WANTScript 2.0

Tutorial & Reference

DRAFT

NOTE: This is work-in-progress. As such, it may contain errors, omissions, and is subject to frequent revisions. Use at your own risk and peril.

By Andrew J. Woźniewicz
Updated September 18, 2008

TABLE OF CONTENTS

Introduction

About This Tutorial

NOTE: This revision of the tutorial refers to WANT version 2.0.4 ALPHA.

This tutorial is intended for programmers who want to use WANTScript – the programming language underlying WANT 2.0. This is not an introductory programming tutorial and a fair amount of knowledge on how to program in an object-oriented language is assumed.

The code examples in this tutorial appear in a distinct font and inside a shaded box:

```
project WANTScriptProject
  ...
end
```

The console output of sample programs is presented in a differently shaded box:

```
This is output of a WANTScript program
More output from the program
```

Language syntax is written inside a box like this:

```
while <condition> do
  <statements>
end
```

The syntax-highlighting displayed by the example code is the default syntax-highlighting employed by WIDE, the WANT IDE.

The code appearing in this tutorial was pasted from the WIDE editor by using `Edit|Copy As|Copy as HTML` functionality of the IDE.

Note that, in the interest of expediency, fragments of “Python Tutorial”, “Python Library Reference”, and “Python Reference Manual” by Guido van Rossum (Release 2.5, 19 Sep. 2006, Copyright © 2001-2006 Python Software Foundation; All Rights Reserved. See www.python.org) may have been quoted verbatim – or with minor changes – whenever WANTScript semantics closely correspond to Python semantics. This is acknowledged here up front and no further acknowledgements will be made in the body of this tutorial.

Setting Up WANT

In order to execute any of the examples in this tutorial, you will need the WANT executable, version 2.0 or later. You can download the latest WANT distribution from `http://www.want-tool.org/downloads`. Place the command-line executable called `want.exe` from the distribution somewhere in your path. When you type `want` on the command line, you should see something like this (the exact version and build numbers will vary):

```
WANT 2.0.4 (Build 2008.09.26.13.19)

Usage: WANT [options] <filename> [[options] argument argument ...]
  where
  -c = Log calls
  -xnnnnnnn = Generate intermediate XML files, e.g. -x1357
  -o = Output folder for files
```

This means WANT is installed and working properly. It also tells you that, at the minimum, you need to supply the name of the script to execute on the command line. That is, unless you happen to have a file named `build.want` in your folder, because WANT will use `build.want` as the default script name and will execute it, if present, without your having to supply the name explicitly.

NOTE: A graphical Integrated Development Environment (IDE) for WANT, called WIDE, is also available in the distribution as `WIDE.exe`. Although all examples here will execute equally well in WANT or WIDE, this tutorial assumes that you are using WANT (the command-line interpreter) for simplicity. For more information on how to use WIDE, see the separate WIDE Tutorial.

What Is WANTScript?

WANT executes programs (source scripts) written in WANTScript. WANTScript is an interpreted, block-structured scripting language – a light version of Modula-7 (www.modula7.org). WANTScript programs are called projects. WANT projects are typically software build-scripts performing routine tasks of compiling, linking, and packaging software products, given that WANT is primarily a software build automation tool. WANTScript is also a general purpose programming language, however, so that programs can be written for many other uses.

Creating Projects

A WANTScript project is simply an ASCII text file containing a `project` module. Type the following into any ASCII editor, save it as `MyProject.want`, and you have a skeleton WANTScript project:

```
project MyProject
```

```
  // project code goes here
end
```

The identifier following the keyword **project** is the project name. Use the same rules for naming a WANTScript project as you would use to name a Pascal program or unit.

WANTScript project files should have the extension `.want`, e.g. `MyProject.want`.

Notice that the double-slash acts as the single-line comment delimiter in WANTScript, just as it does in Delphi/Pascal (or C++, etc.). There are other comment delimiters which will be described shortly.

Also notice a conspicuous absence of semicolons. In WANTScript, semicolons are strictly optional. You can put them in where you feel they belong, for example after the name of the project, or after the final **end**, but you can just as well omit them.

> The examples in this tutorial generally omit any semicolons. When the semicolons are present, they are included solely for illustrative purposes, or because the author was conditioned by the long-term use of other languages. They are permitted – where appropriate – by the WANTScript syntax, but in no way required.

Another difference between WANTScript project and a Pascal program is that the file containing the source does not need to have the same name as the project itself. So, you could save the above project to a file named `test1.want`, for example. More on the relationship between the file name and the project name later.

TODO: Files versus projects in the section on imports

The Hello World Project

"HelloWorld" is the traditional first program in just about any programming language tutorial, so we will continue the tradition here.

To output (or “write”) a line of text to the console, you can use the built-in `System.IO.Console.WriteLn` function. A possible `HelloWord` program looks like this in WANTScript:

```
System.IO.Console.WriteLn("Hello world!")
```

That’s all there is to it – a `HelloWorld` WANTScript project is a single line of code. You can save this one line into a text file `hello.want` and execute it. WANT automatically creates a project named `DEFAULT` if you don't explicitly specify a project name. The above single line of code is thus equivalent to the following:

```
project DEFAULT
  System.IO.Console.WriteLn("Hello world!")
end
```

Now, if you want to output more than a single line of text, or if you are a Pascal programmer (or both), you would probably object to writing the fully-qualified name `System.IO.Console.WriteLn` every time you want to output a line. Good news is that, instead of always having to use fully qualified names, you could just import the entire `System.IO.Console` module, which – besides `WriteLn` – contains some other useful functions, and then use these functions directly, without qualification. This is analogous to the `uses`-clause in Pascal. You can import another module like this:

```
project HelloWorld
  import System.IO.Console
  WriteLn("Hello world!")
end
```

When you use the `import` directive, it makes all the publicly visible symbols from the imported module available in the importing module. That way, you can use the `WriteLn` function pretty much the same way you would use it in Pascal, without any qualification, like the above example illustrates.

Now, even better news is that you don't need to bother importing `System.IO.Console` explicitly at all, since it is automatically and implicitly imported for you by the WANT Runtime. So, you could have omitted the import directive entirely, and written the `HelloWorld` project simply as follows:

```
project HelloWorld
  WriteLn("Hello world!")
end
```

That's not all, though. Since you already know that you can omit the project header – if you are happy with the default project name of `DEFAULT` – the entire `HelloWorld` program in WANTScript boils down to this single line of code:

```
WriteLn("Hello world!")
```

Not bad, but it gets even better! In WANTScript, you can use a literal value, like the `"Hello world!"` above, which is a string literal, in place of a statement. It is automatically understood by the interpreter to mean a call to the `WriteLn` built-in procedure, with the literal as its argument. So, the above line is equivalent to

```
"Hello world!"
```

Now, even this can be simplified. WANTScript does not require that you terminate your string literals. It will automatically terminate them for you at the end-of-line. So, you can rewrite the above to

```
"Hello world!
```

Finally, since you can use either single or double quotes, you could equally well have written just

```
'Hello world!
```

which arguably holds the world's record for the shortest HelloWorld program ever.

So now we've come full circle. The HelloWorld program in WANTScript could be a very simple single line of code, if you wanted, but having seen these many different variations on that theme should have given you a better feel for some of WANTScript syntax.

WANTScript Comments

In this section you will se how to include comments in a WANTScript file.

WANTScript supports all the comment delimiters supported in Object Pascal (Delphi):

- Single-line comments starting with two forward slashes //:

```
//This is a single-line comment
```

- Multi-line comments delimited by a pair of braces `{ }` :

```
{
This is a multi-line (or block) comment.
This style of comment is inherited from Pascal.
}
```

- Multi-line comments delimited by the sequences `(*` and `*)`, also known as ANSI comments in Pascal:

```
(*
This is Pascal-style multi-line (block) comment
known as the ANSI-style comment.
*)
```

In addition to those three kinds of comments directly inherited from Pascal, WANTScript also supports C/C++ style of multi-line comments delimited with `/*` and `*/`, for example:

```
/*
```

```
This is C-style multi-line (block) comment
typical for C, C++, Java, SQL, and a host of
other languages.
*/
```

Comments are generally ignored by the interpreter and treated as blank space. You cannot nest comments of the same type. Comments cannot occur within string literals.

A Simple Build Script

WAnt is about building things using other tools. The primary focus is therefore the ability to compile and package a software release. Along those lines, let's look at a script that performs a very simple build:

TODO: A simple build script example

Useful Example Projects

Following is a number of examples of WANTScript projects performing various useful tasks, from executing other programs (`WANT.FileTasks.ShellExec` and `Exec`), to … , to running automated tests with DUnit (`WANT.Tests.DUnit`).

TODO: List of useful tasks

Executing Other Programs

A very useful capability in a build management tool is the ability to execute another program, as if invoked from the command line. You can execute an external executable in one of two ways: using `WANT.FileTasks.ShellExec`, or using `WANT.FileTasks.Exec`.

Executing with WANT.Filetasks.ShellExec

The `WANT.Filetasks.ShellExec` is a thin disguise for the Windows API call `ShellExecute`: it performs an operation on a specified file. The operation could be one of 'open', 'edit', 'explore', 'find', or 'print'; the default is 'open'.

The following short script is an example of using `ShellExec`:

```
project TestShellExec

  import WANT.FileTasks
```

```
  "**** RUNNING: TestShellExec
  call ShellExec with
    Executable := "D:\temp\test.txt"
    Operation := 'edit'
  end
  "**** TestShellExec END.

end
```

In this example, `ShellExec` is called to open the file `d:\temp\test.txt` for editing. Assuming the specified file exists, it will be open in the default editor, which on Windows is usually Notepad for files with the extension of `.txt`. If the file does not exist, the call will fail with an error code of 2.

The three most important things you have to know about `ShellExec` are:

- It launches external programs as INdependent processes; the call to `ShellExec` returns almost immediately, without waiting for the executed program to close;
- It does NOT allow the WANT console window to capture the output of the executed program (you must use `WANT.FileTasks.Exec` for that, as described below);
- It is capable of launching not only executable programs (.exe), but files, in which case the associated program is launched; folders, in which case Windows Explorer opens; URLs, in which case the default browser opens; and other system objects that open their associated applications.

The `ShellExec` function returns a numeric result, with the value greater than 32 if successful, or an error value that is less than or equal to 32 otherwise.

