The Topaz Programming Language

Mark Chenoweth

7th November 2002

Contents

1 Introduction

1.1 Language Features

Topaz is a powerful, light-weight, efficient programming language specifically designed for the PalmOS Computing Platform. It is dynamically typed, interpreted from byte-code, has automatic memory management, object oriented features, access to native PalmOS gui widgets and more.

Topaz also Features:

- A simple, short, and flexible syntax
- A simple Class/Object model for object oriented programming
- Built in floating point math and basic functions (sin,cos,tan,etc...)
- Built in graphics functions (drawpoint,drawline,drawrect,etc...)
- Built in native user interface classes (Button,Field,CheckBox,etc...)
- Automatic type conversion
- Easy string handling
- Integrated editor/compiler/interpreter
- A single, small (<70k) .prc file with no extra libraries needed
- Good documentation

Topaz is NOT:

- Meant to create complex multi-form applications
- Meant to replace more mature programming languages

1.2 Quick Start

To create and run the canonical "hello, world" program:

1. Install topaz.prc into your Palm device running PalmOS V3.0 or greater and launch Topaz.
2. Press the "New" button on the Main Form to create a new program and display the Editor Form.
3. Enter this program:

```
# Hello world example
println('hello, world')
```

4. To execute the program, press the "Run" button. The Output Form will appear, and "hello, world" should be printed on it. If there are any errors, an error dialog box will notify you.
5. After the program completes executing, the Output Form will continue to be displayed. You may return to either the Editor or the Main Form by selecting the Menu icon, and then Goto|Editor or Goto|Main.
6. Basic program structure is as follows:
   - The main form will always display the first line of a program as the program title, so programs generally start with a one line descriptive comment followed by one or more statements.
   - Comments begin with a # and continue until the end of the line.
   - Follow the initial comment with one or more statements.

For a more complicated example, try this:

```
# Scribble Demo
btn = Button.new
btn.setXY(10,125)
btn.setSize(40,20)
btn.setLabel('Clear')
btn.show
while true
  e=event(100); y=peny
  if e==btn.id then
    eraserect(0,0,160,120)
  elsif y < 115
    drawrect(penx,y,3,3)
  end
end
```

2 Language Syntax

2.1 Program Structure

A Topaz program consists of a series of statements, interspersed with any function and/or class definitions. Statement, function, and class definitions can be declared in any order, anywhere in a program, but may not be referenced until after they are declared.

By convention, the Main form of the IDE displays the first line of a program as a title of the program. Therefore it is a good idea (but not required) to start a program with a one line descriptive comment. Follow that with any global variable declarations, then any function or class definitions, and finally program statements.

Semi-colons can be used to separate statements for clarity, but are not required.

2.2 Lexical Conventions

Identifiers in Topaz can be any string of letters, digits, and underscores, up to sixteen characters long and must begin with a letter.

The following identifiers are keywords, and may not be used otherwise:

```
and       break    case    class     def
do        else     elsif    end      for
if        mod       not      or    private
protected public  return   self    super
then      unless  until    when    while
```

The following strings denote other tokens recognized by the parser:

```
( ) [ ] { } . ; , & + - * / || && == = > >= < <= <> !
```

And the following keywords are reserved for future use:

```
in include foreach module next require static
```

Topaz is a case sensitive language. “Case” and “case” are two separate identifiers.

String Constants can be up to 64 characters long, delimited by matching single or double quotes, and can contain the C-like escape sequences:

new line:	\n
carriage return:	\r
double quote:	\”
single quote:	\’
tab:	\t
backslash:	\\

Examples of valid string constants are:

- `"hello, world"`
- `'Programming should be fun'`
- `"This \'word\' is quoted"`

Numerical Constants - Standard octal, hexadecimal, and scientific notation is supported for numerical constants. Examples of valid numerical constants are:

- 42
- 0377
- `0xffff`
- `3.14159`
- `707e-3`
- `1.45E5`

Whitespace is ignored outside of string constants and consists of: blanks, tabs, newlines, and carriage returns.

Comments are introduced with a # and continue until the next new line character.

2.3 Data Types

2.3.1 Primitive Types

There are three primitive types built into Topaz:

Integer	a 32 bit signed integer
String	a single or double quote delimited string not more than 64 characters long
Real	a 32 bit IEEE single precision float

In addition to the basic data types, Topaz also supports Arrays, Hashes, and Objects.

2.3.2 Arrays

Arrays are created by assigning an *array initializer* to a variable. An array initializer is a comma-separated string of values between square brackets. You can use the special *array repeat operator* to easily assign multiple values. For example:

```
arr = [ 1, 3.14159, ''some text'', 7x0 ]
```

will create an array of ten elements -

```
arr[0] = 1
arr[1] = 3.14159
arr[2] = ''some text''
arr[3] = 0
arr[9] = 0
```

The format of the array repeat operator is: *count* `x` *value*. The count must be an integer constant, the "`x`" must be lowercase, and the value can be any valid expression. It is only valid inside array initializers.

