The Ferite Programming Language 1.1.19

November 5, 2010

Contents

1 Introduction

1.1 What is ferite?

ferite is a small robust scripting engine providing straight forward application integration, with the ability for the API to be extended very easily. The design goals for ferite are lightweight - small memory and CPU footprint, fast, thread-safe, and straight forward both for the programmer of the parent application and the programmer programming ferite scripts to learn the system.

1.2 What does this documentation provide?

This document is the official commentary on ferite, including language information such as constructs and known issues. An API guide for the standard objects provided with every ferite distribution and the means in which to embed ferite are provided separately.

1.3 Why should I choose ferite?

ferite is designed to be added into other applications. Providing a consistent API, your application will be able to stay binary compatible with the latest ferite engine, allowing you, the application programmer, to add powerful scripting to your application without the worry of the internals of ferite.

ferite provides a language very similar to that of C and Java with additional features from other languages such as: closures from scheme and namespaces from C++. This means that the skill set acquired through learning existing main-stream languages can be instantly applied to the creation of ferite scripts. ferite is not a heavy language and prides itself on being clean and to the point; it is easy to pick up a script written long ago and instantly understand what is going on, which, unfortunately, can not be said for some other languages.

ferite provides a framework for creating structured programs: variables must be declared at the beginning of a block, global variables must declared as such, critical variables and functions within a thread can be clearly defined as thread-safe. ferite has classes, objects, namespaces, modules, threading and exception handling. The scripts are more verbose than other languages but ferite makes it easy to write, maintain and debug.

If you are looking for a scripting engine that is thread-safe, allowing for thread-safe use within an application, ferite is the way to go. Not only does ferite remain thread-safe, it also uses the operating systems native threads; you can now have scripts that run safely across multiple processors.

2 Language Reference

2.1 Conventions Used

Within this document there are a number of conventions used to aid the delivery of examples when discussing various parts of the language.

To write a complete example for each and every part of the language would be both confusing and over kill. To aid the understanding of examples the following convention is used: There will be a -> symbol which means that the result of doing what is on the left hand side is on the right hand side. If you wish to copy the example you must remove the -> and the text that follows it. E.g.

```
1 + 2 -> 3
```

The above example demonstrates that the result of 1 + 2 is 3.

2.2 Scripts

A script file is a plain text file that usually ends in **.fe**. Each file consists of three main parts:

- A set of imports using the **uses** keyword to give you access to extra functionality (such as printing out text, string or mathematic functions).
- Zero or more function, class and namespace declarations.
- The code to be executed when the script is first run. E.g. If you can have a script that does some processing on each argument passed to it; you could have a loop going through each argument within the startup code which calls a function you have declared in the middle section of a script.

For example:

```
// The importing of extra functionality
uses "console", "array";

// Declaration of a function.  Classes and Namespaces also go here
function processArgument( string argument ) {
  Console.println( "Argument:  " + argument );
}

// The startup code
Array.each( argv ) using ( argument ) {
  processArgument( argument );
};
```

First thing; do not worry if you do not understand the above code, you will know what it means at the end of this manual.

The **startup code** is what is called when the script is run. The **startup code** is equivalent to the **main** method within a C or Java program.

Having just shown you a more involved script, here is the famous *Hello World* program:

```
uses "console";
Console.println( "Hello World from ferite" );
```

The 'uses' statement is used to import API either from an external module or from another script. Its use is described in greater depth later on in this manual. You may be wondering why it takes two lines of code to print out a line of text; `ferite` does not have any built in functions - they all have to be pulled in by the programmer. The result? The only thing used within `ferite` is what you need.

2.3 Comments

Writing documentation is not fun, but, it can make your and others lives much easier in the long run. Commenting code is the same boat. You can reduce the amount of commenting you do through the use of meaningful variable, function and class names, and writing clear code. However, there are times when it is not obvious what is happening within a block of code, which is when comments become important.

`ferite` supports two methods of commenting code: using block comments (/* */) or single line comments (//). Block comments cause `ferite` to ignore everything from the start of the comment '/*' to the end of the comment '*/'. Single line comments cause `ferite` to ignore everything from, and including, the comment start '//' to the end of the line. Comments can be used throughout the scripts you are writing.

```
// This is a single line comment
/*
 This is another comment.
 But it is for blocks.
 And can span multiple lines.
*/
```

`ferite` can handle nested block comments. This allows easy commenting out of code regardless of the comments or code that is within the block. Note that this is different from the default behavior of some other languages.

```
/*
  /* Print out some information */
  Console.println( "Today" );
*/
```

2.4 Types

A type is a hint from you, the programmer, to `ferite` on how to look after and deal with your information. `ferite` requires you to state the type of each variable or parameter you declare. This keeps writing code and debugging easier because `ferite` can stop you making mistakes - such as trying to add a string to an object. There are a number of types within the type system which fall into two main categories: simple types (**boolean**, **number**, and **string**) and complicated types (**array**, **object** and **void**).

In this section we talk about not only types but some operators and variables which are discussed later on in the manual. It is suggested that you read this section and come back to it when you have made more progress with the other sections.

2.4.1 boolean

The **boolean** type encapsulates a simple numerical type with the value true or false. It is interchangeable with the number type for values 1 and 0. `ferite` has two build in keywords `true` and `false` that provide the only two possible values a boolean variable can hold. All logical operators within the language reduce to a boolean value of either true or false.

```
boolean someValue = true;
boolean someOtherValue = false;
```

2.4.2 number

This type encapsulates all integer and floating point numbers within the 64bit IEEE specification and will automatically handle issues regarding overflow and conversion. In layman terms that means that they can store really big numbers and will convert between integer numbers (E.g. 1, 2, 14 - numbers without a decimal point) and floating point numbers (E.g. 3.14, 23.5 - numbers with a decimal point). Once the conversion has taken place, the number will remain a floating point number.

- All numbers start out as 64bit signed integers. When the value of a number goes above what can be stored in 64bits, the number will switch over to being a 64bit floating point number (allowing for much larger numbers; it should be noted that accuracy can potentially be tainted when this occurs). This is also true if the value to be stored is a floating point value. The point where the switch occurs is +-(2^64/2).
- Comparisons can be made between numbers but it should be noted that once a number has internally become a floating point number, equality comparisons can potentially give unexpected results. To try and solve this problem, when floating point numbers are being compared there is a slight amount of tolerance involved

which means that they do not need to be identical but very close in value. The default tolerance is **0.000001**. At the time of writing the default tolerance can not be changed by the programmer.

```
number someValue = 10;
number someOtherValue = 1.21;
number newValue = someValue + someOtherValue;
```

It is possible to use various different notations for numbers within a script. The different methods allow for different number bases. `ferite` supports decimal, real, binary, octal and hexadecimal notation. Most of the examples within this document use either decimal or real as they are easily recognized. These notations all end up being stored the same internally, however there are times when it is more useful to use either hexadecimal or binary to define values depending on what data you wish to manipulate. The following examples demonstrate how to use the binary, octal and hexadecimal notations.

To define a binary number, a set of '1' and '0' should be prefixed with '0b'.

```
number one = 0b1 -> one is the numerical value 1
number four = 0b100 -> four is the numerical value 4
number twentyone = 0b10101 -> twentyone is the numerical value 21
```

To define an octal number, a set of numbers from '0' to '7' should be prefixed with '0'.

```
number one = 01 -> one is the numerical value 1
number four = 04 -> four is the numerical value 4
number twentyone = 025 -> twentyone is the numerical value 21
```

To define a hexadecimal number, a set of numbers '0' to '9', and a set of alpha characters 'a' to 'f', should be prefixed with a '0x'.

```
number one = 0x1 -> one is the numerical value 1
number four = 0x4 -> four is the numerical value 4
number twentyone = 0x15 -> twentyone is the numerical value 21
```

Even though they provide different notations, the above should demonstrate how `ferite` interprets them to numbers of the same value.

2.4.3 string

Strings are specified using double quote ("") and contain a string of characters. It is possible to embed variables and expressions within a string to make complex string construction easier (such as generating XML). To access individual characters within a string you can use square brackets along with an index or range. There are a number of control characters that can be used within a string to provide various formatting options

such as a tab or new line character. These control characters are described in the list below:

- `\n` – Adds a new line to the string – Non visible character
- `\r` – On windows this character provides a line feed.
- `\t` – Adds a tab character to the string – Non visible character
- `\"` – Adds a double quote character to the string.
- `\x??` – Character with the hexadecimal value e.g. \x20 would provide the space character. The question marks are where the hexadecimal digits go.
- `\???` – Character with the octal value e.g. \040 would provide the space character. The question marks are where the octal digits go.
- `\b????????` – Output e.g. \b00100000 would provide the space character. The question marks are where the binary digits go.
- `\a` – Audible bell – Cause the computer to beep.
- `\f` – Form feed – Tell the output to create a new page.

You can reference variables or even place small expressions within a strings that get evaluated at runtime. This allows for the complicated construction of strings to be less painful and easier to understand. To reference a variable you simply prefix its name with a dollar symbol '$'. When the value of the referenced variable is placed in the string, it is important to note that the string representation for the variable will be used. For objects the function `toString()` will be called and the return value used. To place an expression within a string you can use a dollar symbol followed by a set of curly brackets '{ }' with the expression between them. The following code will print out "Hello World" and then print out 2. It is important to note that expressions within a string have effect outside. For instance using `++` on a variable within the braces would cause the variable to remain incremented after the string is constructed.

```
string test = "Hello";
Console.println( "$test World" );
Console.println( "${(1 + 1)}" );
```

Strings can also be defined using single quotes ('). These strings differ from the double quote notation because everything between the single quotes is use verbatim. As a result control characters are ignored as well as embedded expressions. There is only one control code that is obeyed and that is `\'` which will insert a single quote within the string.