Please, refer to the Windows API documentation for more information about `ShellExec`'s capabilities. Inside the WANTScript environment, `WANT.Filetasks.ShellExec` is (implicitly) declared as follows:

```
module WANT.FileTasks
  ...
  function ShellExec(FileName, Arguments="", Dir="",
    Operation="", ShowCmd = 1, FailOnError=True)
    ...
  end
  ...
end
```

The only difference with the Windows protoplast is that WANTScript's `ShellExec` does not take a window handle as a parameter, and makes all parameters other than the `FileName` optional.

Executing with WANT.FileTasks.Exec

`WANT.FileTasks.Exec` function has similar purpose to `ShellExec`: to launch an external application. It differs from `ShellExec` in a number of important ways:

- It launches external programs as child processes of the executing WANTScript host program (WIDE or WANT, as the case may be), and waits for the executed program to finish.
- It captures the standard output of the external program and directs it to the script console; the output of the executed program becomes part of the script output.
- It can only launch executable programs (.exe, .com); it cannot launch files directly.

Here is an example of using `WANT.FileTasks.Exec` to compile a C source file with an external C compiler (gcc, the GNU C compiler):

TODO: Example of Exec

Working with Text Files

TODO: Working with text files

Working with INI Files

TODO: Working with INI files

Working with XML

TODO: Working with XML

Unit Testing

This section is dedicated to integrating unit testing with DUnit into your build scripts.

DUnit is “an xtreme testing framework for Delphi programs” – a Delphi port of JUnit. DUnit, just like WANT, is an open-source project hosted on SourceForge (http://sourceforge.net/projects/dunit/). It is a testing framework that helps automate unit testing of Delphi code. There is a natural and historical synergy between WANT and DUnit, which is not surprising considering that both have been conceived and originally written by the same person (Juancarlo Añez).

WANT has traditionally incorporated and continues to support DUnit testing framework (at this time, version 9.3.0 released on Feb. 13, 2007 is current) and is able to run test libraries created with a compatible version of DUnit.

> It is imperative that the version of DUnit used to compile WANT is the same version that you use to compile your test libraries with. Otherwise, difficult-to-debug errors will ensue. If you are running your tests with the pre-compiled WANT executables downloaded from the WANT project site, you must compile your test libraries with the version 9.3.0 of DUnit. The DUnit source code used to build the official WANT binaries is included with the source distribution.
>
> If, on the other hand, you choose to compile your test libraries with a version of DUnit other than 9.3.0, you must recompile the WANT binaries (WANT and WIDE) with that version of DUnit and use your newly created executables rather than the official binaries.

Creating a Test Library

Although this section is not intended as a tutorial on using DUnit, a few words on creating WANT-compatible DUnit-test libraries are in order. For more information on creating unit-tests with DUnit, see the DUnit web site at http://dunit.sourceforge.net.

The first step to incorporate unit testing into a WANT build script is to create a compatible test library. Here is an example of a trivial test library – a test-library-equivalent of a HelloWorld program – that you can use to exercise the unit-testing capabilities of WANT:

```
library TestClassLib;

uses
  ShareMem,
  TestFramework,
  TestClass;

{$R *.RES}

exports
  RegisteredTests name 'Test';
```

```
end.
```

TestClassLib is a library project that – when compiled – produces TestClassLib.dll – a DUnit test-library. The library is in the canonical form for DUnit-WANT integration: it exports a single function, RegisteredTests from the TestFramework unit, with the exported name of Test. Your own test libraries will look essentially the same.

> When creating your own DUnit test libraries, be sure that you always export the correct function (RegisteredTests) and that it is always exported as Test.

WANT expects to find a function called Test in every DUnit library it loads, and the function is supposed to return the ITestSuite interface declared in the TestFramework unit – just like RegisteredTests does.

The actual test code that the TestClassLib library exercises is contained in a separate unit, in this example, called TestClass:

```
unit TestClass;

interface

uses
  TestFramework;

type
  TTestClassTest = class(TTestCase)
  protected
    procedure SetUp; override;
    procedure TearDown; override;
  published
    procedure Test;
  end;

implementation

uses
  SysUtils;

{ TTestClassTest }

procedure TTestClassTest.SetUp;
begin
  //Do nothing
end;

procedure TTestClassTest.TearDown;
begin
  //Do nothing
end;

procedure TTestClassTest.Test;
begin
  Check(True);
end;

initialization
```

```
  RegisterTest(TTestClassTest.Suite);
end.
```

The `TestClass` unit is the equivalent of the user-supplied DUnit test case that exercises some aspect of your target program.

It defines a very simple test case, `TTestClassTest`, which – for the purposes of this exercise – never fails: the `Test` procedure, which is the only published test procedure in this case, calls `Check(True)` which ensures that the test succeeds. Your own test cases will likely be more elaborate, but this is not a DUnit tutorial after all.

The initialization section of the unit calls `RegisterTest` to register the test suite with the framework, and the previously-shown `TestClassLib` library project includes `TestClass` unit in its `uses`-clause.

Ensuring that `ShareMem` is the first unit mentioned in the **`uses`**-clause of the library project completes the preparations of the test library. You should compile the project to produce `TestClassLib.dll`.

Creating a Test Script

Once you have a DUnit test library (`TestClassLib.dll`) prepared, you can write WANTScript code to exercise the tests in the library.

The script to use the library is very simple:

```
project TestLibTest

  import WANT.Test

  call DUnit with
    TestLib := "d:\WANT2\TestClassLib.dll"
  end

end
```

The exact path to the test library in your script will vary, depending on where you put the library DLL, but you should get output similar to what follows when you run the above script:

```
Test success: Test
Test success: TTestClassTest
Test success: TestClassLib.dll
```

This kind of output is typical of a test case that succeeds. To see what happens when a test case fails, change the line inside the `TestClass` unit that reads

```
Check(True);
```

to

```
Check(False);
```

and recompile the library project. This time, when you execute the test library, you should receive an error message to the effect that a test failed.

That's all there really is to integrating with DUnit. Of course, you don't need to recompile the test library manually, separately from running the script that exercises the library; you can combine both operations (compile test library, and run tests from the library) in a single script:

TODO: An example of compiling and exercising a test lib

Passing Functions As Parameters

In WANTScript, modules are first-class values, that is, they can be passed as arguments, returned from function calls, bound to variable names, etc., just like simple types such as strings, Booleans, and numbers.

Blocks of code that can be passed as arguments are commonly (and often incorrectly) referred to as closures. The purpose of a closure is to simplify the process of passing a piece of functionality to a module, together with an evaluation environment containing bound variables.

By design, WANTScript does not support true closures. It allows the passing of existing named blocks (modules) as parameters to other modules, but it does not bind the environment of the passed named block. In that regard it is more similar to Pascal and C++, with their support for function pointers, than to Smalltalk, Ruby, or Scala, which support true closures.

Here is an example of passing a function as a parameter in WANTScript:

```
project FuncAsParam //test033

  function Add(N1, N2)
    return N1 + N2
  end

  function Mult(N1, N2)
    return N1 * N2
  end

  function Max(N1, N2)
    if N1 >= N2 then
      return N1
    else
      return N2
    end
  end

  function Min(N1, N2)
    if N1 <= N2 then
      return N1
    else
      return N2
```

```
    end
  end

  procedure Evaluate(Action,
    Param1, Param2, ExpectedValue)
    var Result
    Result := Action(Param1,Param2)
    WriteLn( PadLeft(Result,2),
      " (expecting ", ExpectedValue, ")")
  end

  Evaluate(@Add,7,5, 12)
  Evaluate(@Mult,7,5, 35)
  Evaluate(@Max,7,5, 7)
  Evaluate(@Min,7,5, 5)

end
```

The project `FuncAsParam` defines a number of inner functions, each of which takes two numeric parameters. It also defines an `Evaluate` procedure that takes an `Action` parameter, as well as two numeric parameters `Param1`, and `Param2`, and a numeric `ExpectedValue` parameter that is being passed in for display purposes.

The `Evaluate` procedure treats its `Action` parameter like a function that takes two arguments: it calls it with the arguments `Param1`, and `Param2`, the values of which it itself received from the caller.

The caller in this case is the main module `FuncAsParam`, which calls the `Evaluate` procedure repeatedly, each time with a different `Action`. First, the `Action` is the `Add` function; then the `Mult` function, then the `Max` function, and finally the `Min` function are passed into `Evaluate`.

The output of this program is as follows:

```
12 (expecting 12)
35 (expecting 35)
 7 (expecting 7)
 5 (expecting 5)
```

You can see how different actions produce different results, even though the numeric arguments passed into `Evaluate` are the same in each case.

The key to passing a module as a parameter is the module-reference operator denoted by the @-sign (analogous to the @, or the "address of" Pascal operator). If you were to list a function name, such as `Add`, without the @-sign as one of the parameters to `Evaluate`, it would be understood as a function call and the interpreter would complain about missing parameters:

```
Evaluate(Add, 7,5, 12) //Does NOT work!!!
```

The first argument here is a reference to `Add`, which is treated like a call to the function, and – since the call does not include any arguments – it is bound to fail. If you want to pass the function as a whole as an argument, you must use the module-reference operator @:

```
Evaluate(@Add,7,5, 12)
```

In this case, thanks to the @-sign in front of the function name, instead of trying to invoke the specified function, the runtime is passing the entire function as a parameter for `Evaluate` to call internally.

WANTScript Reference

Lexical Analysis

WANTScript uses the 8-bit ANSI character set for program text. A WANTScript project (program) is read by a parser and converted into an in-memory “parse-tree” representation that is further manipulated by the analyzer and eventually executed by the runtime.

The whitespace characters space, tab and formfeed can be used interchangeably to separate tokens. Whitespace is needed between two tokens only if their concatenation could otherwise be interpreted as a different token (e.g., ab is one token, but a b is two tokens).

Unlike in Python, indentation does not carry intrinsic meaning and is not enforced by the interpreter – it is merely a stylistic tool to help program readability.

WANTScript, like Pascal, and unlike C/C++, is not case sensitive. Names like `MyModule`, `MYMODULE`, and `mYmOdUlE` are equivalent.

For readability, so called camel capitalization is recommended, with every word of a compound identifier starting with a capital, e.g. `MyProgram`, `VeryLongModuleName`, etc. keywords should be written in all lowercase.

Literals

Literals are notations for constant values of some built-in types.

Strings

String literals are instances of `System.String` module. Examples are:

```
"Hello World!"
'1234567890'
'Shakespeare wrote "Hamlet"'
"Ed's question was:"
```

A string is a sequence of zero or more ANSI characters enclosed within single or double quotes (`'` or `"`), known as string delimiters. Double-quote characters may be contained within strings delimited by single-quote characters, and single-quote characters may be contained within strings delimited by double quotes.

