

# **Vecscript**

## **User Manual**

By Mike Hahn

Copyright © 2008 Vecset.net

Date Last Edited: 8-Oct-2008

# Table of Contents

<b>Overview .....</b>	<b>1</b>
<u>Introduction .....</u>	<u>1</u>
<u>Vecscript Overview .....</u>	<u>1</u>
<b>Editor .....</b>	<b>2</b>
<u>User Interface .....</u>	<u>2</u>
<u>Navigation Keys .....</u>	<u>3</u>
<b>Language Specs .....</b>	<b>5</b>
<u>Language Description .....</u>	<u>5</u>
Packages .....	5
Comments .....	5
Identifiers .....	6
Primitive Data Types and Literals .....	6
Classes and Objects .....	7
Static and Object Initializers .....	12
Interfaces .....	12
Local Variable Declarations .....	13
Blocks .....	13
Statements .....	13
Operators .....	17
<u>Language Grammar .....</u>	<u>19</u>
<u>VEC Grammar .....</u>	<u>29</u>
<u>Sample Code .....</u>	<u>37</u>
Delphi Code Counter .....	37
Vecscript Code Counter .....	41
<b>Language Implementation .....</b>	<b>46</b>
<u>VECSET Block Diagram .....</u>	<u>46</u>
<u>Lists .....</u>	<u>46</u>
<u>Memory Management .....</u>	<u>48</u>
<u>Game Server .....</u>	<u>48</u>

# Overview

## Introduction

[ [Home](#) ]

Vecset can be used to create multiplayer board games, as well as animated games. You can log on to [vecsworld.com](#) and play these games with other Vecset users. Vecset games are coded in a built-in scripting language called Vecscript. Non-programmers can create drag-and-drop games, and programmers can add functionality to these games. The Vecscript scripting language is based on Java, but with a Lisp-like syntax.

## Vecscript Overview

All Vecset games are written in a built-in scripting language called Vecscript. Vecscript is a powerful object-oriented language which is designed with beginner programmers in mind. Pressing the question mark (?) key when in the code editor brings up a popup menu of choices valid in the context of the text cursor position. Optional Structure-Editor mode eases code entry for naïve users. The default operator/operand mode is prefix (operators come before their operands) as opposed to optional infix (binary operators come in between their operands). Vecscript is based on Java, and when infix mode is selected, Vecscript code strongly resembles Java, with a touch of Object Pascal thrown in for good measure.

## Language Features

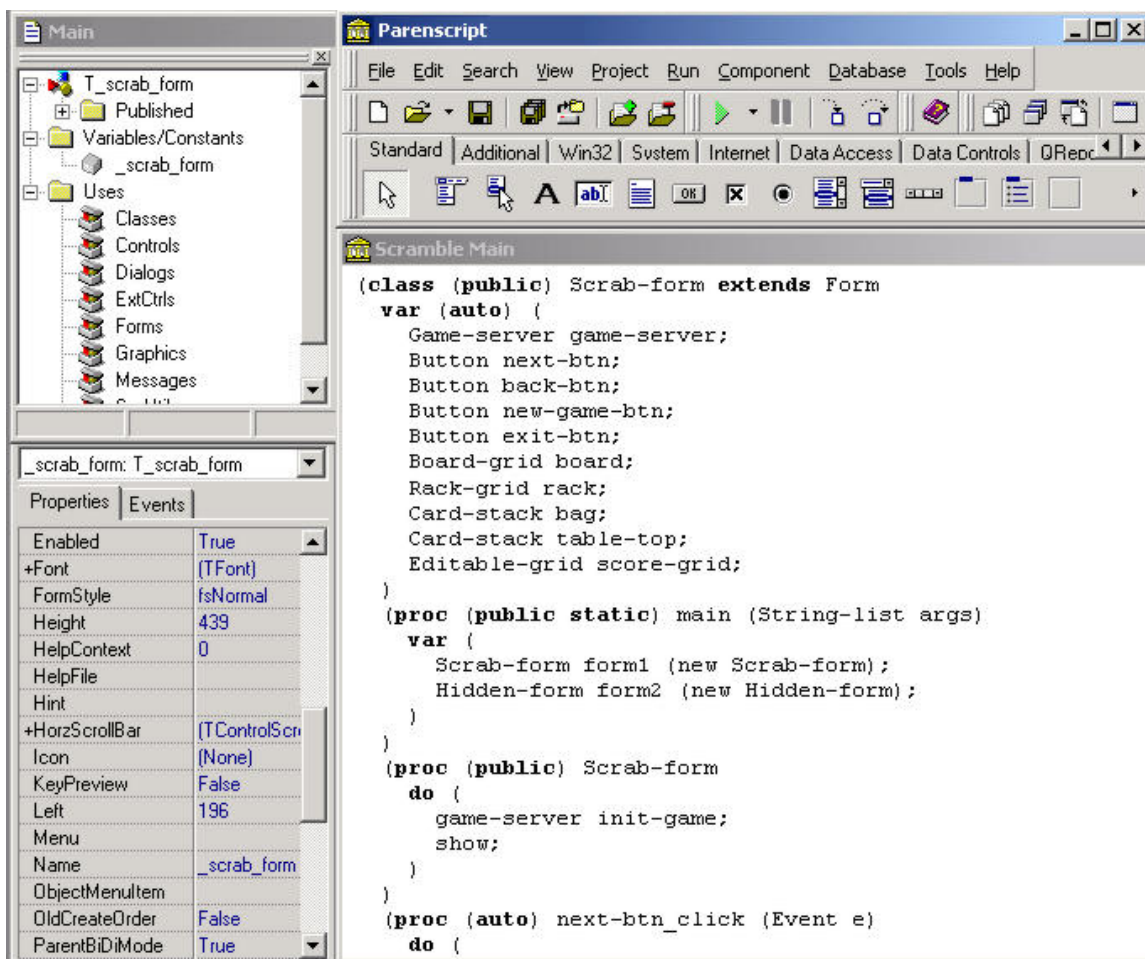
- **Syntax:** Vecscript is a subset of Java, although the syntax differs radically from Java, in that all operators are, by default, prefix (like Lisp) rather than infix, and parentheses are used for grouping.
- **Semicolons:** Statements and declarations are semicolon-delimited.
- **Case:** Vecscript is case-sensitive: the convention for identifiers is all lower case, hyphens being used to separate parts of multi-word identifiers. Class names are an exception: the initial letter is upper case.
- **Keyboard Aid:** When enabled, this feature allows the user to enter a hyphen followed by a lower case letter by typing an upper case letter (but only when the text cursor is in the middle or at the end of an identifier). Alternatively, the user may enter a hyphen by typing a quote ('). Also, commas and periods are immediately converted into parentheses if desired (if the period key is used to enter ')', then the ')' key may be used to enter a decimal point). The user may toggle Code Menu mode by typing slash (/) instead of question mark (?). Keyboard Aid is always disabled inside comments and string literals.
- **Meta-Programming:** Vecscript programs are lists, which can act as data for other Vecscript programs. Structure editor mode lets newbies create simple event handlers without having to know the syntax of the Vecscript programming language.

# Editor

## User Interface

Vecscript is similar to Visual Basic or Delphi, in which the game designer selects components from the Object Palette, places them on the main window of the game under development, and modifies their properties using the Object Inspector.

- **Browser Window:** Occupying the leftmost quarter of the screen, this window contains a tree-view of the current project, and below that is the 2-column Object Inspector (property names and values). Between the tree-view and the Object Inspector is a splitter that moves up and down.
- **Main Window:** The rest of the screen has a menu bar, a button bar, a hierarchical Object Palette (0.75 inches high, 3/4 of the screen wide), and the code pane. When the game designer is not coding, the code pane can be hidden, revealing the game window under development.



# Navigation Keys

There exist 3 text-entry modes in the Vecscript code editor:

1. Free Form
2. Structure Editor
3. Code Menu

Pressing Esc toggles between Free Form and Structure Editor modes. Pressing F1 displays context-sensitive help in all modes. Pressing the question mark (?) enters Code Menu Mode, in which a menu of available context-sensitive options is displayed. Selecting an option may bring up another, lower-level menu, which may lead to yet another lower-level menu, and so on.

## Structure Mode Commands

A bottom-level token (e.g. a keyword, identifier, operator, or constant) or an entire list is often highlighted. Using the Shift key in conjunction with the Up/Down Arrow keys, it is possible to select more than one token/list at a time.

- **Esc** – toggle between Free Form and Structure Editor modes
- **Up Arrow** - go to previous list element
- **Down Arrow** - go to next list element
- **Left Arrow** - go to parent list
- **Right Arrow** - go to first child element (if none, display text cursor following current bottom-level token)
- **Shift+Up/Down Arrow** - select a range of tokens/lists
- **Printable Char.** - incrementally select valid matching menu item (if any)
- **Backspace** - undo insertion of previous printable char.
- **Delete** - delete current token/list
- **Enter** - display text cursor, insert space after cursor (or insert result of incremental menu selection)
- **Space** - display text cursor, insert space before cursor (or insert result of incremental menu selection)
- **Ctrl+Enter** – if at end of line, append blank line (otherwise break line into 2 lines)

## Code Menu Commands

A popup menu above or below text cursor (and including text cursor) is displayed. The contents of this menu include all valid code elements in the context of the text cursor (ignoring anything after the text cursor). If the current menu item refers to a list, the entire list is highlighted (defaults to light gray if background color of whitespace is white).

