nil
  - method : public : native : Append(c : Char), Nil
  - method : public : native : Append(b : Byte), Nil
  - method : public : native : Append(i : Int), Nil
  - method : public : native : Append(f : Float), Nil
  - method : public : native : Append(a : Char[]), Nil
- `Find` - returns the index of the first occurrence of a given Character

  - method : public : native : Find(c : Char), Int
- `Find` - returns the index of the first occurrence of a given String
  - method : public : native : Find(s : String), Int
- `Size` - returns the size of the String
  - method : public : native : Size(), Int
- `Get` - returns the Character at the given index or -1 if not found
  - method : public : native : Get(i : Int), Char
- `ToCharArray` - converts a string to a `Char[]`
  - method : public : native : ToCharArray(), Char[]
- `ToInt` - converts a string to a `Int`
  - method : public : native : ToInt(), Int
- `ToFloat` - converts a string to a `Float`
  - method : public : native : ToFloat(), Int
- `SubString` - creates a new string that contains a subset of the string's contents
  - method : public : native : SubString(offset : Int), String
  - method : public : native : SubString(offset : Int, length : Int), String
- `Trim` - removes all leading and trailing whitespace
  - method : public : native : Trim(), String
- `StartsWith` - returns true if `String` starts with matching pattern; returns false otherwise
  - method : public : native : StartsWith(), Bool
- `EndsWith` - returns true if `String` ends with matching pattern; returns false otherwise

- method : public : native : EndsWith(), Bool

- `ToUpper` - coverts all lowercase characters to uppercase characters
  - method : public : native : ToUpper(), String
- `ToLower` - coverts all uppercase characters to lowercase characters
  - method : public : native : ToLower(), String
- `Reverse` - returns a string with reversed characters
  - method : public : native : Reverse(), String
- `Equals` - compares two string returns `true` if they are equal
  - method : public : Equals(rhs : String), Bool
- `Compare` - compares two string returns 0 if they are equal
  - method : public : native : Compare(rhs : Compare), Int
- `Print` - prints the current value
  - method : native : Print(), Nil
- `PrintLine` - prints the current value along with a line return
  - method : native : PrintLine(), Nil

6.2 Data Structures

6.2.1 Compare

Base class for all objects that use comparative algorithms.

- `Compare` - compares two object instances; 0 if equal, less-then 0 if less and greater-then 0 if greater.
  - method : virtual : public : Compare(rhs : System.Compare), Int
- `HashID` - returns a unique value for the given class type
  - method : virtual : public : HashID(), Int

6.2.2 List/IntList/FloatList

The List class allow values to be added, removed and deleted from a list. There are two specialized version of this class: `IntList` and `FloatList`. The `IntList` and `FloatList` classes use `Int` and `Float` types respectively instead of `Compare` type. The general class supports the following operations:

- `AddBack` - adds a new value to the back of the list
  - method : public : native : AddBack(value : Compare), Nil
- `RemoveBack` - removes the last element in the list
  - method : public : RemoveBack(), Nil
- `AddFront` - adds a new value to the front of the list
  - method : public : native : AddFront(value : Compare), Nil
- `RemoveFront` - removes the first element in the list
  - method : public : RemoveFront(), Nil
- `Find` - finds an element in the list
  - method : public : Find(value : Compare), Bool
- `Has` - returns true if item is in list
  - method : public : Has(value : Compare), Bool
- `Insert` - inserts a new value in the position pointed to the cursor
  - method : public : Insert(value : Compare), Bool
- `Remove` - removes the last element in the list
  - method : public : Remove(), Nil
- `Insert` - inserts an element into the current cursor position
  - method : public : Insert(value : Compare), Bool
- `Next` - advances the internal cursor by one element

- method : public : Next(), Nil

- `Pervious` - retreats the internal cursor by one element
  - method : public : Pervious(), Nil
- `Get` - returns the value of the element pointed to by the cursor
  - method : public : Get(), Compare
- `Forward` - moves the cursor to the end of the list
  - method : public : Forward(), Nil
- `Rewind` - moves the cursor to the start of the list
  - method : public : Rewind(), Nil
- `IsStart` - returns true if cursor is at the start of the list
  - method : public : IsStart(), Bool
- `IsEnd` - returns true if cursor is at the end of the list
  - method : public : IsEnd(), Bool
- `Size` - returns the size of the list

6.2.3 Stack/IntStack/FloatStack

The Stack class support the concept of a growing stack. There are two specialized version of this class: `IntStack` and `FloatStack`. The `IntStack` and `FloatStack` classes use `Int` and `Float` types respectively instead of `Base` type. The general class supports the following operations:

- `Push` - pushes a new value onto the stack
  - method : public : Push(value : Base), Nil
- `Pop` - pops a values from the top of the stack
  - method : public : Pop(), Base
- `Top` - retrieves the top value from the stack (without popping the stack)
  - method : public : Top(), Base

6.2.4 Vector/CompareVector/IntVector/FloatVector

The Vector class support the concept of a growing array. There are three specialized version of this class: `CompareVector`, `IntVector` and `FloatVector`. The `IntVector`, `FloatVector` and `CompareVector` classes use `Int`, `Float` and `Compare` types respectively instead of `Base` type. The general class supports the following operations:

- `AddBack` - adds a new value to the back of the vector
  - method : public : AddBack(value : Base), Nil
- `RemoveBack` - removes the last element in the vector
  - method : public : RemoveBack(), Nil
- `Get` - returns the value of the element pointed to by the cursor
  - method : public : Get(index : Int), Base
- `Set` - replaces the list value based upon the given index
  - method : public : Set(value : Base, index : Int), Bool
- `Size` - returns the size of the list
  - method : public : Size(), Int
- `ToArray` - returns an array of elements
  - method : public : ToArray(), Base

  Additional methods for CompareVector/IntVector/FloatVector

- `Sort` - sorts a vector of values using a merge sort algorithm in $O(n \log_2 n)$ time.
  - method : public : Sort(), Nil
- `BinarySearch` - performs a binary search for an element in a *sorted* vector using an iterative binary search algorithm in $O(\log_2 n)$ time.
  - method : public : BinarySearch(v : Compare), Nil

Additional methods for IntVector/FloatVector

- `Min` - returns the smallest value in the vector
  - method : public : native : Min(), Int/Float
- `Max` - returns the largest value in the vector
  - method : public : native : Max(), Int/Float
- `Average` - returns the calculated average for all values in the list
  - method : public : native : Average(), Int/Float

6.2.5 Map/IntMap/FloatMap/StringMap

The Map class supports the concept of an associative array with key/value pairs. The class implements a balance binary tree algorithm such that inserts, deletes and searches are $O(\log_2 n)$.

- `Insert` - adds a new value to the tree
  - method : public : Insert(key : Compare, value : Base), Nil
- `Remove` - removes a value from the tree
  - method : public : Remove(key : Compare), Nil
- `Find` - searches for a value based upon a key
  - method : public : Find(key : Compare), Base
- `Has` - returns true if item is in map
  - method : public : Has(value : Compare), Bool
- `GetKeys` - returns a vector of keys
  - method : public : GetKeys(), Vector
- `Gets` - returns a vector of values
  - method : public : Gets(), Vector

6.2.6 Hash/StringHash

The Hash class supports the concept of an associative array with key/value pairs. The class implements a hashing algorithm that is optimized for pairs of 256 or less (consider using the `Map` class for larger sets):

- `Insert` - adds a new value to the hash table
  - method : public : Insert(key : Compare, value : Base), Nil
- `Remove` - removes a value from the hash table
  - method : public : Remove(key : Compare), Nil
- `Find` - searches for a value based upon a key
  - method : public : Find(key : Compare), Base
- `Has` - returns true if item is in hash
  - method : public : Has(value : Compare), Bool
- `GetKeys` - returns a vector of keys
  - method : public : GetKeys(), Vector
- `Gets` - returns a vector of values
  - method : public : Gets(), Vector

.

6.3 System Libraries

6.3.1 Console

The Console class allows programmers to read and write information to the system console. The class supports the following operations:

- `GetInstance` - returns the console instance
  - function : GetInstance(), Console

- `Print` - prints all basic types including `String` and `Char[]` to standard out.
- `PrintLine` - prints all basic types including `String` and `Char[]` to standard out followed by a newline.
- `ReadString` - reads in a line of text as a `Char[]` from standard in.
  - method : public : ReadString(), String

6.3.2 Time

The Time class allows programmers gain access to the current system time. The class supports the following operations:

- `New`
  - New()
- `GetDay` - return the current day as an `Int`.
  - method : public : GetDay(), Int
- `GetMonth` - return the current month as an `Int`.
  - method : public : GetMonth(), Int
- `GetYear` - return the current year as an `Int`.
  - method : public : GetYear(), Int
- `GetHours` - return the current hour as an `Int`.
  - method : public : GetHours(), Int
- `GetMinutes` - return the current minutes as an `Int`.
  - method : public : GetMinutes(), Int
- `GetSeconds` - return the current seconds as an `Int`.
  - method : public : GetSeconds(), Int
- `IsSavingsTime` - return true if daylights saving time, false otherwise
  - method : public : IsSavingsTime(), Bool