String literals must be written on a single script line – they may not be broken across two lines. If a matching closing string delimiter is not present on the same script line as the opening delimiter, the string literal is considered automatically terminated at the end of the line.

Strings can be concatenated (glued together) with the + operator:

```
WriteLn("Hello" + ' ' + "World" + '!')
```

For more information on using strings, see the section on `System.String` built-in type (page 55).

Numbers

Numeric literals are instances of the built-in type `System.Number`. Numeric literals are written naturally as sequences of digits, with an optional decimal point. Scientific notation, which includes the character ‘E’ (or ‘e’) and the exponent, may also be used. [**TODO: Hex**] For example:

```
12345
3.1415
1.2E4
3.14
10.
1e100
3.14e-10
0E0
```

Note that numeric literals do not include a sign; a phrase like `-1` is actually an expression composed of the unary operator ‘`-`‘ and the literal `1`.

Buglet: Numeric literals starting with the decimal period, such as `.001`, are not recognized. One must write `0.001` instead.

Booleans

Boolean literals are instances of the built-in type `System.Boolean`. There are exactly two Boolean values:

```
True
False
```

The Boolean literals are reserved words in WANTScript.

Lists

A list literal is an instance of `System.List` built-in module. List literals are written as lists of comma-separated values (items) between square brackets:

```
[1,2,3]
[1, True, 'Yes', 1.14]
["Sat", "Sun"]
```

For more information on lists, see *Built-In Type: System.List* section (page 58).

Modules

The most fundamental building block of a WANTScript program is a module. A module is a combination of some data (variables and parameters), and a single block of code that is associated with the data (executable statements). Modules are WANTScript's abstraction mechanism that provides a convenient way to encapsulate a list of statements, and a set of variables.

MODULE = DATA (parameters+variables) + CODE (statements)

> There is a subtle distinction between the definition of a module just given and a typical definition of a class in an object-oriented language. A class similarly defines data (fields) and code (methods with executable statements); but notice that the code in the case of a class is not a simple, monolithic block (list of statements), but rather a collection of individual blocks (methods), each of which can be considered a module in its own right. Consequently, the concept of a WANTSCript module is more fundamental than that of a class, as classes can be built by lexically nesting method modules inside a class module.

The basic structure of a module declaration is very simple:

```
module <ModuleName>
  <declarations-directives-and-statements>
end
```

A module declaration consists of the `module`-keyword, followed by the module name (an identifier), the module "block" consisting of embedded declarations, statements, and directives, and finally the keyword `end`.

Here is an example of a module declaration:

```
module MyModule
  // This is the module block
end
```

The keyword `module` has several aliases in WANTSCript. The following is a list of module keywords: `module`, `project`, `procedure`, `function`, `library`, `class`, and `unit`. Most of these keywords are completely interchangeable – they are simply aliases for the same thing – a module. In some cases, the module-alias keyword has additional meaning, which will be explained in due course.

A module can be declared anywhere within the declaration of an enclosing module, or at the top level in a script file. When the declaration appears as the outermost module in a file, it is called a top-level module. Otherwise, the module is called a nested (or local) module.

A module can be accessed from anywhere its name is visible.

The Project Module

The keyword `project` used in the prior examples is just an alias to indicate a top-level `module`. The other keywords that you could use in place of `project` are `module`, `library`, and `class`. These are "top-level" module keywords, so called because they must be used at the top-level of a script file – as the outermost block in a file – they cannot be nested.

Other types of modules include `procedure`, `function`, and `target`, whose definitions must be nested inside an enclosing top-level module. So the following is a valid WANTScript project:

```
procedure Test08
  procedure Embedded
  end
end
```

This procedure translates to the following:

```
module Test08
  module Embedded
  end
end
```

On the other hand, you may not nest top-level modules inside other modules. The following is *not* a valid WANTScript project, because the keyword `program` may only be used to designate a top-level module in a script file:

```
procedure Test08
  //This is invalid!
  program Embedded
  end
end
```

> The differences among various types of blocks, such as procedures and classes, classes and units, units and libraries, etc., are not as pronounced in WANTScript as they are in other programming languages. All of these blocks are examples of WANTScript modules, and all of them can be containers of nested modules, such as procedures and classes. The restrictions on which keywords are top-level and which are not are purely to aid the programmer in organizing the code. You can always use the `module` keyword for everything.

This universal-nesting, and all-modules-are-equivalent paradigm takes some getting used to, but once understood, it becomes a very powerful and expressive abstraction tool.

There is very little structure imposed on the organization of a WANTScript module. This means that declarations of module's variables, for example, can appear anywhere within the module, in any order.

In WANTScript it is not necessary for a variable declaration to precede its use in a statement. WANT is a multi-pass interpreter and will resolve the names whose declarations appear later in the module. The following module is thus perfectly valid:

```
project Test03;
  import System.IO.Console;
  WriteLn("The answer is: ",X);
  WriteLn("What was the question?");
  var X := 42;
end;
```

This example also illustrates how to declare and initialize variables. Notice that you don't explicitly declare the type of a variable. Its type will be inferred by the WANT Runtime from its use. In this case the variable will hold a Number. Finally, notice that the `System.IO.Console.WriteLn` module (procedure) permits multiple and variable number of arguments, just like the Pascal version of it. Unlike in Pascal, however, you can write your own modules (procedures, functions) that allow variable number of parameters.

Not surprisingly, the statements inside the module will be executed in the order in which they are written within a module definition, top to bottom, in sequence. The declarations, such as `var`, and directives, such as `import`, are all processed before the module starts executing, thus making the respective data elements available and visible to all statements inside the module.

Consequently, the order of declarations within a module does not matter; the order of statements – does, as is illustrated by the following example:

```
project Test04
```

```
  WriteLn("The answer is: ",X)

  var X := TheUltimateAnswer
  const TheUltimateAnswer = 42
  import System.IO.Console

  WriteLn('What was the question?')
end
```

Notice how the declarations above appear in exactly the opposite order in which you would expect them in a Pascal program. Incidentally, this example illustrates the use of a `const`-declaration to declare a constant; a `Number` constant in this case. Also note the way strings are delimited in WANTSCript. You can use either double, or single quotes to delimit a string; more on this later.

Invoking Modules

To invoke a module means to call it like a procedure. In WANTScript, all modules are callable subroutines that can be invoked by referring to their name whenever a statement is allowed. For example:

```
module Test06

  Test1
  Test2

  module Test1
    WriteLn("Test1 called");
  end

  module Test2
    WriteLn("Test2 called");
  end

end
```

The two nested modules, named `Test1` and `Test2`, are invoked inside the module `Test06`, just like procedures would be. In fact, given that you can use the keywords `program` and `procedure` as an alias for `module` in this context, the preceding example can be re-written in the following way, to make it more intuitive to a Pascal programmer:

```
program Test06

  procedure  Test1
    WriteLn("Test1 called");
  end

  procedure Test2
    WriteLn("Test2 called");
  end

  Test1
  Test2

end
```

The output of both versions of this program is – predictably – as follows:

```
Test1 called
Test2 called
```

Not-so-obvious to a Pascal programmer is the fact that one can also invoke the top-level module Test06 just like a procedure. This is akin to calling a unit directly by name, which is not allowed in Pascal (the code of a unit – its initialization section - is automatically called by the runtime).

```
program ExerciseTest06

  import Test06

  Test06

end
```

This program produces exactly the same output as before:

```
Test1 called
Test2 called
```

The difference with the previous example is that the module `Test06` is called as a procedure from within `ExerciseTest06` – it is no longer used as a top-level module, but as a callable subroutine.

In general, in WANTScript, any module can potentially be called as a subroutine. Calling a module as a subroutine runs its statements, if any.

A top-level module run from the command line or IDE is simply being called like a subroutine by the WANT Runtime Engine.

Module Parameters

Just like procedures in Pascal, or functions in C/C++, modules can accept parameters. The following example illustrates a module (procedure `Test`) with parameters that are passed by-value:

```
program ParamTest
//Test008

  Test("Calling Test the first time")
  Test("Calling Test the second time")

  procedure Test(Msg)
    WriteLn(Msg);
  end

end
```

The output of this program is:

```
Calling Test the first time
Calling Test the second time
```

Notice that the formal parameter `Msg` of the `Test` procedure is declared without specifying its type. In general, you do not specify the type of a variable or a parameter in their declaration in WANTScript.

Multiple parameters can be declared in the normal way:

```
procedure TestProc(A, B, C)
  WriteLn("A=",A, " B=", B, " C=", C)
end
```

Multiple arguments in a call to such a procedure are handled in the usual way: actual arguments are bound by position with formal parameters; thus the first argument is bound to the first formal parameter; the second argument, to the second formal parameter; etc.

There is also an option of binding arguments to formal parameters explicitly by name, in which case they can be listed in any order. Here is an example of arguments bound by name:

```
program ArgumentsByName
//Test009

  //Normal binding, by position
  TestProc("ArgA","ArgB","ArgC")

  //Bind all by name
  TestProc(B := "ArgB", C:= "ArgC", A := "ArgA")

  //Bind some by position, some by name
  TestProc("ArgA", C := "ArgC", B := "ArgB")
  TestProc("ArgA", B := "ArgB", C := "ArgC")
  TestProc("ArgA", "ArgB", C := "ArgC")

  //Invalid: Unnamed argument follows a named one
  //TestProc("ArgA", B := "ArgB", "ArgC")

  //Illegal: Binding the same argument multiple times
  //TestProc("ArgA",C:="ArgC",B:="ArgB",A:= "Arg1")

  procedure TestProc(A, B, C)
    WriteLn("A=",A, " B=", B, " C=", C)
  end

end
```

The rule is that all arguments bound by position must be listed first, followed by arguments bound by name. Once an argument explicitly bound by name appears in the argument list, all remaining (unbound) parameters must be bound by name. An attempt to violate this rule will result in a "too many actual arguments" compile-time error.

A formal parameter can be bound only once to an actual argument. An attempt to bind multiple times to a parameter is an error. That's why the other commented-out line in the example is illegal: it attempts to bind the same parameter A first by position, then by name.

Value versus Reference Parameters

Arguments to modules in WANTScript can be passed by value, or by reference. Unless otherwise specified, arguments in WANTScript are passed by value, i.e. a copy of the actual argument's value is made for use inside the module and any modifications to the formal parameter of the module will only have localized effect within that module.

To define an argument as being passed by reference, you must prefix the argument declaration with one of the following keywords:

- `var`
- `ref`
- `out`
- `output`

All four of these keywords have the same effect of declaring the parameter to be passed by reference.

