### Modules and Applications

#### Creating modules

A module consists of a sub-directory inside the top level '/modules/' directory, that contains a 'root' monkey2 file with the same name as the sub-directory and a 'module.json' file , eg:

/modules/my-module/  
/modules/my-module/my-module.monkey2  
/modules/my-module/module.json

Module names can contain any valid identifier or the - character. However, the - character cannot be used twice or more in succession, eg:

legal-module-name  
illegal--module--name

Module names live in a 'flat' namespace so should be as unique as possible.

The 'module.json' file must contain a json object with the following fields:

* "module" : string - the name of the module. Must be the same as the directory name.
* "author" : string - the author of the module.
* "version" : string - the version of the module. This should be in number 'dot' number 'dot' number format, eg: "1.0.0".
* "depends" : string array - All modules this module depends on. This will generally include all other modukes #imported by the module, eg: ["libc","std"].

A simple module.json file might look like this:

{  
 "module" : "my-module",  
 "version" : "1.0.0",  
 "depends" : ["libc","std"]  
}

To rebuild a module, use the mx2cc 'makemods' option, eg:

mx2cc makemods my-module

This will cause the root 'modules/my-module/my-module.monkey2' file to be built, along with any local files it imports.

You can also rebuild all modules with:

mx2cc makemods

This will use the "depends" information in the module.json files to determine the correct order to build the modules in.

Modules must not have cyclic dependancies.

Each module may also declare a single Main:Void() function that can be used to initialize the module.

This function is called *after* global variables (including global Consts) have been initialized but *before* the application's Main is called.

Since modules can't have cyclic dependencies, Mains will always execute in the correct order, eg: if module X imports module Y, then module Y's Main is guaranteed to be called before module X's.

#### Importing modules

Once built, a module can be imported into other modules using 'import', eg:

#Import "<my-module>"

#### Applications

An application is really just a special type of module. It doesn't have to reside in the '/modules/' directory, can't be imported by other modules and produces executable files instead of archives, but is otherwise dealt with just like a module.

An application must declare a Main:Void() function. Actually, there's no difference between the Main declared in the app, and Mains declared in modules. Since the app depends on ALL modules, and NO modules depend on the app, the app's Main just happens to be the last one called.

### Namespaces and Using

All identifiers declared in a monkey2 program file end up inside a 'namespace'. Namespaces are hierarchical, so in addition to identifiers declared by monkey2 code, namespaces can also contain other namespaces.

#### Declaring namespaces

You control which namespace the identifiers declared in a monkey2 file go with the namespace directive:

Namespace *namespace-path*

This directive must appear at the top of the program file, before any actual declarations are made.

A namespace path is a 'dot' separated sequence of identifiers, eg:

monkey.types std.collections

The 'dot' separator indicates the namespace hierarchy, eg: monkey.types is a 'child' or 'inner' namespace of monkey.

If no namespace is specified in a program file, the identifiers go into a default namespace. This is not recommended though, as the default namespace has no name - so anything inside the default namespace cannot be 'seen' by anything outside.

#### Accessing namespaces

Code can access an identifier in a namespace by prefixing the identifier with the namespace path, eg:

Local list:=New std.collections.List<Int>

Here, std.collections refers to a namespace, while List is an identifier inside the std.collections namespace.

Code inside a particular namespace does not have to use a namespace prefix to find identifiers in the same namespace or in any parent namespace, although it can still do so, eg:

Namespace testing  
  
Function Test()  
End  
  
Function Main()  
 Test() 'works fine...  
 testing.Test() 'also works...  
End

#### The Using directive

To make it easier to deal with long or complex namespace paths, the using directive provides a way to add namespace 'search paths' for locating identifiers. The using directive looks like this:

Using *namespace-path*

A program can have multiple using directives. Using directives must appear at the top of the program file before any program declarations.

Each using directive adds a namespace to a set of 'search paths' that is used to resolve any unknown identifiers in the program code, eg:

#Import "<std>"  
  
Using std.collections  
   
Function Main()  
 Local list:=New List<Int>  
End

Without the using directive, this program would fail to compile because the List identifier cannot be found.

However, the using direct tells the compiler look for List in the std.collections namespace, where it is successfully located.

If you have multiple using directives and an identifier is found in more than one namespace, you will still get a compile error. In this case, you will need to 'fully qualify' the identifier by prefixing it with the correct namespace path.

Some modules declare many namespaces, and it can often be difficult to remember where everything is. To deal with this, Monkey2 provides a convenient 'double dot' form of Using that will use both a namespace AND all namespaces contained in that namespace. For example:

Using std..

The double dots must appear at the end of the using.

This will use the std namespace, and the std.collections, std.filesystem etc namespaces too. This works recursively, so any namespaces inside std.collections and std.filesystem are also used.

This can of course lead to more 'duplicate identifier' clashes but is none-the-less very convenient.

### Monkey2 types

#### Primitive types

The following primtive types are supported by monkey2:

|  |  |
| --- | --- |
| Type | Description |
| Void | No type. |
| Bool | Boolean type. |
| Byte | 8 bit signed integer. |
| UByte | 8 bit unsigned integer. |
| Short | 16 bit signed integer. |
| UShort | 16 bit unsigned integer. |
| Int | 32 bit signed integer. |
| UInt | 32 bit unsigned integer. |
| Long | 64 bit signed integer. |
| ULong | 64 bit signed integer. |
| Float | 32 bit floating point. |
| Double | 64 bit floating point. |
| String | String of 16 bit characters. |
| Enum | 32 bit UInt Enumerated type. |

#### Compound types

The following compound types are supported by monkey2:

|  |  |
| --- | --- |
| Type | Description |
| *Type* [ [,...] ] | Array type |
| *Type* Ptr | Pointer type |
| *Type* ( *Types* ) | Function type |

#### Implicit type conversions

These type conversions are performed automatically:

|  |  |
| --- | --- |
| Source type | Destination type |
| Any numeric type | Bool |
| String or array type | Bool |
| Class or interface type | Bool |
| Any numeric type | Any numeric type |
| Any numeric type | String |
| Any pointer type | Void Ptr |
| Any enum type | Any integral type |
| Class or interface type | Base class type or implemented interface type |

When numeric values are converted to bool, the result will be true if the value is not equal to 0.

When strings and arrays are converted to bool, the result will be true if the length of the string or array is not 0.

When class or interface instances are converted to bool, the result will be true if the instance is not equal to null.

When floating point values are converted to integral values, the fractional part of the floating point value is simply chopped off - no rounding is performed.

#### Explicit type conversions

The Cast < *dest-type* > : *dest-type* ( *expression* ) operator must be used for these type conversions:

|  |  |
| --- | --- |
| Source type | Destination type |
| Bool | Any numeric type |
| String | Any numeric type |
| Any pointer type | Any pointer type, any integral type |
| Any integral type | Any pointer type, any enum type |
| Class type | Derived class type, any interface type |
| Interface type | Any class type, any interface type |

When casting bool values to a numeric type, the result will be 1 for true, 0 for false.

### Arrays

An array is a linear sequence of values that can be addressed using one or more integer indices.

