JavaBASIC

A variant of BASIC compiled to Java bytecode, with a bundled games runtime based on XGameStation's XGS BASIC.

Background

I created JavaBASIC for two reasons. First, to demonstrate my Java bytecode library, which to date has mostly been used in enterprise database automation projects (O/R Mapping). Second, because I seem to have a soft spot for compilers and especially for BASIC.

To spice things up I borrowed some concepts from XGameStation.com, creating a small runtime library for making simple games. It is not meant to be powerful, but it is meant to be fun, and maybe a little "retro". So instead of "Hello World", the demo apps (from XGameStation) are Pong, Tetris, and 3D Spinning Blocks.

Consider it experimental at every level, including syntax.

- Jason Sando

Clarity Systems Group, LLC

Getting Started

1. If you're reading this document then you've probably already downloaded the zip file. If so head to step 2. If not, download the JavaBASIC-x.x.zip distribution, and unzip it.
2. Make sure you have Java 1.4 or newer installed. If you don't or you're not certain, head over to http://www.java.com, and click on the "Get It Now" link on the right. You don't need a Java SDK for JavaBASIC, just a runtime (a JRE). I'm testing on Java 2 v1.4.2_04 right now.
3. Open a command window, and change to the JavaBASIC home directory. If you unzipped to your C:\ drive, this folder might be "c:\JavaBASIC-0.1".
    a. cd \JavaBASIC-0.1
4. Compile and run one of the demos, such as pong.jbas or remtris.jbas
    a. java –jar JavaBASIC.jar etc/remtris.jbas
    b. java –jar etc/remtris.jar (runs the tetris clone, press ESCAPE to quit)
5. Take a look at the demo programs such as etc/remtris.jbas using a text editor (I use TextPad).
6. If you want to play in full-screen mode try this, but BEWARE I DON'T THINK IT'S VERY STABLE!
    a. java –Djavabasic.fullscreen=true –jar etc/remtris.jar

Compiling

The JavaBASIC compiler and runtime is self-contained within the Jar file "JavaBASIC.jar". By default, when you compile your program it will output a new executable JAR file with all required runtime classes in it. This allows you to have a self-contained executable (minus the Java runtime!), which makes it easy to email or post on a website.

To launch the compiler, open a command prompt, and type:

```
java –jar JavaBASIC.jar -?
```

If Java is installed correctly, you should see the following help message:

```
java -jar JavaBASIC.jar [options] target

  Options:
  -c                Compile to classfile, not Jar file
  -p [<file>]       Disassemble bytecode
  -d <dir>          Place output files in <dir>
  -l <path>         Append resource at <path> into Jar file
  -?                Print this message
```

Compiler Options

-c Write output to a ".class" file only, instead of building an executable JAR file. You might use this option if you are debugging a large program with lots of external resources such as images, and on a slow machine perhaps it takes too long to bundle everything into the JAR file. This allows you to manually launch Java with a specified classpath, with just the .class file.

-p [<file>]

Write program disassembly. If no file is specified it will write it to <target>.jasm. This should be the last option specified, otherwise if <file> is omitted the next argument will be taken as the output file. The disassembly is in CSG-Bytecode format, which is quite readable compared to Sun's standard format.

-d <dir>

Write output file(s) to <dir>.

-l <path>

Append resource at <path> into the JAR file. This provides a mechanism to bundle sound, image, and other data files into the JAR file. Note that the resources must be reachable from the Java classloader, and will be written into the JAR file with the same path.

Example

In the simplest form you can give it the source file to compile, and it will produce an executable JAR file for you, in the same folder as your source file. To build the examples, such as pong, do the following:

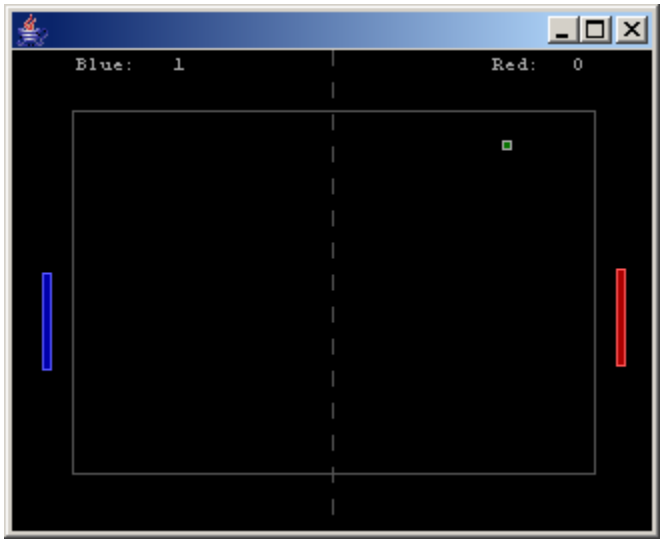
```
java –jar JavaBASIC.jar etc/pong.jbas

Generated etc\pong.jar, use java –jar etc\pong.jar to run.
```

Note the second line, which is the output from the compiler. You can run the compiled program as it says, or in Windows you should be able to double-click the etc\pong.jar file as well. We'll run ours from the command line, since we're there already:

```
java –jar etc\pong.jar
```

You should see the Pong game running, like this:



To exit the Pong game press ESCAPE.