You can also create n dimension arrays by assigning arrays to array elements. For example:

```
arr = [ 0, 1, [10,20] ]
```

will create an array with three elements, the third being another array, and containing the following values:

```
arr[0]    = 0
arr[1]    = 1
arr[2][0] = 10
arr[2][1] = 20
```

Array indexes always start at zero and continue to length-1 . It is a runtime error to try to access arrays with an invalid index. Memory to hold arrays is allocated from the heap, and can be freed by simply assigning a zero to the array variable.

Individual characters of a string variable may be accessed like an array. For example, after executing the two statements:

```
month = ''November''
first = month[0]
```

the variable `first` will contain an integer with the value of 78 (the ascii value of "N").

2.3.3 Hashes

Hashes can be created with *hash initializers*, or by simply assigning key/value pairs. A hash initializer is a list of key/value pairs between braces. Key are separated from values by the sequence "=>". The key value pairs are separated with commas. For example, the next two statements:

```
months = { ''Jan'' => 1, ''Feb'' => 2 }
months{''Mar''} = 3
```

will first create a hash with two elements, and then add another. The final result will be -

```
months{''Jan''} = 1
months{''Feb''} = 2
months{''Mar''} = 3
```

Memory for hashes is allocated from the heap, and can be freed by simply assigning a zero to the hash variable.

2.3.4 Objects

Objects are 32 bit references to an object instance that are created by instantiating a class using its `new` method.

2.4 Variables and Constants

Topaz variables are type-less and are declared by assigning a value to them with an assignment statement. The *value* of a variable has a type associated with it.

- Variables start with a lowercase letter, and are followed by up to 15 other letters, numbers, or underscore characters.
- Constants start with an uppercase letter, and are followed by up to 15 other letters, numbers, or underscore characters. Once a constant is defined, it is a compile-time error to attempt to redefine it.

2.5 Expressions and Operators

2.5.1 Expressions

Expressions combine variables and constants (the operands) with operators to produce new values.

2.5.2 Operators

Topaz supports a standard set of operators. The following table lists all the operators and the types they operate on:

							Valid operand types
logical negation:	`not`	`!`					integer
multiplicative:	`*`	`/`					integer or real
modulus:	`mod`	`%`					integer
additive:	`+`	`-`					integer or real
bitwise logical:	`and`	`&&`	`or`	`||`			integer
relational:	`==`	`<>`	`<`	`<=`	`>`	`>=`	integer or real

Note that are are two operators with identical functions for the *logcial negation, modulus*, *bitwise and* and *bitwise or* operators.The logical negation operators only operate on a single operand. All the rest operate on two operands. Parenthesis can be used to group expressions together.

2.5.3 Operator Precedence

Operator precedence is similar to PASCAL where there are only four levels of operator precedence:

Highest:	`not`	`!`				
	`*`	`/`	`and`	`&&`	`mod`	`%`
	`+`	`-`	`or`	`||`		
Lowest:	`==`	`<>`	`<`	`<=`	`>`	`>=`

Operators at the same precedence level are evaluated from left to right. When in doubt, use parenthesis to group your expressions together.

2.5.4 Automatic Type Conversion

Topaz provides automatic type conversion between values during runtime. When an arithmetic expression is being evaluated using the relational, addititive or multiplicative operators, the two operands will automatically be converted according to the following rules:

1. If an operand is a string, it will be converted to a real if there is a decimal point present. Otherwise, it will be converted to an integer.
2. If both operand types are ints or reals, the resulant type is the same as the operands.
3. An integer and a real make a real.

For example:

- $1 + 3.1415 = 4.1415$
- $1 +$ "2.0" $= 3.0$
- "1.5" $+ 2.0 = 3.5$
- ”10” + ”20” = 30

2.6 Statements

When describing the syntax of Topaz statements, optional keywords are enclosed by square brackets.

2.6.1 The assignment statement

Lhs `=` *Expr*

The assignment statement is used to declare variables as well as assign new values to existing variables. The left hand side can be a new or existing variable. The right hand side can be any normal expression, array initializer, hash initializer, or function or method call.

2.6.2 The `if` statement

The basic format of the `if` statement is as follows:

```
if condition [then]
  statementlist
elsif condition [then]
  statementlist
else
  statementlist
end
```

The condition is an expression which controls what the rest of the statement will do. If condition is true, the the next block of code will be executed. The `elsif` and `else` blocks are optional. Note however that unlike certain other languages, the `end` keyword is required.