- **Question Mark** - toggle between Code Menu and Free Form/Structure Editor modes
- **Esc** - show/hide code menu
- **Up Arrow** - move selection up (scroll up after pressing Esc)
- **Down Arrow** - move selection down (scroll down after pressing Esc)
- **Left Arrow** - go to parent code menu
- **Right Arrow** - go to lower-level code menu, if any
- **Enter** - go to lower-level code menu (if none, insert current menu item, go to next menu item, or if none, go to parent code menu)

- **Space** - go to lower-level code menu (if none, insert current menu item, go to next menu item, or if none, go to parent code menu; exit Code Menu mode)
- **Printable Char.** - incrementally select matching menu item
- **Backspace** - undo operation of previous printable char.
- **Page Up** - page up after pressing Esc
- **Page Down** - page down after pressing Esc
- **Shift Arrow** – only used if current menu item is repeated, such as a statement in a block, a declaration, or a parameter in a parameter list
  - **Shift Up Arrow** – select previous instance of current menu item
  - **Shift Down Arrow** - select next instance of current menu item
  - **Shift Left Arrow** – insert above current menu item
  - **Shift Right Arrow** - insert below current menu item
  - **Semicolon** – toggle parent list: multi-line/single-line

# Language Specs

## Language Description

### Introduction

This Language Summary has been adapted, with kind permission of Macmillan Computer Publishing, from [Java 2 Platform Unleashed](#), Appendix A, "Java Language Summary," by [Jamie Jaworski](#).

A hybrid of Java and Lisp, this OOP language is at once powerful and easy to master. One of the goals of Vecscript is to enable programming newbies to create simple drag-and-drop applications, such as board games, with little or no coding.

### Packages

Vecscript programs are organized into *packages* that contain the source code declarations of Vecscript classes and interfaces. Packages are identified by the `package` statement. It is the first statement in a source code file:

```
package package-name
```

If a package statement is omitted, the classes and interfaces declared within the package are put into the default no-name package. The package name and the CLASSPATH are used to find a class. Only one class or interface may be declared as public for a given source code file (compilation unit). For example, you can define classes X and Y and interface Z within a compilation unit, but only one of these three can be declared public. The name of the compilation unit must be the name of the public class or interface followed by the `.TS` extension.

### The import Statement

The `import` statement is used to reference classes and interfaces that are declared in other packages. There are three forms of the import statement:

- `import ( package-name class-name )`
- `import ( package-name interface-name )`
- `import ( package-name * )`

The first and second forms allow the identified classes and interfaces to be referenced without specifying the name of their package. The third form allows all classes and interfaces in the specified package to be referenced without specifying the name of their package.

### Comments

Vecscript provides both multiline and single line comments:

```
{ This is a  
multiline comment. }
```

```
\ This is a single line comment
```

Brace brackets are used to enclose multiline comments. All text between { and } is treated as a comment. A backslash is used for single-line comments. All text following the \ until the end of the line is treated as a comment.

Comments cannot be nested and cannot appear within string literals.

## Identifiers

Identifiers are used to name Vecscript language entities. They begin with a letter or underscore character (`_`). Subsequent characters consist of these characters, digits, and hyphen characters (`-`). Identifiers are case-sensitive and cannot be the same as a reserved word.

### Identifier Naming Convention

The following is just a programming convention and is not enforced by the Vecscript compiler. By convention, all identifiers and keywords are lower case, with the exception of class names, in which the first letter is upper case. Multi-word identifiers are hyphen-separated, e.g. `str-to-int`

## Primitive Data Types and Literals

Vecscript defines eight primitive types. Variables that are declared as a primitive type are not objects. They are only placeholders to store primitive values. The eight primitive types are `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`.

The `byte`, `short`, `int`, and `long` types represent 8-, 16-, 32-, and 64-bit integer values. The literal values of these types are written using positive or negative decimal, hexadecimal, or octal integers. Hexadecimal values are preceded by `0x` or `0X` and use letters `a` through `f` (upper or lower case) to represent the digits 10 through 15. Octal numbers are preceded by `0`. Long decimal values have an `L` or `l` appended to the end of the number.

The `float` and `double` types represent 32- and 64-bit floating-point numbers. `float` numbers have the `f` or `F` suffix, `double` numbers have `d` or `D`. If no suffix is provided, the default `double` type is assumed. Floating-point numbers may be written in any of the following four forms:

- *digits . optional-digits optional-exponent-part suffix*
- *. optional-digits optional-exponent-part suffix*
- *digits exponent-part suffix*
- `NaN`

The suffix is optional. It consists of `f`, `F`, `d`, or `D`, as described previously.

The exponent part is optional in the first two forms but required in the third form. It consists of an `e` or `E` followed by a signed integer. It is used to identify the exponent of 10 of the number written in scientific notation. For example, `1000000.0` could be represented as `1.0E6`.

The special value `NaN` is used to represent the value "not a number," which occurs as the result of undefined mathematical operations such as division by zero.

The `char` type represents 16-bit Unicode characters. Unicode is a 16-bit superset of the ASCII character set that provides many foreign-language characters. A single character is specified by putting the character within single quotes (`'`). There are two exceptions: single quote (`'`) and backslash (`\`). The backslash character (`\`) is used as an escape code to represent special character values. The character escape codes are:

<i>Escape Code</i>	<i>Character</i>
<code>\b</code>	Backspace
<code>\t</code>	Tab
<code>\n</code>	Linefeed
<code>\f</code>	Form feed
<code>\r</code>	Carriage return
<code>\'</code>	Single quote
<code>\\</code>	Backslash

The backslash can also be followed by an 8-bit octal value, or by `u` or `U` followed by a four-digit hexadecimal value. The four-digit value is used to specify the value of Unicode characters.

The `boolean` type represents the logical values `true` and `false`.

String literals are also provided by Vecscript, although strings are not primitive values. Strings consist of characters enclosed by single quotes (`'`). The character escape codes may be used within strings. In case the string literal consists of a single character, the context is used to determine whether it's a string literal or a character constant.

The literal value `nil` is used to identify the fact that an object is not assigned to a value. It may be used with any variable that is not of a primitive data type.

## Classes and Objects

Objects are the basic elements of Vecscript programs. They are executable entities that contain data and provide methods for manipulating that data. Every object is an instance of a class. Classes are templates from which objects are created. They define the type of data that an object contains and the methods for manipulating that data. Objects are created (or instantiated) via constructors. An object is instantiated by assigning specific values to the field variables defined by the class.

### Class Declarations

Class declarations allow new classes to be defined for use in Vecscript programs. Classes are declared as follows:

```
( class ( class-modifiers ) class-name extends-clause implements-clause class-body )
```

The class modifiers, `extends` clause, and `implements` clause are optional. If there are no class modifiers, the parentheses enclosing `class-modifiers` are omitted.

The class modifiers are `abstract`, `public`, and `final`. An abstract class provides an abstract class declaration that cannot be instantiated. In general, abstract classes are used

as building blocks for the declaration of subclasses. A class that is declared as `public` can be referenced outside its package. If a class is not declared as `public`, it can be referenced only within its package. A `final` class cannot be subclassed. A class cannot be declared as both `final` and `abstract`.

The `extends` clause is used to identify the immediate superclass of a class and thereby position the class within the overall class hierarchy. It is written as follows:

**extends** *immediate-superclass*

The `implements` clause identifies the interfaces that are implemented by a class. It is written as follows:

**implements** ( *interface-names* )

The interface names consist of a list of one or more interface names.

The class body declares the members of a class. It is written as follows:

*member-declarations*

The member declarations consists of zero or more of the following declarations:

- *field-variable-list*
- Constructors and methods
- Static and object initializers
- Inner classes

The *field-variable-list* is written as follows:

**var** ( *variable-modifiers* ) ( *field-variable-declarations* )

The variable modifiers are optional, and if absent, so are the parentheses enclosing them.

## Variable Declarations

*Variables* are used to refer to objects and primitive data types. They are declared as follows:

- ( *variable-modifiers* ) *type* *variable-list* *variable-initialization* ;
- ( *variable-modifiers* ) **array** *dimension-count* *type* *variable-list* *variable-initialization* ;
- ( *variable-modifiers* ) **set** *set-type* *variable-list* *variable-initialization* ;
- ( *variable-modifiers* ) **list** *list-type* *variable-list* *variable-initialization* ;
- ( *variable-modifiers* ) **list** *variable-list* *variable-initialization* ;
- ( *variable-modifiers* ) **enum** *enumerated-type-name* ( *enumeration-list* ) ;

The variable modifiers and variable initialization are optional. If the variable modifiers are absent, so are the parentheses enclosing them. A variable's *type* may be a primitive data type, class type, or interface type. The variable list is a list of variable names enclosed in parentheses. If there is only one variable name, the parentheses are optional.

The variable initialization consists of an expression yielding a value of the variable's type. If the variable being declared is an array, it can be assigned to an array initializer. An array initializer is a list of expressions which all evaluate to values of the indicated type, enclosed in parentheses. For multi-dimensional arrays, the expression lists are nested to the depth indicated by the dimension count (a positive integer constant). If the array is one-dimensional, the dimension count may be omitted.

When declaring a *set*, the set type is either an enumerated type or a subrange, such as (.. 'A' 'Z').

There are seven variable modifiers: `public`, `protected`, `private`, `static`, `final`, `transient`, and `volatile`.

The `public`, `protected`, and `private` modifiers are used to designate the specific manner in which a variable can be accessed. Variables that are declared as `public` can be accessed anywhere that the class in which they are declared can be accessed. Variables that are declared as `protected` can be accessed within the package in which they are declared and in subclasses of the class in which they are declared. Variables that are declared as `private` are only accessible in the class in which they are defined and not in any of its subclasses. If a variable isn't declared as `public`, `protected`, or `private` it can be accessed only within the package in which it is declared.