2.4.4 array

An array allows the sequential or random storage of data that can be retrieved at any point. They can grow as you need them and are able to store any type of data within

them; they are not limited to a specific type meaning that numbers, string, objects and even arrays can be stored side by side. To access the contents of an array you need to use a set of square brackets with a reference between them - this is covered by the Index Operator.

When declared an array variable contains no elements. To add a value it is possible to use one of the provided Array functions, use an set of empty square brackets or use the in-line array notation. The second option allows you to pop a value onto the end of the array very easily and is demonstrated below. The third allows you to create arrays using a very simple syntax: a comma separated list of values (they can be expressions), surrounded in a set of square brackets.

```
array a = [ 1, 2, 3 ]; // Declare an array 'a' and initialize it
to have 3 elements
a[] = 4; // Add the value '4' to the end of the array
a[] = 5; // Add the value '5' to the end of the array
```

The above code shows how to use an array in a linear fashion. To initialize and access arrays using names for look-up and retrieval you can use code similar to the following example.

```
array a = [ 'FirstValue' => 1, 'SecondValue' => 2 ]; // Declare
'a' with two elements
                    // 'FirstValue' maps to the value 1
Console.println( a['FirstValue'] ); // print out '1' to the console
a['ThirdValue'] = 3; // Set 'ThirdValue' to map to the value 3
```

When you create the array using the above notations, it is important to note that `ferite` will copy the value and add it to the array. It will not reference the variable you pass to it. The following correct code should demostrate this fact.

```
void v;
array a = [ v ];
a[0] = 10;
v = "ten";
```

It is possible to mix the different types of values you can store in an array, however, once a value has been set for a spcific index, it must remain that type. For instance the following is *invalid* code because the value at 0 is already a number.

```
array a = [ 2 ];
a[0] = "two";
```

2.4.5 object

Variables of type object either point to an object or they point to **null**. Null allows you to see if an object variable points to something. All object variables, when declared, are

initialized to null. To check if an object variable points to null; it is only necessary to check for equality to null. Pointing to an object is done in two ways: the new operator, which creates a new instance of a class, or by assigning it the value that another object points to. Although this is covered later, it is important to note that two different object **variables** can point to the same object and that an object variable can point to any type of allocated object.

```
object o = null;
object o2 = new SomeObject();
o2 = new SomeOtherObjectType();
```

2.4.6 void

The void type is `ferite`'s semi-polymorphic type; only semi-polymorphic because once it has morphed into a specific type it remains that type. What this means in simpler terms is that the void variable can be seen as a place holder type. Assigning a value to a variable of type void will cause it to change to that type; for instance, declaring a variable of type void and assigning a numerical value to it will cause it to become a number. This allows you to write dynamic code that does not depend on a specific type. It is important to note that a variable declared at the beginning of a function will remain a type for the duration that function is running and will be reset when it is next called. If it is an object's instance variable, it will remain that type for that object for the duration of the object's life time.

It is important to note that the only thing that can not be assigned to a variable of type void is a function. You can assign namespaces and classes to variables of type void and then use them as normal. The ability to assign namespaces and classes is very useful when writing abstraction layers as it allows you to use different concrete implementations and pass those references into functions.

```
void v = null;->v is now an object pointing to null
void v2 = 42; -> v2 is now a number with the value 42
void v3 = SomeNamespace; -> v3 points to SomeNamespace
void v4 = SomeClass; -> v4 points to SomeClass
```

Question Is there any performance penalty for using void? In other words, could a lazy programmer just declare all variables as void and let the type system make all of the decisions, and not pay a price other than maintainability?

Answer There is no performance penality in using void and a lazy programmer can declare **void** if wanted. It is strongly recommended that this practice not be followed because you drop a core feature (declared types) of ferite which is there to make maintenance and debugging less of a headache.

2.4.7 Type Hinting

The type system in `ferite` works great for most code bases, however there are times when less flexibility and more constraints are very useful. In preparation of a future releases, `ferite` currently supports the syntax for type hinting which will become actively used to provide the afore mentioned constraints.

```
object<XML.Element> e = null;
string<UTF8> s = '';
```

The hinting is useful to show intent and the above example shows how hints can be used. A type hint uses the form of type<hint>, with the hint being contained within the less-than and greater-than characters. The intent of the hint on the object variable is to state that object variable will only point to objects of type XML.Element. When enforced the type system would only allow objects of that type to be assigned to that variable. The notion of type hinting becomes much more interesting when the string example is considered. The aim of type hinting to allow customised type constraints written in ferite. The string example could, for instance, guarantee that its contents are always UTF8 encoded, with either exceptions or automatic transcoding of character data upon assignment.

Even though hints are currently not enforced, their use is not wasted as they help provide tools to make the code self documenting and are highly encouraged.

2.5 Variables

Variables are containers for values; the sort of value a variable stores is dependent on its type. Variables come in a number of similar but distinct varieties: local variable within a function, a parameter into a function, class, object and namespace variables. Before it is possible to store, or pass information around, it is necessary to declare your variables.

```
modifiers type name [= expression] [, name [= expression], ...]
```

- modifiers

  There are a number of modifiers that can be applied to variables. This affects how they are treated within `ferite` and allow you, the programmer, to specify what restrictions you wish to put on them. Depending on the context of the variable declaration you can use zero of more of the keywords:

  - final - the value of the variable, once set, can not be changed.
  - atomic - all accesses on the variable (getting the value and setting the value) are atomic, i.e. thread safe, which means that it is unnecessary to create an explicit lock for it. It is important to note that this does have an execution time, and space, overhead to the variable and should therefore be used wisely.

There are a few more that will be discussed in the section about classes and namespaces (they are **public**, **private**, **protected**, **abstract** and **static**).

- type

  This is the type of variable that you wish to declare. It can be void, number, string, array or object.

- name

  The name of the variable to be declared. The name must start with an alpha character (a-z, A-Z) or underscore (_) and after that may contain underscores, digits ([0-9]) and other alpha characters.

- [= expression]

  Variables can be initialized to a custom default value. If the expression is omitted the default value will be '0' for numbers, an empty string for strings, an empty array for arrays and null for objects.

  Please Note!

  When a variable is declared within a function you can specify any valid expression to be used as the variable's initializer (assuming the types are correct). E.g. a return from a function, the addition of two previously declared variables.

  However, when the variable is declared within a global, class or namespace block, it is only possible to use a constant e.g. initialize a number with an integer or real number (e.g. 120 or 1.20 respectively), a string with a double (without expressions) or single quoted string. It is **not** possible to initialize an array or object in a global, class, or namespace block; these will have to be done using some form of initializer function.

- [, ...]

  Rather than having to declare modifiers and type again for a set of variables it is possible to simply add more names and initializers in a comma separated list. The modifiers apply to all of the variables declared within the list which should reduce ambiguity when debugging.

```
number mynumber = 10, another_number;
final string str = "Hello World";
object newObj = null;
array myarray = [ 1, 2, 3 ];
```

A variable's scope is as local as the function in which it is declared. The exception being global variables which can be accessed from any function. Global variables can be accessed anywhere within a script and are declared using the following syntax:

```
global {
  ...variable declarations...
}
```

Unless explicitly defined a variable is considered local. There are a number of predefined global variables within a `ferite` script, these are **argv**, **null** and **err**. **argv** is an array of strings containing the parameters passed into the top level script that first gets executed. **null** is used to allow checking of object variables. **err** is the error object used for exception handling.

2.6 Expressions

Almost everything written in `ferite` is an expression as they are the building blocks of a program. They are combined to build other expressions which are in turned used in others using operators. Expressions are built up using various operators, variables and other expressions, e.g. adding of numbers, or creating an instance of a new object. Expressions are made clearer when discussing operators as these are what are used to build them.

2.6.1 Truth Values

The crux of program flow are true and false. It is important to know what constitutes a truth value.

- A number that is not zero is considered as true, this also means that negative values are also true. It has to be noted that if a number has switched into real format it is never likely to be considered false. Currently `ferite` deals with this by binding false to the range plus/minus 0.00001.

  ```
  0 -> false
  1 -> true
  0.0 -> false
  0.2 -> true
  ```

- A string that has zero characters is considered false, otherwise it is true.

  ```
  "" -> false
  "Any Value" -> true
  ```

- An array with no elements is false, otherwise is considered true.

  ```
  [] -> false
  [ 1, 2, 3 ] -> true
  ```

- An object is considered to be false if it does not reference any instantiated object.

```
null -> false
(new Object()) -> true
```

- A void variable can not be true and therefore will always be false. For a variable of type void to become true, it must be instantiated to a truth value of another type. This side-effect of void variables is useful because it allows you to, knowing that it is void, use a simple if statement to see whether the variable has been initialized or not.

```
void v -> false
```

There are currently two keywords that can be used 'true' and 'false' these are of type number. For passing explicit truth values around these are the considered the correct way.

```
number shouldKeepDoingThings = true;
```

2.7 Operators

An operator applies a operation to one or more values. `ferite` provides a set of operators that allow you to do basic arithmetic, assignment, comparison and a whole load more operations. This section is broken down into a set of groups; each containing like-wise operators. Before we delve into what operators exist, it is important to consider some terminology. A unary operator is one that operates on only one value and usually prefixes the value that it is applied to. A binary operator operates on two values; these two values are called the `left hand side` and `right hand side`.