2.6.3 The `unless` statement

```
unless condition [then]
  statementlist
else
  statementlist
end
```

The `unless` statement is similar the the `if` statement, except that the next block of code will be executed when when the condition is false. The `else` block is optional.

2.6.4 The `case` statement

```
case expression
when condition [then]
  statementlist
else
  statementlist
end
```

The `case` statement is yet another form of the `if` statement. The expression after the case keyword is evaluated. If the result matches the value of the expression in one of the `when` or `else` statements, then that block of code will be executed. There can be zero or more `when` blocks and zero or one `else` blocks.

2.6.5 The `for` statement

```
for( init; condition; incr )
  statementlist
end
```

The `for` statement is very similar its C language counterpart. The *init* part is an assignment statement which is executed at the beginning of the loop. Then *condition* is evaluated, and while it is true, the *statementlist* will be executed. At the end of the loop, *incr* is evaluated, and the loop starts over. All three parts (init,condition,incr) are optional.

2.6.6 The `while` statement

```
while condition [do]
  statementlist
end
```

The `while` statement is the simplest of loops. The condition part is evaluated at the beginning of each loop, and while true, the statement list will be executed.

2.6.7 The `until` statement

```
until condition [do]
  statementlist
end
```

The `until` statement is the opposite of the while. The condition part is evaluated at the beginning of each loop, and while false, the statement list will be executed.

2.6.8 The `break` statement

```
break
```

The `break` statement is used to break out of `for`, `while`, and `until` loops. It transfers control to the next statement after the loop.

2.6.9 The `return` statement

```
return expression
```

The `return` statement is used to return from functions and class methods. *expression* is the optional return value.

2.6.10 The `public` statement

```
public
```

The `public` statement is only valid within class definitions and signals that any following fields or methods will be publicly available.

2.6.11 The `protected` statement

```
protected
```

The `protected` statement is only valid within class definitions and signals that any following fields or methods will be protected. That is, they are only visible within the class they are defined and to sublcasses.

2.6.12 The `private` statement

```
private
```

The `private` statement is only valid within class definitions and signals that any following fields or methods will be private. That is, they are only visible within the class they are defined in.

2.7 Functions

Functions in Topaz are similar to functions in most other procedural programming languages. Their basic syntax is:

```
def functionname(param1,param2,)
  statementlist
end
```

The params enclosed by parenthesis are optional. The `return` statement can optionally be used to return a value.

Functions can be defined anywhere in a program, but can only be referenced after they are defined. Any new variables declared within a function are considered local variables and are only visible within that function. Any variables, functions, or classes declared in the global scope before the current function definition will also be visible.

2.8 Classes

The basic sytax of a class definition is:

```
class Name < Superclass
  body
end
```

The class name must begin with an uppcase letter. The sequence `< Superclass` is optional. A class body consists of a series of one or more field and method definitions. By default, everything in a class has public scope. The `public`, `protected`, and `private` statements can be used the change the visibility of following definitions.

2.8.1 Instantiation

Every class implicitly has a `new` method which will automatically create a new object and initialize its fields. For example:

```
point = Point3D.new
```

will create a new object of class Point3D and assign a reference of it to the variable point.

2.8.2 Fields

Class fields must be initialized with static values - either int, real, string, hash or array. You cannot intialize a field with an object.

2.8.3 Methods

Class methods are simply functions declared within a class definition. The syntax is exactly the same. A method with the name `init` is considered a class constructor. See the section on class constructors.

2.8.4 Scope

Class fields an methods can be at one of three scope levels:

Public - visible anywhere within it's class after its declaration, and in any instantiated objects.

Protected - visible within the class it is defined in, and any subclasses

Private - visible only within the class it is defined in

By default, everything in a class is public. You can change the scope using the public, protected, or private keywords.

Class fields and method declarations only belong to the class they are defined in, therefore they can and will override any previous identifiers.

2.8.5 Constructors

Class fields are automatically initialized when a classes `new` method is called. In addition, you may create a user defined class initializer method called `init`. If the `init` method declaration does not have any formal parameters, it will be automatically called right after the fields are initialized. If it does have parameters, you can pass them via the `new` call. If it does have parameters and you do not pass them via the new call, the `init` method will not automatically be called, however you may explicitly call it yourself (with the proper number of parameters). When a class is instantiated all of its superclasses fields will be initialized, however you must explicitly call any `init` methods. Generally you would call a superclasses `init` method from the base classes `init` method. For example:

```
class Point2D
  x=0; y=0
  def init(x,y)
   self.x=x; self.y=y
  end
end
class Point3D < Point2D
  z=0
  def init(xx,yy,zz)
    super.init(xx,yy)      # or - x=xx; y=yy
    z=zz
  end
end
p1 = Point2D.new           # p1.x=0, p2.y=0
p2 = Point2D.new(10,10)    # p2.x=10, p2.y=10
p3 = Point3D.new           # p3.x=0, p3.y=0, p3.z=0
p4 = Point3D.new(10,10,10) # p4.x=10, p4.y=10, p4.z=10
```

2.8.6 Inheritance

Topaz supports single inheritance with chained subclassing. That is - a class can have only one superclass, however that class can have a superclass, and so on.