Each array has an associated element type. That is, the type of the values actually stored in the array. An array's element type is a purely static property. It is only known at compile time so arrays cannot be 'downcast' at runtime.

The syntax used for declaring values and variables of array type is:

*ElementType* [ [,...] ]

An array can also be multidimensional, in which case the '[]' will contain 1 or more commas.

Here are some example of declaring array variables:

Local ints:Int[] 'One dimensional int array.  
Local map[,] 'Two dimension int array.  
Local funcs:Int()[] 'One dimensional array of functions of type Int().  
Local stacks:Stack<Int>[] 'One dimensional array of stacks of type Int.

#### Creating arrays

Declaring an array does not actually create an array. To do that you must use New.

New can be used to create either an unintialized or preinitialized array. The syntax for creating an uninitialized array is:

New *ElementType* [ *DimensionSizes* ]

(Note: the elements of an 'uninitialized' array are actually initialized to 'Null'!)

The syntax for creating an initialized array is:

New *ElementType*

Here are some examples:

Local ints:Int[]=New Int[10] 'Creates a ten element integer array.  
Local flts:=New Float[]( 1.0,3,5.1,7,9.2 ) 'Creates a 5 element float array initialized to 1.0,3,5.1,7,9.2

#### Iterating through arrays

You can iterate through the elements of an array using Eachin, eg:

Local arr:=New Int[]( 1,3,5,7,9 )  
For Local i:=Eachin arr  
 Print i  
Next

#### Slicing arrays

One dimensional arrays can be sliced using the Slice method, eg:

Local ints:=New Int[]( 1,3,5,7,9 )  
ints=ints.Slice( 1,4 ) 'ints now contains 3,5,7

For more information, see the [[types.Array.Slice|Array.Slice]] API documentation.

#### Resizing arrays

One dimensional arrays can be resized using the Resize method, eg:

Local ints:=New Int[]( 1,2,3 )  
ints=ints.Resize( 5 ) 'ints now contains 1,2,3,0,0

(Note that resize does not actually resize the array! It actually returns a resized *copy* of the array.)

Note that mutidimensional arrays cannot currently be sliced or resized, and you cannot create an initialized multidimensional arrays. These features are planned for the future though.

For more information, see the [[types.Array.Resize|Array.Resize]] API documentation.

### Strings

Values of type String are used to represent sequences of characters, such as text. The exact size of each character in a string value is target dependent, but is at least 8 bits.

String variables are declared using the type name String, for example:

Local test:String="Hello World"

String literals are sequences of characters enclosed in "" (quotation marks). String literals may also include escape sequences, special sequences of characters used to represent unprintable characters.

You can use the following escape sequences in string literals:

|  |  |
| --- | --- |
| Escape sequence | Character code |
| ~q | 34 (quotation mark ") |
| ~n | 10 (newline) |
| ~r | 13 (return) |
| ~t | 9 (tab) |
| ~z | 0 (null) |
| ~~ | 126 (tilde ~) |

For example, to include literal quotation marks in a string...

Local test:="~qHello World~q"

You can index a string using the `[]' operator, eg:

Local str:="Hello World"  
For Local i:=0 Until str.Length  
 Print str[i]  
Next

Indexing a string will return the character code at a given string index as an int.

You can iterate through the characters in a string using Eachin, eg:

For Local chr:=Eachin "Hello World"  
 Print chr  
Next

For more information on strings, please see the [[types.String|String]] API reference. ### Variants

The Variant type is a primitive type that can be used to 'box' values of any type.

The easiest way to create a variant is to cast a value to Variant (much like casting an Int to String etc), eg:

Local v:=Variant( 10 )

An uninitialized variant will contain a 'null' value (of type Void) until you assign something to it:

Local v:Variant  
v=10 'variant now contains an int 10.  
v="hello" 'variant now contains a string "hello".

A variant is 'true' if it contains any value with a non-void type (including a bool false value!) and 'false' if it is uninitialized and has no (void) type.

Any type of value can be implicitly converted to a variant, so you can easily pass anything to a function with variant parameters:

Function Test( v:Variant )  
End  
  
Function Main()  
 Test( 1 )  
 Test( "Hello" )  
 Test( New Int[] )  
 Test( Main )  
End

To retrieve the value contained in a variant, you must explicitly cast the variant to the desired type:

Local v:=Variant( 100 )  
Print Cast<Int>( v )

Note that the cast must specify the exact type of the value already contained in the variant, or a runtime error will occur:

Local v:=Variant( 10 )  
Print Cast<String>( v ) 'Runtime error! Variant contains an Int not a String!

The one exception to this is if the Variant contains a class object, in which case you can cast the variant to any valid base class of the object. ### Enums

Enum is a data type containing a set of UInt constants.

By default the members will receive values starting from zero and incemented by one for each new member. You can assign a chosen value to each member when declaring them.

Enum myBasicEnum  
 a,b,c 'a=0, b=1, c=2  
End

Enum myCustomEnum  
 a=7  
 b=31,c,d 'c=32, d=33  
End

The values can be accessed with the postfix member acces operator (.). Enums values are implicitly converted to integral values when assigned to it.

Local i:UInt=myCustomEnum.b

You can also create Enum variables. An Enum variable contains a Uint variable in addition to it's constant members (default value is zero).

Bitwise operators (|,&,~) can be used with Enums variables and Enums members to compute combinations. Such Enums most often contain powers of 2 numbers as members! (1,2,4,8,16,32,64,... and 0 if needed).

A bitmask Enum example:

Enum Flags 'a classic Enum. (4 bits bitmask)  
 None=0  
 A=$0001 'bin OOOI dec 1  
 B=$0002 'bin OOIO dec 2  
 C=$0004 'bin OIOO dec 4  
 D=$0008 'bin IOOO dec 8  
End

An enum with modifiers example (in this case the bitwise operators should be used with at least one modifier):

Enum Foo '(modifiers on 5th and 6th bit)  
 None=0  
 A=1,B,C,D,E,F,G,H,J,K,L,M ' max 15 because the 5th bit is used for modifier  
 Modifier\_A=$0010 'bin IOOOO dec 16  
 Modifier\_B=$0020 'bin IOOOOO dec 32  
End

Some operations examples:

Local flags1:=Flags.A | Flags.B  
Local flags2:=flags1 | Flags.C  
Local flags3:=Flags.B & (flags1 ~ flags3.D)

If needed, you can 'extract' the variable value by simply assigning your Enum variable to a UInt.

Local i:UInt=flags1

### Variables

#### Local variables

Local variables live on the stack. To declare a local variable:

Local *identifier* : *Type* [ = *Expression* ]

...or...

Local *identifier* := *Expression*

#### Global variables

Global variables live in global memory and exist for the lifetime of the application. To declare a global variable:

Global *Identifier* : *Type* [ = *Expression* ]

...or...

Global *Identifier* := *Expression*

#### Consts

Consts are stored in the same way as globals, but cannot be modified after they are initialized. To declare a const:

Const *Identifier* : *Type* = *Expression*

...or...

Const *Identifier* := *Expression*

### Pointers

Pointers are special variables containing a memory adress. In Monkey2 pointers are mainly used with external C/C++ code. It is not advised to use pointers if not necessary. It can lead to bug or memory acces violation if the pointed adress is not kept "alive". Pointer to globals are safe for example.