2.7.1 Arithmetic Operators

- The addition operator '+' allows you to add two variables together. Addition requires that the left hand side of the operator should be either a **number** or a **string**. If the left hand side is a number, the right hand side can only be a number; otherwise an exception will be thrown. If the left hand side is a string, the right hand side can be of any type: `ferite` will inspect the right hand side and produce a string representation of that value. If the value is an object, `ferite` will invoke the `toString()` method and use its return value as the right hand side.

```
1 + 2 -> 3
"How many times?  " + 10 -> "How many times?  10"
```

- The subtraction operator '-' is mainly used with numbers. Although if the left hand side is a string, the right hand side must also be a string, every occurrence of the string on the right hand side, in the left hand side will be removed.

```
2 - 1 -> 1
"How many times?  " - "How " -> "many times?  "
"A B C B D" - "B " -> "A C D"
```

- Multiplication '*' only applies to number types. The result is the left hand side multiplied by the right hand side.

```
2 * 2 -> 4
```

- Division '/' also only applies to number types. The result is the left hand side divided by the right hand side. If both sides of the operation are integer based numbers, the division will be integer otherwise it will be floating point division.

```
2 / 2 -> 1
2 / 0.5 -> 4
```

- Modulus '%' - Returns the remainder of integer division between two number variables. If the numbers are in real format they will be implicitly cast into integers and then the operation will be done.

```
2 % 2 -> 0
11 % 10 -> 1
```

2.7.2 Bitwise Operators

Please note that a if a real number is passed to a bitwise operator, it will be implicitly cast to a integer number and then the operation applied. The examples within this section show the resulting number in binary format, however, in `ferite`, printing the value out to the console will display the result in decimal format.

`10&11.1 -> **10&11**`

- Bitwise AND '&' - does a bitwise AND on the two values passed to it. The result of the operation is the left hand side value bitwise 'and'ed with the right hand side. In basic terms, this means that for every bit in the right hand side that has a value of 1, the corresponding bit in the left hand side will be noted, otherwise 0 will be used.

```
0b1010 & 0b1000 -> 0b1000
0b1111 & 0b0000 -> 0b0000
```

- Bitwise OR '|' - does a bitwise OR on the two values passed to it. The result of the operation is the left hand side bitwise 'or'ed with the right hand side. In basic terms, this means that for corresponding bits on each side, if either bit is 1, 1 is used otherwise 0.

```
0b1010 | 0b1000 -> 0b1010
0b1111 | 0b0000 -> 0b1111
```

- Bitwise XOR '^' - does a bitwise XOR on the two values passed to it. The result of the operation is the left hand value bitwise 'xor'ed with the right hand side. In basic terms, this means that for corresponding bits on each side, if either bit is 1, 1 is used otherwise if both bits equal 0 or 1, then 0 is used.

```
0b1010 ^ 0b1000 -> 0b0010
0b1111 ^ 0b0000 -> 0b1111
```

- Left Shift '<<' - does a bitwise left shift on the left hand value by the right hand value number of bits. For each bit shift, it is equivalent to multiplying the left hand value by 2. All new bits resulting in the shift will be set to 0.

```
0b0010 << 2 -> 0b1000
0b1111 << 2 -> 0b1100
```

- Right Shift '>>' - does a bitwise right shift on the left hand value by the right hand value number of bits. For each bit shift, it is equivalent to dividing the left hand value by 2. All new bits resulting in the shift will be set to 0.

```
0b0010 >> 2 -> 0b0000
0b1111 >> 2 -> 0b0011
```

2.7.3 Incremental and Decremental Operators

Incremental operators allow in-line incrementing and decrementing of numerical values and should be applied to variables of type number. There are two flavors of both incrementing and decrementing: pre and post operators. When using the prefix operator, the variable's value is incremented and the expression evaluates to the new value. The postfix operator differs slightly because it will evaluate to the variable's current value and increment it afterwards.

- Prefix Increment '++'

```
number var = 10; -> variable 'var' of value 10
10 + (++var) -> 21, var is incremented to 11 and the value is
added to 10
```

- Postfix Increment '++'

```
number var = 10; -> variable 'var' of value 10
10 + (var++) -> 20, var is added to 10, resulting in 20, but
var now has the value 11
```

- Prefix Decrement '–'

```
number var = 10; -> variable 'var' of value 10
10 + (--var) -> 19, var is decremented to 9 and the value is
added to 10
```

- Postfix Decrement '–'

```
number var = 10; -> variable 'var' of value 10
10 + (var--) -> 20, var is added to 10, resulting in 20, but
var now has the value 9
```

The examples hopefully demonstrate the subtle differences of the operators and their variants. These operators are often used within looping.

2.7.4 Assignment Operators

The basic assignment operator is '='. The operator will take the value of the expression on the right hand side and place that value in the variable on the left hand side.

```
number a = 10; -> the variable a contains the numerical value 10
string b = "Hello World"; -> the variable b contains the string
value "Hello World"
```

It is useful to note that an obvject variable can point to any object. This means that an object variable could point to an object of type Foo, and then point to an object of type Bar.

```
class Foo { number idx; }
class Bar { number idx; }

object f = new Foo();
f = new Bar();
```

There are a number of short hands using assignment: they couple one operator with assignment. For instance, in the example below, to add-assign you could use the first expression, however you could use the second shorter and less clunky version.

```
number a = 10; -> a is initialized to 10
a = a + 100; -> a now contains 110
a += 100; -> a now contains 210
```

You can use the short hand of an operator with any of the arithmetic or bitwise operators, just prefix the assignment operator with the operator you wish to use. For instance:

```
b -= "many" -> remove "many" from the string b
c *= 10 -> multiple c by 10
a += 12; -> 'a' now is equal to a+12
b += " From Ferite"; -> 'b' now is equal to b+" From Ferite"
```

A variable of type void can be morphed into a specific type through assignment. In the example below the variable c is morphed into a number, and variable d morphed into a string. These variables can not be changed into a different type once morphed.

```
void c = 42; -> c has been morphed into a number
void d = "Now A String"; -> d has been morphed into a string
```

2.7.5 Comparison Operators

Comparison operators allow the differences between two values to be checked. `ferite` provides a standard set of comparison operators that should be found in all programming languages. It is important to note that `ferite` will not throw an exception if different types are used (such as a string and a number) within a comparison, instead the result will be set to false. Comparison operators are used for conditional statements and loops (discussed later).

- Equal To '==' - true if both sides are equal, false otherwise.

```
10 == 11 -> false, 10 does not equal 11
4 == (2 + 2) -> true, 4 does equal 2 + 2
```

- Not Equal To '!=' - true if both sides are not equal, false otherwise

```
10 != 11 -> true, 10 does not equal 11
4 != (2 + 2) -> false, 4 does equal 2 + 2
```

- Less Than '<' - true if the left hand side is less than the right, false otherwise.

```
10 < 11 -> true, 10 is less then 11
4 < (2 + 2) -> false, 4 is equal to 2 + 2
```

- Less Than Or Equal To '<=' - true if the left is less than or equal to the right hand side, false otherwise.

```
10 <= 11 -> true, 10 is less then 11
4 <= (2 + 2) -> true, 4 is equal to 2 + 2
```

- Greater Than '>' - true if the left hand side is greater than the right, false otherwise.

```
10 > 11 -> false, 10 is less then 11
4 > (2 + 2) -> false, 4 is equal to 2 + 2
```

- Greater Than Or Equal To '>=' - true if the left is greater than or equal to the right hand side, false otherwise.

```
10 >= 11 -> false, 10 is less then 11
4 >= (2 + 2) -> true, 4 is equal to 2 + 2
```

- isa 'isa' - true if the left hand side expression is of type stated on the right hand side.

```
"Hello World" isa string -> true
42 isa string -> false, 42 is a number
```

- instanceOf 'instanceof' - true if the left hand side expression is an instance of the class stated on the right hand side.

```
Console.stdin instanceof Sys.StdioStream -> true
Console.stdin instanceof Test -> false
```

2.7.6 Logical Operators

These operators are useful for connecting comparison operators to build bigger expressions.

- Not '!' - true if the expression it is applied to is false.

```
!(4 > (2+2)) -> true, (4 > (2+2)) evaluates to false, the !
operator flips this to true.
!(10 > 4) -> false, (10 > 4) evaluates to true, the !  operator
flips this to false.
```

  You can also use the keyword **not** to represent the operator. This is present in `ferite` because sometimes the '!' can get lost in the expression. 'not' is the same as '!'.

```
not (4 > (2+2)) -> true
not (10 > 4) -> false
```

- And '&&' - true if both variables/expressions are true.

```
true && (4 > (2+2)) -> false, (4 > (2+2)) is false, therefore
true && false -> false
true && (10 > 4) -> true, (10 > 4) is true, therefore true &&
true -> true
```

  It is also possible to use the keyword `and` to represent this operator.

```
true and (4 > (2+2)) -> false
true and (10 > 4) -> true
```

- Or '||' - true if either variable/expression is true.

```
true || (4 > (2+2)) -> true, (4 > (2+2)) is false, therefore
true || false -> true
true || (10 > 4) -> true, (10 > 4) is true, therefore true ||
true -> true
```

  It is also possible to use the keyword `or` to represent this operator.

```
true or (4 > (2+2)) -> true
true or (10 > 4) -> true
```

It is important to note that to make sure the operators are evaluated in the expected order it is often necessary to use brackets. The example below demonstrates how an extra set of brackets changes the order of evaluation.

```
not (4 > 2) and (4 > 4) -> false
not ((4 > 2) and (4 > 4)) -> true
```

2.7.7 Index Operator

The index operator provides a mechanism for pulling information out of an array or a string. There are a number of variations of the operator that are explained below.

- '[]' - This works on only the array type. When used in this form, the operator adds a new void variable onto the end of the array and then provides it ready for use. The main aim of this variant of the index operator is to provide an easy way of placing new values onto the end of an array.

  ```
  array a;
  a[] = 1;
  a[] = 2;
  a[] = 3; -> a being an array with three elements:  [ 1, 2, 3
  ]
  ```

  For the record the above example can be re-written as shown in the example below. This does not make the above example invalid, we just ask you to imagine lots of code between each line.

  ```
  array a = [ 1, 2, 3 ]
  ```

- [*expression*] - this variant is the main way you can pull either a value or character out of an array or string respectively. When used with an array, if the expression evaluates to a number, the operator will evaluate the value at that index. If the expression is of any other type, the value will be used to create a hash index to get the array value. If the operator is applied to a string, only a numerical expression can be used; the operator will evaluate to the character at that index.

  ```
  array a = [ 1, 2, 'Hello World' => 3 ];
  a[0] -> the first value within the array, in this example '1'
  a["Hello World"] -> the value pointed to by "Hello World", in
  this example '3'
  a[2] -> the third value (created using the 4th line of the example)
  ```

  It is important to note that counting for an index starts at 0 (for historical reasons), therefore a value in position 10 will have an index 9.