If an argument is passed by reference, any changes to the formal parameter bound to that argument will be reflected outside the called module after it is finished with the call. This is the same behavior as `var`-parameters in Pascal, or reference parameters in C++.

For example:

```
program ParamsByReference

  var X = 0

  WriteLn("X=",X);
  Increment(X,2)
  WriteLn("X=",X);
  Decrement(X)
  WriteLn("X=",X);
  Mult(X,2)
  WriteLn("X=",X);
  Divide(X, 2)
  WriteLn("X=",X);


  procedure Increment(var P, IncStep)
    P := P + IncStep
  end

  procedure Decrement(out P, DecStep)
    P := P - DecStep
  end

  procedure Mult(ref P, AFactor)
    P := P * AFactor
  end

  procedure Divide(output P, ADivisor)
    P := P / ADivisor
  end

end
```

The output of this program is:

```
X=0
X=2
X=1
X=2
X=1
```

What happens in this case is that the variable `X` is bound to the formal parameter `P` in either call to the Increment and Decrement subroutine module, whereby its value gets modified inside the respective subroutine.

The CALL Statement

So far, all the examples of passing arguments into subroutines that you have seen followed the traditional “function”, or “expression” notation of listing the actual arguments in parentheses following the target subroutine name. This notation allows you to include the function call as part of a larger expression.

For example, this is the expression-notation call to a function named Fib():

```
N := ( Fib(10) + 5 )/ 2
```

The call to the `Fib()` function, with a single parameter – the numeric literal 10 – fits nicely inside a more extensive arithmetical expression.

The notation of passing arguments using parentheses is very widespread and supported by most programming languages. It should certainly be nothing new to a Pascal, or C/C++, or Java programmer.

The drawback of this notation is that it becomes more difficult to read and understand when the number of parameters increases.

In WANTScript, there is an alternative notation – the **`call-with`** statement – that is much more readable, even with larger number of arguments being passed. The price to pay for this readability is a slight increase in verbosity.

Consider the built-in module `WANT.FileTasks.Exec` implicitly defined inside the WANT runtime as follows:

```
module WANT.FileTasks.Exec(
  Executable,
  Arguments = "",
  Dir = "",
  FailOnError = True,
  ErrorLevel = 0
)
  ...
end
```

Given the above declaration of the `Exec` function, following is an example of a **`call-with`** statement used instead of the traditional expression-notation:

```
call WANT.FileTasks.Exec with
  Executable := "brc32.exe"
  FileName := "want.rc"
  FailOnError := False
end
```

This example shows a call to the `WANT.FileTasks.Exec` function with three arguments, which is exactly equivalent to the following, more traditional call statement:

```
WANT.FileTasks.Exec(Executable := "brc32.exe",
  FileName := "want.rc", FailOnError := False)
```

which, in turn, is equivalent to the following statement:

```
WANT.FileTasks.Exec("brc32.exe", "want.rc",
  "", False)
```

It all boils down to how verbose – or self-documenting – you want your code to be. All three call statements generate essentially the same code. The only difference is that in the third and last example, since the arguments are bound to the formal parameters by position, you have to include an empty string as the third argument, `Dir`, to allow a non-default value `False` to be passed and bound to the fourth parameter, `FailOnError`.

Calling Functions Using a Call Statement

One detail that was omitted in the preceding section was the issue of how you capture the value returned from a function called using the **`call-with`** statement.

In the case of normal expression-notation, you simply assign the value returned from the function by placing the entire call on the right-hand side of an assignment statement:

```
N := Fib(10)
```

You cannot simply replace the call with a **`call-with`** statement, because the **`call-with`** statement is a statement, not an expression. Instead, what you need to do, if you choose to use the call-with statement notation, is to use a **`call-set-with`** statement as follows:

```
Var N
call Fib set N with
  10
end
```

The reserved word set that follows the name of the function to call indicates to the interpreter that you wish to assign the value returned by the function to the indicated variable, `N` in this case.

So, the previous example of executing an external program could be written as follows:

```
call WANT.FileTasks.Exec set ExeCode with
  Executable := "brc32.exe"
  FileName := "want.rc"
  FailOnError := False
end
```

In this case, the result of the Exec call will be stored in the ExeCode variable. The equivalent “traditional” notation for this would be:

```
ExeCode := WANT.FileTasks.Exec(
  Executable := "brc32.exe"
  FileName := "want.rc"
  FailOnError := False
)
```

Again, both of these are equivalent, except that the latter call may be used as part of an expression, and the former one may not.

Parameter Declarations Inside a Module

An oddity of WANTScript, perhaps, is an alternative way of declaring module parameters, reminiscent of K&R C:

```
//Test015
procedure Build
  input Folder
  input File
  input Description = ""

  WriteLn("Folder      : ", Folder)
  WriteLn("File        : ", File)
  WriteLn("Description: ", Description)
end
```

Instead of declaring them inside a pair of round parentheses, you can declare them anywhere in the statement block, prefixed with the keyword **`input`** (for parameters passed by value), or any one of: **`output`**, **`out`**, or **`ref`**, for parameters passed by reference. Note that the keyword **`in`** cannot be used in this context.

The procedure `Build` can be called like a function, or with a **`call`** statement, just like any other:

```
Build("d:\projects\delphi7\want2\","want.dpr")

call Build with
  File := "want.dpr"
  Folder := "d:\projects\delphi7\want2\"
end
```

Here is another example of a module (procedure `GetInput`) that declares its parameters in the statement block, rather than in parentheses. This time, the procedure also declares an output-parameter (a parameter passed by reference):

```
procedure GetInput //Test023
  input Prompt
  output Value

  Write(Prompt)
  Value := ReadLn
end

var S

call GetInput with
  "Please, enter some text: "
  S
end
WriteLn("You entered: '",S,"'")
```

Here is the output produced by this script:

```
Please, enter some text: WANTScript is fun!
You entered: 'WANTScript is fun!'
```

This, admittedly somewhat archaic, style of parameter declaration helps write more readable programs in a more declarative style, if used responsibly. Of course, this feature can easily be abused by scattering parameter declarations throughout the entire module block, interspersed with executable statements (not recommended).

Default Parameters

WANTScript allows defining default parameters for a module, that is, parameters with default values.

You can give a default value to a parameter of a module and then call the module with or without the parameter, thus making it optional. To provide a default value, end the parameter declaration with the `=` (equals) symbol followed by a constant expression.

For example, given the module declaration

```
procedure TestDefParams(A, B, C = 3)
  WriteLn("A=",A," B=",B," C=",C)
end
```

all the following calls are equivalent

```
TestDefParams(A := 1, B := 2)
TestDefParams(1, B := 2)
TestDefParams(1, 2, 3)
TestDefParams(1, 2)
TestDefParams(A := 1, B := 2, C := 3)
```

and each produces the following output:

```
A=1 B=2 C=3
```

Note that parameters with default values must occur at the end of the parameter list, and must be passed by value. In other words, once a default value for a parameter is provided in the list of module parameters, all remaining parameters must have default values. Parameters passed by reference cannot have default values.

When calling modules with more than one default parameter, you cannot skip parameters, like in Visual BASIC. The following example illustrates this:

```
program DefaultParams
//Test013

  TestAllDefParams
  TestAllDefParams(4)
  TestAllDefParams(4,5)
  TestAllDefParams(4,5,6)

  //These are invalid (can't skip parameters):
  //TestAllDefParams(4,,6)
  //TestAllDefParams(4,5,)

  procedure TestAllDefParams(A = 1, B = 2, C = 3)
    WriteLn("A=",A," B=",B," C=",C)
  end

end
```

The commented-out lines would be illegal and generate an "invalid syntax" interpreter error.

Command Line Arguments

Any module can have parameters. This statement applies even to top-level modules and, in particular, to the main module named on the command line to execute when invoking WANT.

When you invoke WANT, you have to give it a command line parameter that indicates the name (and optionally, the path) of the script file that you want WANT to execute. For example, if you type

```
WANT d:\scripts\Test06
```

on the command line, WANT will execute the `Test06.want` script file (the .want extension is assumed by default).

Additionally, if the module to be executed defines parameters, you can also include them on the command-line, using similar rules as you would be using when passing parameters within a script. For example, consider the following module:

```
//test090
program StringParameters(
  Message = "",
  Prompt = "DisplayMessage")

  DisplayMessage(Prompt,Message)

  procedure DisplayMessage(Prefix, Text)
    WriteLn(Prefix,": ",Text)
  end

end
```

When executed as the main module, this program prints two pieces of text: a `Prompt`, followed by a `Message`. The message to print and the prompt to display are parameters to the main module, in this case defined as optional. You can run this module without supplying any parameters at all, but if you choose to supply them, you can do so in one of the following ways (this is not an exhaustive list of possibilities, just some examples):

```
want test090 42 "The answer is"
want test090 Message=42 Prompt="The answer is"
want test090 Prompt="The answer is" Message=42
```

In either of these cases, the value `42` gets bound to the `Message` parameter, and the string `"The answer is"` – to the `Prompt` parameter, producing the same answer in each case:

```
The answer is: 42
```

which, of course, as we all know, is the ultimate answer.

Nesting of Modules

As mentioned, modules can be embedded (nested) inside other modules. The following example illustrates how a module definition can be contained inside other module definitions. The example illustrates four levels of nesting:

```
module Test050
  Test050.Outer.Inner1.Inner2.Inner3("Hello!");

  module Outer
    module Inner1
      module Inner2
        module Inner3(Msg)
          System.IO.Console.WriteLn(Msg);
        end
      end
    end
  end
end
```

The above example also shows how to call a deeply embedded module. You can simply call it by using its fully qualified name, passing any required parameters in the usual way.

Multi-File Projects

A WANT project is not limited to a single script file. You may want to organize your code into multiple files. Perhaps you collect a set of related, reusable procedure modules together and call the collection a “library”. You can then use the library procedures from other scripts by making them available via an import directive.

An import directive is the technique to access the scope of another module. When you import another module into your current project, you can access the definitions contained in the imported module.

TODO: imports, namespaces

TODO: What if two modules declare the same identifier? Explain fully qualified names

Variables

A variable is a "container" for information you want to store. A variable's value can change during the execution of a script. You can refer to a variable by name to see its value, or to change its value.

A variable name must begin with a letter or underscore, followed by zero or more letters, digits, or underscores. Variable names cannot contain a period “.”.

You must declare all variables that are used in a program.

An ordinary (that is, automatic; see below) variable is declared using the keyword `var`, followed by one or more variable names, separated by commas:

```
var <Name1> [, <NameN>]
```

Here are some concrete examples of variable declarations:

```
var FirstName, Salary, Department
var Day, Month, Year
var I
```

You do not declare the types of variables explicitly in WANTScript. The runtime infers the types of variables from their most recent use.