A variable that is declared as `static` is associated with its class and is shared by objects that are instances of its class. A `static` variable is also known as a *class variable*.

A variable that is declared as `final` is a constant and cannot be modified. `final` variables must be initialized before they are used.

A variable that is declared as `transient` is not saved as part of an object when the object is serialized. The `transient` keyword identifies a variable that does not maintain a persistent state.

A variable that is declared as `volatile` refers to objects and primitive values that can be modified asynchronously by separate threads of execution. They are treated in a special manner by the compiler to control the manner in which they can be updated.

## Field Variable Declarations

Field variable declarations are identical to variable declarations, except they may contain an optional *property-specifier*. Valid property specifiers include:

```
( property read )  
( property write )  
( property read write )
```

Field variable declarations are declared as follows:

- ( *variable-modifiers* ) *property-specifier type variable-list* ;

Field variables that include property specifiers are known as *properties*. Read-only properties cannot be written to except in the class in which they are declared. Write-only properties cannot be read from except in the class in which they are declared. Read-write properties can be read from or written to without restrictions.

## Property Methods

Property functions are used to read from read-only (and read-write) properties. Property procedures are used to write to write-only (and read-write) properties. They are declared as follows:

- ( **func** ( *method-modifiers* ) *field-type func-name throws-clause method-body* )
- ( **proc** ( *method-modifiers* ) *proc-name ( field-type field-name ) throws-clause method-body* )

The *field-name* is the name of the associated property. The *field-type* is the type of the associated property. The *func-name* is always **get\_** concatenated with *field-name*. The *proc-name* is always **set\_** concatenated with *field-name*.

For read-only properties, if a property function exists, all access to that property is funneled through the property function. For write-only properties, if a property procedure exists, all access to that property is funneled through the property procedure. For read-write properties, all access to that property is funneled through the property function and/or property procedure, if they exist.

Indexed property methods are declared as follows:

- ( **func** ( *method-modifiers* ) *field-type func-name ( integer-type identifier ) throws-clause method-body* )
- ( **proc** ( *method-modifiers* ) *proc-name ( field-type field-name ; integer-type identifier ) throws-clause method-body* )

## Constructor Declarations

*Constructors* are methods that are used to initialize newly created objects of a class. They are declared as follows:

( **cons** ( *constructor-modifiers* ) *constructor-name ( parameter-list ) throws-clause constructor-body* )

The constructor modifiers are optional, and include `public`, `protected`, and `private`. They control access to the constructor and are used in the same manner as they are for variables. If omitted, the parentheses enclosing them are also omitted.

The constructor name is the same as the class name in which it is declared. It is followed by an optional parameter list, consisting of a list of parameter declarations. If there are no parameter declarations, the enclosing parentheses are omitted. Parameter declarations are written as follows:

- *type variable-list ;*
- **array** *dimension-count type variable-list ;*
- **set** *set-type variable-list ;*
- **list** *list-type variable-list ;*
- **list** *variable-list ;*

Each parameter declaration consists of a type followed by a list of parameter names enclosed in parentheses. If there is only one parameter in the list, the parentheses are optional. For array parameter declarations, if the array is one-dimensional, the dimension count may be omitted.

The `throws` clause identifies all uncaught exceptions that are thrown within the constructor. It is written as follows:

```
throws ( uncaught-exceptions )
```

The body of a constructor is written as follows:

```
local-variable-list do ( constructor-call-statement statements )
```

The local variable list, constructor call statement, and statements are optional. The local variable list consists of zero or more of the following:

```
var ( variable-modifiers ) ( variable-declarations )
```

The constructor call statement allows another constructor of the class or its superclass to be invoked before the constructor's block body. It is written using one of the two following forms:

- **this** *argument-list* ;
- **super** *argument-list* ;

The first form results in a constructor for the current class being invoked with the specified arguments. The second form results in the constructor of the class's superclass being invoked. The argument list consists of expressions that evaluate to the allowed values of a particular constructor.

If no constructor call statement is specified, a default `super` constructor is invoked before the constructor block body.

## Method Declarations

Methods are used to perform operations on the data contained in object. They are written as follows:

- ( **proc** ( *method-modifiers* ) *method-name* ( *parameter-list* ) *throws-clause* *method-body* )
- ( **func** ( *method-modifiers* ) *return-type* *method-name* ( *parameter-list* ) *throws-clause* *method-body* )

The parameters and `throws` clause are declared in the same method as in constructor declarations.

The method body differs from the constructor body in that it does not allow a constructor call statement.

The method modifiers include the `public`, `protected`, and `private` modifiers defined for constructors, as well as the `final`, `static`, `abstract`, and `synchronized` modifiers.

The `final` modifier identifies a method that cannot be overridden.

The `static` modifier identifies a class method. Class methods are only allowed to access static class variables. `static` methods are `final`.

An `abstract` method is used to identify a method that cannot be invoked and must be overridden by any non-abstract subclasses of the class in which it is declared. An `abstract` method does not have a method body.

A `synchronized` method is a method that must acquire a lock on an object or on a class before it can be executed.

## Static and Object Initializers

A *static initializer* is a block of code that is used to initialize the `static` variables of a class. It is written as follows:

```
static statement-block
```

Static initializers can only access static class variables. They are executed in the order in which they appear in a class declaration. A statement block is a list of zero or more statements enclosed in parentheses. The parentheses are mandatory, even if there are no statements in the statement block.

Object initializers are used to initialize non-static variables of a class and are executed immediately after a class's superclass constructor is invoked. They are written as follows:

```
do statement-block
```

## Interfaces

An *interface* specifies a collection of abstract methods that must be overridden by classes that implement the interface. Interfaces are declared as follows:

```
( interface ( interface-modifiers ) interface-name extends-clause interface-body )
```

The interface modifiers are `public` and `abstract`. `public` interfaces can be accessed in other packages. All interfaces are `abstract`. The `abstract` modifier is superfluous. If there are no interface modifiers, the parentheses enclosing them are omitted.

The optional `extends` clause is used to identify any interfaces that are extended by an interface. It is written as follows:

```
extends ( interface-names )
```

An interface inherits all the methods of all interfaces that it extends.

The interface body consists of zero or more field variable lists and abstract method declarations. All variables declared in these field variable lists are `public`, `static`, and `final`. Methods are `public` and `abstract`. These variable and method modifiers need not be specified.

## Local Variable Declarations

Local variables are declared in the same manner that field variables are declared, except that local variables do not include modifiers. They are accessible within the `do` block of the method or constructor in which they are declared. The `this` and `super` variables are predefined. They refer to the current object for which a method is invoked and the superclass of the current object being invoked.

## Blocks

Blocks consist of a list of statements, each of which is terminated with a semicolon, and enclosed with parentheses, and are written as follows:

( *statement-list* )

Note that the parentheses are mandatory, even if the statement list is empty.

## Statements

The programming statements supported by Vecscript are described in the following subsections.

### Method Invocation

A *method invocation* invokes a method for an object or class. Method invocations may be used within an expression or as a separate statement. If used as a separate statement, the enclosing parentheses are omitted. Method invocations take the following forms:

- ( *method-name argument-list* )
- ( : *object-name method-name argument-list* )
- ( : *class-name method-name argument-list* )

The *argument-list* consists of a list of zero or more expressions that are consistent with the method's parameters.

### Allocation Statements

When an object is *allocated*, it is typically assigned to a variable. However, it is not required to be assigned when it is allocated. An allocation statement is of the following form:

**new** *constructor argument-list*

The `new` operator is used to allocate an object of the class specified by the constructor. The constructor is then invoked to initialize the object using the arguments specified in the *argument-list*.

### Assignment Statements

The *assignment statement* assigns an object or value to a variable. Its general form is

= *variable-name expression*

where the expression yields a value that is consistent with the variable's type.

Other assignment operators may be used in addition to the = operator. See the section titled [Operators](#).

## The if Statement

The `if` statement is used to select among alternative paths of execution. It is written as follows:

```
if boolean-expression then block elseif-clause else-clause
```

The `elseif` and `else` clauses are optional. The `elseif` clause may occur zero or more times. They are written as follows:

- `elseif` *boolean-expression* **then** *block*
- `else` *block*

Each block is only executed if the preceding `boolean` expression is true. At most one block in the `if` statement is executed. The block inside the `else` clause is only executed if the preceding `boolean` expression is false.

## The switch Statement

The `switch` statement is similar to the `if` statement in that it enables a selection from alternative paths of execution. It is written as follows:

```
switch expression ( case-clause default-clause )
```

The expression must evaluate to a `byte`, `char`, `short`, or `int` value. Control is transferred to the block following the constant list containing a value matching the expression.

The `case` clause may occur one or more times. If the constant list contains only one value, the enclosing parentheses are optional. The `default` clause is optional. They are written as follows:

- `case` ( *constant-list* ) **do** *block*
- `default` *block*

If none of the values contained in any of the constant lists match the expression, and a `default` clause exists, control is transferred to the `default` clause block.