- [*lower bound expression..upper bound expression*] (also referred to as the slice operator) - This is a range expression. With strings and arrays it allows you to take

a slice of the variable; in the case of an array, the operator will evaluate to a new array containing the values described within the range, in the case of a string, a new string with the described range will be evaluated to.

The range can be ascending - in which case the order in the variable is preserved, or descending in which case, the slice is made with the contents being reversed. It is possible to leave out the upper or lower bound expression dictating that the operator should go to the end or from the beginning respectively. If a negative number is given, `ferite` interprets the expression to imply an offset from the end of the variable's range.

```
string s = "Hello";
string t = s[-1..0]; -> a slice of the entire string and reverse
it
string u = s[..2]; -> a slice of the first 3 characters in the
string s
array a = [ 1, 2, 3 ];
array b = a[1..]; -> a slice of a containing [ 2, 3 ]
```

2.7.8 Complex Operators

These are not complicated operators. They are actually very friendly and nice operators. In-fact they are quite simple. Just slightly more complicated than the previous operators.

I wish to apologize. I am listening to a goon show and couldn't come up with a better introduction and I felt it was necessary to write something rather than leave it blank. Besides, it is not often you get something silly about something nice and important ;)

- Instantiate an object 'new' - this operator takes a class name and a set of values and will create a new object based upon that class. It will call the constructor of that class and then return the object.

  ```
  new SomeClass( 10 ) -> a new object from SomeClass
  ```

  It should be noted that multiple *object* variables can point to the same object created using the new keyword. This is discussed later on within **Objects and Classes section** .

  ```
  object newObject = new SomeClass( "aString", 10 );
  object anoObject = newObject; -> both variables point to the
  same object
  ```

- Evaluate a string 'eval' - The eval operator allows you to on the fly compile and execute a script **and** get a return value. They say a picture is worth a thousand

words, so here is an example of eval. If the string supplied to the operator is invalid, an exception will be thrown.

```
eval( "Console.println(\"Hello World\");" );
```

The above example is the same as running the following script:

```
Console.println("Hello World");
```

To return a value, you just use the return keyword (mentioned within the function documentation in the next section). The code below will return '42', which will in turn be assigned to the variable **value**.

```
number value = eval( "return 42;" );
```

This is of course a very simple example and does not show what a useful operator it is, but it does allow you to, at runtime, modify the behavior of code. It should also be noted that there are potential security risks involved with this operator and it should be considered carefully; it is advised that you never eval a string that is from an unknown source.

Later on in this manual, the operator **include** is discussed. This operator is similar to eval except it will evaluate a file.

2.8 Statements and Blocks

Statements are basically a collection of expressions followed by a ';'.

```
x = 1 + 2;
x++;
```

A block is a set of statements sat between two braces '{}'. It is possible to declare new variables but they must be declared at the beginning of a block.

```
{
  number x = 10;
  x = x + 2;
  x++;
}
```

Sometimes it is useful to isolate variables to a specific block in a large chunk of code. To make writing code easier, `ferite` allows you to nest blocks allowing you to, if required, reduce the number of variable declarations at the beginning of functions and/or re-use variable names. The example below shows this in action.

```
{
  number x = 10;
  {
    string str = "Some Value";
```

```
  }
}
```

Even though it is possible to re-use variable names within blocks it is not recommended as it can introduce ambiguity. However, `ferite` will correctly use the most locally declared variable.

```
{
  number x = 10; // 1st x
  x = 20; // 1st x
  {
    number x = 20; // 2nd x
    x = 30; // 2nd x
  }
  x = 40; // 1st x
}
```

2.9 Control Structures

`ferite` contains methods for changing the program flow, these are formally called control structures. The control structures can be nested as deeply as you need, however, we strongly urge you to not nest them too deeply as it can cause confusion and ambiguity.

2.9.1 If, Then and Else

This allows for the conditional execution of `ferite` scripts based upon the result of a test in the form of an expression. There are two types you can employ: one that will execute code if test is true, the other will do the same as the first but also execute a different bit of code if the test is false.

```
if ( expression ) {
  statements if the expression is true
}
```

```
if ( expression ) {
  statements if the expression is true
} else {
  statements if the expression is false
}
```

It is not necessary to place braces around the statement block if it is only one statement.

```
if ( a < 10 )
  a = a + 2;
```

The truth value of the expression is determined when the control structure is executed based upon the definition of the truth values in the **truth values** .

```
if ( a < b )
  Console.println( "A is less than B" );

if ( b > c ) {
  Console.println( "B is greater than C" );
  Console.println( "This could be fun." );
} else {
  Console.println( "It's all good." );
}
```

There is the age old problem of the *dangling else* problem. With `ferite` the else binds to the closest if statement within the same block. For instance:

```
if (foo) if (bar) Console.println("one"); else Console.println("two");
```

Is equivalent to:

```
if (foo) {
  if (bar) {
    Console.println("one");
  } else {
    Console.println("two");
  }
}
```

2.9.2 Looping

You can not do very much without the ability to loop over data. Each time the block of a loop is executed it is called an iteration. There are several forms of iterating over data in `ferite`: **while**, **for** and **do..while** loop.

while Loop

The first type of loop `ferite` supports is the while loop; this allows you to keep looping until the expression, used as the test, evaluates to false. Each time the loop executes, the test is evaluated and if it is true, the body of the while will run.

while (*expression*) {
  statements if the expression is true
}

while (*expression*)
  single statement if the expression is true

The following code will keep looping until n is equal to 10. On each loop the code will print out the current value of n from 0 to 9 on a new line and then increment the value of n by 1.

```
number n = 0;
while ( n < 10 ) {
  Console.println( "$n" );
  n++;
}
```

for Loop

All of `ferite`'s looping constructs can be modeled using the while loop, however, it is often useful to use the for loop.

for (*initiator*; *test*; *post-block*) {
  statements if the expression is true
}

for (*initiator*; *test*; *post-block*)
  single statement if the expression is true

The initiator expression is executed unconditionally at the beginning of the loop. This is useful, as an example, for setting the initial value of a loop variable. The test is used in the same fashion as a while loop; if it evaluates to true the looping continues, if it evaluates to false the loop will terminate. The main difference to the while loop is the ability to execute a statement at the end of each iteration - often used to increment a variable.

If the test is empty it will automatically evaluate to true, causing the loop to continue until **break** is used.

```
number i = 0;
for ( i = 0; i < 10; i++ )
  Console.println( "variable i currently equals " + i ); // print
out the value of i
```

It is possible to declare a loop variable within the first section of a for loop declaration. This allows you to use loop only variables, knowing that there will be no impact on the surrounding code. This is commonly done when using for loops. The above example, re-written to take advantage of the feature looks like this:

```
for ( number i = 0; i < 10; i++ )
  Console.println( "variable i currently equals " + i ); // print
```

```
out the value of i
```

do .. while Loop

The *do .. while* loop is a variation of the *while* loop, the one difference being that it guarantees at least one execution of its body. It will only then complete looping until the expression evaluates to false.

do {
  statements if the expression is true
} **while** (*expression*)

do
  single statement if the expression is true
while (*expression*)

2.9.3 Manipulating Loops

Break

break will end the current *for, while, do .. while, or switch* loop it is executed in. It allows you to easily escape the current loop if you wish to.

Continue

continue will cause execution flow to jump to the end of the block of the current*for, while, do .. while, or switch* loop it is executed in. continure is useful to move onto the next iteration before the current one has finished.

2.9.4 Exception Handling

This control structure provides the exception handling within `ferite`. There are three main components to this structure: the code to execute that may throw an exception, the code to execute if an exception is thrown and an optional block to execute if an exception is **not** thrown.

monitor {
  statements
} **handle** {
  statements to clean up in case of an exception
} **else** {
  statements if no exception has occurred
}

It is possible to nest **monitor-handle-else** blocks. When an exception does occur a global variable called err is instantiated and given information relating to the thrown

exception. The object has two attributes, a string 'str' and number 'num' - these provide information on the error that occurred. Exceptions are propagated up through the system until a handler is found; if no monitor-handle-else block catches the exception, `ferite` will cease execution of the script.

2.9.5 Switch

This allows you to write blocks of code that are only executed when an expression evaluates to a certain value. This is roughly equivalent to doing a number of successive **if** blocks, but it is cleaner, tidier and easier to understand.

```
switch ( expression ) {
  case expression:
    ...  code ...
    ... more case blocks ...
  default:
    ...  code ...
}
```

When the switch statement is executed, the expression at the top is evaluated and its value stored for comparison. A **case** expression is evaluated to see if it equals the value previously stored for comparison. If it does, the cases block is executed until the end of the switch statement (including other case blocks). To restrict the flow of execution to that of the matching case block, the **break** is used. 'break' will cause the execution to jump to the end of the switch statement. If you use **continue**, the first expression will be re-evaluated, effectively causing the switch statement to start again.

It is possible to define a catch-all case block using the **default** identifier. If no case blocks match the switch expression, this case block will be executed.

```
switch( input ) {
  case 0:
   Console.println( "case 0" );
  case 1:
   Console.println( "case 1" );
   break;
  case 2:
   input++;
   Console.println( "case 2" );
   continue;
  case "3":
   Console.println( "case 3" );
   break;
  default:
```

```
    Console.println( "default" );
}
```

When the variable input is equal to 0, the following will be printed out to the console. The first two case blocks are executed because there is no **break** at the end of the first case block.

```
case 0
case 1
```

When input = 1, the following will be output-ed to the console. The break at the end of the second case block causes execution of the switch statement to finish.

```
case 1
```

When input = 2, the following will be output-ed to the console. The third case block increments the value of input and causes the execution of the switch statement to be restarted. The value of input increases to the numerical value of 3; as there is no case block that can catch that value the **default** case block is executed. At first it may seem that the fourth case block may be used, but its expression is the string representation of 3.

```
case 2
default
```

When input = "3", the following will be output-ed to the console. Once again, the break causes the switch statement to finish.

```
case 3
```

When input is anything else:

```
default
```

It is very important to note that you can use any valid expression as the case expression. It can even have different types than the first switch expression (there won't be any exceptions thrown). This makes switch a very powerful construct. You do not need to supply a **default** block if you do not wish to have one.