Variables in WANTScript can be initialized at the time they are declared. To initialize a variable within its declaration, follow the name with either an assignment operator (`:=`), or with an equal-sign (`=`):

```
var <Name1> [:= <Value1>] [, <NameN> [:= <ValueN>]]
var <Name1> [= <Value1>] [, <NameN> [= <ValueN>]]
```

The values you assign when initializing a variable (`Value1`, … `ValueN`) must be computable at the time the module containing the variable is activated.

You can mix the `=` symbol and the `:=` operator when initializing variables within the same `var`-declaration.

Here are some examples:

```
var Day = 1, Month := 1, Year = 2008
var I = 0
var J := 1
```

Variables in WANTScript are automatically initialized to “zero”/False/empty-string/nil even if you don’t explicitly initialize them.

This means that you can use in an expression a variable that was not explicitly initialized or assigned to and will produce predictable results. This is contrast with Pascal or C, where an uninitialized variable contains “garbage”, or undefined value.

Here is an example:

```
program Test027_AutoInitVars
//Variables are automatically
//initialized to "zero"/""/"false"

  var X
  X := X + 1
  WriteLn(X)

  var S
  S := S + "XYZ"
  WriteLn(S)

  var B
  if B then
    WriteLn("True")
  else
    Write("False")
  end

end
```

The output of this program is consistent with an uninitialized variable being zero, empty string, or False, respectively:

```
1
XYZ
False
```

Static versus Automatic Variables

There are two types of variables in WANTScript: automatic, and static.

An automatic variable is available only when its containing module is active or instantiated. A module is active when its code is executing; a module gets instantiated when one of its constructors is called.

- **Activation by a Call**
  In case of module activation via a call, the automatic variables of a module are allocated on the runtime stack at the time of the call, and de-allocated just before the control returns to the caller. In that regard, they act analogously to local variables of a Pascal procedure or function.

- **Instantiation by Constructor**
  If a module is instantiated via a call to its constructor, the automatic variables are allocated on the heap as data members of the instance. In that regard they are analogous to fields in a Pascal class. They persist as long as the object that contains them persists, and they vanish when the object is destroyed.

Static variables, also known as class variables or module variables, on the other hand, are allocated when the module that contains them is first loaded by the Runtime. In that respect, they are analogous to class variables in Pascal, or static variables in C/C++/C#. They persist throughout the entire project run and are deallocated only when the module that contains them is unloaded, at the end of the run.

Static variables are declared with one of the following three keywords: `staticvar`, `classvar`, `modulevar`. Otherwise, the syntax is exactly the same as in the case of automatic variable declarations:

```
staticvar <Name1> [, <NameN>]
classvar  <Name1> [, <NameN>]
modulevar <Name1> [, <NameN>]
```

Static variables can also be initialized at the time when they are declared. Care must be taken to initialize them with constant-expressions only, or in other words, with expressions that are computable before the program actually starts running.

Here are some examples of static variable declarations:

```
modulevar M = 1
classvar C = 2
staticvar S = 3, X, Y, Z := 7
```

The variables declared and initialized in this way retain their values throughout the execution of a project (unless re-assigned during the execution, of course, in which case they retain their new values until re-assigned again, or until the program terminates).

Constants

Constants are symbolic names for certain key values in a program to make programs more readable for humans.

Constants keep the same value throughout the execution of a program. At runtime, the Runtime Engine replaces all references to a constant with the constant value itself throughout the program.

You declare a constant using the keyword `const`:

```
const <Name1> = <ValueExpr>
```

Here are some examples of constant declarations:

```
const PI = 3.14;
const DaysInAYear = 365 + 1;
const SecondsInADay = 60*60*24;
const PressKey = "Press any key to continue..."
```

The rules for naming a constant are the same as those for naming a variable (start with a letter or underscore, followed by letters, digits, and underscores).

As illustrated above, certain arithmetical and string operations are allowed when defining constants, as long as the values can be computed at compile-time. No calls to functions or references to variables or parameters are permitted in a constant declaration; only references to other constants and literals are allowed.

You cannot assign values to constants. The following expression is therefore illegal, given the declaration that precedes it:

```
const DaysInAYear = 365;
DaysInAYear := 366; //Invalid
```

Statements

The purpose of most useful programs is to carry out some action. WANT was originally designed as a build automation tool, and the most common types of actions coded in WANTScript are those related to creating software builds.

WANTScript actions are coded as statements. There are several different types of statements one can use in a WANTScript program:

- Assignment statement
- Call statement
- Loop statement
- Conditional statement
- Return statement
- Depends statement

Assignment Statement

The most common simple statement is an assignment statement. An assignment statement syntax is:

```
<LeftHandSide> := <Expression>
```

The assignment operator `:=` evaluates the expression on its right-hand side and assigns the computed value to the variable specified on the left-hand side.

You have already seen many examples of assignments in this tutorial. Here are some more:

```
X := X + "D"
S1 := "The Question"
X := X + 1.2
M := Cos(X) + I*Sin(X)
```

Assigning a value to a variable changes that variable's type. So, for example, given:

```
X := 1
X := "A"
```

the first assignment sets the variable `X` to be of type `System.Number`, and to have the value of `1;` the subsequent assignment, changes the type of the variable X to `System.String`, and sets its value to the string `"A"`.

Unlike in statically-typed languages like Pascal or Java, a variable's type in WANTScript depends solely on the most recently assigned value. It is therefore possible for a variable to change its type – based on the most recently assigned value – many times throughout the execution of a program.

CALL Statements

As a programmer, you know about at least one way of invoking (calling) subroutines: listing their name, followed by any comma-separated parameters in parentheses. This is how it is done in most programming languages, and it is what you could call a "normal" or "standard" subroutine call notation.

In WANTScript, “normal” call style is also supported and works with either “procedures” (modules that don’t return a value) or “functions” (modules that do return a value).

For example, this is a call to the standard procedure `WriteLn` with a single string argument:

```
WriteLn("Hello User!")
```

If a module does not define any required parameters, (empty) parentheses are not necessary (but are permitted) following the module name. For example:

```
WriteLn()
```

CALL-WITH Statements

The standard function-call notation is not the only way to invoke a module in WANTScript, however. An alternative is to use an explicit **`call-with`** statement.

A **`call-with`** statement provides special syntax to invoke modules. The complete syntax of the **`call-with`** statement is as follows:

```
call <ModuleName> set <VariableName> with
  <argument-list>
end
```

Here is an example of calling the same module using two different notational styles:

```
project TestCallWith //test028

  procedure Test(Param1, Param2, Param3 = "KLM")
    WriteLn("Param1=",Param1)
    WriteLn("Param2=",Param2)
    WriteLn("Param3=",Param3)
    WriteLn
  end

  Test("First Call", 1, "XYZ")

  call Test with
    "Second Call"
    2
    "XYZ"
  end

  call Test with
    Param3 := "XYZ"
    Param1 := "Third Call"
    Param2 := 3
  end

  Test(Param1 := "Fourth Call", Param2 := 4)

  call Test with
```

```
    Param1 := "Fifth Call"
    Param2 := 5
  end

end
```

This is the output produced by the program:

```
Param1=First Call
Param2=1
Param3=XYZ

Param1=Second Call
Param2=2
Param3=XYZ

Param1=Third Call
Param2=3
Param3=XYZ

Param1=Fourth Call
Param2=4
Param3=KLM

Param1=Fifth Call
Param2=5
Param3=KLM
```

Looking at the code of the program, the module being called is defined as a procedure `Test` taking three parameters, the third of which is optional. The same procedure is then called using several different styles of programming:

- A function-like call with all parameters bound by order of appearance in the calling list, i.e. by position:

  ```
  Test("First Call", 1, "XYZ")
  ```

- A **`call-with`** statement, in which the three parameters are also bound by position:

  ```
  call Test with
    "Second Call"
    2
    "XYZ"
  end
  ```

- A **`call-with`** statement in which the parameters are bound explicitly by name:

  ```
  call Test with
    Param3 := "XYZ"
    Param1 := "Third Call"
    Param2 := 3
  end
  ```

- A traditional function-call, in which the parameters are bound explicitly by name, and only required parameters are passed:

```
Test(Param1 := "Fourth Call", Param2 := 4)
```

- A `call-with` statement in which the parameters are bound explicitly by name and only required parameters are passed:

```
call Test with
  Param1 := "Fifth Call"
  Param2 := 5
end
```

CALL-SET-WITH Statements

The call-set-with statement is an extension of the call-with statement that allows the caller to capture the value returned by the called subroutine, if any.

Following is an example of using a `call-set-with` statement, contrasted with the “normal” function call:

```
project TestCallWith //test029
  var TestResult

  function Test(ATerm1, ATerm2, AFactor = 1)
    return ATerm1 + ATerm2*AFactor
  end

  TestResult := Test(3,2,1)
  WriteLn('TestResult =', TestResult)

  call Test set TestResult with
    3
    2
    1
  end
  WriteLn('TestResult = ', TestResult)

  call Test set TestResult with
    AFactor := 1
    ATerm2 := 2
    ATerm1 := 3
  end
  WriteLn('TestResult = ', TestResult)

  TestResult := Test(ATerm2 := 2, ATerm1 := 3)
  WriteLn('TestResult = ', TestResult)

  call Test set TestResult with
    ATerm2 := 2
    ATerm1 := 3
  end
  WriteLn('TestResult = ', TestResult)
end
```

The program produces the following output:

```
TestResult = 5
TestResult = 5
TestResult = 5
TestResult = 5
```

```
TestResult = 5
```

As you can see, all five calls are exactly equivalent, producing the same result in each case. The result of each call is stored in the variable `TestResult` and is equal to `5` after every call.

Looking again at the code, a function `Test` is defined to take three numeric parameters: `ATerm1`, `ATerm2`, and `AFactor`.

There are five calls to the function `Test`, as follows:

- A traditional function call using an expression-notation, passing all three actual arguments by position:

```
TestResult := Test(3,2,1)
```

- A call-set-with statement, also passing all three actual arguments by position:

```
call Test set TestResult with
  3
  2
  1
end
```

- A call-set-with statement passing all three actual parameters explicitly by name:

```
call Test set TestResult with
  AFactor := 1
  ATerm2 := 2
  ATerm1 := 3
end
```

- A traditional function call, passing only the two required parameters by name:

```
TestResult := Test(ATerm2 := 2, ATerm1 := 3)
```

- A call-set-with statement, passing only the two required parameters by name:

```
call Test set TestResult with
  ATerm2 := 2
  ATerm1 := 3
end
```

As mentioned, the same result is produced in every case.