6.3.3 File

The File class allows programmers manipulate system files. The class supports the following operations:

- `New`
  - New(name : String)
- `IsOpen` - returns true if file is open.
  - method : public : IsOpen(), Bool
- `IsEOF` - returns true if the file pointer is at the EOF.
  - method : public : IsEOF(), Bool
- `Seek` - seeks to a position in a file.
  - method : public : Seek(p : Int), Bool
- `Rewind` - moves the file pointer to the beginning of a file.
  - method : public : Rewind(), Nil
- `Size` - returns the size of the file.
  - function : Size(name : String), Int
- `Remove` - deletes a file.
  - function : Remove(n : String), Bool
- `Exists` - returns true if the file exists.
  - function : Exists(n : String), Bool
- `Rename` - renames a file
  - function : Rename(from : String, to : String), Bool

6.3.4 FileReader

The FileReader is inherited from the File class and allows programmers read files. The class supports the following operations:

- `New`
  - New(name : String)
- `Close` - closes a file.
  - method : public : Close(), Nil
- `ReadByte` - reads a byte from a file.
  - method : public : ReadByte(), Byte
- `ReadBuffer` - reads n number of bytes from a file.
  - method : public : ReadBuffer(offset : Int, num : Int, buffer : Byte[]), Int
- `ReadString` - reads a line from a file.
  - method : public : ReadString(), String

6.3.5 FileWriter

The FileReader is inherited from the File class and allows programmers read files. The class supports the following operations:

- `Close` - closes a file.
  - method : public : Close(), Nil
- `WriteByte` - writes a byte to a file.
  - method : public : WriteByte(b : Int), Bool
- `WriteBuffer` - writes n number of bytes to a file.
  - method : public : WriteBuffer(offset : Int, num : Int, buffer : Byte[]), Int
- `WriteString` - writes a string to a file.
  - method : public : WriteString(s : String), Nil

6.3.6 Directory

The Directory class allows programmers manipulate file system directories. The class supports the following operations:

- `Create` - creates a new directory.
  - function : Create(n : String), Bool
- `Exists` - returns true if the directory exists.
  - function : Exists(n : String), Bool
- `List` - returns vector of file and directory names.
  - function : List(n : String), String[]

6.3.7 TCPSocket

The TCPSocket class allows programmers connect to TCP/IP socket servers. The class supports the following operations:

- `New`
  - New(address : String, port : Int)
- `IsOpen` - returns true if the socket is connected.
  - method : public : IsOpen(), Bool
- `Close` - closes a connected socket.
  - method : public : Close(), Nil
- `WriteByte` - writes a byte to a file.
  - method : public : WriteByte(b : Int), Bool
- `WriteBuffer` - writes n number of bytes to a file.
  - method : public : WriteBuffer(offset : Int, num : Int, buffer : Byte[]), Int
- `WriteString` - writes a string to a file.

  - method : public : WriteString(s : String), Nil

- `ReadByte` - reads a byte from a file.
  - method : public : ReadByte(), Byte
- `ReadBuffer` - reads n number of bytes from a file.
  - method : public : ReadBuffer(offset : Int, num : Int, buffer : Byte[]), Int
- `ReadString` - reads a line from a file.
  - method : public : ReadString(), String

6.3.8 HttpClient

The HttpClient class allows programmers access HTTP 1.1 websites:

- `New`
  - New(url : String, port : Int)
- `New`
  - New(url : String, content_type: String, port : Int)
- `Post` - Performs a HTTP POST to the connected website, returns true is successful
  - method : public : Post(content : String), Bool
- `Get` - Performs a HTTP GET from the connected website, returns a `Vector` of strings
  - method : public : Get(), Vector
- `GetHeaders` - Returns a `Map` of headers information. HTTP header information is populated after a HTTP GET request.
  - method : GetHeaders(), Map

7 Examples

7.1 Prime Numbers

```
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->PrintLine();
        };
        candidate := candidate + 1;
      };
    }
  }
}
```

7.2 Simple HTTP client

```
use Net;
use IO;
use Structure;

bundle Default {
  class HttpTest {
    function : Main(args : String[]), Nil {
      if(args->Size() = 1) {
        client := HttpClient->New(args[0], 80);
        lines := client->Get();

        for(i := 0; i < lines->Size(); i := i + 1;) {
          lines->Get(i)->As(String)->PrintLine();
        };
      };
    }
  }
}
```