2.10 Functions

Blocks of statements have been discussed before and functions are just a block of code with a name attached to it. Using this name you can re-use the block of code, creating a library of functions to solve common tasks.

A function consists of a name, a parameter list declaration and a block of code - but before delving into the creating of functions, it is important to look at how to call a function.

```
x = add_numbers( 1, 2 );
```

To call a function, you use its name and pass it a set of values. The above example shows a function call to a function named **add_numbers**; it is passed two values: the number **1** and the number **2**. Any value can be passed to a function and is often the result of evaluating an expression; `ferite` will throw an exception if the types mismatch those of the function being called. All functions may return a value and, in the above example, it is assigned to some variable x. You can ignore the return value from a function call if you so wish, ferite will automatically handle the return and discard it as needed.

As mentioned before, a function consists of a name, a parameter list and a block of code. It looks like the example below.

function *function_name*(*parameter declarations*){
  block of code
}

- *function_name* - This is the name of the function and must obey the same rules for naming as the rules for naming **variables** . It is recommended that the name should reflect the purpose of the function, e.g. open, close, add; rather than how the function works.
- *parameter declarations* - This is a comma separated list of variables. For each variable it is only necessary to supply the type of a variable and its name. Sometimes it is necessary to pass a variable by reference; `ferite` allows this to happen by allowing the name of the variable to be prefixed with a '&'. Passing variables by reference is covered later in **passing variables by reference** .
- *block of code* - To what a block consists of you should see the section **statements** .

```
/*
 This function will add the string "foo" onto the end of the string
it has been
 given and then return it.
*/
function foo( string bar ) {
  bar += "foo";
  return bar;
}
```

It is important to note that `ferite` falls into the category of call-by-value languages. This means that when a function is called, `ferite` takes each value passed to the function, copies it and assigns it to the variable for that parameter. This means that variables passed into a function call will not be modified by that function call and will remain the same afterwards. This is the same for all types, including arrays, strings and objects. There is a slight oddity with objects: whereas with strings and arrays, you get a complete

copy of the original value, with an object variable you get a copy of the reference to the object it points to. Therefore making a function call, or changing a variable, on an object reference (even though it is copied) will change that object and that change will be seen after the function call; this is because the reference to the object and **not** the object itself that has been copied.

The following example demonstrates this effect with arrays and objects. (It is assumed that StringObject used within the example holds a string that can be set using **setString()** and obtained by calling **stringValue()** and using the return value. For more information about classes and objects please read the section **Classes and Objects**

```
function test( array a, object b ) {
  a[] = "Add Item On The End"; // Add an item to the end of the array
  b.setString( "Set A String In Test" );
}

array a = [ 1, 2, 3, 4 ];
object b = new StringObject( "Hello World" );
Console.println( "Before call to test:  ${a}, ${b.stringValue()}"
);
test( a );
Console.println( "After call to test:  ${a}, ${b.stringValue()}"
);
```

The output of the above program is shown below. Note how the array has remained the same but the object has been changed.

```
Before call to test:  [ 1, 2, 3, 4 ], Hello World
After call to test:  [ 1, 2, 3, 4 ], Set A String In Test
```

Functions provide an easy way of grouping statements together to perform a task and are the corner stone of any nutritious `ferite` script.

2.10.1 Variable Argument Functions

Sometimes it is hard to know how many values you wish to pass into a function; to aid you, `ferite` provides an easy mechanism to write functions that depend on an unknown set of values. When executed it is possible for the function to get a list of the arguments by invoking the **arguments()** strong. This returns an array containing the values passed into the function, with the last value being the name of the function being called.

Although **arguments()** is often used in a variable argument setting, it is possible to use it within any function.

Declaring a function with variable arguments is the same as declaring a normal function,

except the last item in a parameter list is **...**. This tells `ferite` that the function can accept more values including and beyond that point within the parameter list. The following program listing shows how to declare and use variable argument functions.

```
uses "array", "console";

function test( string fmt, ... ){
 number i = 0;
 array fncArgs = arguments();

 Console.println( "test() called with ${ Array.size(fncArgs) } args"
);
 Console.println( fmt );

 for( i = 0; i < Array.size(fncArgs); i++ ){
  Console.println( "Arg[$i]:  ${ fncArgs[i] }" );
 }
}

test( "nice" );
test( "nice", "two", "pretty" );
```

This is an important concept as it is used elsewhere in `ferite` to provide a clean method of dynamic programming when used with **dynamic programming with objects** .

2.10.2 Returning A Value

If there is not an explicit return statement then the function will return nothing in the form of a void variable. To return a value from a function, it is as simple as using the **return** keyword.**return** takes an expression and returns the result of its evaluation to the caller of the function. Just as values are passed in by copying them to the parameter variables, the return value from a function is also copied to the function calling it.

```
return someValue * 10;
return 0;
return "Hello World";
```

The default return from a function is a void variable. The two functions below will return exactly the same value to the caller. `ferite` has a default return value to remove the necessity of requiring the programmer to always supply one.

```
function foo() {
 void x;
 return x;
```

```
}
function bar() {

}
```

Annotating Return Type

Future versions of `ferite` will provide improved static type checking. With this shift, `ferite` supports the ability to provide return type annotations on functions. This is an enforced requirement on native interface functions and will remain optional on ferite functions. Annotations are inserted after the parameter declaration and before the function contents:

```
function version :  number {
  return 1.0;
}
```

The annotation should be the type that the function returns. There are a couple of additions to make the annotation more concise:

- If a function does not return a value, the keyword `undefined` should be used.
- If a function returns multiple different types the keyword `void` should be used.
- If a function returns an object you can not only use the `object` keyword, but also specify the class.

```
function version :  XML.Element {
  return new XML.Element();
}
```

Annotations are fast becoming a powerful ally in isolating, at compile time, potential type clashes within the code that might not surface until significantly later in seldom run code. Although the full benefits of annotations have yet to arrive, even in their current state they provide clarity and improved understanding of function outputs which is invaluable in all bug free code bases.

2.10.3 Function Overloading

There are times when you wish to have the same operation applied to different data types, for example, a print method where you wish to handle various different types and/or number of arguments. `ferite` provides a function overloading mechanism which allows you to write a set of functions all with the same name but with different parameters. When the program is run `ferite` will automatically choose the best function for the job based upon the types being passed into the function.

```
uses "console";
```

```
function print( number n ){
  Console.println( "Number:  $n" );
}

function print( string s ){
  Console.println( "String:  $s" );
}

print( 10 );
print( "Hello World" );
```

The above code declares two functions with the name `print`. If the script is run the following output would occur.

```
Number:  10
String:  Hello World
```

2.10.4 One Line Functions

Some functions return the result of a single expression, or call a different function. Using braces is somewhat over kill and can cause clutter within the code. `ferite` allows you to use one line functions; these are best explained through the use of an example.

```
uses "console";

function printNumber( number n )
  Console.println( "Number:  $n" );

function printString( string s )
  Console.println( "String:  $s" );

printNumber( 10 );
printString( "Hello World" );
```

2.10.5 Zero Parameter Functions

Along the same lines as one line functions, `ferite` allows you to skip the brackets on function declaration when the function does not take any arguments. It is important to note that you can only skip the brackets on declaration, you will still have to provide them when calling the function.

```
uses "console";

function version
```

```
  return 1.0;

Console.println(version());
```

2.10.6 Pass By Reference

Earlier in this section on functions, it was mentioned that `ferite` passes parameters in by value. This is fine for small strings and arrays, but there are times when you do not want the overhead of copying a large array or large string from one function to another. Or you would like a function to return more than one value. To make life easier `ferite` allows you to pass a variable in by reference; this means that when you change the value of a parameter, the value of the variable in the calling function is modified - there is no copying.

This feature is enabled through the use of a '&' before the name of the parameter. The following example should help demonstrate what happens and how it differs from the normal behavior of `ferite`. In the example, the function test has two parameters: a number 'b' which is marked as pass-by-reference and a string str which will maintain normal behavior.

```
uses "console";

function test( number &b, string str ){
  b = 2;
  str += "From Foo";
}

number a = 1;
string str = "bar";

Console.println( "This should be 1:  $a, should be 'bar':  $str"
);
test( a, str );
Console.println( "This should be 2:  $a, should be 'bar':  $str"
);
```

The result of running the program, is the number 'a' passed into 'test' is passed by reference which means that rather than the value being passed the variable is passed into test. When test changes the value of the parameter to be '2', the variable 'a' is changed. Even though test changes the value of the string parameter, it is only changing the value of the copy. The output of this code running is shown below.

```
This should be 1:  1, should be 'bar':  bar
This should be 2:  2, should be 'bar':  bar
```

This can be a very useful tool for speeding up programs or returning multiple values; it is recommended that it is used sparingly because it can potentially cause confusion.

2.11 Classes and Objects

Object oriented programming (OOP) is a proven methodology for implementing large complicated systems. The core idea behind object orientation is the grouping of likewise data and methods into special containers called objects. Each object provides a specific purpose, for instance: you could have an 'Employee' object that stores data about an employee, such as name, age and salary and has a set of functions that do things with the data within the object, such as increasing the salary or incrementing the age of the employee. OOP allows you to think of a programs structure as you do the real world and therefore plays a critical role within `ferite`.