Runtime Options

When you run your program you can also pass some additional options to the Java runtime. To pass these options use "-Doption=value" when launching your program, such as:

```
java –Djavabasic.fullscreen=true –jar etc\pong.jar
```

Option	Usage
javabasic.fullscreen	true | false Signifies to go into full-screen exclusive mode. The default is "false", and makes it run in a 320x240 window.
javabasic.scale	true | false Signifies to scale the off-screen buffer when writing to the main window, based on the current window size. This allows you to stretch the window larger than 320x240. It definitely degrades performance on slower machines. The default is "false".

JavaBASIC Language

Compilation, Execution

JavaBASIC is compiled into Java bytecode and executed within a Java Virtual Machine (JVM). The Java type system is used directly. The JVM data types are used directly, so an *int* in JavaBASIC is compiled into an *int* reference in bytecode.

A JavaBASIC program is compiled into a single class, having the same name as the JavaBASIC source file. Any statements that appear outside of a function declaration will be appended to the *main* method of the Java class. Any variable declarations outside a function declaration (i.e., "global" variables) will be generated as *public static* fields in the Java class. Variables declared within a function are created as local variables.

A key difference between most BASIC's and this one is the lack of "built-in" functions. JavaBASIC follows my preference for not polluting the namespace with built-in functions (such as MID$, LEN, OPEN, etc). Instead library routines are used to organize the built-in functions. Also there is no option to omit function parenthesis in an expression statement.

Following is a short JavaBASIC program, and the resulting (disassembly of the) class file generated.

Sample "etc/hello.jbas"

```
' JavaBASIC "Hello World"
for i = 1 to 10: HelloWorld (): next

function void HelloWorld ()
    System.Print ("Hello, World!")
    System.PrintLn ()
end
```

Output from running "etc/hello.jbas":

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

Disassembly of generated classfile:

```
.source "etc/hello.jbas"
import com/claritysys/javabasic/runtime/System

public class hello extends Object {

       public static void main (String[] local_0) {

              int i
       .line 2:

              iconst_1
              istore i

       L0002:
              iload  i
              bipush 10
              if_icmpgt    L0011

       .line 2:
              invokestatic  HelloWorld()void
              iinc   i += 1
              goto   L0002

       L0011:
              nop
              iconst_m1
              invokestatic  Display$SetMode(int)void
              iconst_0
              invokestatic  System$exit(int)void
              return
       }

       public static void HelloWorld () {

       .line 5:
              ldc    "Hello, World!"
              invokestatic  com/claritysys/javabasic/runtime/System$Print(String)void

       .line 6:
              invokestatic  com/claritysys/javabasic/runtime/System$PrintLn()void
              return
       }
}
```

Data Types

JavaBASIC has only *int, long,* arrays of each of these, and string constants. There are currently no string operations (such as concatenation). There is no *boolean* type, instead integers are used for relational operations, and zero signifies false, non-zero is true. This is consistent with the JVM, as well as many other versions of BASIC.

There is no support for compound types, *new*, or even array assignment (to another array). Arrays cannot be passed as parameters, nor can they be returned from functions (I just noticed this while writing the formal grammar!).

A JavaBASIC program can reference *public static* fields and methods in other (compiled) JavaBASIC programs and other Java classes. The class must be accessible by the

classloader of the compiler *at compile time* so that it can use Java reflection to verify the reference and generate the proper bytecode. Of course the classes must also be accessible at *runtime*, in order to execute. The JavaBASIC runtime API is located in package *com.claritysys.javabasic.runtime*. Any classes in this package are automatically "imported". There is currently no means to import any other package; therefore at this moment only classes in the *default* package can be accessed. It should be easy to add some type of "import" statement, similar to Java's.

Because JavaBASIC has only 3 data types (int, long, and String literal), any referenced Java fields or methods can only use these types; however, the Java types *boolean, char,* and *short* are all compiled into *int* references, so JavaBASIC can still reference them.

Statements, Comments

A comment starts with an apostrophe and continues to the end of that line. Any text within a comment is ignored.

Each statement in JavaBASIC is terminated by either an end-of-line or colon (":"). The end-of-line allows both Unix and DOS encoding, i.e. either LF or CRLF. The ":" allows multiple statements to be put on the same line.