A pointer can point to any kind of type, even garbage collected types. This can lead to bad things too as the garbage collector is not 'aware' of pointers.

#### Declaration

Use the Ptr keyword to declare a pointer.

Local myPtr:int Ptr  
  
Local anotherPtr:Void Ptr

#### Referencing

Use the VarPtr operator to reference a pointer

Local i:int=1  
Local myPtr:int Ptr  
  
myPtr=VarPtr i

The myPtr pointer now points to the variable i

#### Dereferencing with []

You can access the pointed value(s) with the [] index operator

Local i:int=1  
Local myPtr:int Ptr  
  
myPtr=VarPtr i  
Print myPtr[0]

Will print 1, the value of i. Note you can use pointer arythmetics with the index operator([]) but you have to be sure you have access to that part of the memory or you'll get a memory acces violation!

#### Dereferencing with ->

You can acces user defined types fields, methods,.. with the -> operator. It is equivalent to [0].

Struct str  
 Field i:Int=1  
End  
  
Function Main()  
 Local s:=New str  
 Local strPtr:str Ptr  
 strPtr=VarPtr s  
  
 Print strPtr->i   
End

will show the value of the struct's field i

#### Casting

You can Cast a pointer and do some explicit conversions with the Cast operator.

Cast<*Type*>(*adress*)

An example with a useless conversion from Int to Void to Int:

Local i:int=1  
Local myVoidPtr:Void Ptr  
  
myVoidPtr=Cast<Void Ptr>(VarPtr i)  
  
Local j:int  
Local myIntPtr:Int Ptr  
  
myIntPtr=Cast<Int Ptr>(myVoidPtr)  
j=myIntPtr[0]

j receives to value of i but does not have the same adress. myIntPtr and myVoidPtr both points to the same adress(VarPtr i) but have different types. ### Functions

#### Global functions

To declare a global function:

Function *Identifier* [ *GenericParams* ] [ : *ReturnType* ] ( *Parameters* )

...Statements...

End

*ReturnType* defaults to Void if omitted.

*Parameters* is a comma separated list of parameter declarations.

#### Class methods

The syntax for declaring a class method is:

Method *Identifier* [ *GenericParams* ] [ : *ReturnType* ] ( *Parameters* ) [ Virtual|Abstract|Override`|Final|Override Final] ``` ...Statements... ```End`

If a method is declared Virtual or Abstract, it can be overriden by methods in derived classes. Overriding methods must have the same return type and parameter types as the class method, and must be declared Override.

If a method is declared Abstract, no implementation may be provided (ie: no 'statements' or 'End'). Such a method must be overriden by a method in a derived class, and also makes its enclosing class implictly abstract (an abstract class cannot be instantiated).

If a method is declared Override or Override Final, it must override a virtual method in a base class.

If a method is declared Final or Override Final, it cannot be overriden by any methods in derived classes.

By default, class methods are final.

#### Lambda functions

To declare a lambda function:

...Lambda [ : *ReturnType* ] ( *Parameters* )

...Statements...

End...

Lambda declarations must appear within an expression, and therefore should not start on a new line.

For example:

Local myLambda:=Lambda()  
 Print "My Lambda!"  
End  
  
myLambda()

To pass a lambda to a function:

SomeFunc( Lambda()  
 Print "MyLambda"  
End )

Note the closing ) after the End to match the opening ( after SomeFunc.

#### Function values

Monkey2 supports 'first class' functions.

This means function 'values' can be stored in variables and arrays, passed to other functions and returned from functions. ### Loop statements #### While

The While loop allows you to execute a block of statements repeatedly while a boolean expression evaluates to true.

Note that a While loop may never actually execute any of it's statements if the expression evaluates to false when the loop is entered.

The syntax for the While loop is:

While *Expression*

\_Statements...\_

Wend

End or End While may be used instead of Wend.

Exit and Continue may be used within a While loop to abruptly terminate or continue loop execution.

#### Repeat

Like the While loop, the Repeat loop also allows you to execute a block of statement repeatedly while a boolean expression evaluates to true.

However, unlike a While loop, a Repeat loop is guaranteed to execute at least once, as the boolean expression is not evaluated until the end of the loop.

The syntax for Repeat/Until loops is:

Repeat

\_Statements...\_

Until *Expression*

..or..

Repeat

\_Statements...\_

Forever

Exit and Continue may be used within a While loop to abruptly terminate or continue loop execution.

#### For (Numeric)

A numeric For loop will continue executing until the value of a numeric index variable reaches an exit value.

The index variable is automatically updated every loop iteration by adding a constant step value.

The syntax for a numeric For loop is:

For [ Local ] *IndexVariable* [:]= *FirstValue* To | Until *LastValue* [ Step *StepValue* ]

\_Statements...\_

Next

End or End For may be used instead of Next.

If present, Local will create a new local index variable that only exists for the duration of the loop. In addition, IndexVariable must include the variable type, or := must be used instead of = to implicitly set the variable's type.

If Local is not present, IndexVariable must be a valid, existing variable.

The use of To or Until determines whether LastValue should be inclusive or exclusive.

If To is used, the loop will exit once the index variable is greater than LastValue (or less than if StepValue is negative).

If Until is used, the loop will exit once the index variable is greater than or equal to LastValue (or less than or equal to if StepValue is negative).

If omitted, StepValue defaults to 1.

Exit and Continue may be used within a numeric For loop to abruptly terminate or continue loop execution.

#### For (EachIn)

A For EachIn loop allows you to iterate through the elements of a collection.

A collection is either an array, a string, or a specially designed object.

The syntax for a For EachIn loop is:

For [ Local ] *IndexVariable* [:]= EachIn *Collection*

\_Statements...\_

Next

End or End For may be used instead of Next.

If present, Local will create a new local index variable that only exists for the duration of the loop. In addition, IndexVariable must include the variable type, or := must be used instead of = to implicitly set the variable's type.

If Local is not present, IndexVariable must be a valid, existing variable.

If Collection is an array, the loop will iterate through each element of the array, and the type of the index variable must match the element type of the array.

If Collection is a string, the loop will iterate through each character code of the string, and the type of the index variable must be numeric.

If Collection is an object, it must implement the std.collections.Icontainer interface. See std-std-collections-icontainer.

#### Exit

Exit can be used within While, Repeat and For loops to abruptly exit the loop before the loop termination condition has been met.

#### Continue

Continue can be used within While, Repeat and For loops to force the loop to abruptly skip to the next loop iteration, skipping over any statements that may be remaining in the current loop iteration. ### Conditional statements

#### If

The If statement allows you to conditionally execute a block of statements depending on the result of a series of boolean expressions.

The first boolean expression that evaluates to true will cause the associated block of statements to be executed. No further boolean expressions will be evaluated.

If no boolean expression evaluates to true, then the final else block will be executed if present.

The syntax for the If statement is:

If *Expression* [ Then ]

\_Statements...\_

ElseIf *Expression* [ Then ]

\_Statements...\_

Else

\_Statements...\_

EndIf

There may be any number of ElseIf blocks, or none. The final Else block is optional.