Loops

There are three kinds of loop constructs available in WANTScript right now: **`WHILE`**-loop, **`REPEAT`**-loop, and **`FOR`**-loop. [**TODO: foreach**] They generally correspond to the respective constructs in the Object Pascal programming language as implemented in Delphi.

One difference with Pascal is that the `FOR`-loop has an optional `step` clause, more in line with BASIC.

Interestingly, the keywords `loop`, `do`, or `repeat` are synonyms, so that `while-do` is equivalent to `while-loop` or `while-repeat`, `for-do` is equivalent to `for-loop` or `for-repeat`, and `repeat-until` is equivalent to `loop-until` or `do-until`.

In other words,

- `loop` can appear whenever `do` or `repeat` can;
- `do` can appear whenever `repeat` or `loop` can;
- `repeat` can appear whenever `do` or `loop` can.

WHILE Loop

A while-loop is a simple loop that keeps iterating *while* something is true. The condition is checked at the beginning of the loop, hence the statements inside the loop may be executed zero or more times.

The wile-loop is constructed in one of the following ways:

```
while <condition> do
  <statements>
end
```

or

```
while <condition> loop
  <statements>
end
```

or

```
while <condition> repeat
  <statements>
end
```

Here is an example of a simple WANTScript project that uses a while-loop:

```
project Test181_WHILELoop

  var I

  "WHILE-DO
  I := 0
  while I < 10 do
    i++
    Write(I," ")
  end
```

```
  "

  "WHILE-LOOP
  I := 0
  while I < 10 loop
    i++
    Write(I," ")
  end
  "

  "WHILE-REPEAT
  I := 0
  while I < 10 repeat
    i++
    Write(I," ")
  end
  "

end
```

The output is:

```
WHILE-DO
1 2 3 4 5 6 7 8 9 10
WHILE-LOOP
1 2 3 4 5 6 7 8 9 10
WHILE-REPEAT
1 2 3 4 5 6 7 8 9 10
```

UNTIL-Loop

The `until`-loop is a loop that evaluates its condition at the end and continues iterating *until* the condition becomes true. Since the loop-condition is checked at the end of the block comprising the loop, the statements inside the loop will be executed at least once.

The syntax for the `until`-loop is:

```
repeat
  <statements>
until <condition>
```

Alternatively:

```
loop
  <statements>
until <condition>
```

As yet another alternative:

```
do
  <statements>
until <condition>
```

Here is an example of using the `until`-loop in all three of its incarnations:

```
project Test182_UNTILLoop
```

```
  var I

  "DO-UNTIL
  I := 0
  do
    i++
    Write(I," ")
  until I >= 10
  "

  "REPEAT-UNTIL
  I := 0
  repeat
    i++
    Write(I," ")
  until I >= 10
  "

  "LOOP-UNTIL
  I := 0
  loop
    i++
    Write(I," ")
  until I >= 10
  "

end
```

The output produced by this program is:

```
DO-UNTIL
1 2 3 4 5 6 7 8 9 10
REPEAT-UNTIL
1 2 3 4 5 6 7 8 9 10
LOOP-UNTIL
1 2 3 4 5 6 7 8 9 10
```

FOR-TO-Loop

A `for`-loop iterates a predefined number of times. In its most basic form, the `for`-loop is structured as follows:

```
for <CounterVar> := <StartValueExpr>
  to <EndValueExpr> do
  <statements>
end
```

A for-`to`-loop requires that a counter variable be declared either outside the loop, in the scope of the enclosing module, or as part of the loop statement.

To declare a counter variable inline, as part of the `for`-loop statement, follow the initial keyword `for` with the keyword `var`. Note that variables declared as part of a `for`-loop are created in the scope of the enclosing module that contains the loop and survive the execution of the loop that declared them.

For example, the following **for**-loop both declares and initializes its counter variable `I` in a single statement:

```
for var I := 0 to 10 do
  WriteLn(I)
end
```

The counter variable of a **for**-loop can be initialized with an arithmetic expression (`StartValueExpr`). The end value can also be specified via an arithmetic expression (`EndValueExpr`).

The counter is incremented by one each time through the loop (logically, at the end of the loop), then checked against the end-value specified by the to-expression (at the beginning of the loop iteration), and the loop enters its next iteration if the counter hasn't yet reached or *exceeded* its end value.

Given that the keyword **do** can be replaced with either **loop** or **repeat**, the syntax gives rise to three alternatives, as illustrated in the following example:

```
project Test183_FORLoop

  var I

  "FOR-DO
  for I := 0 to 10 do
    Write(I," ")
  end
  "

  "FOR-LOOP
  for I := 0 to 10 loop
    Write(I," ")
  end
  "

  "FOR-REPEAT
  for I := 0 to 10 repeat
    Write(I," ")
  end
  "

end
```

The output:

```
FOR-DO
0 1 2 3 4 5 6 7 8 9 10
FOR-LOOP
0 1 2 3 4 5 6 7 8 9 10
FOR-REPEAT
0 1 2 3 4 5 6 7 8 9 10
```

The for-to loop works by first initializing the counter variable to the specific value, and then incrementing the counter by one upon each iteration of the loop, until the end value is reached or exceeded.

This means that the initial value of the loop counter must be numeric, or else the attempt at incrementing the counter will fail. It also means that the loop may not execute at all, if the end value is less than the starting value of the counter.

FOR-DOWNTO Loop

Like in Pascal, there is also an alternative syntax counting the loop variable down instead of up, i.e. *decrementing* the value by one each time, instead of incrementing it:

```
for <CounterVar> := <StartValueExpr>
  downto <EndValueExpr> do
  <statements>
end
```

Again, the keyword `do` can be substituted by either `loop`, or `repeat`. The loop will be entered if the starting value is greater than the end value of the counter, and will continue until the counter reaches the end value.

Both, the `StartValueExpr` and the `EndValueExpr`, are arithmetic expressions that must evaluate to a numeric result.

Non-Integer Loop Counters

Note that nothing in the preceding prevents the loop counter from taking non-integer values:

```
  for I := 10.5 downto 0.5 do
    Write(I," ")
  end
```

This happily produces:

```
10.5 9.5 8.5 7.5 6.5 5.5 4.5 3.5 2.5 1.5 0.5
```

As long as the value of the loop counter is numeric and thus can be incremented/decremented, the loop will proceed as normal.

> Both a `for-to` and a `for-downto` loop will terminate if the counter passes *through* the end value, even if it never exactly *equals* the end value.

Even if the end-value is never exactly equal to the current value of the counter, the `for`-loop terminates if the counter reaches or exceeds the end-value in the case of a `for-to` loop, or when the counter gets below the end value, in the case of a `for-downto` loop.

Here is an example:

```
project Test187_InfiniteFORLoopAttempt
  import System.Utilities
```

```
  var I

  //This loop is NOT really infinite
  for I := 0 to 9.5 do
    Write(PadLeft(I,2))
  end
  WriteLn

  //Neither is this one
  for I := 0.5 to 9.5 do
    Write(PadLeft(I,4))
  end
  WriteLn

end
```

Both of the loops in the example above terminate when the counter `I` reaches or exceeds the value `9.5`.

The output of this program is:

```
 0 1 2 3 4 5 6 7 8 9
 0.5 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5
```

This is not to say that a `for`-loop is always guaranteed to terminate after a finite number of iterations. If you work at it, you can create an infinite `for`-loop.

For example, it is possible, although not advisable, to modify the value of the loop counter variable inside the loop:

```
for var I := 0 to 10 do
  I++
  WriteLn(I)
end
```

Don't try this at home, but if – instead of the increment operator ++ being used inside the loop – a decrement operator were used, the loop would never terminate.

It is also possible to create an infinite `for`-loop using the technique from the next section. Typically, you don't want to implement infinite loops, and when you do, a `for`-loop is a poor choice for implementing them, so consider yourself forewarned.

FOR-Loop with a STEP Clause

A `for`-loop is not limited to an increment or decrement exactly by (the integer) one. You can specify an almost arbitrary numeric value as the loop increment by using the `for-to-step` or `for-downto-step` syntax. The following syntax is thus the most complete description of the for-loop structure:

```
for <Counter> := <StartValueExpr>
```

```
  to <EndValueExpr> step <StepValueExpr> do
  <statements>
end
```

```
for <Counter> := <StartValueExpr>
  downto <EndValueExpr> step <StepValueExpr> do
  <statements>
end
```

In each case, the optional step-clause containing a step-expression (`StepValueExpr`) determines the amount added to (or subtracted from) the loop counter variable after each iteration. The **`step`**-expression must evaluate to a *positive* numeric value.

It is important that the step value is a positive number because otherwise, if the step-increment is negative, you will create an infinite loop that counts “the wrong way”.

Likewise, if the step-expression evaluates to zero, the loop will not be counting at all – it will potentially iterate indefinitely, without changing the counter at all.

TODO: Throw an exception if step < 0?

Here is an example of properly using a **`for-to-step`** and a **`for-downto-step`** loop:

```
project Test184_FORLoopStep

  var I

  "for I := 0 to 10 step 2 do
  for I := 0 to 10 step 2 do
    Write(I," ")
  end
  "

  "for I := 10 downto 0 step 2 do
  for I := 10 downto 0 step 2 do
    Write(I," ")
  end
  "

  "for I := 0 to 10 step 3 loop
  for I := 0 to 10 step 3 loop
    Write(I," ")
  end
  "

  "for I := 10 downto 0 step 3 repeat
  for I := 10 downto 0 step 3 repeat
    Write(I," ")
  end
  "

end
```

The output of this program is:

```
for I := 0 to 10 step 2 do
0 2 4 6 8 10
for I := 10 downto 0 step 2 do
10 8 6 4 2 0
for I := 0 to 10 step 3 loop
0 3 6 9
for I := 10 downto 0 step 3 repeat
10 7 4 1
```