## The for Statement

The `for` statement is used to iteratively execute a block. It takes the following forms:

- `for` *index-variable* ( *initial-expression* *limit-expression* ) *by-clause* **do** *block*
- `for` *index-variable* *list-variable* *by-clause* **do** *block*

The `by-clause` is optional. It is written as follows:

- `by` *step-expression*

- **by** ( *step-expression from-expression* )

The index variable must be a local variable of ordinal type (*byte, short, int, or long*). In regards to the first form of the `for` statement, each of the 2 expressions contained within the parentheses, as well as the step expression, must evaluate to the same type as the index variable. The initial expression is assigned to the index variable at the beginning of the `for` statement. The step expression defaults to 1. At the beginning of each loop, if the value of the index variable is greater than the limit expression (or less than the limit expression if the step expression is negative), then the `for` statement terminates. Otherwise the block is executed, and then the step expression is added to the index variable, another comparison between the index variable and the limit expression takes place, and so on.

In regards to the other form of the `for` statement, the loop iterates through the list defined by the list variable. If the step expression is negative, the `for` statement iterates from the end of the list to the beginning of the list. The step expression must be of ordinal type. A step expression of 2 iterates through every other element of the list, a step expression of 3 iterates through every third element of the list, and so on. The from-expression defaults to 0. The current list index is initially either the from-expression or, in case of a negative step expression, the no. of elements in the list minus 1 minus the from-expression.

## The while Statement

The `while` statement is used to execute a block while a `boolean` expression is true. It is written as follows:

```
while boolean-expression do block
```

The `boolean` expression is evaluated; if it is true, the block is executed. It continues to execute until the `boolean` expression is false.

## The do-while Statement

The `do while` statement is used to execute a block until a `boolean` expression becomes false. Unlike the `while` statement, the block is always executed at least once, since the `boolean` expression is evaluated at the bottom of the loop. The `do while` statement is written as follows:

```
do block while boolean-expression
```

## The break Statement

The `break` statement is used to transfer control out of a loop block to the statement immediately following the `for`, `while`, or `do-while` statement.

## The continue Statement

Like the `break` statement, the `continue` statement is used to transfer control out of a loop block, but then instead of terminating the loop, the `boolean` expression is evaluated (or in the case of a `for` statement, the step expression is added to the index variable and then compared to the limit expression), and the loop statement terminates if the `boolean` expression evaluates to the appropriate value. Otherwise, the `for`, `while`, or `do-while` statement continues executing.

## The return Statement

The `return` statement is used to return an object or value as the result of a method's invocation. It is written as follows:

```
return expression
```

The value of the expression must match the return value in the method's declaration. If the method is a procedure, the expression is omitted, since procedures don't return anything, and the method terminates.

## The with Statement

The `with` statement (borrowed from Object Pascal) is used instead of the colon operator (`:`) as a means of abbreviating numerous and/or lengthy colon expressions. The following 2 colon expressions (procedure calls, in this case):

```
: a b p;  
: a b (q x y z);
```

may be abbreviated as follows:

```
with (a b) do (  
    p;  
    q x y z;  
);
```

The `with` statement is written as follows:

```
with ( identifier... ) do block
```

The parentheses may be omitted if there is only one identifier between `with` and `do`. The `with` statement (without the terminating semicolon) can also be used in place of the statement block, or body, of a method declaration. For example:

```
(proc p  
    with (a b) do (  
        ...  
    )  
)
```

## The synchronized Statement

The `synchronized` statement is used to execute a block after acquiring a lock on an object. It is written as follows:

```
synchronized expression block
```

The expression yields the object for which the lock must be acquired.

## The try Statement

The `try` statement executes a block while setting up exception handlers. If an exception occurs, the appropriate handler, if any, is executed to handle the exception. A `finally` clause may also be specified to perform absolutely required processing.

The `try` statement is written as follows:

```
try block catch-clauses finally-clause
```

At least one `catch` clause or a `finally` clause must be provided.

The format of the `catch` clause is as follows:

**catch** ( *exception-declaration* ) *block*

If an exception is thrown within the block executed by the `try` statement and it can be assigned to the type of exception declared in the `catch` clause, the block of the `catch` clause is executed.

The `finally` clause, if it is provided, is always executed regardless of whether an exception is generated.

## Operators

Vecscript defines arithmetic, relational, logical, shift, set, assignment, list, string, object, caste, instance, allocation, and selection operators. The following table summarizes these operators.

<i>Operator Type</i>	<i>Operator</i>	<i>Description</i>	<i>Example</i>
Arithmetic	+	Addition	( + a b c )
	-	Subtraction	( - a b )
	*	Multiplication	( * a b c )
	/	Division	( / a b )
	%	Modulus	( % a b )
Relational	>	Greater than	( > a b )
	<	Less than	( < a b )
	>=	Greater than or equal	( >= a b )
	<=	Less than or equal	( <= a b )
	<>	Not equal	( <> a b )
	==	Equal	( == a b )
Logical	not	Not	( not a )
	and	AND	( and a b c )
	or	OR	( or a b c )

	xor	Exclusive or	(xor a b)
Shift	<<	Left-shift	(<< a b)
	>>	Right-shift	(>> a b)
	>>>	Zero-filled right-shift	(>>> a b)
Set	+	Set union	(+ a b c)
	-	Set difference	(- a b)
	*	Set intersection	(* a b c)
	==	Set equality	(== a b)
	<>	Set inequality	(<> a b)
	<=	Set inclusion	(<= a b)
	in	Set membership	(in a b)
	..	Subrange	(.. a b)
Assignment	=	Assignment	= a b
	++	Increment and assign	++ a
	--	Decrement and assign	-- a
	+=	Add and assign	+= a b
	-=	Subtract and assign	-= a b
	*=	Multiply and assign	*= a b
	/=	Divide and assign	/= a b
	%=	Take modulus and assign	%= a b
	=	OR and assign	= a b
	=&	AND and assign	&= a b
	=^	XOR and assign	^= a b
	==<<	Left-shift and assign	<<= a b
	==>>	Right-shift and assign	>>= a b

	<code>=&gt;&gt;&gt;</code>	Zero-filled right-shift and assign	<code>&gt;&gt;&gt;= a b</code>
List	<code>pchild</code>	Get child pointer	<code>(pchild a)</code>
	<code>pnext</code>	Get next pointer	<code>(pnext a)</code>
	<code>pchild</code>	Set child pointer	<code>pchild a b</code>
	<code>pnext</code>	Set next pointer	<code>pnext a b</code>
	<code>atomic</code>	Is an atom?	<code>(atomic a)</code>
	<code>:</code>	List member	<code>(: a n)</code>
String	<code>+</code>	Concatenation	<code>(+ a b c)</code>
Object	<code>:</code>	Property/Method	<code>(: a b c)</code>
Cast	Type	Convert to type	<code>(char a)</code>
Instance	<code>instanceof</code>	Is instance of class?	<code>(instanceof a b)</code>
Allocation	<code>new</code>	Create a new object of a class	<code>(new a)</code>
	<code>resize</code>	Change array size	<code>resize a m n</code>
Selection	<code>quest</code>	If...then selection	<code>(quest a b c)</code>

## Language Grammar

The following BNF Grammar defines the syntax of the Vecscript programming language. Non-terminal symbols are written in italics, e.g. *while-stmt*. Square brackets ([]) enclose something that is optional (can occur 0 or 1 times). Square brackets followed by an ellipsis (...) enclose something that repeats (can occur 0 or more times). A non-terminal symbol followed by an ellipsis indicates repetition (can occur 1 or more times). If a given non-terminal can be replaced by multiple expressions, each one is usually on a separate line. Otherwise, the vertical slash (|) is used to separate them. A backslash (\) indicates that the rest of the line is a comment.

*source-file*:

- `[pkg-stmt][import-stmt]... [priv-cls-def]... pub-cls-def [priv-cls-def]...`

*pkg-stmt*:

- `package pkg-list`

*pkg-list*:

- `pkg-name`

- ( *pkg-name...* )

*pkg-name*:

- *identifier*

*import-stmt*:

- **import** *import-list*

*import-list*:

- ( *pkg-name... cls-name* )
- ( *pkg-name... int-name* )
- ( *pkg-name... \** )

*pub-cls-def*:

- *pub-class-def*
- *pub-interf-def*

*priv-cls-def*:

- *priv-class-def*
- *priv-interf-def*

*class-def*:

- *pub-class-def*
- *priv-class-def*

*interf-def*:

- *pub-interf-def*
- *priv-interf-def*

*pub-class-def*:

- ( **class** *pub-class-modif class-name [extends-clause][implements-clause] class-body* )

*priv-class-def*:

- ( **class** [*priv-class-modif*] *class-name [extends-clause][implements-clause] class-body* )

*pub-class-modif*:

- ( **public abstract** )
- ( **public final** )
- ( **public** )

*priv-class-modif:*

- ( **abstract** )
- ( **final** )

*class-name:*

- *cls-name*
- ( *pkg-name... cls-name* )

*extends-clause:*

- **extends** *class-name*

*pub-interf-def:*

- ( **interface** ( **public** ) *interf-name* [*extends-interf-clause*] *interf-body* )

*priv-interf-def:*

- ( **interface** *interf-name* [*extends-interf-clause*] *interf-body* )

*extends-interf-clause:*

- **extends** ( [*interf-name*]... )

*interf-name:*

- *int-name*
- ( *pkg-name... int-name* )

*class-body:*

- [*field-var-list*]... [*stat-obj-init*] *method-def*...

*interf-body:*

- [*field-var-list*]... *method-hdr*...