End or End If may be used instead of EndIf, and Else If may be used instead of ElseIf.

In addtion, a simple one line version of If is also supported:

If *Expression* [ Then ] *Statement* [ Else *Statement* ]

#### Select

The Select statement allows you to execute a block of statements depending on a series of comparisons.

The first comparison to produce a match will cause the associated block of statements to be executed.

If no comparisons produce a match, then the final Default block will be executed if present.

The syntax for the Select statement is:

Select *Expression*

Case *Expression* [ , *Expression*... ]

\_Statements...\_

Default

\_Statements...\_

End [ Select ]

There may be any number of Case blocks, or none. The final Default block is optional. If the default block is present, it must appear after all Case blocks.

#### ? Else

the ? Else operator is used to assign a value with a condition:

*variable*=*Expression* ? *Expression-A* Else *Expression-B*

the *variable* will receive the value of *Expression-A* if *Expression* is True, else it will receive the value of *Expression-B*.

i=j>2 ? 5 else j+7

### Expressions

#### Operators

|  |  |  |
| --- | --- | --- |
| Operator | Description | Precedence |
| New | New object or array | 1 |
| Null | Null value |  |
| Self | Self instance |  |
| Super | Super instance |  |
| True | Boolean true |  |
| False | Boolean false |  |
| Typeof | Typeof operator |  |
| Cast | Cast operator |  |
| Lambda | Lambda function |  |
| *identifier* | Identifier |  |
| *literal* | Literal value |  |
|  |  |  |
| . | Postfix member acccess | 2 |
| ( ) | Postfix Invoke |  |
| [ ] | Postfix Index |  |
| < > | Postfix Generic instance |  |
|  |  |  |
| Varptr | Unary variable address | 3 |
| - | Unary numeric negate |  |
| ~ | Unary integer complement |  |
| Not | Unary boolean invert |  |
|  |  |  |
| \* | Numeric multiplication | 4 |
| / | Numeric division |  |
| Mod | Numeric modulo |  |
|  |  |  |
| + | Numeric addition | 5 |
| - | Numeric subtraction |  |
|  |  |  |
| Shl | Integer shift left | 6 |
| Shr | Integer shift right |  |
|  |  |  |
| & | Integer and | 7 |
| ~ | Integer xor |  |
|  |  |  |
| \| | Integer or | 8 |
|  |  |  |
| <=> | Compare | 9 |
|  |  |  |
| < | Less than | 10 |
| > | Greater than |  |
| <= | Less than or equal |  |
| >= | Greater than or equal |  |
|  |  |  |
| = | Equal | 11 |
| <> | Not equal |  |
|  |  |  |
| And | Boolean and | 12 |
|  |  |  |
| Or | Boolean or | 13 |
|  |  |  |
| ? Else | If-then-else | 14 |

#### Type balancing

When evaluating an operator's operands, it is sometimes necessary to adjust the type of one or both operands.

When evaluating the operands of arithemetic or comparison operators, the following rules are used:

* If either operator String, the other is converted to String.
* Else If either operand is Double, the other is converted to Double.
* Else if either operand is Float, the other is converted to Float.
* Else if either operand is ULong, the other is converted to ULong.
* Else if either operand is Long, the other is converted to Long.
* Else if either operand is UInt, the other is converted to UInt.
* Else if either operand is unsigned, both are converted to UInt.
* Else both operands are converted to Int.

When evaluating the operands of the &, | and ^ integer operators, both operands must be integral types and are converted as follows:

* If either operand is ULong, the other is converted to ULong.
* Else if either operand is Long, the other is converted to Long.
* Else if either operand is UInt, the other is converted to UInt.
* Else if either operand is unsigned, both are converted to UInt.
* Else both operands are converted to Int.

When evaluating the operand of the Shl and Shr integer operators, the left-hand-side must be an integral type, while the right-hand-side 'shift amount' operand is converted to Int.

#### Operator overloading

Operator overloading allows you to customize the behavior of the built-in monkey2 operators for classes and structs.

You overload an operator by writing an 'operator method', which is effectively just a special kind of method. Operators must appear inside classes/structs - they cannot currently be 'global'.

Here is a simple example:

The 'Operator+' declaration here defines an addition operator for Vec2. This is then used whenever a Vec2 appears as the 'left hand side' of an addition. For example:

The following unary operators can be overloaded: + - ~

The following binary operators can be overloaded: \* / Mod + - Shl Shr & | ~ = <> < > <= >= <=>

The following assignment operators can be overloaded: \*= /= Mod= += -= Shl= Shr= &= |= ~=

Indexing behaviour can also be overloaded using [] and []=

Note that you cannot overload Not, And, Or or plain assignment =

Operators can return any type of value, and can take any type of value for their 'right hand side' argument(s). However, the precedence of operators cannot be changed.

The [] and []= operators allow you to define 'indexing' like behaviour. The [] operator is used when an object is indexed, and []= is used when an object is indexed and assigned. Both of these operators can accept any number of parameters of any type. The []= operator requires an additional parameter that is the value to be assigned. This must appear at the end of the parameter list.

Here is an example of some indexing operators for the Vec2 class above:

With these additions, you can access Vec2 coordinates 'by index', eg:

You can also overload assignment operators, for example:

If you have already written an Operator+ (as is the case here) this is not strictly necessary, as monkey2 will generate the code for Operator+= for you. However, you may still want to provide a custom version for Operator+= if your code can do so in a more efficient way.

### User defined types

#### Classes

A class is a kind of 'blueprint' for creating objects at runtime.

The syntax for declaring a class is:

Class *Identifier* [ < *GenericTypeIdents* > ] [ Extends *SuperClass* ] [ Implements *Interfaces* ] [ *Modifier* ]  
...Class Members... End

*SuperClass* defaults to Object if omitted.

*Interfaces* is a comma separated list of interface types.

*Modifier* can be one of:

* Abstract - class cannot be instantiated with 'New', it must be extended.
* Final - class cannot be extended.

Classes can contain const, global, field, method and function declarations, as well as other user defined types.

Once you have declared a class, you can create objects (or 'instances') of that class at runtime using the New operator.

Classes are 'reference types', meaning that class instances are really just a 'handle' or 'pointer' to the actual class data.

#### Structs

Structs are similar classes, but differ in several important ways:

* A struct is a 'value type', whereas a class is a 'reference type'. This means that when you assign a struct to a variable, pass a struct to a function or return a struct from a function, the entire struct is copied in the process.
* Stucts are statically typed, whereas classes are dynamically typed.
* Struct methods cannot be virtual.
* A struct cannot extend anything.

To declare a struct:

Struct *Identifier* [ < *GenericTypeIdents* > ] ...Struct members... End

A struct can contain const, global, field, method and function declaratins, as well as other user defined types.

#### Interfaces

To declare an interface:

Interface *Identifier* [ < *GenericTypeIdents* > ] [ Extends *Interfaces* ] ...Interface members... End

*Interfaces* is a comma separated list of interface types.

An interface can contain consts, globals, fields, methods, functions and other user defined types.

Interface methods are always 'abstract' and cannot declare any code.