Loop Example: Floyd's Triangle

Here is a more extensive and complete example of how to use various loops in WANTScript:

```
project FloydsTriangle //Test190

{
Loops. WHILE-DO, FOR-TO, REPEAT-UNTIL
}

  WriteLn("Floyd's Triangle using WHILE loop")
  FloydsTriangleWHILE(5)
  WriteLn
  WriteLn("Floyd's Triangle using FOR loop")
  FloydsTriangleFOR(5)
  WriteLn
  WriteLn("Floyd's Triangle using UNTIL loop")
  FloydsTriangleUNTIL(5)

   {
    Floyd's triangle is a right angled triangular
    array of natural numbers. It is named after
    Robert Floyd. It is defined by filling the
    rows of    the triangle with consecutive numbers,
    starting with a 1 in the top left corner:

         1
         2  3
         4  5  6
         7  8  9  10
         11 12 13 14 15

    The numbers along the right edge of the triangle
    are the triangular numbers.

    Source:
      www.wikipedia.org/wiki/Floyd%27s_triangle
   }


  import System.Utilities

  procedure FloydsTriangleFOR(MaxRows)
    var Row, Column
    var Number

    Number = 0
    for Row := 1 to MaxRows loop
```

```
      for Column := 1 to Row do
        Number++
        Write( PadLeft(Number,3) )
      end
      WriteLn
    end
  end


  procedure FloydsTriangleWHILE(MaxRows)
    var Row, Column
    var Number

    Number = 0
    Row := 1
    while Row <= MaxRows loop
      Column := 1
      while Column <= Row do
        Number++
        Write( PadLeft(Number,3) )
        Column++
      end
      WriteLn
      Row++
    end
  end


  procedure FloydsTriangleUNTIL(MaxRows)
    var Row, Column
    var Number

    Number = 0
    Row := 1
    loop
      Column := 1
      repeat
        Number++
        Write( PadLeft(Number,3) )
        Column++
      until Column > Row
      WriteLn
      Row++
    until Row > MaxRows
  end

end
```

Here is the output:

```
Floyd's Triangle using WHILE loop
  1
  2  3
  4  5  6
  7  8  9 10
 11 12 13 14 15

Floyd's Triangle using FOR loop
  1
  2  3
  4  5  6
  7  8  9 10
 11 12 13 14 15
```

```
Floyd's Triangle using UNTIL loop
  1
  2  3
  4  5  6
  7  8  9 10
 11 12 13 14 15
```

The example runs three versions of the same algorithm, one that produces the first five rows of the Floyd's triangle. Each version of the algorithm is implemented with a different type of loop:

- `FloydsTriangleFOR` uses two **`FOR`**-loops to produce the output;
- `FloydsTriangleWHILE` uses two **`WHILE`**-loops to do the same thing;
- `FloydsTriangleUNTIL` uses two **`UNTIL`**-loops to produce the exact same output.

Conditional Statements

There are two types of conditional statements in WANTScript: the **`IF`**-statement , and the **`SWITCH`** statement, a.k.a. the **`SELECT`**-statement.

IF-THEN Conditional Statement

The keyword `elseif` is short for "else if", and is useful to avoid excessive indentation.. Theer can be zero or more elseif parts, and the else part is optional.

```
var A = 0

if A > 0 then
  WriteLn("A is positive"
elseif A = 0 then
  WriteLn("A is ZERO")
else
  WriteLn("A is negative")
end
```

SWITCH/SELECT Conditional Statement

The keyword **`switch`**, which is equivalent to the keyword **`select`**, introduces a multi-branch conditional statement. The two keywords, **`switch`** and **`select`**, can be used interchangeably.

```
switch Selector
  case 1, 2, 3:
    WriteLn("1,2,3")
  case 4, 5, 6:
    WriteLn("4,5,6")
  else
    WriteLn("other")
end
```

The colon at the end of each `case`-clause is optional and may be omitted. Alternatively, and also optionally, it can be replaced with a semicolon.

The preceding switch-statement is exactly equivalent to the following `if-then` statement:

```
if Selector = 1 or Selector = 2 or Selector = 3 then
  WriteLn("1,2,3")
elseif Selector = 4 or Selector = 5
or Selector = 6 then
  WriteLn("4,5,6")
else
  WriteLn("other")
end
```

The same statement can also be written as follows:

```
select True
  case Selector=1 or Selector=2 or Selector=3;
    WriteLn("1,2,3")
  case Selector=4 or Selector=5 or Selector=6;
    WriteLn("4,5,6")
  else
    WriteLn("other")
end
```

Notice that the expressions in each `case`-clause need not be constant expressions, like in the Pascal `case`-statement. In WANTScript, they can be arbitrary expressions as long as they evaluate to the same type as the selector expression.

Also notice that there can be more than one expression in each `case`-clause, separated by a comma from the previous one. Consequently, it is possible to re-write the preceding `select`-statement as follows:

```
select True
  case Selector=1, Selector=2, Selector=3:
    WriteLn("1,2,3")
  case Selector=4, Selector=5, Selector=6:
    WriteLn("4,5,6")
  else
    WriteLn("other")
end
```

There is an implied OR-operator in place of a comma in this case. Notice that the notation

```
case Selector=1, Selector=2, Selector=3:
```

is consistent with the Pascal-inspired notation:

```
case 1, 2, 3:
```

It's just that the matching expression in a WANTScript `case`-branch may be more than a simple constant: it can be an arbitrarily complex expression, that includes calls to functions and other operators. For example:

```
case UserFunction(Param)=1:
```

The fact that you can use arbitrary expressions in `switch-case` branches also means that you are not limited to ordinal types. Actually, there is not even a distinction between ordinal and other numeric types in WANTScript: there is only `System.Number`.

Hence, you can write `select` (or `switch`) statements that match on strings, which is quite useful. For example, a dispatcher for a user-typed command could be written as follows:

```
var Command = ""
WriteLn("Simple Command Processor")
repeat
  Write("command>")
  Command := ReadLn
  switch Command
    case "dir", "ls", "list":
      WriteLn("DIR selected")
    case "cpy", "copy":
      WriteLn("COPY selected")
    case "mkdir", "md":
      WriteLn("MKDIR selected")
    case "remdir", "rd":
      WriteLn("RMDIR selected")
    else
      WriteLn("You typed: ",Command)
  end
until command="quit" or command="exit"
WriteLn("Program terminated.")
```

The above sample could produce the following session, assuming that you type the indicated commands, when prompted:

```
Simple Command Processor
command>ls
DIR selected
command>copy
COPY selected
command>help
You typed: help
command>exit
You typed: exit
Program terminated.
```

Namespaces, Visibility, and Scope

Scope of an identifier is the range of lines code in which the identifier is “visible” and can be referenced.

Generally, identifiers defined inside a module are visible in the module that defines them.

They are also visible inside any module that explicitly imports the module in question.

They are also visible in any module embedded in the module in question.

A module definition creates a "namespace", i.e. a container providing context for the items it holds, and allowing disambiguation of items having the same name but residing in different namespaces.

TODO: Visibility and Scope

Built-In Types

Built-In Type: System.String

Strings can be concatenated (glued together) with the + operator. Strings can be subscripted (indexed); like in C, the first character of a string has subscript (index) 0. There is no separate character type; a character is simply a string of size one.

Like in Icon, substrings can be specified with the *slice notation*: two indices separated by a colon. Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

Using Quotes with Strings

The example programs in this tutorial use strings of characters, such as `"Hello world!"`. In this case, the string was delimited with double quotes.

WANTScript also permits strings delimited with single quotes, such as `'Hello world!'`. It is possible to embed a single quote inside a string delimited with double quotes and vice-versa, to embed a double quote inside a string delimited with single quotes:

```
"Ed asked about Q's response"
'regarding the "loop" statement.'
```

WANTScript also supports Pascal-like convention of embedding character codes using the `#` symbol plus the decimal ASCII code of the character to be embedded. The above examples can thus be re-written as:

```
"Ed asked about Q"#39"s response"
'regarding the '#34'loop'#34' statement.'
```

Multiple adjacent string literals (delimited by whitespace, such as space, tab, new-line-sequence, etc.), possibly using different quoting conventions, are allowed, and their meaning is the same as their concatenation. In other words, two or more string constants directly following one another, without any intervening non-whitespace tokens, are considered parts of a single string and concatenated together.

If the two strings are on separate lines, an end-of-line sequence (on Windows: carriage-return+line-feed, or `#13#10`) is inserted between the two constants. So, the two lines above are really a single string constant, with an end-of-line after the word "response".

Last, but not least, an un-terminated string, that is, a string that does not have its corresponding closing delimiter, is considered automatically terminated at the end of the line, but it does not include the end-of-line character sequence. So, all four of the following are exactly equivalent:

```
"Hello world!"
'Hello world!'
"Hello world!
'Hello world!
```

Here is a complete script that illustrates just about all of the preceding points:

```
//Experiments with quoted strings
//

// "Unterminated" strings
"Hello world!"
'Hello world!'
"Hello world!
'Hello world!

//Yes, the line below is a valid statement
'

//Single quote embedded inside
//doubly-quoted string
"Ed asked about Q's response"

//Double quote embedded inside
//singly quoted string
'regarding the "loop" statement.'

//Single quote embedded via character code
"Q"#39"s response was"
//Double quotes embedded via character codes
'characteristically '#34'damifino'#34', as usual.'

//Substrings can be quoted with either quotes
"Ed asked again about Q"#39's response'
'regarding the '#34'loop'#34" statement."

// When quoted inappropriately...
"Ed asked about Q"'"s response'
'regarding the "#34"loop"#34" statement."
```

```
(*
//But be *really* careful about what happens
//when you mix quotes inappropriately...
"Ed asked about Q"'"s response'
'regarding the "#34'loop'#34" statement."
*)
//WARNING: The commented out lines above
//execute an infinite loop printing
//'#34" statement.' every iteration.
//Fortunately, Ctrl+C works!
```

The above script produces the following output when run:

```
Hello world!
Hello world!
Hello world!
Hello world!

Ed asked about Q's response
regarding the "loop" statement.
Q's response was
characteristically "damifino", as usual.
Ed asked again about Q's response
regarding the "loop" statement.
Ed asked about Q"s response
regarding the "#34"loop"#34" statement."
```

NOTE: A string literal `"XYZ"` used in place of a statement resolves to `System.IO.Console.WriteLn("XYZ")`. The following two statements are therefore equivalent – perform exactly the same action:

```
"Ed asked about Q's response"

System.Console.WriteLn("Ed asked about Q's response")
```

The first of those is simply a convenient syntactical shortcut.

The project below illustrates how to output a large amount of text without having to type a whole lot of extra syntax. This could be useful, for example, when generating HTML, or printing a lot of status/progress messages.

Things like this are much more difficult in other languages...

In WANTScript:

- String literals are considered automatically terminated at the end of the line.
- Auto-terminated string literals do not normally include the end-of-line characters.
- Two or more string literals following each other are automatically concatenated and considered a single string literal.

- If the two string literals are on separate lines, an end-of-line sequence will be inserted between them.

- A string literal by itself, in place of a statement within a module, is an implicit call to `System.IO.Console.WriteLn()`.