The underscore ("_") can be used as a *line-continuation* character if it appears at the end of a line by itself. This allows you to break a long statement, such as a large IF expression or an array initializer, into multiple lines to make it easier to read and maintain.

Currently the language is ***case-sensitive***, so all keywords must be lowercase, and all variable and function names must match exactly.

Examples:

```
' Multiple statements on one line
int a = 3, b = 2: if a = 3 then a = 10: b = 3

' Line continuation
int DATA[20] = [ _ ' The underscore means "continue on the next line"
      1,   2,  3,  4,  5,  6,  7,  8,  9, 10, _
      11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

Literals

Literal values can be written as integers, such as "250". If needed you can also list integer literals in hexadecimal by preceding them with a dollar sign. So instead of "250", you would write "$FA" or "$fa".

JavaBASIC also allows string literals within double quotes. Unfortunately no string operations currently exist, and there is no string data type, so string literals can only be used to pass to Java methods in the runtime library (i.e., for printing predefined messages on the screen).

Examples:

```
const int SCREEN_WIDTH = 320              ‘ 320 is an integer literal
const int JAVA_MAGIC = $cafebabe          ‘ $CAFEBABE is also an integer literal
System.Print (“The magic number is: “)    ‘ String literal passed to function
System.Print (JAVA_MAGIC)
System.PrintLn ()
```

Variables

Variables *must be declared before you can use them*. You will get a compiler error if you fail to declare a variable before using it.

Declarations follow more of a C syntax than traditional BASIC: the type comes first. Like in Java, you can declare variables anywhere in a block, you don’t have to declare them all at the top.

Arrays are supported, with up to 10 dimensions. You can use a static initializer with a single dimension array (see example below). You can use this to include large data sets in your program, because there is no DATA/READ/RESTORE command. Current arrays cannot be assigned directly, e.g. as a reference to another array. Also arrays cannot be passed as parameters to functions or returned as values from functions.

Examples:

```
‘ Declaration allows assignment at the same time
int y = $ff, z = -40000, a = 42

‘ Long var is 64 bits (its a Java long)
long loop_time = System.Time()

‘ Arrays use BASIC syntax but brackets like in C or Java
int sprite_x[4], map[20, 20]

‘ Single dimension arrays can have a static initializer.
‘ Use this instead of the DATA/READ/RESTORE commands.
int POSITIONS[10] = [ $ff, $00, $fc, $ee, $00, $bb, $cc, $99, $9a, $42 ]

‘ You can declare variables anywhere
for i = 1 to 10
      int b = i * 25
      ‘ do something with b ...
next
```

If/Then/ElseIf/Else

You can use either the single line or the block format for IF/THEN statements. The expression is evaluated and the THEN statements are executed if the result was non-zero.

Note that the block format is concluded with **end**, not “end if” as is traditional.

Examples:

```
‘ This single line IF/ELSE
if a = 3 then bx = bx + 4 else bx = bx – 4
```

```
‘ ... can also be written as a block statement.
if a = 3 then
      bx = bx + 4
else
      bx = bx – 4
end

‘ You can also have multiple elseif clauses.
if a = 3 then
      bx = bx + 4
elseif a = 5 then
      bx = bx + 25
elseif a = 7 then
      bx = bx + 35
else
      bx = bx – 4
end
```

For/Next

The FOR/NEXT statement builds a loop with an increment variable. Note that unlike most BASICs, the loop variable is not named in the NEXT statement; that is, you don't write "next i", you just write "next". This is because if multiple for/next loops are nested, each "next" statement is bound to the closest "for" statement, so the variable name is superfluous.

(I am undecided whether to continue to use "next", or to be consistent with all statement blocks and use "end". I think using "do/loop", "for/next", and "if/end if" makes it more readable.)

JavaBASIC does support the **break** keyword, which allows you to cancel the nearest enclosing loop. It currently does not support a break label (like Java does).

Example:

```
int found_x = 0
for x = 1 to 10   ‘ default step of 1 is used
      for y = 10 to 1 step –1       ‘ if we forget the “step –1” it won’t work!
            if map[y, x] = 42 then
                  found_x = x
                  break             ‘ breaks out of nearest loop
            end
      next

      if found_x = 1 then break
next
```

Do While/Until

Currently the WHILE/UNTIL, which is optional, can only appear at the top of the loop.

The "do" must have a matching "end" statement. This may be changed to "loop" in the future.

The **break** keyword can be used to break out of the nearest enclosing loop.

Example:

```
do while x < 10
      x = x – 1
end

‘ Without a condition, using break inside the loop
do
      x = x – 1
      if x >= 10 then break
end
```

Return

The **return** keyword allows you to return from a function, optionally returning a value (if the function is declared to return a value).

Example:

```
function int Max (int a, int b)
      if a > b then return a else return b
end
```

Expressions

The following table lists the operators and their usage.

Operator	Usage
+	Addition
-	Subtraction
*****	Multiplication
/	Division
and	Bitwise and. Relational expressions always evaluate to zero or one, so subsequent bitwise operators work as logical operators.
or	Bitwise or.
xor	Bitwise exclusive-or
not	Bitwise not.
mod	Modulus function, eg “x mod 3”

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
=	Equals
<>	Not equals
()	Subexpression

Functions

Functions can be declared anywhere. You do not need to declare a function before calling it (ie, forward references are allowed).

Arrays cannot be passed to or returned from functions, this is a current limition.

Example:

```
SayHi ()
System.Print (Max (50, 100))

function void SayHi ()
      System.Print (“Hi! The biggest number in this program is: ”)
end

function int Max (int a, int b)
      if a > b then return a else return b
end
```

JavaBASIC Syntax Specification

Keywords, Symbols, and Grammar Rules

Bold normal font indicates a keyword. Italics indicate a rule of the grammar.

When brackets or parenthesis occur in the syntax literally they are both bold and in quotations (see *ArrayDimensions*).

Alternatives

The bar “|” indicates alternatives. When you see a bar, you must choose one of the items from the list.

a | b |c

means “a” *or* “b” *or* “c”.

Grouping

Parenthesis are used to group alternatives. Given a rule such as

(This | That) Way

The parentheses are indicating that you must choose one item from the list, either “This” or “That”. The rule given can be expanded as “This Way”, or “That Way”.

Multiplicity

Brackets are used to indicate zero-or-one occurrences, e.g. an *optional* item.

[Go] This Way!

This can be read “Go This Way!”, or simply “This Way!” since the “Go” is optional.

An asterisk indicates zero-or-more occurances.

[Go]* This Way!

This can be read as:

“This Way!”

“Go This Way!”

“Go Go Go Go This Way!”

A plus symbol indicates one-or-more occurrences.

Formal Grammar Specification

```
Program:    [FunctionDeclaration | Statement]*

FunctionDeclaration:
            function TypeSpecifier Name “(“ [ParamList] “)”
                  [Statement]*
            end

TypeSpecifier:
            int | long | void

Name:       Letter | Underscore [Letter | Underscore | Digit]*

ParamList:  Param [, Param]*

Param:      TypeSpecifier Name

Statement:  DeclarationStatement |
            ExpressionStatement |
            ForStatement |
            DoStatement |
            IfStatement |
            ReturnStatement |
            BreakStatement
```

```
DeclarationStatement:
             [static] [const] TypeSpecifier Declaration [, Declaration]*

Declaration:
            Name [ArrayDimensions] [Initializer | ArrayInitializer]

ArrayDimensions:
            "[" ExpressionList "]"

Initializer:
            = Expression

ArrayInitializer:
            = [ ExpressionList ]

ReturnStatement:
            return [Expression]

BreakStatement:
            break

IfStatement:
            if Expression then Statement+ [ElseIfClause]* [ElseClause]
            if Expression then
                  [Statement]*
            [ElseIfClause]*
            [ElseClause]
            end

ForStatement:
            for Name = Expression to Expression [step Expression]
                  [Statement]*
            next

DoStatement:
            do [ (while | until) Expression ]
                  [Statement]*
            end

ExpressionList:
            Expression [, Expression]*

ExpressionStatement:
            Assignment | FunctionCall

Assignment: Name = Expression

FunctionCall:
            Name "(" [ExpressionList] ")"

Expression: OrExpr [ xor OrExpr ]

OrExpr:     AndExpr [ or AndExpr ]

AndExpr:    NotExpr [ and NotExpr ]
```

NotExpr: *[* **not** *] RelExpr*

RelExpr: *AddExpr [(“***<**” | “**<=**” |”**<>**” | “**=**” | “**>**” | “**>=**”*) AddExpr]*

AddExpr: *ModExpr [(“***+**” | “**-**“ *) ModExpr]**

ModExpr: *MultExpr [* **mod** *MultExpr]*

MultExpr: *UnaryExpr [(“******” | “**/**”*) UnaryExpr]**

UnaryExpr: *[“***+**” | “**-**“ *] PrimExpr*

PrimExpr: *“***(**“ *Expression “***)**”
| *FunctionCall*
| *Name*
| *Literal*

Literal: *IntegerLiteral* | *HexLiteral* | *StringLiteral*

Runtime API

The runtime library provides functions for interacting with the display and keyboard, for generating random numbers, and accessing the system time. The classes are implemented as Java class in the package *com.claritysys.javabasic.runtime*, which is automatically searched each time a function or field reference occurs that contains a "." in the name. This means if you reference "HunkaHunka.BurningLove(5)" its going to look for a Java class named "HunkaHunka" in the aforementioned package.

The JavaDocs are included in the *etc/api* folder of the distribution, please refer to them for detailed information.