#### Fields

Fields are variables that live inside the memory allocated for an instance of a class or struct. To declare a field variable:

Field *identifier* : *Type* [ = *Expression* ]

...or...

Field *identifier* := *Expression*

For struct fields, *Expression* must not contain any code that has side effects.

#### Methods

To declare a method:

Method *Identifier* [ < *GenericTypeIdents* > ] [ : *ReturnType* ] ( *Arguments* ) [ *Modifiers* ] ...Statements... End

*ReturnType* defaults to Void if omitted.

*Arguments* is a comma separated list of parameter declarations.

*Modifiers* can only be used with class methods, and can be one of:

* Abstract - method is abstract and has no statements block or End terminator. Any class with an abstract method is implicitly abstract.
* Virtual - method is virtual and can be dynamically overridden by a subclass method.
* Override - method is virtual and overrides a super class or interface method.
* Override Final - method is virtual, overrides a super class or interace method and cannot be overridden by subclasses.
* Final - method is non-virtual and cannot be overridden by a subclass method.

Methods are 'Final' by default.

#### Properties

To declare a read/write property:

Property *Identifier* : *Type* () ...getter code... Setter ( *Identifier* : *Type* ) ...setter code... End

To declare a read only property:

Property *Identifier* : *Type* () ...getter code... End

To declare a write only property:

Property ( *Identifier* : *Type* ) ...setter code... End

#### Conversion Operators

You can also add 'conversion operators' to classes and structs. These allow you to convert from a custom class or struct type to an unrelated type, such as another class or struct type, or a primitive type such as String.

The syntax for declaring a conversion operator is:

Operator To [ < GenericTypeIdents > ] : *Type* () ...Statements... End

Conversion operators cannot be used to convert a class type to a base class type, or from any type to bool.

For example, we can add a string conversion operator to a class like this:

Struct Vec2  
  
 Field x:Float  
 Field y:Float  
  
 Method New( x:Float,y:Float )  
 Self.x=x  
 Self.y=y  
 End  
  
 Method ToString:String()  
 Return "Vec2("+x+","+y+")"  
 End  
  
 ' The string conversion operator  
 Operator To:String()  
 Return "Vec2("+x+","+y+")"  
 End  
End

This will allow Vec2 values to be implictly converted to strings where possible, for example:

Local v:=New Vec2  
  
Print v

We no longer need to use '.ToString()' when printing the string. Since Print() takes a string argument, and Vec2 has a conversion operator that returns a string, the conversion operator is automatically called for you.

#### Extensions

Extensions allow you to add extra methods and functions to existing classes or structs. Fields cannot be added this way. Private members cannot be accessed by extensions.

Struct Foo  
 Field i:Int=0  
End

Struct Foo Extension  
 Method Increment()  
 i+=1  
 End  
End

#### Encapsulation

There are three Levels of encapsulation for class and struct members:

-Public members can be accessed from anywhere. It is the default encapsulation level.

-Protected members can only be accessed by the base class and the derived ones or by class/struct extensions. Code existing in the same source file have acces to Protected members too.

-Private members can only be accessed by the base class. Code existing in the same source file have acces to Private members too.

example:

Class Foo  
 'public by default'  
 Field i:Int  
  
 Protected  
  
 Field someProtectedThing:Int  
 Method doSomething()  
 Print "Doing something"  
 End  
  
 Private  
  
 Field \_somePrivateThing:String  
End

### Preprocessor

Monkey2 include a simple preprocessor that allows you to conditionally compile code depending on a number of build setttings.

The preprocessor supports the following statements: #If, #Else, #ElseIf, #EndIf, #Rem, #End. Preprocessor statements must begin on a new line.

Preprocessor expressions may only use the 'And', 'Or' and comparison operators.

The following symbols may be used in preprocessor expressions:

|  |  |  |
| --- | --- | --- |
| Symbol | Type | Meaning |
| \_\_TARGET\_\_ | String | The current build target. One of: "windows", "macos", "linux", "android", "ios", "emscripten" |
| \_\_CONFIG\_\_ | String | The current build config. One of: "release", "debug" |
| \_\_DESKTOP\_TARGET\_\_ | Bool | True if the current build target is windows, macos or linux. |
| \_\_MOBILE\_TARGET\_\_ | Bool | True if the current build target is android or ios. |
| \_\_WEB\_TARGET\_\_ | Bool | True if the current build target is emscripten. |
| \_\_DEBUG\_\_ | Bool | True if the current build config is debug. |
| \_\_RELEASE\_\_ | Bool | True if the current build config is release. |

For example, to include code in debug builds only, use something like:

#If \_\_DEBUG\_\_  
Print "This code is only included in debug builds."  
#Endif

To include code on desktop or mobile builds, use:

#If \_\_DESKTOP\_TARGET\_\_ Or \_\_MOBILE\_TARGET\_\_  
Print "This code is only include in desktop and mobile builds."  
#Endif

### Reflection

#### Typeof and TypeInfo

The Typeof operator return a TypeInfo object, that contains various properties and methods for inspecting types at runtime. There are 2 ways to use Typeof:

Local type:=Typeof( expression )  
Local type:=Typeof< type >

The use of seperate () and <> delimeters is to prevent the parser getting confused by complex expressions.

TypeInfo objects have a To:String operator (mainly for debugging) so can be printed directly:

Print Typeof<Int>  
Print Typeof<Int Ptr>  
Local t:=10  
Print Typeof( t )  
Print Typeof( "yes" )

Typeof returns the 'static' type of a class object. To get the actual instance type, use the Object.InstanceType property:

Class C  
End  
  
Class D Extends C  
End  
  
Function Main()  
 Local c:C=new D  
 Print Typeof( c ) 'Class default.C  
 Print c.InstanceType 'Class default.D  
End

You can retrieve the type of the value contained in a variant using the Variant.Type property:

Local v:=Variant( 10 ) 'creates a variant containing an int.  
Print v.Type 'prints 'Int'

TypeInfo also includes functions for inspecting all user defined types:

Function TypeInfo.GetType( name:String )

Returns the TypeInfo for a named type. A named type is a namespace or class declared by your app - it does not include primitive types, pointer types, array types etc. Class names must be prefixed by the namespace they are declared in.

To get an array of ALL named types:

Function TypeInfo.GetTypes:TypeInfo[]()

#### DeclInfo objects

TypeInfo objects for namespaces and classes also contain a set of DeclInfo objects. A DeclInfo represents the member declarations inside of a namespace or class. Currently, only global, field, method and function members are supported. DeclInfo objects also have a To:String operator to help with debugging.

You can inspect the member decls of a type using the TypeInfo.GetDecls method:

Namespace mynamespace  
  
Global test:Int  
  
Function Main()  
  
 Local type:=TypeInfo.GetType( "mynamespace.MyClass" )  
  
 For Local decl:=Eachin type.GetDecls()  
 Print decl  
 Next  
End

You can retrieve a single unique member using TypeInfo.GetDecl:

Local type:=TypeInfo.GetType( "mynamespace.MyClass" )  
  
Local ctor:=type.GetDecl( "New" )

There may be several decls with the same name due to method and function overloading, in which case the simple GetDecl above will fail and return null. In this case, you either need to inspect each decl individually to find the one you want, or you can pass an additional TypeInfo parameter to GetDecl:

Local type:=TypeInfo.GetType( "MyNamespace.MyClass" )  
  
Local ctor:=type.GetDecl( "New",Typeof<Void()> )

This will return the default constructor for MyClass, assuming there is one.