*method-def:*

- *proc-def*
- *func-def*
- *cons-def*
- *event-handler*

*proc-def:*

- ( **proc** [*modifier-list*] *method-name* [*parm-list*] [*throws-clause*] [*loc-var-list*]... *do-with block* )

*func-def:*

- ( **func** [*modifier-list*] *type* *method-name* [*parm-list*] [*throws-clause*] [*loc-var-list*]... *do-with block* )

*cons-def:*

- ( **cons** [*modifier-list*] *class-name* [*parm-list*] [*throws-clause*] [*loc-var-list*]... *do-with-cons-block* )

*event-handler:*

- ( **proc** ( **auto** ) *method-name* ( *event-class-name identifier* ) [*throws-clause*] [*loc-var-list*]... *do-with-block* )

*method-hdr:*

- *proc-hdr*
- *func-hdr*

*proc-hdr:*

- ( **proc** [*modifier-list*] *method-name* [*parm-list*] [*throws-clause*] )

*func-hdr:*

- ( **func** [*modifier-list*] *type method-name* [*parm-list*] [*throws-clause*] )

*throws-clause:*

- **throws** *exception-list*

*exception-list:*

- *exception*
- ( *exception...* )

*exception:*

- *exception-class-name*

*modifier-list:*

- ( *modifier...* )

*modifier:*

- **public**
- **protected**
- **private**
- **final**
- **static**
- **abstract** \ \_\_\_methods only
- **synchronized** \ \_\_\_methods only
- **transient** \ \_\_\_variables only

- **volatile** \ \_\_\_variables only

*stat-obj-init:*

- [*stat-init*] [*obj-init*]
- [*obj-init*] [*stat-init*]

*stat-init:*

- **static block**

*obj-init:*

- **do block**

*parm-list:*

- ( *parm* [ ; *parm* ]... )

*parm:*

- *type id-list*

*field-var-list:*

- **var** [*modifier-list*] ( *field-decl* [ ; *field-decl* ]... )

*loc-var-list:*

- **var** [*modifier-list*] ( *loc-decl* [ ; *loc-decl* ]... )

*field-decl:*

- [*modifier-list*][*property-decl*] *type id-list* [*expr*]
- [*modifier-list*] **typedef** *type type-name*

*loc-decl:*

- [*modifier-list*] *type id-list* [*expr*]
- [*modifier-list*] **typedef** *type type-name*

*property-decl:*

- ( **property read** )
- ( **property write** )
- ( **property read write** )

*id-list:*

- *identifier*
- ( *identifier*... )

*type:*

- *simple-type*
- *class-name*
- **list** [*type*]
- **set** *set-type*
- **array** [*dim-size*] *type*
- **enum** ( *identifier...* )

*simple-type:*

- **byte**
- **short**
- **int**
- **long**
- **float**
- **double**
- **char**
- **boolean**

*set-type:*

- *enum-type*
- ( .. *const const* )

*dim-size:*

- *pos-integer* \ zero-based
- *neg-integer* \ one-based
- 0

*pos-integer:*

*neg-integer:*

- *integer-constant*

*do-with:*

- **do**
- *with-do*

*with-do:*

- **with** *with-obj* **do**

*with-obj:*

- *identifier*
- ( *identifier*... )

*block*:

- ( [*stmt* ; ]... )
- ( *stmt* [ ; *stmt* ]... )

*stmt*:

- *if-stmt*
- *switch-stmt*
- *while-stmt*
- *for-stmt*
- *with-stmt*
- *assignment-stmt*
- *inc-dec-stmt*
- *ptr-stmt*
- *call-stmt*
- *jump-stmt*
- *try-stmt*
- *throw-stmt*
- *synch-stmt*

*if-stmt*:

- **if** *bool-expr* **then** *block* [*elseif-clause*]... [ **else** *block* ]

*elseif-clause*:

- **elseif** *bool-expr* **then** *block*

*switch-stmt*:

- **switch** *expr* ( *case-clause*... [ **default** *block* ] )

*case-clause*:

- **case** *expr-list* **do** *block*

*while-stmt*:

- **while** *bool-expr* **do** *block*
- **do** *block* **while** *bool-expr*

*for-stmt*:

- **for** *identifier* ( *init-expr limit-expr* ) [ **by** *step-expr* ] **do** *block*
- **for** *identifier list-variable* [ **by** *step-expr* ] **do** *block*

*with-stmt:*

- **with-do** *block*

*assignment-stmt:*

- *asst-op var-expr expr*

*inc-dec-stmt:*

- **++** *var-expr*
- **--** *var-expr*

*ptr-stmt:*

- **pchild** *var-expr expr*
- **pnext** *var-expr expr*

*call-stmt:*

- *proc-name* [*expr*]...
- **:** *colon-head* [*colon-expr*]... *proc-name*
- **:** *colon-head* [*colon-expr*]... ( *proc-name expr*... )

*jump-stmt:*

- **break**
- **continue**
- **return** [*expr*]

*try-stmt:*

- **try** *block* [*catch-clause*]... [ **finally** *block* ]

*catch-clause:*

- **catch** ( *exception exception-var* ) **do** *block*

*throw-stmt:*

- **throw** *throw-obj*

*synch-stmt:*

- **synchronized** *obj-name* **do** *block*

*expr:*

- *const-expr*

- *var-expr*
- ( *op expr...* )
- *func-call*
- *array-ref*
- *obj-expr*
- *list-expr*

*var-expr:*

- *var-name*
- *prop-name*
- *array-ref*
- ( : *colon-head [colon-expr]... var-end-expr* )

*var-end-expr:*

- *prop-name*
- *array-ref*

*obj-expr:*

- *obj-name*
- ( : *colon-head colon-expr...* )

*func-call:*

- *func-name*
- ( *func-name expr...* )

*array-ref:*

- *array-name*
- ( *array-name expr...* )

*colon-head:*

- *obj-name*
- *func-call*
- ( *array-name expr...* )

*colon-expr:*

- *func-call*
- *prop-name*
- *class-name*

- *array-ref*

*expr-list:*

- *expr*
- ( *expr...* )

*list-expr:*

- ( **list** *expr...* )

*const-expr:*

- *int-const*
- *float-const*
- *string-const*
- *bool-const*
- **nil**

*asst-op:*

- = | += | -= | \*= | /= | %= | =& | ^= | <<= | => | =>>
- =|

*op:*

- + | - | \* | / | % | & | ^ | << | >> | >>> | ?
- |
- == | <> | > | < | >= | <= | not | and | or | xor | in
- pchild | pnext | new | resize | instanceof

*bool-const:*

- **true**
- **false**

*cls-name:*

*int-name:*

*method-name:*

*proc-name:*

*func-name:*

*obj-name:*

*prop-name:*

*array-name:*

*type-name:*

*list-variable:*

*exception-class-name:*

- *identifier*

*identifier*: \ \_\_\_no white space allowed between tokens

- *first-char* [*id-char*]...

*first-char*:

- *underscore*
- *letter*

*id-char*:

- *underscore*
- *letter*
- *digit*
- *hyphen*

*alphanumeric*:

- *digit*
- *letter*

*digit*:

- 0 | 1 | ... | 9

*letter*:

- A | B | ... | Z
- a | b | ... | z

*underscore*:

- \_

*hyphen*: \ \_\_\_must be immediately preceded and succeeded by *alphanumeric*

- -

## VEC Grammar

The Virtual Environment Code is an intermediate language which specifies the format of the .VEC files, which are the output of the Vecscript Compiler and the input of the VEC Loader. The VEC Loader converts the . VEC files to an in-memory format, which is then executed by the Vecset Runtime.

### Grammar Format

The following Grammar defines the syntax of the .VEC files.

- Non-terminal symbols are written in italics, e.g. *while-stmt*. A non-terminal symbol is a place-holder for an entity in the Grammar which can be replaced by lower-level symbols.

- Terminal symbols appear in a monospaced font, e.g. `#func`. Terminal symbols are bottom-level entities in the Grammar, which cannot be replaced by any other symbols.
- Square brackets (`[]`) enclose something that is optional (can occur 0 or 1 times).
- Square brackets followed by an ellipsis (`...`) enclose something that repeats (can occur 0 or more times).
- A non-terminal symbol followed by an ellipsis indicates repetition (can occur 1 or more times).
- If a given non-terminal can be replaced by multiple expressions, each one is usually on a separate bulleted line. Otherwise, the vertical slash (`|`) is used to separate them.
- A backslash (`\`) indicates that the rest of the line is a comment.

## VEC Design Specs

*class-decl:*

```
(  
  id  
  modifs \ set  
  imp-idx  
  cls-idx  
  ( interf-ref... )  
  ( fld-decl... )  
  static-blk  
  do-blk  
  ( meth-decl... )  
)
```

*interf-ref:*

```
imp-idx interf-idx
```

*meth-decl:*

```
(  
  meth-typ  
  func-typ | ()  
  modifs \ set  
  ( parm-decl... )  
  ( loc-decl... )  
  ( except-typ... )  
  ( stmt... )  
  id  
)
```

*meth-typ:*

- #proc
- #func
- #cons
- #event-hndlr

*func-typ:*

```
(  
  scalar-typ  
  ( arr-lst... )  
  [ imp-idx cls-idx ]  
)
```

*loc-decl:*

```
(  
  value  
  scalar-typ  
  ( arr-lst... )  
  imp-idx  
  cls-idx  
  ( minval maxval ) \ int  
  id  
  [ expr ] \ loc-decl only  
)
```

*parm-decl:*

```
loc-decl
```

*fld-decl:*

```
(  
  value  
  scalar-typ  
  ( arr-lst... )  
  imp-idx  
  cls-idx  
  modifs \ set  
  property  
  ( minval maxval ) \ int  
  id  
  [ expr ]  
)
```

*simp-typ:*

- \$st-byte
- \$st-short
- \$st-int
- \$st-long
- \$st-float
- \$st-double
- \$st-char
- \$st-boolean
- \$st-object
- \$st-listnode
- \$st-set
- \$st-setlong