Objects within `ferite` have already been mentioned before, however, this section covers them in great depth: such as creating classes (the templates that objects are created from), instantiating objects and how to use `ferite` to implement a clean object base architecture. Before we can talk about objects, we must discuss classes. A class is a description of a structure; it describes how an object is put together: what information it can hold and what functions can be called on that data. Every single object within `ferite` is an instance of a class; the following code shows what a class looks like.

```
class ExampleClass {
  string stringValue;

  constructor( string str ){
    self.stringValue = str;
  }

  function printString(){
    Console.println( self.stringValue );
  }
}
```

The above code snippet declares a class called **ExampleClass**. The class declares the variable **stringValue** and two functions, a constructor **constructor** and normal function **printString**. When an object is created from a class definition it said to have been 'instantiated' from that class; we say that the object is an instance. It is possible, and often the case, that each class will have more than one instance within a program. When an object is instantiated, `ferite` takes the description provided by the class, creates the variables within that object and then calls the constructor. constructors are covered in **constructors section** .

To create an instance of a class the **new** keyword is used; it takes the name of the class, the values to be passed onto the class's constructor and will return an object. The code

below shows how to create a new object and then call the function **printString** on it.

```
object someObj = new ExampleClass( "Hello World" );
someObj.printString(); // will output Hello World
```

What good is an object if you can not do anything with it? To access the variables, or call a function, in an object requires two things to be known: what you want to access/call and which object to use. Using the above example, we call the function **printString()** on the object **someObj**; the trick is with the '.', it tells `ferite` to apply the function call on the right hand side to the object on the left hand side.

To reference variables and functions from within an object's function, it is necessary to prefix the variable with **self.** or simply a **.**. Just as in the above example, we use **someObj** on the outside of the object, on the inside - the class description - we use **self** to refer to the object we are in. This can be seen in use in the definition of the class **ExampleClass**; in the function **printString** we use the **self** variable to access the string contained in the object.

Lots of objects can cause a lot of headaches, especially when you have to make sure to clean them up when finished. `ferite` comes to the rescue with a garbage collector; a behind the scenes helper that deletes objects from memory when they are no longer in use. An object is considered in use when there is at least one variable of type object pointing to it. When the number of variables referencing an object drops to 0, the object will be deleted because it is no longer reachable from the program. `ferite`'s garbage collector is designed to deal with large and small programs and should have minimal performance impact on the execution of a program. There are no guarantees as to when an object will be deleted.

2.11.1 What is a constructor?

A constructor is a special function that is called, if it exists, when a object is instantiated from a class. It provides a mechanism for setting up the default values for an object when it is created. Like normal functions, a constructor may have a parameter list and obtains the value through the **new** keyword. An example constructor can be seen in the body of **ExampleClass**. This constructor assigns the value of the parameter to the string contained in the object. All constructors are called **constructor**. In a previous life, `ferite` required you to declare constructors with the function keyword. This is no longer the case, however ferite will continue to parse code that still provides it.

2.11.2 Inheritance

One of the best pieces of advice when developing a program is to re-use code. Inheritance provides a clean mechanism of re-using the functionality in your classes. As an example,

let's say that a program needs to deal with vehicle information. You want to store information about different vehicles and for simplicity we shall say that there are three types: motor bikes, cars and trains. There is a common set of details that can be stored about them: the size of engine, make, model, top speed, number of wheels. We have to create a new class, for each vehicle type, that deals with all the pieces of information; this can introduce bugs, inconsistencies and make life complicated. Inheritance to the rescue! To solve this problem, we create a class that contains all the information that can be shared. We then create a set of classes that inherit from our class; each new class contains the vehicle specific information. The example below shows how to implement the solution within `ferite`.

```
class Vehicle {
  string make;
  string model;
  number engineSize;
  number topSpeed;
  number wheelCount;
}

class MotorBike extends Vehicle {
  number handleBarType;
}

class Car extends Vehicle {
  number doorCount;
}

class Train extends Vehicle {
  number carriageCount;
}
```

To extend a class, the **extends** keyword is used; it tells `ferite` to note that this class inherits the variables and methods from the class named on the right of the keyword. In the example above, a class **Motor Bike** inherits from the class **Vehicle**. When an an instance of Motor Bike has been created; it is possible to set the value of each variable named in both Motor Bike and Vehicle. The code below demonstrates how it is possible to use the variables.

```
object bike = new MotorBike();
bike.handleBarType = 1; // From 'MotorBike'
bike.make = "Honda"; // From 'Vehicle'
bike.model = "Fireblade"; // From 'Vehicle'
```

`ferite` only supports single inheritance; a class may inherit from a maximum of one other class. This distinction is being made because some languages support multiple

inheritance. If we have two classes, A and B, and B inherits from A, we call A the 'super' class, and B the 'sub' class.

There are a few important points that should be noted about inheritance:

- It is possible to create a function in a subclass with the same name as a function in the super class. This allows you to create custom functionality and change the way the super class's implementation may work.

```
class A {
  function someFunction() {
    Console.println( "someFunction in A" );
  }
}

class B extends A {
  function someFunction() {
    Console.println( "someFunction in B" );
  }
}
```

  If an object of type A is created and **someFunction** called, the output will be:

```
someFunction in A
```

  However, as we have changed the **someFunction** in B, the output for an object from B will be:

```
someFunction in B
```

- When we inherit values or functions from a super class we sometimes want to access them as if we are an object of that super class; `ferite` allows you to prefix items with the keyword **super**. This is similar to **self** except that the object is seen as an instance of its super class. E.g. `super.someFunction()` will call the function `someFunction` as if it was coming from an object created from the parent class.

```
class B extends A {
  function someFunction() {
    super.someFunction(); // Call the someFunction in A
    Console.println( "someFunction in B" );
  }
}
```

  The result in making the above addition to **someFunction** in B, will cause the following output, if **someFunction** called.

```
someFunction in A
someFunction in B
```

- As mentioned before, it is possible for `ferite` to invoke a constructor on an object when it is created. It is important that if the class the object is being instantiated from inherits from another class, there is a call to the super class's constructor - it is not done automatically. This can either be done by doing super.NameOfconstructor() or super().

```
class B extends A {
  function constructor() {
    super(); // Let the super class initialize the object
  }
}
```

- You do not need to re-implement each function within a class. If a function is called on an object, and the subclass does not have an implementation for it, `ferite` will go up the tree of inheritance until it can find one.

```
class A {
  function someFunction() {
    Console.println( "someFunction in A" );
  }
}

class B extends A {
}

object o = new B();
o.someFunction(); // Will use the implementation in class A
```

The output to the above example will be:

```
someFunction in A
```

- You can inherit from a class as many times as you like and you can inherit from classes that inherit from other classes.

```
class A {
}

class B extends A {
}

class C extends B {
}

class D extends B {
}
```

```
class E extends D {
}
```

- It is possible to call the constructor of the parent class by using the special function **super()**.

```
class A {
  function constructor() {
    Console.println( "A" );
  }
}

class B extends A {
  function constructor() {
    super(); // Call the constructor in A
    Console.println( "B" );
  }
}
```

2.11.3 Static Members

When `ferite` creates an object from a class, all the variables and functions are added to the object; however, it is potentially useful to tie variables or functions to a class. An example of this need is generating a unique id for each object that has been created: the current id is stored in the class and a function to get the next id. To tell `ferite` that you want this behavior, it is necessary to use the **static** modifier. When `ferite` sees the keyword it notes that the function or variable it is used with must remain within the class and not appear in any instances.

The code below shows how to solve the unique id problem mentioned above.

```
class SpecialClass {
  // The class only items

  static number currentID;

  static function getNextUniqueID() {
    // self refers to the class
    self.currentID++;
    return self.currentID;
  }

  // The object only items

  number myID;
```

```
  function constructor() {
    // self refers to the object that has been created
    self.myID = SpecialClass.getNextUniqueID();
  }
}
```

There are a number of important things to note about the use of static items: the keyword **static** prefixes the items to be made static; to reference a static member of a class you must first use the class name and then the member you want - even if you are making the reference from within an object from that class. If you do try and access a static member within an object through self, `ferite` will throw an exception.

There is another trick that can be used with classes: static constructors. These allow you to run initialization code on a class once `ferite` has finished compiling it. The code will run during compilation **not** execution. Static constructors are the same as normal constructors with a few differences: there is no guaranteed order of execution (class A may be called before class B but it is not guaranteed), they must have no parameters and they have the static keyword prefix. An example of how to write a static constructor is below.

```
class A {
  static function constructor() {
    Console.println( "Boot Strapping Class A" );
  }
}
```

2.11.4 Access Control: Public, Protected, Private, Abstract and Final

Programmers are notoriously good at creating bugs, it is, therefore, sometimes necessary to lay down some rules on how classes and objects should be used. `ferite` has a set of access controls that allow you, the programmer, to specify how functions and variables can be accessed from outside or inside an object. Why would you want to add restrictions? Sometimes it is important to hide the mechanisms within an object or class and add a well specified interface; if you change the way things work in the background, impact on existing code should be minimal. The variables within an object are that object's data and it should be up to the object to change the values, not an external entity fishing about. If some other piece of code changed a variable without the object's consent, all manner of mayhem could occur.

What restrictions can be specified? Public, Protected, Private, Abstract and Final.