#### Getting and setting variables

Member decls that represent variables (ie: fields and globals) can be read and written using the DeclInfo.Get and Decl.Info.Set methods:

Namespace mynamespace  
  
Global MyGlobal:Int  
  
Function Main()  
  
 Local vdecl:=TypeInfo.GetType( "mynamespace" ).GetDecl( "MyGlobal" )  
   
 vdecl.Set( Null,10 )  
   
 Print MyGlobal  
   
 Print Cast<Int>( vdecl.Get( Null ) )  
End

The first parameter of Set and Get is an object instance, which must be non-null for getting and setting fields.

The second parameter of Set is a variant, and is the value to assign to the variable. The type of the value contained in the variant must match the variable type exactly, or a runtime error will occur.

Note that since any value can be cast to a variant, we can just provide the literal value '10' for Set and it will be implictly converted to a variant for us. On the other hand, we must explicitly cast the result of Get() from a variant back to the type of value we want.

#### Invoking methods and functions

To invoke methods and functions, use the DeclInfo.Invoke method:

namespace mynamespace  
  
Function Test( msg:String )  
  
 Print "Test! msg="+msg  
End  
  
Function Main()  
  
 Local fdecl:=TypeInfo.GetType( "mynamespace" ).GetDecl( "Test" )  
   
 fdecl.Invoke( Null,New Variant[]( "Hello Test!" ) )  
End

The first parameter of Invoke is an object instance, which must be non-null for invoking methods.

The second parameter of Invoke is an array of variants that represents the parameters for the call. The types of these parameters must match the parameter types of the actual method or function exactly, or a runtime error will occur.

#### Limitations

Currently, typeinfo is only generated for non-generic, non-extension, non-extern 100% pure monkey2 globals, fields, function, methods, classes and namespaces. You can still use other types (structs etc) with variants etc, but you wont be able to inspect their members.

Typeinfo may be stripped out by the linker. I've added a little hack to mojo to keep module typeinfo 'alive', but there is still work to do here. If you find the linker stripping out typeinfo, you can prevent it doing so for now by adding a 'Typeof' to Main() referencing the type you want to keep alive. Or, you can set MX2\_WHOLE\_ARCHIVE in bin/env\_blah.txt to '1' to force the linker to include ALL code, but this will of course produce larger executables. ### Error handling

A Try/Catch block is an error-handling construct that allows custom code to be executed in situations which may otherwise cause undesirable behaviour.

The Try/Catch block opens with Try and closes with End (or End Try). The code to be executed within must be followed by at least one Catch section.

In the event of an error occurring within the Try/Catch block, an exception object (based on the native Throwable class) should be 'thrown' via the Throw instruction.

If an exception occurs, program flow jumps to a Catch section declared explicitly for the given exception type. The exception object is 'caught' and the relevant error-handling code is executed.

You can declare multiple exception classes to handle different types of exception and should create a matching Catch section for each one.

After an exception is caught and handled, program flow exits the Try/Catch block and continues.

When a try block has multiple catch blocks and an exception is thrown, the first catch block capable of handling the exception is executed. If no suitable catch block can be found, the exception is passed to the next most recently executed try block, and so on.

If no catch block can be found to catch an exception, a runtime error occurs and the application is terminated.

The Try/Catch method of error-handling allows code to be written without the need to manually check for errors at each step, provided an exception has been set up to handle any errors that are likely to be encountered.

Syntax:

Try

*...code (sould contain at least one throw)...*

Catch exception

*...error handling code...*

End

Example code:

#Import "<std>"  
Using std..  
  
Class custoEx Extends Throwable  
 Field msg:String  
  
 Method New (mesag:String)  
 Self.msg = mesag  
 End  
End  
  
Function Main()  
 Local somethingWrong:=True  
 Try  
 If somethingWrong=True Then Throw New custoEx ("Custom Exception detected")  
 Catch err:custoEx  
 Print err.msg  
 End  
End

### Assets management

TO BE COMPLETED! This page might be completed with:

import syntax (single data file, wildcards, @ directive for imports, ...?)

differences between stream/stream files, file system files and an explanation on assets management on mobile devices (packed datas)

supported file formats for each target

### Integration with native code

In order to allow monkey2 code access to native code, monkey2 provides the 'extern' directive.

Extern begins an 'extern block' and must appear at file scope. Extern cannot be used inside a class or function. An extern block is ended by a plain 'public' or 'private' directive.

Declarations that appear inside an extern block describe the monkey2 interface to native code. Therefore, functions and methods that appear inside an extern block cannot have any implementation code, as they are already implemented natively.

Otherwise, declarations inside an extern block are very similar to normal monkey2 declarations, eg:

Extern  
  
Struct S  
 Field x:Int  
 Field y:Int  
   
 Method Update() 'note: no code here - it's already written.  
 Method Render() 'ditto...  
End  
  
Global Counter:Int  
  
Function DoSomething( x:int,y:Int )

#### Extern symbols

By default, monkey2 will use the name of an extern declaration as its 'symbol'. That is, when monkey2 code that refers to an extern declaration is compiled, it will use the name of the declaration directly in the generated output code.

You can modify this behaviour by providing an 'extern symbol' immediately after the declarations type, eg:

Extern  
  
Global Player:Actor="mylib::Player"  
  
Class Actor="mylib::Actor"  
 Method Update()  
 Method Render()  
 Function Clear()="mylib::Actor::Clear"  
End

#### Extern classes

Extern classes are assumed by default to be *real* monkey2 classes - that is, they must extend the native bbObject class.

However, you can override this by declaring an extern class that extends Void. Objects of such a class are said to be native objects and differ from normal monkey object in several ways:

* A native object is not memory managed in any way. It is up to you to 'delete' or otherwise destroy it.
* A native object has no runtime type information, so it cannot be downcast using the Cast<> operator.

### The build system

Monkey2 includes a simple build system that converts monkey2 files to C++, compiles the C++ code, and links the resultant object files.

The build system also allows you to import several types of non-monkey2 files into a project for compilation and/or linking. This is done using a system import directive:

#Import "<*system\_file*>"

...or or a local import directive:

#Import"*local\_file*"`

Import directives can appear any where in a monkey2 source file, but it's generally tidiest if they are placed at the top of the file.

#### System Imports

System files are files that are generally provided with the compiler toolset, and that the compiler and/or linker are configured to find automatically. Monkey2 recognizes the following system file types:

|  |  |
| --- | --- |
| System file type suffix | System file type |
| .o, .obj, .a, .lib | Static library. |
| .so, .dll, .dylib | Dynamic library. |
| .framework | MacOS framework. |
| .h, .hh, .hpp | C/C++/Objective C header. |
| .monkey2 | Monkey2 module. |

Note that system file names are enclosed by < and > characters, while local file names are not.