2.9 Scope

The scope rules in Topaz are pretty simple.

- The built-in constants, functions, and classes are always accessible anywhere in a program
- A user defined constant, variable, function, or class is not accessible until after it is declared.

- All function and class names are global as well as any constants or variables declared outside of function or class definitions.
- Any global constants, variables, functions, or classes will be accessible within any subsequent function or class definitions.
- If a constant or variable is declared for the first time within a function or class method, it is considered local and will only be visible within that function or method.
- Class fields and methods are alway local to the class they are defined in and can override global identifiers.

2.10 Memory Management

Topaz automatically allocates and frees memory to hold values as needed. Array, Hashes, and Objects are all allocated from the PalmOS 64k heap area when they are declared. Caution is recommended when creating many objects or large arrays and hashes. Any memory allocated inside of a method is automatically freed (except for any return value) when returning from the method. If you allocate a large array or hash, and would later like to free the memory associated with it, simply assign a zero to the variable.

3 Standard Library Reference

3.1 Constants

`True`
The integer one.

`False`
The integer zero.

3.2 Functions

Functions are described with the convention: return_type=function(typeName), where type can be "i" for integer, "r" for real, "s" for string, or "e" for expression.

3.2.1 Types

`s=getType(e)`
Returns: "int", "real", "string", "array", "hash", or "object".

`v=setType(e)`
Returns: e converted to type sType; where sType can be - "int", "real", or "string".

`i=isInt(e)`

Returns: True if var is an int, False otherwise.

`i=toInt(e)`
Returns: e converted to an integer.

`i=isReal(e)`
Returns: True if var is a real, False otherwise.

`r=toReal(e)`
Returns: e converted to a real.

`i=isStr(e)`
Returns: True if var is a string, False otherwise.

`s=toString(e)`
Returns: e converted to a string.

3.2.2 Math

`r=sin(rDeg)`
The sine of x in degrees.

`r=cos(rDeg)`
The cosine of x in degrees.

`r=tan(rDeg)`
The tangent of x in degrees.

`r=sqrt(rDeg)`
The square root of x.

`r=abs(rDeg)`
The absolute value of x.

3.2.3 Graphics and Events

`i=event(iTimeout)`
Waits iTimeout ticks for an event. Events returned are:
0 - nilEvent
1 - penDown
2 - penMove
3 - penUp
4 - keyDown
3000 or greater - Control ID after control select event.

`yield`
Gives the OS a chance to process events. Use inside of loops to prevent lockups.

`i=penx`
Returns: the last x coordinate of the stylus.

`i=peny`

Returns: the last y coordinate of the stylus.

`i=key`
Returns: the code of the last Graffiti character entered.

`clear`
Clears the entire screen.

`drawpoint(iX,iY)`
Draws a pixel at X,Y.

`drawline(iX1,iY1,iX2,iY2)`
Draws a line from X1,Y1 to X2,Y2.

`drawrect(iTopLeftX,iTopLeftY,iExtentX,iExtentY)`
Draws a rectangle from TopLeftX,TopLeftY with the sides length ExtentX and ExtentY.

`title(sTitle)`
Draws a PalmOS style title bar at the top of the screen.

3.2.4 Miscellaneous

`print(v{,v})`
`println(v{,v})`
Prints a comma separated list of values to the console using default formatting. Println appends a newline character, print does not.

`printf(sFormat,v{,v})`
`printxy(iX,iy,sFormat,v,{,v})`
Works just like the standard C printf. Printf formats and prints to the next line on the console, printxy prints starting at pixel location X,Y. Supports all of the standard width and precision modifiers and the following format codes:

```
%d - decimal
%s - string
%c - character
%o - octal
%x - lower case hex
%X - upper case hex
%f - real
%e - lower case exponential notation
%E - upper case exponential notation
%i - integer
%b - binary
```

`i=len(e)`
Returns the length of e. If e is an array or hash len will return the number of elements. For a string, it will return the length of the string. For int's and real's, it returns one.

`beep`

Triggers the Palm's built-in piezo element.

`delay(iTicks)`
Waits for a specified amount of time.

`i=random(i)`
Returns: a random number between 0 and i.

`dup(v)`
Creates a copy of the argument.

3.3 Classes

3.3.1 Misc

`Object`

3.3.2 User Interface

`Fld`

`Btn`

`PBtn`

`RBtn`

`CBox`

`Lst`

`Trg`

`Sel`