8 Appendix A: Example Debugging Session

8.1 Sample Source

```
bundle Default {
  class Bar {
    v1 : Float;
    v2 : Int;

    New() {
      v1 := 2.31;
      v2 := 26;
    }
  }

  class Foo {
    bar : Bar;
    value : Int;

    New(v : Int) {
      value := v;z
    }

    method : public : Get(), Int {
      return value;
    }

    method : public : SetBar(), Nil {
      bar := Bar->New();
    }
  }

  class Test {
    function : Main(args : System.String[]), Nil {
      d : Float := 11.12;
      z := Int->New[5,6];
      z[2,3] := 27;
```

```
      f := Foo->New(24);
      f->SetBar();
      v := f->Get();
    }
  }
}
```

The sample file is named `debug.obs`.

8.2 Compiling the Source and Starting the Debugger

```
obc -src test_src\debug.obs -dest a.obe -debug
obd -exe ..\..\compiler\a.obe -src ..\..\compiler\test_src

-------------------------------------
Objeck v1.1.0 - Interactive Debugger
-------------------------------------
loaded executable: file='../../compiler/a.obe'
source files: path='../../compiler/test_src/'
```

8.3 Setting a Breakpoint and Running the Program

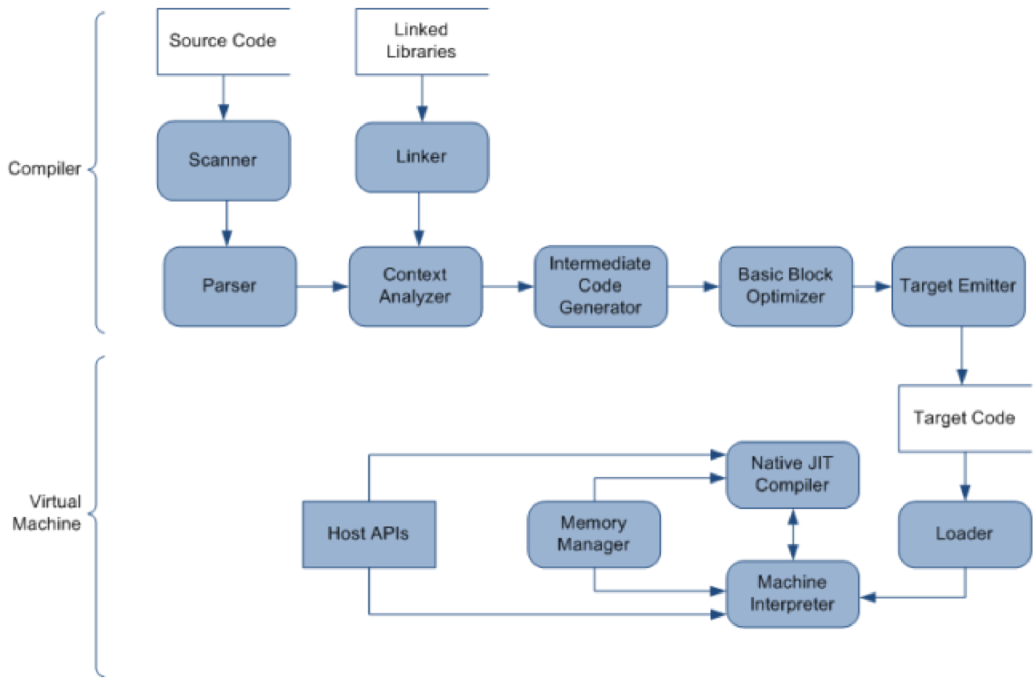
```
> b debug.obs:31
added break point: file='debug.obs:31'
> r
break: file='debug.obs:31', method='Test->Main(..)'
> l
  List
     26:  }
     27:  }
     28:
     29:  class Test {
     30:  function : Main(args : System.String[]), Nil {
=>   31:  d : Float := 11.12;
     32:  z := Int->New[5,6];
     33:  z[2,3] := 27;
     34:
     35:  f := Foo->New(24);
```

```
     36:  f->SetBar();
> n
break: file='debug2.obs:32', method='Test->Main(..)'
```

8.4 Printing a Value

```
> p d
print: type=Float, value=11.12
> b debug.obs:37
added break point: file='debug.obs:37'
> c
break: file='debug2.obs:37', method='Test->Main(..)'
> p z
print: type=Int[], value=2197556(0x218834), dimension=2, size=30
> p z[2,3]
print: type=Int[], value=27(0x1b)
>  p f->value
print: type=Int, value=24
> p f->bar
print: type=Bar, value=0x218864
> p f->bar->v1
print: type=Float, value=2.31
> q
> p f->bar->v1 * 3.5
print: type=Float, value=8.085
goodbye.
```

9 Appendix B: Compiler and VM Design



The following section gives a brief overview of the major architectural components the comprise the Objeck language compiler and virtual machine.