Here is a project that illustrates all of the above points:

```
project Test141

{
Long text declaration
}

  WriteLn("BEGIN")

  "Gallia est omnis divisa in partes tres, quarum
  "unam incolunt Belgae, aliam Aquitani, tertiam
  "qui ipsorum lingua Celtae, nostra Galli appellantur.
  "Hi omnes lingua, institutis, legibus inter se differunt.
  "
  "Gallos ab Aquitanis Garumna flumen, a Belgis
  "Matrona et Sequana dividit.

  WriteLn("END")

end
```

Built-In Type: System.List

`System.List` is a built-in type representing an ordered collection of elements. The elements of a `List` need not be all of the same type. You can easily mix and match `Numbers`, `Strings`, `Booleans`, and other objects, including other `Lists`.

A new instance of a `List` is created via a call to its default constructor `Create`:

```
var L := System.List.Create
```

Once the instance has been created, you can use its `Add` method to add new elements:

```
L.Add(1)
L.Add("Hello")
L.Add(3.1415)
L.Add(True)
```

A `List` also defines a `Delete` method that you can use to delete a particular element from the list. The element is chosen via its zero-based index:

```
L.Delete(0)
```

To find out how many elements a list contains, call its `Count` function (or its alias `Size`):

```
WriteLn("L.Count=",L.Count)
WriteLn("L.Size=",L.Size)
```

There are two ways of retrieving a particular element from a list. The first method is via `System.List.Get(n)` function, where `n` represents the zero-based index of the element from the list:

```
WriteLn( L.Get(2) )
```

The aternative way of accessing an element in a list is to use the customary array subscript notation:

```
WriteLn( L[2] )
```

These two methods of retrieving elements from a list are exactly equivalent.

Likewise, there are two equivalent ways of replacing existing elements in a list. The first method is `System.List.Put(n, value)`, where `n` is the zero-based index of the element to be replaced, and `value` is the replacement value. For example:

```
L.Put(2, 12345.6789)
```

This puts the value of `12345.6789` into the third element of the list `L`. If the list `L` does not contain at least three elements, an exception is raised.

The alternative way of replacing list elements is, again, to use the normal array notation, as follows:

```
L[2] := 'Peek-A-Boo'
```

This places the string `'Peek-A-Boo'` as the third element of the list `L`, if the list has at least three elements.

Note that you cannot grow a list simply by referring to an element beyond the end of the list. Referencing a non-existent element in a list results in an exception being raised. To add elements to the list, you must use the `System.List.Add` function.

Lists of strings can easily be stored to, and retrieved from, disk files. To store a list's content into a disk file named `mylist.txt` you can use the following code:

```
L.SaveToTextFile('mylist1.txt')
```

To load a text file into a list, use the following call:

```
L.LoadFromTextFile('mylist1.txt')
```

The file you are loading into a list does not have to be a previously saved list. You can load any arbitrary text file using the

System.List.LoadFromTextFile call. This capability provides a powerful way of processing text files, analogous to the way you would use TStringList in Delphi/Pascal. You can load a text file into a list and then access the individual lines of the file as String-typed list elements.

Often, you need to retrieve the first element of a list. This can be done either by accessing the 0-th element of the list in one of the two methods described above (Get, or the array subscript []) , or by calling the First function on a List object. Thus

```
L.First
```

is equivalent to either of the two following lines:

```
L[0]
L.Get(0)
```

Correspondingly, there is the System.List.Last function that makes it easier to access the last element of the list:

```
L.Last
```

This is equivalent to either of the following:

```
L.Get(L.Count-1)
L[L.Count-1]
```

Note that both First and Last are functions returning an element value, and therefore cannot be used on the left-hand side of an assignment.

To delete all elements from the list in one shot, call the System.List.Clear method:

```
L.Clear
```

A list can be initialized with a literal expression, as follows:

```
L := [1, 2, 3, 4, 5]
```

List literals are enclosed in square brackets and contain the elements, which themselves must be literals or constants, separated by commas.

Since list items need not all have the same type, the following list literal is valid:

```
var A = ['Mouse', True, 100, 12.34]
```

Here is a more comprehensive and realistic example of list initialization and usage:

```
//test506_ListLiterals

//Static/class variable
"Days of the week"
```

```
staticvar L := ['Mon', 'Tue', 'Wed', 'Thu',
  'Fri', 'Sat', 'Sun']
WriteLn('NumDays=',L.Count);
PrintList(L)

WriteLn

//Instance variable
"Months of the year:
var M
M := ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
  'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
WriteLn('NumMonths=',M.Count);
PrintList(M)

procedure PrintList(AList)
  var I
  for I := 0 to AList.Count-1 do
    WriteLn(AList[I])
  end
end
```

The output of this program is as follows:

```
Days of the week
NumDays=7
Mon
Tue
Wed
Thu
Fri
Sat
Sun

Months of the year:
NumMonths=12
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
```

In this example, two lists `L` and `M` are created using a literal list expression. The list `L` is populated with the weekday-name abbreviations; the list `M` is initialized with the short versions of the month names. The list `L` is defined as a static variable, while the list `M` is an automatic (instance, or local) variable. This is just to illustrate that either type of variable will work in this context.

In either case, the number of elements in the list is displayed, and then the list elements are printed to the console using the `PrintList` procedure.

Built-In Type: System.Dictionary

`System.Dictionary` is a built-in type representing an associative array of elements. An associative array (a.k.a. map, hash, lookup table) stores name-value pairs – referred to as the key and item, respectively – in such a way that each key is unique and associated with one value.

Each item in the dictionary has a unique key. The key is used to retrieve (or "lookup") an individual item and is usually an integer or a string, but can be just about anything. The value corresponding to each key can be of any type, also.

Here is a simple – but typical – example of how an associative array, a.k.a. dictionary, can be used in an application:

```
project AssociativeArrays //Test511

  var D
  D := System.Dictionary.Create

  D["Poland"]    := "Warsaw"
  D["Venezuela"] := "Caracas"
  D["France"]    := "Paris"
  D["USA"]       := "Washington, D.C."
  D["Canada"]    := "Ottawa"
  D["Sweden"]    := "Stockholm"
  D["Spain"]     := "Madrid"

  PrintTheCapitalOf("Venezuela")
  PrintTheCapitalOf("USA")
  PrintTheCapitalOf("Poland")
  PrintTheCapitalOf("Sweden")
  PrintTheCapitalOf("Spain")
  PrintTheCapitalOf("France")
  PrintTheCapitalOf("Canada")

  Find(D,"Poland")
  Find(D,"Italy")

  Find(D,"USA")
  D.Delete("USA")
  Find(D,"USA")

  WriteLn("D.Count=",D.Count)
  D.Clear
  WriteLn("D.Count=",D.Count)

  procedure PrintTheCapitalOf(Country)
    WriteLn("The capital of ",Country,
      " is ", D[Country])
  end

  procedure Find(Dict, Country)
    if Dict.Contains(Country) then
      WriteLn( "Contains ", Country)
    else
      WriteLn( "Does NOT contain ", Country )
    end
  end
```

```
end
```

The program produces the following output:

```
The capital of Venezuela is Caracas
The capital of USA is Washington, D.C.
The capital of Poland is Warsaw
The capital of Sweden is Stockholm
The capital of Spain is Madrid
The capital of France is Paris
The capital of Canada is Ottawa
Contains Poland
Does NOT contain Italy
Contains USA
Does NOT contain USA
D.Count=6
D.Count=0
```

To create a new instance of a dictionary object, you need to call its default constructor `Create`:

```
var D
D := System.Dictionary.Create
```

To add items to a `System.Dictionary`, you just access the item in question by its key, for example:

```
D[0] := "ZERO"
D[1] := True
D[0.5] := "Half"
D["QUARTER"] := 0.25
D[True] := 1
```

Notice that, in this example, the type of the "array index" – or the key – used to access the element varies. You can see here an example of an integer key `1`, a floating point key `0.5`, a Boolean key `True`, and a string key `"QUARTER"`. You can also see various element types: a string, an integer, a float, and a Boolean.

The assignment to an element identified by the key automatically expands the dictionary as necessary. This is in contrast with lists, where the element must exist before it can be accessed via a list index.

To retrieve – or "lookup" – the item corresponding to a key in a dictionary, simply refer to the item using the normal array notation:

```
WriteLn( D[0.5] )
```

To inquire about the current size of a dictionary (the count of its elements), use the `System.Dictionary.Count` method, or its alias, `System.Dictionary.Size`:

```
WriteLn('D.Count=',D.Count)
WriteLn('D.Size=',D.Size)
```

To find out whether a dictionary contains a particular key, call the `System.Dictionary.Contains` function, which returns a `Boolean` result:

```
if D.Contains("Poland") then
  WriteLn( "Dictionary contains Poland")
else
  WriteLn( "Dictionary does NOT contain Poland" )
end
```

To delete a single item from a dictionary, use the `System.Dictionary.Delete` method, or its alias `System.Dictionary.Remove`, passing it the key of the item to be deleted:

```
D.Delete("USA")
D.Remove("France")
```

To empty a dictionary completely, use the `System.Dictionary.Clear` method:

```
D.Clear
```

You can retrieve all keys and all items from a dictionary using the following methods: `System.Dictionary.Keys` and `System.Dictionary.Items`.

The `System.Dictionary.Keys` function returns a list (of `System.List` type) of all the keys in the dictionary object:

```
var Keys
Keys := D.Keys
WriteLn("D contains ", Keys.Length, " keys")
```

The `System.Dictionary.Items` function returns a list (of `System.List` type) of all the items, i.e. all objects corresponding to the keys, in the dictionary object:

```
var Items
Items := D.Items
WriteLn("D contains ", Keys.Items, " items")
```

TODO: This needs to change. Note that the keys retrieved via a call to `System.Dictionary.Keys` function – being of string type – are always returned in alphabetical order. On the other hand, the items returned by `System.Dictionary.Items` are potentially of various types, so that there is no intrinsic order to them inside the dictionary, and are returned in an arbitrary order, usually different from the alphabetical ordering of their keys.

Keywords

Following is the list of all keywords in WANTScript; these keywords are also recognized by the syntax-highlighting editor in the IDE:

```
abstract and as break call case class classvar const
constructor depends destructor div do downto else elseif
end exit except false finally for foreach from function if
implementation import in inherited inline input interface
is library loop mod module modulevar new nil not null
object of on operator or out output overload override
private procedure program project property protected public
published raise ref reintroduce record repeat return select
set shl shr static staticvar step switch target then to
true try type unit until uses using var virtual while with
xor
```

ALPHA-RELEASE NOTE: At the moment, not all of these keywords perform their intended function yet. Some are reserved for future use.