- **public**: a public variable or function that may be accessed from any part of a program; there are no restrictions. By default everything is made public.

```
class A {
  public number objectID;
}
```

- **private**: when applied to a function or variable, it will only be accessible from that object or class. **private** will hide the variable or function from any subclasses that may exist.

```
class A {
  private string mutableBuffer;
}
```

- **protected**: protected sits between private and public; a protected variable or function may only be accessed from within an object it is part of, however, unlike private, a protected item can be accessed from any subclass.

```
class A {
  protected number databaseRefID;
}
```

- **final**: Sometimes it is necessary to make sure that a variable can not be changed. **final** does a couple of things. With a variable, **final** tells `ferite` that, once a value has been assigned to it, the variable must not be modified. This is a great way of setting up constants within a program. If a program attempts to assign a new value to a final variable an exception will be thrown. When used with a class, final will make sure that another class can not inherit from it. This is useful feature allowing you, as the programmer, to dictate whether or not you want people changing the behavior of your class.

```
class A {
  static final number maxObjectCount = 1024;
}
```

- **abstract**: this can only be applied to classes. An abstract class can not be instantiated; a programmer may only inherit from an abstract class. This is useful when enforcing a factory-methods style of programming. There is the top level abstract class which all concrete implementations inherit from; functionality is driven using the abstract class. E.g. an abstract protocol class with concrete implementations for HTTP, ftp and HTTPS.

```
abstract class Protocol {
  function sendRequest( string req ) { ...  };
  function receiveRequest() { ...  };
}

class HTTP extends Protocol {
```

```
    ...
  }
```

2.11.5 Protocols

Protocols are your friend and solve the problem of multiple inheritance. What are they for? Sometimes it is necessary to have an agreed upon protocol between a collection of objects. For example, if there is an event engine that you wish to register an object with, it is important that two properties are true: the event engine can quickly and easily check to see if the object responds to the function calls the event engine expects, and that a programmer can easily craft an object that contains the correct functions.

Protocols allow, in our example, the event engine developer to define the interface that it expects all registered objects to respond to. The definition contains a set of functions without implementation and, using this protocol, developing a client to the event engine, programmers can specify that the class they are developing should conform to the protocol. When `ferite` finishes compiling a class it first executes, if it exists, the class's static constructor and then checks to see if the class implements all the functions described in the protocol list. If the class fails to conform to any of the protocols, compilation will halt with a compile error.

```
protocol EventHandler {
  function respondsToEvent( string event );
  function handleEvent( string event, array data );
}
```

This is a protocol definition. It defines a protocol called **EventHandler** with two functions. A class that implements this protocol must have two functions with the same names and signatures as these two functions. Protocols look very similar to classes in their structure.

```
class EventEngine {
  function registerEventHandler( object eventh ) {
    if( eventh.getClass().conformsToProtocol( EventHandler ) ) {
      // register the event
      return true;
    }
    return false;
  }
}
```

This code shows how to test if an object conforms to a protocol. The **getClass()** function returns the class that the object is instantiated from. **conformsToProtocol** is a function that all classes respond to; it takes one parameter and that is the name

of the protocol to check for. It returns true if the class supports the protocol and false otherwise. If true, it is safe to call the functions described within the protocol.

```
class KeystrokeEvent implements EventHandler {
  function respondsToEvent( string event ) {
    return true;
  }
  function handleEvent( string event, array data ) {
    ...
  }
}
```

This class shows how to tell `ferite` that a class should conform to a protocol. The important part of the above example is the **implements EventHandler**; it tells `ferite` that the class should 'implement' the protocol 'EventHandler'. It is possible for a class to conform to more that one protocol at a time; instead of a single protocol name, you use a comma separated list.

```
class Keystroke implements EventHandler, LogClient {
  ...
}
```

It is important to note that if you are using protocols, the extends clause must come before the implements clause.

```
class SomeClass extends AnotherClass implements SomeProtocol {
  ...
}
```

2.11.6 Dynamic Objects

Sometimes you do not know what you will have to deal with at runtime as information and structure can change. `ferite` provides mechanisms to catch missing variables and functions within objects. The mechanism makes building dynamic systems easier; for instance, a generic database table object does not know what fields may exist. To keep usage consistent with the way values are obtained from an object, the table object can implement **attribute_missing**. When `ferite` can not locate a variable within an object, the runtime will call, if it exists, the **attribute_missing** function. It passes a string to the function which contains the name of the required variable. In the case of the table object, the **attribute_missing** function loads the value from the database and returns it.

```
class DatabaseTable {
  function attribute_missing( string variable ) {
    void v = load_value_from_db( "sometable", variable );
    return v;
```

```
  }
}
```

The mechanism for missing functions is **method_missing**. When `ferite` calls this function it provides all the original parameters as well as the name of the function as the last parameter. To get the parameters it is possible to use the **variable arguments** mechanism. This functionality is used extensively within `ferite`'s rmi module.

```
class RemoteSystem {
  function method_missing( ... )
  {
    number i = 0;
    array args = arguments();

    Console.println( "Function called:  " + args[Array.size(args)-1]
);
    Console.println( "With arguments:\n" );
    for( i = 0; i < Array.size(args)-1; i++ )
      Console.println( "\tArgument ${(i+1)} = '" +
            args[i] +
            "' (${Reflection.type(args[i])})" );
    Console.println( "" );
  }
}
```

The above code prints out the name of the function and then prints out each argument and the argument's type.

2.11.7 Modifying Existing Classes

`ferite` has a number of features that allow you to modify existing classes. Why is this useful? Well, say you have a class that is used all over the place, let's say **File**, and you wish to debug a method, or re-implement a method to work around a bug, or even just add a method. It transparently allows you to shape an existing class to be how you want it to be.

To do this you use a few keywords: modifies, alias and rename. Here is an example:

```
class modifies File {

  rename readln oldReadln;
  rename open oldOpen;

  function readln(){
    return self.oldReadln(1024);
```

```
  }

  function open( string file, string mode ){
    Console.println( "Opening file $file" );
    self.oldOpen( file, mode, "" );
  }

  function toString(){
    string str = "";

    while( !self.eof() )
      str += self.readln();
    return str;
  }

  function newName() {
    ...
  }
  alias oldName newName;
}
```

To modify a class you use the syntax '**class modifies** nameOfClass', this will tell `ferite` that the target for modification is 'nameOfClass'; the class must exist otherwise you will get a compile error. Once this is done you can add new methods and variables, and manipulate the existing ones.

rename - this takes two labels, the current name and the new name and renames it. The advantage of this approach is that you can drop in a replacement method and still call the old method within your new method. The above example re-implements the **readln** method within the **File** class such that it does not require the passing of a number of bytes to read.

alias - this allows you to create a pointer to a function using a different name. This is useful when an API gets renamed to keep existing code functioning whilst it is moved over. In the above example the name **oldName** is aliased to the function **newName**.

The above example also adds a new **toString()** which will return the file's contents in a string.

WARNING: you can potentially cause a lot of confusion using this mechanism, but it is very useful for debugging and various other uses. You can modify any class within a program. This mechanism can not be applied to a class with a final modifier - it attempted, `ferite` will halt the compliation.

2.12 Namespaces

Namespaces are defined in the following manner:

namespace *name of namespace* {
  variable, namespace, class, and function declarations
}

Namespaces are a means of grouping likewise data and functions into a box to reduce the potential for name conflicts. Functions, Variables, Classes and even other namespaces can be defined within a namespace. For example, say we have several different methods for delivering error messages to a user depending on their preference. We have a couple of ways we can do it: the non namespace approach or with namespaces.

```
function text_deliverErrorMessage( string msg ) {}
function gtk_deliverErrorMessage( string msg ) {}
function qt_deliverErrorMessage( string msg ) {}
function network_deliverErrorMessage( string msg ) {}
```

It quickly gets confusing. It can only get worse when you start to add yet another function e.g. for delivering a warning. What we need is a little organization and to the rescue are namespaces.

```
namespace Text {
  function deliverErrorMessage( string msg ){}
}
namespace Gtk {
  function deliverErrorMessage( string msg ){}
}
namespace Qt {
  function deliverErrorMessage( string msg ){}
}
namespace Network {
  function deliverErrorMessage( string msg ){}
}
```

The namespace approach is cleaner and more concise. We know that all we have to do is call the 'deliverErrorMessage' in the correct namespace. The code below shows how you can use the magic of void variables to get a handle on a namespace and then call the functions within it.

```
void outputMechanism;

if( wantGtk )
  outputMechanism = Gtk;
else
  outputMechanism = Text;
```

```
outputMechanism.deliverErrorMessage( "We have an Error" );
```

They promote clean and precise code. When a function is defined within a namespace it has to reference stuff within the namespace as code outside does, e.g. `someNamespace.resource`.

Some readers will note that this can also be achieved with classes and static members, however namespaces are useful for grouping similar functionality that does not necessarily operate on the same data; such as a namespace full of mathematical functions for numbers or string routines.

2.12.1 Modifying Existing Namespaces

There is also an alternative syntax for namespaces allowing you to extend an already existing namespace or create a new one if it does not already exist. This is done like so:

namespace modifies *name of namespace* {
  variable, namespace, class and function declarations
}

When this modifies the namespace it places all items within it in the block in the namespace mentioned. e.g:

```
namespace foo {
  number i;
}

namespace modifies foo {
  number j;
}
```

In the above example the namespace **foo** has a number i and a number j. The main reason for this syntax is to allow module writers to easily intermingle native and script code within the namespace. There are times when placing something in another namespace makes more sense. e.g. placing a custom written network protocol within a Network namespace.

It is possible to use the same set of commands on namespaces as you can on classes. See **modifying existing classes** for more information.

2.13 Closures

A closure is best described as an environment capturing anonymous function. They can provide the core mechanism for a number of different methods of programming.

The most prominent use is iteration: closures can provide a very natural method and syntax for iterating over a set of elements no matter what their source. Another use is registering code to be executed when necessary. Both these will be covered within this section.

We shall first observe how closures can be used in the most basic of forms.

```
number x = 1;
number y = 1;

object o = closure {
  number z = y;

  y = 1000;
  return x + z;
};

y = 10;
x = 10;

Console.println( "If this works the next number should be 20:  ${o.invoke()}"
);
Console.println( "And y should be 1000:  $y" );
```

The above code does the following: declares two variables, creates a closure, sets the default values for the variables and then invokes the closure by calling **invoke** on it. All parts are straight forward except creating a closure. So what happens when `ferite` compiles a closure?