An example of importing a system library:

#Import "<wsock32.a>"

If no extension is provided for a system import, Monkey2 will assume you are importing a monkey2 module, eg:

#Import "<std>"

This will import the monkey2 'std' module. This is effectively the same as:

#Import "<std.monkey2>"

#### Local Imports

Local files are files that are located relative to the monkey2 file that imports them.

Monkey2 recognizes the following local file types:

|  |  |
| --- | --- |
| Local file type suffix | Local file type |
| .o, .obj | Object file. |
| .a, .lib | Static library. |
| .so, .dll, .dylib | Dynamic library. |
| .framework | MacOS framework. |
| .exe | Windows executable. |
| .c, .cc, .cxx, .cpp, .m, .mm | C/C++/Objective C source code. |
| .h, .hh, .hpp | C/C++/Objective C header. |
| .monkey2 | Monkey2 source code. |

It is also possible to add local 'include directories', 'library directories' and 'framework directories' with import. This is done using syntax similar to a local import, but replacing the filename with a wildcard.

For example, to add an include directory:

#Import "*include\_directory*/\*.h"

This will allow you to import any header file inside 'include\_directory' using...

#Import "<*include\_file*>"

...without having to specify the full path of the file.

To add a library directory:

#Import "*staticlib\_directory*/\*.a"

To add a MacOS framework directory:

#Import "*framework\_directory*"/\*.framework" ### Miscellaneous

#### Code lines splitting

Lines can currently only be split after ‘[‘, ‘(‘ or ‘,’ tokens.

#### $ Hexadecimal

Hexadecimal numbers can be entered using the $ symbol

Local i:=$A0F

#### Inline Code comments

Inline comments can be done with the ' character.

Print "hello!" 'this line prints hello on the output console

# Articles and Tutorials

### Operator Overloading

Operator overloading is a very cool feature that allows you to customize the behaviour of the built-in monkey2 operators for classes and structs.

You overload an operator by writing an ‘operator method’, which is effectively just a special kind of method. Operators must appear inside classes/structs – they cannot currently be ‘global’.

Here is a simple example:

Struct Vec2  
  
 Field x:Float  
 Field y:Float  
  
 Method New( x:Float,y:Float )  
 Self.x=x  
 Self.y=y  
 End  
  
 Method ToString:String()  
 Return "Vec2("+x+","+y+")"  
 End  
  
 'Overload the addition operator.  
 Operator+:Vec2( rhs:Vec2 )  
 Return New Vec2( x+rhs.x,y+rhs.y )  
 End  
  
End

The ‘Operator+’ declaration here defines an addition operator for Vec2. This is then used whenever a Vec2 appears as the ‘left hand side’ of an addition. For example:

Function Main()  
 Local v1:=New Vec2( 10.0,20.0 )  
 Local v2:=New Vec2( 30.0,40.0 )  
 Local v3:=v1+v2 'note: calls Operator+ in Vec2.  
 Print v3.ToString()  
End

The following unary operators can be overloaded: + – ~

The following binary operators can be overloaded: \* / Mod + – Shl Shr & | ~ = <> < > <= >= <=>

The following assignment operators can be overloaded: \*= /= Mod= += -= Shl= Shr= &= |= ~=

Indexing behaviour can also be overloaded using: [] []=

Note that you cannot override ‘Not’, ‘And’ or ‘Or’ - would just be too confusing if the meaning of these weren't consistent IMO!

Operators can return any type of value, and can take any type of value for their ‘right hand side’ argument(s). However, the precedence of operators cannot be changed.

The ‘[]’ and ‘[]=’ operators allow you to define ‘indexing’ like behaviour. The ‘[]’ operator is used when an object is indexed, and ‘[]=’ is used when an object is indexed and assigned. Both of these operators can accept any number of parameters of any type. The ‘[]=’ operator requires an additional parameter that is the value to be assigned. This must appear at the end of the parameter list.

Here is an example of some indexing operators for the Vec2 class above:

Struct Vec2  
  
 ...etc...  
  
 Operator[]:Float( index:Int )  
 Assert( index=0 Or index=1 )  
 If index=0 Return x Else Return y  
 End  
  
 Operator[]=( index:Int,value:Float )  
 Assert( index=0 Or index=1 )  
 If index=0 Then x=value Else y=value  
 End  
End

With these additions, you can access Vec2 coordinates ‘by index’, eg:

Function Main()  
 Local v:=New Vec2  
 v[0]=10.0  
 v[1]=20.0  
 Print v[0]  
 Print v[1]  
End

You can also overload assignment operators, for example:

Struct Vec2  
  
 ...etc...  
  
 Operator+=( v:Vec2 )  
 x+=v.x  
 y+=v.y  
 End  
End

If you have already written an Operator+ (as is the case here) this is not strictly necessary, as monkey2 will generate the code for Operator+= for you. However, you may still want to provide a custom version for Operator+= if your code can do so more efficiently.

Note that you cannot overload the plain assignment operator '='.

### What are 'lambda functions'?

A lambda function is a special type of function that can be declared in the middle of an expression.

You can think of a lambda function a bit like a ‘temporary’ function – instead of having to declare an entirely separate function to do what you need, you can just declare a lambda function ‘on the fly’ in the middle of an expression.

A lambda function is anonymous. It has no name so can only be used by the expression it is declared within.

A lambda functions can make use of the same local variables as the expression it is declared within. It does this by ‘capturing’ these variables, which means the lambda function receives a copy of the local variable’s value at the point the lambda function is declared. This means that a lambda function will not ‘see’ any future modifications to local variables. A lambda function cannot ‘see’ local variables that have not been declared yet!

If a lambda function is declared within a method, it can also see (and modify) object fields, and call object methods.

Here is a simple example:

Function Test( func:Void() )  
   
 func()  
  
End  
  
Function Main()  
  
 Test( Lambda()  
  
 Print "Hello from lambda!"  
  
 End )  
  
End

### Namespaces and using.

Monkey2 provides simple support for namespaces.

Each file can have a Namespace directive at the top that specifies the ‘scope’ of all the declarations (functions, globals, classes etc) in the file. For example:

'\*\*\*\*\* file1.monkey2 \*\*\*\*\*  
'  
Namespace myapp 'declare namesapce  
   
Global SomeGlobal:Int  
   
Function SomeFunction()  
End

The ‘namespace myapp’ at the top here means that the SomeGlobal and SomeFunction declarations end up in the ‘myapp’ namespace. If you don’t have a Namespace at the top of a source file, a ‘default’ namespace is used. It is recommended that you use Namespace for all substantial projects though.

To access stuff declared in a namespace, use the ‘.’ operator. For example, you can access the SomeGlobal variable above using myapp.SomeGlobal.

However, you don’t need to do this if the declaration being accessed is in the same namespace (or a ‘parent’ namespace…see below) as the code doing the accessing. For example, any code within the above file can use SomeGlobal and SomeFunction without the need for a ‘myapp.’ prefix, as that code is also in the myapp namespace.

This also applies to multifile projects. If 2 separate monkey2 files are in the same namespace, then they can freely access each other declarations without the need for a namespace prefix.

