# Standalone functions

Standalone functions always return a value and are therefore used in contexts that expect a value, such as in the right hand side of a variable declaration (var) or assignment (set) statement, either on their own or as part of a larger expression. Standalone functions usually require at least one argument to be passed in brackets – corresponding to the parameters defined for that function.

## Processing strings

### unicode

Takes an argument of type Int representing a Unicode value – which it is often convenient to specify in hex format – and returns the string character for that code. Very useful in the context of Graphics, for defining graphical symbols. Example of use:

var gr set to new Graphics  
var heart set to unicode(0x2665)  
set gr to gr.putChar(3, 4, heart)

### parseAsInt and parseAsFloat

Mechanisms for converting a string representation of a number into a numeric type. Both methods take a string as an argument, and return a 2-tuple, the first value of which is a Boolean, indicating whether or not the string was successfully parsed as the type required. The second value of the tuple provides the numeric value in the required type. You should not read the second value unless the first is true, because the second will default to zero. Note that the tuple may be ‘decomposed’ into two variables as shown below:

Example of use:

input a  
var (parsed, result) set to a  
if parsed  
 then  
 print result \* result  
end if

Note that many potential needs for these parse methods can be avoided by using Validated input methods.

### indexOf

### length

## Processing numbers

### div and mod

Both methods take two integer arguments and divide the first by the second. div returns the result rounded down to the nearest integer; mod returns the remainder (the ‘modulus’). Example of use:

var n set to inputInt("Number ? ")  
var d set to inputInt("Divisor ? ")  
print "div: {div(n, d)} mod: {mod(n, d)}"

### floor and ceiling

Both methods take a single Float argument, and return an Int. floor returns the nearest integer value below (or equal to) the argument, ceiling returns the nearest integer value above (or equal to) the argument. Example of use:

var x set to input  
print "floor: {floor(x)}, ceiling: {ceiling(x)}"

### round

rounds a Float value to the number of decimal places specified. Example of use:

print round(1/3, 4)

Notes:

* round is especially useful when writing tests that involve Float results, where you do not always want to specify *all* the digits of the expected result. For example:

A screenshot of a computer code

Description automatically generated

### range

returns an immutable list of Int values covering the range specified by the two (inclusive) bounds. Example of use:

print range(1, 10)

## Maths functions

### pi

returns the constant value 3.141592653589793

### abs(n as Float) return Float

returns the absolute value of n.

### acos(n as Float) return Float

returns the arccosine, in radians, of n

### acosDeg(n as Float) return Float

returns the arccosine, in degrees, of n.

### asin(n as Float) return Float

returns the arcsine, in radians, of n.

### asinDeg(n as Float) return Float

returns the arcsine, in degrees, of n.

### atan(n as Float) return Float

returns the arctangent, in radians, of n.

### atanDeg(n as Float) return Float

returns the arctangent, in degrees, of n.

### cos(n as Float) return Float

returns the cosine of n (in radians).

### cosDeg(n as Float) return Float

returns the cosine of n (in degrees).

### exp(n as Float) return Float

returns ex, where x is the argument, and e is Euler's number (2.718…)

### logE(n as Float) return Float

returns the natural logarithm of n.

### log10(n as Float) return Float

returns the base-10 logarithm of n.

### log2(n as Float) return Float

returns the base-2 logarithm of n.

### sin(n as Float) return Float

returns the sine of n (in radians)

### sinDeg(n as Float) return Float

returns the sine of n (in degrees)

### sqrt(n as Float) return Float

returns the positive square root of n.

### tan(n as Float) return Float

returns the tangent of n (in radians).

### tanDeg(n as Float) return Float

returns the tangent of n (in degress).

### degToRad(n as Float) return Float

converts degrees to radians.

### radToDeg(n as Float) return Float

converts radians to degrees.

A screenshot of a computer code

Description automatically generated

## Arrays

### createArray(n as Int, value as <ValueType>) return [<ValueType>)]

returns a new ArrayList with exactly n elements, all initialised to the value specified. The value must be of type: Int, Float, Boolean, or String.