*property:*

- \$rw-read
- \$rw-write
- \$rw-readwrite

*arr-lst:*

- A [ *size* ]
- L

*scalar-typ:*

```
simp-typ | meth-typ
```

*meth-typ:*

```
(  
  #proc-typ | #func-typ  
  simp-parm...  
)
```

*simp-parm:*

```
(  
  simp-typ  
  id  
  [ imp-idx cls-idx ]  
)
```

```

source-file:
  pkg-stmt
  ( imp-stmt... )
  pub-cls-def
  [ priv-cls-def... ]

pkg-stmt:
  ( cls-name... )

imp-stmt:
  (
    imp-typ
    ( cls-name... )
    [ cls-idx | interf-idx ]
  )
imp-typ:
  • $imp-class
  • $imp-interf
  • $imp-all

pub-cls-def:
priv-cls-def:
  • class-decl
  • interf-decl

interf-decl:
  (
    id
    modifs \ set
    ( interf-ref... )
    ( fld-decl... )
    ( meth-hdr... )
  )
meth-hdr:
  (
    #proc-hdr | #func-hdr
    func-typ | ()
    modifs \ set
    ( parm-decl... )
    ( except-typ... )
    id
  )

call-stmt:
  • proc-ref
  • colon-proc-expr

colon-proc-expr:
  (
    #colon-proc
    obj-expr
    proc-ref
  )
asst-stmt:
  (
    asst-op
    var-expr
    expr
  )
var-expr:
  • var-ref
  • arr-ref
  • colon-prop-expr

static-blk:
do-blk:
  (
    #stat-blk | #do-blk
    ( loc-decl... )
    block
  )

```

```

colon-prop-expr:
(
  #colon-prop
  obj-expr
  var-ref | arr-ref
)
colon-meth-expr:
(
  #colon-meth
  obj-expr
  meth-ref
)
var-ref:
(
  #var-ref
  value
  decl-idx
  is-local
  scalar-typ
  id
)
obj-ref:
(
  #obj-ref
  value
  decl-idx
  is-local
  imp-idx
  cls-idx
  id
)
meth-ref:
(
  #proc-ref | #func-ref
  decl-idx
  scalar-typ
  ( [ expr ]... )
  id
)
obj-expr:

- obj-ref
- func-ref
- arr-ref
- colon-prop-expr
- colon-meth-expr

func-ref:
proc-ref:
  meth-ref

```

```

arr-ref:
(
  #arr-ref
  value
  decl-idx
  is-local
  scalar-typ
  ( expr... )
  id
)
expr:

- const-expr
- var-expr
- op-expr
- func-ref
- arr-ref
- obj-expr
- list-expr

const-expr:

- int-const
- float-const
- string-const
- bool-const
- nil

op-expr:

- bin-op-expr
- multi-op-expr
- unary-op-expr
- misc-expr

bin-op-expr:
(
  bin-op
  expr
  expr
)
multi-op-expr:
(
  multi-op
  expr...
)
unary-op-expr:
(
  unary-op
  expr
)
list-expr:
(
  #list-expr
  expr...
)

```

```

block:
(
  [ stmt... ]
)
stmt:
• if-stmt
• switch-stmt
• while-stmt
• for-stmt
• for-list-stmt
• asst-stmt
• inc-dec-stmt
• ptr-stmt
• call-stmt
• jump-stmt
• try-stmt
• throw-stmt
• synch-stmt
• resize-stmt

if-stmt:
(
  $stmt-if
  if-clause...
)
if-clause:
  bool-expr
  block

switch-stmt:
(
  $stmt-switch
  expr
  ( case-clause... )
  [ block ]
)
case-clause:
(
  ( expr... )
  block
)
while-stmt:
(
  $stmt-while | $stmt-repeat
  bool-expr
  block
)
throw-stmt:
(
  $stmt-throw
  obj-ref
)

for-stmt:
(
  $stmt-for
  decl-idx
  ( init-expr limit-expr step-expr )
  block
)
for-list-stmt:
(
  $stmt-for-list
  var-expr \ list
  is-reverse
  block
)
inc-dec-stmt:
(
  $stmt-inc | $stmt-dec
  var-expr
)
ptr-stmt:
(
  $stmt-left | $stmt-right
  var-expr
  expr
)
jump-stmt:
• return-stmt
• $stmt-brk
• $stmt-cont

return-stmt:
(
  $stmt-rtn
  [ expr ]
)
try-stmt:
(
  $stmt-try
  block
  ( catch-clause... )
  [ block ]
)
catch-clause:
(
  decl-idx \ exception decl.
  block
)
synch-stmt:
(
  $stmt-sync
  obj-ref
  block
)

```

*misc-expr:*

- *cond-expr*
- *cast-cls*
- *cast-typ*
- *inst-of*
- *new-expr*
- *quote-expr*

*cond-expr:*

```
(  
  #cond-expr  
  bool-expr  
  expr  
  expr  
)
```

*cast-cls:*

```
(  
  $bop-cast-cls  
  imp-idx  
  cls-idx  
  expr  
)
```

*cast-typ:*

```
(  
  $bop-cast-typ  
  simp-typ  
  expr  
)
```

*inst-of:*

```
(  
  $bop-inst-of  
  imp-idx  
  cls-idx  
  expr  
)
```

*new-expr:*

```
(  
  $uop-new  
  imp-idx  
  cls-idx  
  [ expr... ]  
)
```

*resize-stmt:*

```
(  
  $stmt-resize  
  colon-prop-expr | arr-ref  
  expr...  
)
```

*quote-expr:*

```
(  
  $uop-quote  
  expr  
)
```

*int-const:*

[ *hyphen* ] *digit...*

*float-const:*

F [ *hyphen* ] *digit...* [ *dec* ][ *exp* ]

*string-const:*

*quote* [ *char...* ] *quote*

*bool-const:*

- Y
- N

*dec:*

*dec-pt digit...*

*exp:*

E [ *hyphen* ] *digit...*

*id:*

*underscore identifier*

*identifier:*

*id-start id-char...*

*underscore:*

–

*hyphen:*

-

*quote:*

`

*dec-pt:*

.

*id-start:*

- *underscore*
- *letter*

*id-char:*

- *underscore*
- *letter*
- *digit*
- *hyphen*

*bin-op:*

- \$bop-plus
- \$bop-minus
- \$bop-mult
- \$bop-div
- \$bop-mod
- \$bop-eq
- \$bop-ne
- \$bop-lt
- \$bop-le
- \$bop-ge
- \$bop-gt
- \$bop-and
- \$bop-or
- \$bop-xor
- \$bop-shl
- \$bop-shr
- \$bop-zshr
- \$bop-setun
- \$bop-setdiff
- \$bop-setinter
- \$bop-seteq
- \$bop-setne
- \$bop-setinc
- \$bop-setin
- \$bop-concat
- \$bop-lstidx
- \$bop-cast-cls
- \$bop-cast-typ
- \$bop-inst-of

*multi-op:*

- \$mop-plus
- \$mop-mult
- \$mop-and
- \$mop-or
- \$mop-xor
- \$mop-setun
- \$mop-setinter
- \$mop-concat

*unary-op:*

- \$uop-neg
- \$uop-left
- \$uop-right
- \$uop-not
- \$uop-atom
- \$uop-new
- \$uop-quote

*asst-op:*

- \$asst
- \$aop-plus
- \$aop-minus
- \$aop-mult
- \$aop-div
- \$aop-mod
- \$aop-and
- \$aop-or
- \$aop-xor
- \$aop-shl
- \$aop-shr
- \$aop-zshr
- \$aop-concat

*modifiers:*

- \$mod-pub
- \$mod-prot
- \$mod-priv
- \$mod-fin
- \$mod-stat
- \$mod-ab
- \$mod-sync
- \$mod-trans
- \$mod-vol
- \$mod-auto

## Sample Code

The following Delphi/Vecscript programs count the no. of lines of code, minus white space, in all Pascal source files contained in all selected Delphi project file(s). The user must select the desired Delphi project file(s) using a standard Open File dialog box. Only the lines of code between "implementation" and "end." are counted. The Delphi code listing is immediately followed by the code listing of an equivalent Vecscript program.