9.1 Compiler

The language compiler is written in C++ and makes heavy use of the C++ STL for portability across platforms. As mentioned in the introduction, the compiler accepts source files and shared libraries as inputs and produces either executables or shared libraries. Note, the compiler has two modes of operation: `User Mode` compiles traditional end-user programs, while `System Mode` compiles system libraries and processes special system language directives.

9.1.1 Scanner and Parser

The scanner component reads source files and parses the text into tokens. The scanner works in conjugation with the *LL(k)* parser by providing k lookahead tokens for parsing. Note, the scanner can only scan system language directives while in `System Mode`. The source parser is a recursive-decent parser that generates an abstract parser tree, which is passed to the Contextual Analyser for validation.

9.1.2 Contextual Analyser

The Contextual Analyser is responsible for ensuring that a source program is valid. In addition, the context analyser also creates relationships between contextually resolved entities (i.e. methods $\longleftrightarrow$ method calls). The analyser accepts an abstract parser tree and shared libraries as input and produces a decorated parse tree as output. The decorated parse tree is then passed to the Intermediate Code Generator for the production of VM instructions.

9.1.3 Intermediate Code Generator and Optimzier

The Intermediate Code Generator accpets a decorated parse and produces a flat list of VM stack instructions. These instruction lists are then passed to the Optimizer for basic block optimizations (constant folding, strength reduction, instruction simplification and method inlining).

9.1.4 Target Emitter

Finally, the improved intermediate code is passed to code emitter component, which writes it to a file.

9.2 Virtual Machine

The language VM is written in C/C++ and was designed to be highly portable. The VM makes heavy use of operating system specific APIs (i.e. WIN32 and POSIX) but does so in an abstracted manner. The JIT compiler is targeted to produce machine code for the IA-32 and AMD64 (future) hardware architectures.

9.2.1 Loader

The loader component allows the VM to read target code structures such as classes, methods and VM instructions. The loader create an in-memory representation of this information, which is used by the VM interpreter and

JIT compiler. In addition, the loader processes command-line parameters that are passed into the VM prior to execution.

9.2.2 Interpreter

The Interpreter executes stack based VM instructions (listed below) and manages two primary stacks: the execution stack and call stack. The execution stack is used to manage the data that is needed for VM calculations. The call stack is used to manage function/method calls and the states between those calls.

9.2.3 JIT Compiler

The JIT compiler translates stack based VM instruction into processor specific machine code (i.e. IA-32). The JIT compiler is evoked by the interpreter and methods are translated in a separate execution thread. This process allow methods to be executed concurrently in an interpreted manner while they are being compiled into machine code. Note, methods are only converted into machine code once.

9.2.4 Memory Manager

The Memory Manager component allows the runtime system to manage the user allocation/deallocation of heap memory. The memory mangers implementes a multi-thread "mark and sweep" algorithm. The marking stage of the process is multi-thread, such that, each root in scanned in a separate thread. The sweeping stage is done in a single thread since the data structures that are needed to manage the state of the running program are modified.

10 Appendix C: VM Instruction Set

The appendix below lists the types of stack instructions that are executed by the Objeck VM. The VM was designed to be 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_FUNC_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
STOR_FUNC_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

Logical Operators		
Mnemonic	*Opcode(s)*	*Description*
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*
SWAP_INT	n/a	swaps the top two integer values on the stack
POP_INT	n/a	control pop of an integer from the stack
POP_FLOAT	n/a	control pop of a float from the stack
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
DYN_MTHD_CALL	pops integer values for class id and method id	dynamic 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)

Objects/Methods/Traps		
Mnemonic	*Opcode(s)*	*Description*
THREAD_CREATE	n/a	creates an new thread instance (calculation stack and stack pointer)
THREAD_WAIT	n/a	waits for worker threads to stop execution
CRITICAL_START	n/a	creates a mutex such that only one thread can execute in a given section
CRITICAL_END	n/a	releases a system mutex
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
LIB_NEW_OBJ_INST	n/a	symbolic library link for a new object instance
LIB_MTHD_CALL	n/a	symbolic library link for a method call
LIB_OBJ_INST_CAST	n/a	symbolic library link for an object cast