You can almost think of namespace as simple classes – albeit classes that can’t be new’d so can’t have fields or methods. The name of the class provides a ‘scope’ for the globals and functions declared in the class, and declarations with the class can directly access other declarations in the same class.

Namespaces are also hierarchical. While ‘Namespace myapp’ creates a simple ‘top level’ namespace, it’s also possible to create child namespaces using ‘.’. For example ‘Namespace myapp.utils’ referes to a ‘utils’ namespace within the top level ‘myapp’ namespace.

Finally, the Using directive can make it easier to access frequently used declarations inside a namespace. For example, the ChangeDir and CurrentDir functions are declared in the ‘std.filesystem’ namespace, but (depending on your self discipline level) it can be a hassle having to use std.filesystem.ChangeDir and std.filesystem.CurrentDir all the time.

To help out here, the ‘Using’ directive can be used to instruct the compiler to search a particular namespace for identifiers that it can’t normally find. For example:

Namespace myapp  
   
Using std.filesystem  
   
Function Main()  
 ChangeDir( ".." )  
 Print CurrentDir()  
End

Without the Using declaration in the above code, you would need to use std.filesystem.ChangeDir and std.filesystem.CurrentDir.

You can have multiple Usings in an app, and Usings must appear at the top of a file, before any declarations.

The namespace specified in a Using must be ‘absolute’. That is, the namespace of the file is not taken into account when resolving the Using namespace.

### Multifile projects and #Import.

To add additional source files to a monkey2 project, you use the #Import directive. #Import can also be used to import other stuff into a project, but more on that late

#Imports should appear at the top of a source file before any declarations occur. #Import takes one parameter – the path to the file to import. If the file is a '.monkey2' file, the extensions can be omitted, eg:

'file1.monkey2  
'  
#Import "file2"  
#Import "file3"  
  
Function Something()  
End

The import path can be relative or absolute, and contain “../” etc, making it easy to get at source files located anywhere.

When you build a monkey2 app (or module), the compiler starts with a single ‘root’ monkey2 source file and searches for all other monkey2 files reachable – directly or indirectly – from that root file via #Import directives. All files found via #Import this way will ultimately be included in the project and built by the compiler.

You only need to #Import a particular file once per project – duplicate #Imports of the same file are ignored by the compiler.

Code in any imported monkey2 file can use code in any other imported monkey2 file, regardless of whether or not the files #Import each other. For example:

'\*\*\*\*\* file1.monkey2 \*\*\*\*\*  
'  
#Import "file2"  
#Import "file3"  
  
Function Func1()  
 Func1()  
 Func2()  
 Func3()  
End  
  
'\*\*\*\*\* file2.monkey2 \*\*\*\*\*  
  
Function Func2()  
 Func1()  
 Func2()  
 Func3()  
End  
  
'\*\*\*\*\* file3.monkey2 \*\*\*\*\*  
  
Function Func3()  
 Func1()  
 Func2()  
 Func3()  
End

This is perfectly valid, as long as file1.monkey2 is the ‘root file’ you compile.

You may encapsulate some code within a file by using the Private keyword. That code will only be accessible within the file. The Public keyword allows you to go back to the default public privacy level.

'\*\*\*\*\* file1.monkey2 \*\*\*\*\*  
'  
#Import "file2"  
  
Function Func1()  
 Func1()  
 Func2()  
 Func3() 'this call is not valid, Func3 is private to file2.monkey!  
 Func4()  
End  
  
'\*\*\*\*\* file2.monkey2 \*\*\*\*\*  
  
Function Func2()  
 Func1()  
 Func3() 'here the call is valid  
End  
  
Private 'entering private declarations  
  
Function Func3()  
 'some statements  
End  
  
Public 'back to public declarations  
  
Function Func4()  
 'some statements  
End

### Monkey2 Target SDKs

Monkey2 target SDKs.

#### The Windows Desktop Target

Monkey2 can use either the mingw or msvc express 2015 compilers to build desktop apps.

To use mingw, you can use the mingw build tools available at [[http://monkeycoder.co.nz/monkey2-files]]. Simply download the mingw build tools package, run it (it's a self extracting exe), and select your monkey2 'devtools' dir for installation.

Note that the prebuilt binaries available from itch.io already include mingw in the 'devtools' dir.

To use msvc instead of mingw, you will need to install msvc express 2015 and change the following line in bin/env\_windows.txt:

#MX2\_USE\_MSVC=1

You will need to rebuild all modules after doing this.

Downloads for msvc express can be found here - https://www.visualstudio.com/vs/visual-studio-express/

#### The Macos Desktop Target

Monkey2 uses the command line tools included with xcode to build desktop apps for macos.

#### The Linux Desktop Target

Monkey2 uses the 'gcc' command line tools to build desktop apps for linux.

#### The Emscripten and Wasm Targets

Monkey2 uses the emscripten sdk compilers and tools to build emscripten and wasm apps.

To install the emscripten sdk, please see this page: [[https://github.com/juj/emsdk/blob/master/README.md]].

#### The Android Target

Monkey2 uses the android NDK (native development kit) to build android apps.

Setting up for Android development:

1. Install android studio and make sure it works, ie: you can build and run one of the simple template projects on a device or emulator. Android studio is available here: [[https://developer.android.com/studio/index.html]].
2. Install the 'NDK' (native development kit) using android studio via 'SDK Manager->SDK Tools'.
3. Install the Android 7.0 (Nougat) SDK Platform (API Level 24) using android studio via 'SDK Manager'.
4. Edit your monkey2 bin/env\_windows.txt file and change the ndk-bundle 'PATH' setting so it points to the NDK. Or, you can just add the ndk-bundle directory to your system PATH.
5. Fire up Ted2 and select 'Build->Rebuild Modules->Android'. Wait...

Building an Android app:

1. Build your app in Ted2 using 'Build->Build Only' with 'Build Target->Android' selected.
2. Open the generated android studio project (at myapp.products/Android) in android studio.

Note: I recommend *disabling* the following android studio setting for mx2 dev:

File->Settings->Build, Execution, Deployment->Instant Run->Enable Instant Run

With this enabled, android studio doesn't seem to notice when external project files change.

#### The iOS Target

Monkey2 uses the command line tools included with xcode to build ios apps.

# The mx2cc compiler

Mx2cc is the command line compiler for monkey2. The actual executable is named differently depending on the OS:

* Windows: /bin/mx2cc\_windows.exe
* MacOS: /bin/mx2cc\_macos
* Linux: /bin/mx2cc\_linux

The command line options for mx2cc are:

mx2cc *command* *options* *input*

Valid commands are:

* makeapp - make an app. *input* should be a monkey2 file path.
* makemods - make a set of modules. *input* should be a space separated list of module names, or nothing to make all modules.
* makedocs - make the documentation for a set of modules. *input* should be a space separated list of module names, or nothing to make all modules.

Valid options are:

* clean - rebuilds everything from scratch.
* verbose - provides more information while building.
* target=*target* - set target to desktop (the default) or windows, macos, linux, emscripten, wasm, android, ios. desktop is an alias for current host.
* config=*config* - set config to debug (the default) or release.
* apptype=*apptype* set apptype to gui (the default) or console.