## Delphi Code Counter

This Delphi program counts the no. of lines of code in all source files of the user-selected Delphi project file(s).

```
unit CountMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs,
  IniFiles, StdCtrls;

type
  TZCountForm = class(TForm)
    SelBtn: TButton;
    XLb_Count: TLabel;
    CountBtn: TButton;
    OpenFileDialog: TOpenDialog;
    XLb_ProjName: TLabel;
    procedure SelBtnClick(Sender: TObject);
    procedure CountBtnClick(Sender: TObject);
    procedure FormActivate(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    { Private declarations }
    IniFullFileName: string;
    DprFullFileName: string;
    DprFilePath: string;
    DprFileList: TStringList;
    PasFileList: TStringList;
    DprFiles: TStrings;
    KeyFullFileName: string;
    KeywordList: TStringList;
    procedure SetBtnsAndLabels;
    function GetDprCount(DprFullFileName: string): LongInt;
    function GetLineCount(FullFileName: string): LongInt;
    function IsWhiteSpace(Buf: string): Boolean;
  public
    { Public declarations }
  end;
```

```

var
    ZCountForm: TZCountForm;

implementation

{$R *.DFM}

const
    DefSect = 'Settings';

procedure TZCountForm.FormCreate(Sender: TObject);
begin
    DprFileList := TStringList.Create;
    PasFileList := TStringList.Create;
    KeywordList := TStringList.Create;
end;

procedure TZCountForm.FormDestroy(Sender: TObject);
begin
    DprFileList.Free;
    PasFileList.Free;
    KeywordList.Free;
end;

procedure TZCountForm.FormActivate(Sender: TObject);
var
    I: LongInt;
    IniFile: TIniFile;
    ZCountPath: string;
begin
    XLb_Count.Caption := '';
    ZCountPath := ExtractFilePath(Application.ExeName);
    KeyFullFileName := ZCountPath + 'Keywords.txt';
    KeywordList.LoadFromFile(KeyFullFileName);
    for I := KeywordList.Count - 1 downto 0 do
    begin
        KeywordList[I] := Trim(LowerCase(KeywordList[I]));
        if KeywordList[I] = '' then
            KeywordList.Delete(I);
        end;
    end;
    IniFullFileName := ZCountPath +
        'ZCount.ini';
    IniFile := TIniFile.Create(IniFullFileName);
    try
        DprFiles := nil;
        DprFullFileName := IniFile.ReadString(DefSect,
            'Path', '');
        SetBtnsAndLabels;
    finally
        IniFile.Free;
    end;
end;

procedure TZCountForm.SetBtnsAndLabels;
begin
    CountBtn.Enabled := FileExists(DprFullFileName);
    XLb_Count.Caption := '';

```

```

    XLb_ProjName.Caption := DprFullFileName;
    DprFilePath := ExtractFilePath(DprFullFileName);
end;

procedure TZCountForm.SelBtnClick(Sender: TObject);
var
    IniFile: TIniFile;
begin
    OpenFileDialog.InitialDir := ExtractFilePath(DprFullFileName);
    if not OpenFileDialog.Execute then
        Exit;
    DprFullFileName := OpenFileDialog.FileName;
    DprFiles := OpenFileDialog.Files;
    IniFile := TIniFile.Create(IniFullFileName);
    try
        IniFile.WriteString(DefSect, 'Path', DprFullFileName);
        SetBtnsAndLabels;
    finally
        IniFile.Free;
    end;
end;

procedure TZCountForm.CountBtnClick(Sender: TObject);
var
    LineCount: LongInt;
    I: LongInt;
    FullFileName: string;
begin
    LineCount := 0;
    Screen.Cursor := crHourGlass;
    try
        if DprFiles = nil then
            begin
                LineCount := GetDprCount(DprFullFileName);
                XLb_Count.Caption := IntToStr(LineCount);
                Update;
            end
        else begin
            for I := 0 to DprFiles.Count - 1 do
                begin
                    FullFileName := DprFiles[I];
                    LineCount := LineCount + GetDprCount(FullFileName);
                    XLb_Count.Caption := IntToStr(LineCount);
                    Update;
                end;
            end;
        finally
            Screen.Cursor := crDefault;
        end;
    end;
end;

function TZCountForm.GetDprCount(DprFullFileName: string): LongInt;
var
    LineCount: LongInt;
    I, J: LongInt;
    Buf: string;
    FileName: string;

```

```

    FullFileName: string;
    QuoteStr: string;
begin
    LineCount := 0;
    QuoteStr := '';
    DprFileList.LoadFromFile(DprFullFileName);
    for I := 0 to DprFileList.Count - 1 do
    begin
        Buf := DprFileList[I];
        J := Pos(QuoteStr, Buf);
        if J <= 0 then
            Continue;
        FileName := Copy(Buf, J + 1, Length(Buf));
        FileName := Copy(FileName, 1, Pos(QuoteStr, FileName) - 1);
        FileName := Trim(FileName);
        FullFileName := DprFilePath + FileName;
        LineCount := LineCount + GetLineCount(FullFileName);
    end;
    Result := LineCount;
end;

function TZCountForm.GetLineCount(FullFileName: string): LongInt;
var
    I: LongInt;
    Buf: string;
    StartLn, EndLn: LongInt;
    TopWSpaceCount: LongInt;
    WSpaceCount: LongInt;
begin
    Result := 0;
    if not FileExists(FullFileName) then
        Exit;
    PasFileList.LoadFromFile(FullFileName);
    TopWSpaceCount := 0;
    WSpaceCount := 0;
    StartLn := 1;
    EndLn := PasFileList.Count;
    for I := 0 to PasFileList.Count - 1 do
    begin
        Buf := Trim(LowerCase(PasFileList[I]));
        if Buf = 'implementation' then
            StartLn := I + 1
        else if Buf = 'end.' then
            begin
                EndLn := I - 1;
                Break;
            end
        else if not IsWhiteSpace(Buf) then
            else if StartLn > 1 then
                Inc(WSpaceCount)
            else
                Inc(TopWSpaceCount)
        end;
        if WSpaceCount <= 0 then
            WSpaceCount := TopWSpaceCount;
    end;
    Result := EndLn - StartLn - WSpaceCount + 1;
end;

```

```

function TZCountForm.IsWhiteSpace(Buf: string): Boolean;
var
  I: LongInt;
  Ch: Char;
  WordStr: string;
begin
  Result := False;
  Buf := Buf + ' ';
  WordStr := '';
  for I := 1 to Length(Buf) do
  begin
    Ch := Buf[I];
    if Ch in ['a'..'z'] then
    begin
      WordStr := WordStr + Ch;
    end
    else if WordStr <> '' then
    begin
      Result := KeywordList.IndexOf(WordStr) >= 0;
      if not Result then
        Exit;
      Result := False;
      WordStr := '';
    end
    else if not (Ch in [' ', ';']) then
      Exit
    else begin
      WordStr := '';
    end;
  end;
  Result := True;
end;

end.

```

## Vecscript Code Counter

This Vecscript program counts the no. of lines of code in all Pascal source files of the user-selected Delphi project file(s). It has exactly the same functionality as the previous Delphi code listing.

```

import (windows *)
import (ini-files *)

(class (public) Count-form extends Form
  var (auto) (
    Button sel-btn;
    Button count-btn;
    Label xlb-count;
    Label xlb-proj-name;
    Open-dialog open-dialog;
  )
  var (private) (
    String ini-full-file-name;
    String dpr-full-file-name;
  )
end

```

```

String key-full-file-name;
String dpr-file-path;
list String dpr-file-list;
list String pas-file-list;
list String keyword-list;
list String dpr-files;
)
var (static) (
String def-sect ('Settings');
)
(proc (public static) main (String-list args)
var (
Count-form form1 (new Count-form);
)
)
(cons (public) Count-form
do (
= dpr-file-list (new list String);
= pas-file-list (new list String);
= key-file-list (new list String);
show;
)
)
(proc (auto) Count-form_activate (Event e)
var (
int i;
Ini-file ini-file;
String zcount-path;
)
do (
= (: xlb-count caption) '';
= zcount-path (extract-file-path
(: application exe-name));
= key-full-file-name (+ zcount-path 'Keywords.txt');
: keyword-list (load-from-file key-full-file-name);
for i keyword-list do (
= (: keyword-list i) (trim (lower-case
(: keyword-list i)));
if (== (: keyword-list i) '') then (
: keyword-list (delete i);
-- i;
);
);
= ini-full-file-name (+ zcount-path 'ZCount.ini');
= ini-file (new (Ini-file ini-full-file-name));
= dpr-files nil;
= dpr-full-file-name (: ini-file (read-string
def-sect 'Path' ''));
set-btns-and-labels;
)
)
(proc (private) set-btns-and-labels
do (
= (: count-btn enabled)
(file-exists dpr-full-file-name);
= (: xlb-count caption) '';
= (: xlb-proj-name caption) dpr-full-file-name;

```

```

    = dpr-file-path (extract-file-path dpr-full-file-name);
  );
)
(proc (auto) sel-btn_click (Event e)
  var (
    Ini-file ini-file;
  )
  do (
    = (: open-dialog initial-dir)
      (extract-file-path dpr-full-file-nams);
    if (not (open-dialog execute)) then (
      exit;
    );
    = dpr-full-file-name (open-dialog file-name);
    = dpr-files (open-dialog files);
    = ini-file (new (Ini-file ini-full-file-name));
    : ini-file (write-string def-sect 'Path'
      dpr-full-file-name);
    set-btns-and-labels;
  )
)
(proc (auto) count-btn_click (Event e)
  var (
    int line-count;
    int i;
    String full-file-name;
  )
  do (
    = line-count 0;
    = (: screen cursor) cr-hour-glass;
    try (
      if (== dpr-files nil) then (
        = line-count (get-dpr-count dpr-full-file-name);
        = (: xlb-count caption) line-count;
        update;
      )
      else (
        for i dpr-files do (
          = full-file-name (: dpr-files i);
          += line-count (get-dpr-count full-file-name);
          = (: xlb-count caption) line-count;
          update;
        );
      );
    )
    finally (
      = (: screen cursor) cr-default;
    );
  )
)
(func int get-dpr-count (String dpr-full-file-name)
  var (
    int line-count;
    int (i j);
    String buf;
    String file-name;
    String full-file-name;

