Squirrel Language Reference Sheet

Variable Types

Squirrel variables are dynamically typed with no type declarations.

Integer	32-bit signed. Stored as a value.
Float	32-bit signed. Stored as a value.
Bool	Logical value: true or false. Stored as a value.
String	Not null terminated. May contain null bytes. Characters can be accessed by index. Precede literals with @ for a verbatim string. Immutable. Stored as a reference.
Array	One-dimensional, specified/indexed using square brackets. Mutable. May contain values of any type simultaneously. Stored as a reference.
Table	Collects key-value pairs (called 'slots'). Specified by braces. Keys are typically strings, but may be of any Squirrel data type. Values may be accessed with index notation, eg. *myVariable = myTable[true]*. Mutable. Stored as a reference.
Blob	Custom binary data. Accessed via read/write pointer. Mutable. Stored as a reference.

Assign Variables

Variables must be assigned as global or local to the main program or a function.

Create a Global Variable	Create a Local Variable
myVariable <- 42	*local myVariable = 42*
myVariable <- 23.75	*local myVariable = 23.75*
myVariable <- true	*local myVariable = true*
myVariable <- "I am an Electric Imp"	*local myVariable = "I am an Electric Imp"*
myVariable <- [0, 1, 2, 3, 4]	*local myVariable = [0, 1, 2, 3, 4]*
myVariable <- {"A":65 , "B":66 , "C":67}	*local myVariable = {"A":65 , "B":66 , "C":67}*
myVariable <- {A = 65, B = 66, C = 67}	*local myVariable = {A = 65, B = 66, C = 67}*

Once assigned, all variables can be re-assigned with the = operator. Within a class definition, global variable assignments require the scope operator, ::, be placed before the name.

Semi-colons

Semi-colons at the end of lines are optional unless the line of includes multiple statements.

Operators

Arithmetic	
+	Addition
–	Subtraction
/	Division
*	Multiplication
%	Modulo
Compact Arithmetic	
+=	Compact addition
–=	Compact subtraction
/=	Compact division
*=	Compact multiplication
%=	Compact modulo
Relational	
!	NOT
!=	Not equal
||	OR
&&	AND
==	Equal
<=	Less than or equal
>=	Greater than or equal
>	Greater than
<	Less than
Misc	
::	Scope resolution
<=>	Three-way compare
?:	Conditional
,	Combination
Bitwise	
&	AND
|	OR
^	Exclusive OR
~	NOT
>>	Bit-shift right
<<	Bit-shift left
>>>	Unsigned bit-shift right

Tables

Tables comprise key-value pairs called slots which can hold any variable type, function or class instance.

Create a Slot and Assign a Value	Re-assign a Value	Delete a Slot
local myTable = {}; *myTable.keyOne <- "I am an Electric Imp";*	*myTable.keyOne = false;*	*local value = delete myTable.keyOne;*
myTable["keyOne"] <- "I am an Electric Imp";	*myTable["keyOne"] = false;*	

Functions

Function parameters are implicitly local.
Functions can also be stored in variables: *local myFunction = function(param){. . . }*

Function, No Parameters	Function, Parameters	Function, Parameters with Default Values
function myFunction() *{. . . }*	*function myFunction(param1, param2)* *{. . . }*	*function myFunction(param1=42, param2="I am an Electric Imp")* *{. . . }*

Flow Control

foreach (itemVariable in collectionVariable)
{. . . }
Note Do not add or remove table keys with a *foreach* loop.

foreach (indexVariable, itemVariable in collectionVariable)
{. . . }
Note Do not add or remove table keys with a *foreach* loop.

for (local index = 0 ; index < maxValue ; index += increment)
{. . . }

do
{. . . }
while (CONDITION);

while (CONDITION)
{. . . }

Squirrel Language Reference Sheet

Define a Class

Class properties are unique to each class instance by default. The static keyword can be used to change this: a static property belongs to the class itself and so is shared by all instances. Properties can be initialized in a class declaration, but the initializer is evaluated only once and its value assigned to all instances. For reference-type properties, including arrays and tables, that means that all instances initially refer to the same object. To have an initializer re-evaluated afresh for each instance, initialize the property in a constructor function. All methods and properties are public; Squirrel does not support private class members.

Instantiate a Class

```
local myInstance = MyClass();
local myInstance = MyClass(constructorParameterOne, constructorParameterTwo);
myInstance <- MyClass();
```

Accessing Instance Properties

```
myInstance.propertyOne = 42;
myInstance.propertyThree = "I am an Electric Imp";
local myVariable = myInstance.propertyOne;
myVariable <- myInstance.propertyOne;
```

Calling Instance Methods

```
myInstance.methodOne();
myInstance.methodTwo(methodParameterOne, methodParameterTwo);
```

The Context Object

Squirrel passes a hidden parameter to all methods and other function calls which contains a reference to the calling context. Within the called method, this reference is accessed through the variable this. When registering methods as callbacks, it is often useful to provide the function – which, if called as a callback will be called out of context – with a suitable context object. This is done with the method *bindenv()* ('bind to environment'). This creates a closure combining the method and the content object passed to *bindenv()* as a parameter:

```
imp.wakeup(2.0, aFunction.bindenv(this));
```

Constants

Constants are denoted by the keyword const.

Comments

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

Conditional Structures

```
if (CONDITION_ONE) {
    . . .
} else if (CONDITION_TWO) {
    . . .
} else if (CONDITION_THREE) {
    . . .
} else {
    . . .
}
```

```
switch (myVariable) {
    case 0:
        . . .
        break;

    case 1:
        . . .
        break;

    default:
        . . .
}
```

A Class without a Constructor

```
class MyClass {
    // Scalar property (unique by default)
    propertyOne = 42;

    // Scalar property (shared by all instances)
    static propertyTwo = true;

    // Non-scalar property (shared by all instances but may be re-initialized by an instance)
    propertyThree = "I am an Electric Imp";

    // Non-scalar property (shared by all instances)
    static propertyFour = {"A":65 , "B":66 , "C":67};

    // Methods
    function methodOne() {
        . . .
    }

    function methodTwo(methodTwoParameterOne, methodTwoParameterTwo) {
        . . .
    }
}
```

A Class with a Constructor Function

```
class MyClass {
    // Scalar property (unique by default)
    propertyOne = 42;

    // Scalar property (shared by all instances)
    static propertyTwo = true;

    // Non-scalar properties (shared by all instances but may be re-initialized by an instance)
    propertyThree = "I am an Electric Imp";

    // Non-scalar property (shared by all instances)
    static propertyFour = [0, 1, 2, 3, 4];

    // Non-scalar properties initialized by Constructor are unique – must be declared null
    propertyFive = null;
    propertySix = null;

    // Constructor function
    constructor(constructorParameterOne, constructorParameterTwo) {
        propertyFive = constructorParameterOne;
        propertySix = constructorParameterTwo;
    }

    // Methods
    function methodOne() {
        . . .
    }

    function methodTwo(methodTwoParameterOne, methodTwoParameterTwo) {
        . . .
    }
}
```