A function is created and compiled as you would expect, however there is one exception: if there is a reference to a variable that is not declared within the closure, `ferite` will create a binding to the variable outside the closure. The binding means that the closure can be passed to another function, stored in an array and accessed at a later point and the bound variables will be accessible. In the above example the variables x and y are both referenced within the closure, when a change to the value of x and y is done, the variables that they are bound to are updated. In the above example the variable y is changed from 1 (declared value) to 1000 and this change is reflected in the above code. This form of binding is known as static binding: it means that the variables are bound in the context of the closure's creation rather than at the time of the closure's execution.

Closures are seen by `ferite` as objects making it easy to pass them from one function to another. The code below is a modification to the above example and demonstrates passing values into a closure.

```
function incrementBy( object c, number value ) {
```

```
  number x = 99;
  return c.invoke( value );
}

function closureEx() {
  number x = 1;
  object c = closure( ivalue ) {
    x = x + ivalue;
  };

  // x = 1
  incrementBy( c, 10 );
  // x = 11
  incrementBy( c, 100 );
  // x = 111
}
```

The above function *closureEx* creates a closure that accepts one value, in this example called *ivalue*, and adds it to the current value of x. The closure binds to the x variable declared above it. The function *incrementBy* takes a closure object and a number, invokes the closure passing it the number. The function *incrementBy* is provided to demonstrate the nature of binding: even though there is an x variable in incrementBy, the closure has already bound to the x variable in closureEx.

Passing values into a closure is straight forward, it is the same as calling a function. Declaring is as easy as declaring the parameters a function requires, except you do not declare the types. Each parameter is of type void making closures useful for dynamic programming.

```
function nTimes( number multiplier ) {
  return closure( base ) {
    return base * multiplier;
  };
}

object times_two = nTimes( 2 );
object times_ten = nTimes( 10 );

Console.println( "multiple of 2:  2:  ${times_two.invoke(2)}" );
// (1)
Console.println( "multiple of 2:  5:  ${times_two.invoke(5)}" );
// (2)
Console.println( "multiple of 10:  2:  ${times_ten.invoke(2)}" );
// (3)
Console.println( "multiple of 10:  5:  ${times_ten.invoke(5)}" );
```

```
// (4)
```

The above is yet another example of binding: even when a function exits, a variable bound within a closure will not be deleted from the program until the closure itself is deleted (by the garbage collector). The function *nTimes* creates a closure that will return the multiplication of the base parameter and the bound multiplier value. The creation of the **times_two** closure binds its multiplier value to '2' and the creation of the **times_ten** binds its multiplier value to '10'. Therefore the results of the printout lines are as follows:

```
multiple of 2:  2:  4
multiple of 2:  5:  10
multiple of 10:  2:  20
multiple of 10:  5:  50
```

Hopefully you should have a reasonable understanding of the core features of closures. More examples on how they can be applied to real world programming problems will be presented in section 3.

2.13.1 Recipients, Deliver and Using

Closures are a great way of injecting custom actions into generic code. For instance, if you have written an advanced structure, you may want to provide a generic sorting function or a generic walking function. However there is a slight problem: you do not know what type of information may be stored in your structure. The solution is to write a generic walking or sorting algorithm and use a passed in closure that can deal with the specifics of the data and provide the overall functionality. This solution can be applied to a lot of different mechanisms and to make code less ambiguous, `ferite` provides a special interface to passing a closure into a function, checking to see if a function has received a closure and executing the supplied closure.

To demonstrate how to use the special interface here is the implementation of the function **Array.each** and some code that uses it.

```
namespace Array {
  function each( array a ) {
    if( recipient() != null )
    {
      number i = 0;
      number size = Array.size(a);

      for( i = 0; i < size; i++ )
      {
        void val = deliver( a[i] );
        if( val == false or val == null )
```

```
        break;
      }
    }
  }
}

array a = [ 1, 2, 3, 4, 5 ];
Array.each( a ) using ( value ) {
  Console.println( "Array value:  $value" );
};
```

The above example passes an array and a closure to Array.each, which then iterates over each element within the array and delivers the value to the supplied closure. When the closure is invoked it prints out the value passed to it.

The new keywords introduced in this example are highlighted in bold: **recipient()**, **deliver()** and **using**. 'recipient()' is a special function, like 'arguments()', that is core to the `ferite` engine; the job of the call is to return an object or null if there is, or is not, a supplied recipient: a closure. 'deliver()' is another special function that delivers a set of values to a closure, it simply gets the recipient and calls the **invoke** function on it. If there is no recipient, deliver will throw an exception. 'using' is how you provide a closure to the function call: you append using after the function call and then define your closure *without* the **closure** keyword.

The code below demonstrates the difference in manually writing the above mechanism.

```
namespace Array {
  function each( array a, object c ) {
    if( c != null ) {
      number i = 0;
      number size = Array.size(a);

      for( i = 0; i < size; i++ ) {
        void val = c.invoke( a[i] );
        if( val == false or val == null )
          break;
      }
    }
  }
}

array a = [ 1, 2, 3, 4, 5 ];
Array.each( a, closure ( value ) {
  Console.println( "Array value:  $value" );
```

```
} );
```

It is generally more untidy and and less clear what the intended operation for Array.each is; hence the addition of this syntax. It is possible to pass a closure onto another function by using recipient() rather than a new closure.

```
function someFunction() {
  anotherFunction() using recipient();
}

someFunction() using ( v ) {
  ...
};
```

2.14 Uses and Include

Both the **uses** and the **include()** instructions tell `ferite` to include another script within the current one. The main difference is that **uses** is a compile time directive and **include()** is a runtime directive.

It is important to note that `ferite` will, when using or including a script, use a set of paths to resolve relative file names. It will always try and find the script relative to the current script file first and then search in the global paths.

In the case of the `ferite` strong line application, `$prefix/lib/ferite` is searched for scripts plus any directories that are added on the strong line by using the -I flag. Native modules are placed in `$prefix/lib/ferite/$platform`, where $platform is of the form `os-cpu`. If either the script or the module can not be found the compilation of the script will cease with an error. It is suggested that these are placed at the top of the script (although this is not a requirement).

2.14.1 Uses

The uses keyword is used to import API from other external modules and scripts. The uses keyword is a compile time directive and provides the method for building up the environment. It can either pull in an external module, or compile in another script. The syntax is as follows:

uses *"name of module or script file", ...*;

The name must be in quotes. When `ferite` gets this call it will do the following: if there is no extension it will try loading a script in the system's library paths trying the extensions '.fe', '.fec' and '.feh'. The paths for the native and scripts are defined by the parent application. If an extension is given, `ferite` will check to see if it equals **.lib**, if it does it will load the correct native module, e.g. `uses "array.lib";` will cause `ferite`

to load `array.so` under unices and `array.dll` under windows. This gives a platform independent method to tell `ferite` to load a native library. This is the method used to load a native module. If it does not equal **.lib**, `ferite` will treat it as a script and load it.

2.14.2 Include

include() operates the same way as **uses**, except that it can currently only import other scripts. Once the call has been made - the facilities provided by the imported script can be used. It should be noted that the return value from the include() call is the return of the main method when the script is loaded. This allows items to be passed to the parent script.

```
void v = include ( "someScript.fe" );
```

The above code will assign the return value from 'someScript.fe's main code to the variable v. This is a useful mechanism for writing plug-in architectures: include a script and have the return be an object that represents the loaded plug-in.

2.15 Directives

A directive is a function that is only used at compile time by the compiler. They were initially added to `ferite` to provide a means to attach meta data to objects, classes and namespaces within the run time. They have since evolved into a very clean mechanism that almost mimics macros in other languages. Lets dip into an example:

```
class modifies Obj { // (1)
 directive reader( string type, string name ) { // (2)
  eval( " // (3)
    class modifies ${Class.name(self)} {
      private $type $name;
      function get$name() {
        return .$name;
      }
    }" );
 }
}

class Example {
  [reader number X]; // (4)
  [reader string Y];
  [reader boolean Z];
}
```

Firstly, the example uses the **modifies** keyword to inject a directive into all classes by modifying the root class 'Obj' (1). Declaring a **directive** is no different to writing a **function** except that instead of the`function` keyword you use `directive` (2). This directive is called reader, takes two arguments that are used to specify the type and name of a variable. The directive uses these values to run an **eval** expression that modifies the class that directive is used on to, adding a variable and a getter `function` (3). The power of directives comes from their application. Looking at point (4) in the example, the directive is invoked and causes a number variable called X to be declared and a function getX added to the class.

All directive applications are structured the same:

`[`*directive_name parameter_1 ... parameter_n*`];`

There are a few key pieces of information that should be considered when using directives:

- Applied directives are always evaluated when a class has finished being parsed and are called *before* the static constructor.
- Applied directives are always evaluated in the order they appear within the class declaration.
- Directives are inherited.
- Invoking directives using the square bracket notation allows commas between arguments but it is not required.
- To allow more expressive directive application, `ferite` will convert the keywords `boolean`, `number`, `string`, `void`, `object`, `array`, `class` and `namespace`, when used within the square brackets, into strings representing the same value.
- You can use numerical, string and boolean constants within the directive application, however expression are not supported.

Directives provide a very nice mechanism that can be very powerful when you wish to apply repetitive actions in a clean fashion. The ferite webframework uses directives extensively to mark functions as accessible to external RPC mechanisms, binding components to variables within the controllers, setting up attributes on components to name a few. All can be done without directives, but not nearly as cleanly or elegantly.

2.16 Conclusion

We have been on an epic journey: a once foreign language should be familiar and dancing around in your head, making life easier. It is important that we can now look back at the first example given in this manual: it should now be easy to recognize and understand what is going on. To aid you a little further, keywords have been made bold.

```
// The importing of extra functionality
uses "console", "array";

// Declaration of a function. Classes and Namespaces also go here
function processArgument( string argument ) {
  Console.println( "Argument:  " + argument );
}

// The startup code
Array.each( argv ) using ( argument ) {
  processArgument( argument );
};
```