```

```

    String quote-str;
  )
do (
  = line-count 0;
  = quote-str '\';
  : dpr-file-list (load-from-file dpr-full-file-name);
  for i dpr-file-list do (
    = buf (: dpr-file-list i);
    = j (pos quote-str buf);
    if (<= j 0) then (
      continue;
    );
    = file-name (copy buf (+ j 1) (length buf));
    = file-name (copy file-name 1
      (- (pos quote-str file-name) 1));
    = file-name (trim file-name);
    = full-file-name (+ dpr-file-path file-name);
    += line-count (get-line-count full-file-name);
  );
  return line-count;
)
)
(func int get-line-count (String full-file-name)
  var (
    int i;
    String buf;
    int start-ln;
    int end-ln;
    int top-wspace-count;
    int wspace-count;
  )
do (
  if (not (file-exists full-file-name)) then (
    return 0;
  );
  : pas-file-list (load-from-file full-file-name);
  = top-wspace-count 0;
  = wspace-count 0;
  = start-ln 1;
  = end-ln (: pas-file-list count);
  for i pas-file-list do (
    = buf (trim (lower-case (: pas-file-list i)));
    if (== buf 'implementation') then (
      = start-ln (+ i 1);
    )
    elseif (== buf 'end.') then (
      = end-ln (- i 1);
      break;
    )
    elseif (not (is-white-space buf)) then ( )
    elseif (> start-ln 1) then (
      ++ wspace-count;
    )
    else (
      ++ top-wspace-count;
    );
  );
);
);

```

```

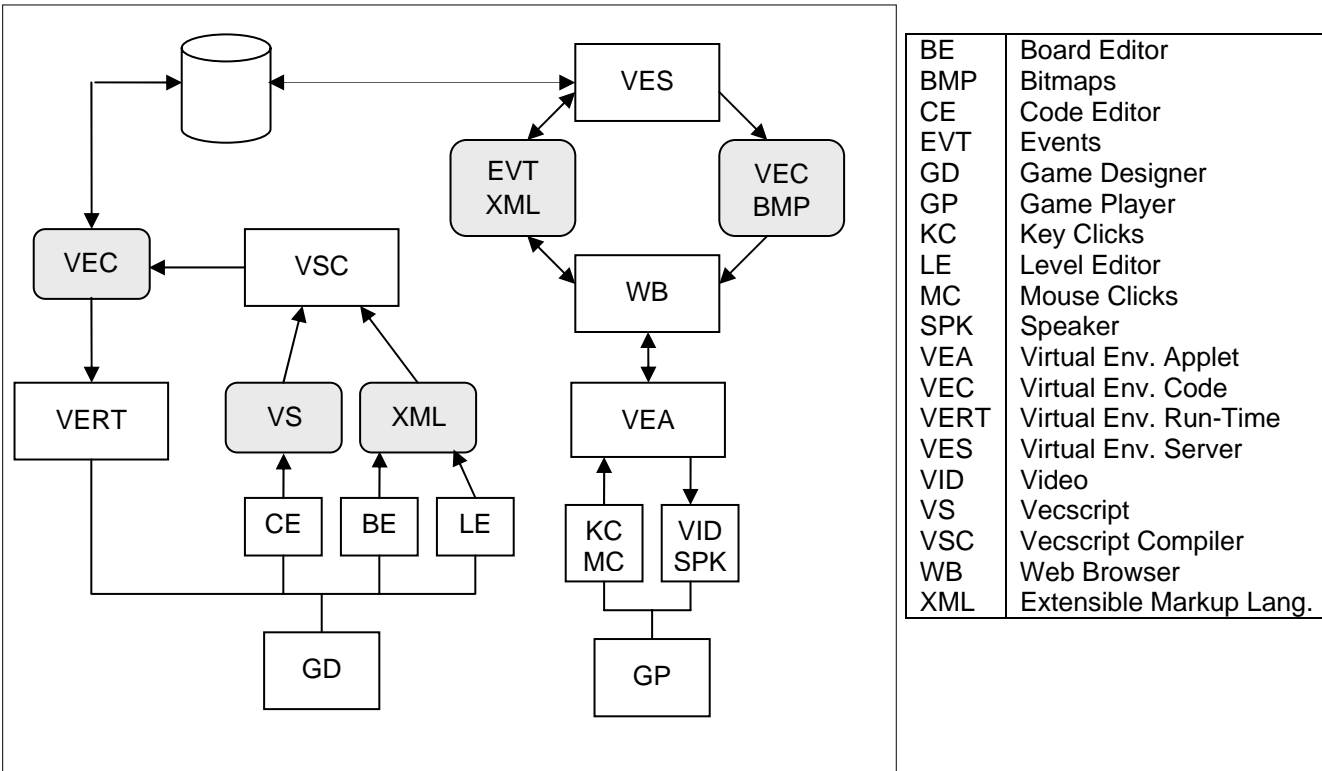
    if (<= wspace-count 0) then (
      = wspace-count top-wspace-count;
    );
    return (+ end-ln (- start-ln) (- wspace-count) 1);
  )
)
(func boolean is-white-space (String buf)
  var (
    int i;
    char ch;
    String word-str;
    boolean result;
  )
  do (
    += buf ' ';
    = word-str '';
    for i (1 (length buf)) do (
      = ch (buf i);
      if (in ch (... 'a' 'z')) then (
        += word-str ch;
      )
      elseif (<> word-str '') then (
        = result (>=
          (: keyword-list (index-of word-str)) 0);
        if (not result) then (
          return false;
        );
        = word-str '';
      )
      elseif (not (in ch (' ' ';))) then
        return false;
      )
      else (
        = word-str '';
      );
    );
  );
  return true;
)
)
)

```

# Language Implementation

The following block diagram illustrates how the different parts of Vecset are interrelated. The VES resides on the VECsworld.com server.

## VECSET Block Diagram



## Lists

Vecscript has built-in support for linked lists and list trees. Both are based on a data structure called a *list node*, which contains 2 pointers, a left pointer and a right pointer. Every linked list has an associated data type, in which the left pointer always points to an object of this type or any of its subclasses, and the right pointer always points to another list node or is nil.

List trees consist of one or more list nodes, in which both the left and the right pointers can point to another list node or an object of any type. To determine whether or not a given pointer points to another list node, use the built-in boolean function *atomic*. This function returns false if its single argument points to a list node, and true if it points to an ordinary object.

## List Trees

The following procedure demonstrates how to create a simple list tree containing 2 list nodes:

```
(proc tree-demo
  var (
    list my-node (new list);
    list root;
    My-class my-obj (new My-class);
  )
  do (
    pchild my-node my-obj;
    pnext my-node (new list);
    = root my-node;
    = my-node (pnext my-node);
    if (atomic (pchild root)) then (
      \ yes it is atomic
    );
    if (not (atomic root)) then (
      \ no it's not atomic
    );
  )
)
```

## Linked Lists

The List class is used to maintain linked lists in which the left pointer of each list node points to an object of a given type. To declare an object of class List, use the keyword "list" followed by a class name, followed by the name of the object. Example:

```
list Foo my-list;
```

All elements of my-list are assumed to be objects of the class Foo or any of its sub-classes. Use the colon (:) operator to refer to the n-th element of the list, example:

```
(: my-list n)
```

```
\ Summary of List class
```

```
(class List
  var (private)(
    list first-node;
    list last-node;
    list curr-node;
    int curr-idx;      \ index of current element
    int count;        \ no. of elements in list
  )
  (proc first ... )   \ go to top of list
  (proc last ... )   \ go to bottom of list
  (proc next ... )   \ go to next element of list
  (proc prior ... )  \ go to previous element of list
  (func boolean is-top ... )   \ top of list?
  (func boolean is-bottom ... ) \ bottom of list?
  (func int get-curr-idx ... )  \ get index of current element
  (func int get-count ... )    \ get no. of elements in list
  (func list get-obj ... )     \ get ptr to current element
  (func list get-node ... )    \ get ptr to current list node
```

```

(func list get-nth (int idx) ... ) \ get ptr to nth element
(func boolean goto (int idx) ... ) \ go to nth element of list
(proc add (list obj) ... ) \ insert obj at end of list
(proc insert (list obj) ... ) \ insert obj before current element
(proc replace (list obj) ... ) \ replace current element with obj
(proc delete ... ) \ delete current element
(proc set-nth (int idx; list obj) ... ) \ set ptr to nth element
)

```

## Memory Management

All Vecscript data in memory is stored in 256-byte *pages*. There are 2 kinds of pages: list node pages and array pages. List node pages consist of 25 *list nodes*, while array pages consist of one list node followed by 240 bytes of array data. Each element of this array is either 1 byte or 2 bytes long. List nodes are 10 bytes long, and consist of a 16-bit header and two 32-bit pointers. The first pointer is referred to as the *left pointer*, and the second pointer is referred to as the *right pointer*.

### Data Structures

- **Linked List:** used for operator stack, operand stack, expressions, arrays of 4- or 8-byte values, lists of array pages, and the free-node list.
- **Tree:** similar to a linked list, except that each element of the list can itself be a list, to an arbitrary depth. Used for program code, multi-dimensional arrays, and large one-dimensional arrays.
- **Array Pages:** used for arrays of 1- or 2-byte values. If array is larger than 240 bytes, a linked list of array pages is used to represent the array.
- **Dynamic Arrays:** all arrays are dynamic, meaning that they can grow or shrink at run-time.
- **Sets:** stored in array pages; maximum length =  $240 \times 8 = 1920$  bits (no. of elements).
- **Atomic Values:** the values of primitive data types are stored in list nodes, replacing the left pointer (except for 8-byte values, which replace both the left and right pointers).

### Garbage Collection

The linked list of pages containing list nodes is sorted by no. of free nodes, so that list node allocation occurs in pages having the fewest free nodes.

Garbage collection is automatic, freeing the programmer from having to worry about freeing up memory used by data which is no longer needed.

## Game Server

The Game Server broadcasts XML files to the other game players whenever a given player makes a move or executes a command.

Whenever a non-local component changes, such as the push/pop events of the Board-grid object being triggered, an XML file is broadcast to all Vecscript clients (except the client who generated the event).