This method mimics the creation of a traditional ‘array’ – which had to be initialised to a fixed size. While the returned result is still an ArrayList (and therefore may still be expanded dynamically) – the advantage of using this method is that it will have a defined number of elements, which may all be accessed by index straight away.

### create2DArray(n as Int, m as Int, value as <ValueType>) return [[<ValueType>)]]

returns a new ArrayList of ArrayLists that mimics a traditional ‘2D array’. The returned array list will contain n ArrayLists, each of which will have m members, all initialised to the value specified. The value must be of type: Int, Float, Boolean, or String.

Individual elements may be accessed via a ‘double-index’ e.g.:

var board set to create2DArray(8,8,"")

set board[0][3] to "Black King"

### indexOf

### length

## Higher order functions

### filter

### map

### reduce

### max

### maxBy

### min

### minBy

### any

# Standalone procedures

### print & printTab

These procedures may be called as an alternative to using the print statement. The differences are that the print or printTab *procedure*:

does not automatically add a ‘newline’ at the end, so you may subsequently print something else on the same line. If you wish to use the print procedure and include one or more newlines in specific places, just include \n (the standard form for a newline) within the string.

Require the data to be printed to be of type String. If you want to print a value of another type, you will either need to add .asString() to it, or put the value into braces within an ‘interpolated’ string.

For print, the data to be printed is the only argument. For example:

for I from 1 to 10 step 1  
 call print("{i}")  
end for

printTab helps in the layout of information printed to the console, in particular, the printing of columns of data. printTab works like the print procedure, but requires an additional argument specifying the tab position (number of characters from the left of the display). For example:

call printTab(0, "No.")  
call printTab(10, "Square")  
call printTab(20, "Cube\n")  
for x from 1 to 10 step 1  
 call printTab(0, x.asString())  
 call printTab(10, "{x^2}")  
 call printTab(20, "{x^3}\n")  
end for

### pause

Pauses the execution of the program for a specified number of milliseconds (minimum 1). For example, to pause for 1/10th of a second:

call pause(100)

There are two uses of pause:

* to control the speed of events – for example in a dynamic game
* to allow the Console and/or Graphics displays to refresh. See Console and Graphics. (For this purpose, pause(1) is sufficient to enable the display refresh and causes minimum delay in program execution).

### clearConsole

Equivalent to pressing the Clear button on the Console, but automatically at specific point(s) in the program execution:

call clearConsole()

# System methods

‘System methods’ refers to a set of specific methods provided by the Elan standard library, that depend on the system (outside of the Elan language) in some way. They *appear* to work like functions – in that they return a value – and may be used in the same way as a regular function, *but may be used only within* main *or a* procedure*.* This is because each system method has a dependency on something more than the arguments (if any) passed into it, and/or generates side effects. Thus, system methods are *not* ‘pure’ functions,

## Validated input methods

Input methods are an alternative to using the standard input *statement* (e.g. input a). Input methods provide some validation of the input type and, optionally, values. Each of these methods, also defines a prompt string, which will be printed immediately before, and on the same line, as the input cursor – and will be repeated if a given user-input is not valid.

### inputString inputStringWithLimits inputStringFromOptions inputInt inputIntBetween inputFloat inputFloatBetween

Examples of use:

var name set to inputString("Your name? ", 2, 50)

var name set to inputStringWithLimits("Your name? ", 2, 50)

var action set to inputStringFromOptions("Action ?", {"add", "remove", "exit")?

Note that the options must be specified as an *immutable* list i.e. within curly braces if specified as a literal list, as above.

var moveSquares set to inputInt("Move squares)

var age set to inputInt("Your age in years? ", 5, 21)

var payment set to inputFloatBetween("Payment value: ", 0.0, 99.99)

## Clock

The clock methods returns an integer representing the current time in milliseconds since ‘the epoch’ (midnight, January 1, 1970 UTC). This is useful for measuring elapsed time. For example:

var startTime set to clock()  
# Your code here  
print "Elapsed time in milliseconds {clock - startTime}"

## Random numbers

### random

### randomInt

# Dot methods

‘dot-methods’ are invoked on a variable or property of the appropriate type, for example, using ‘dot syntax).

## On String

substring

trim

indexOf

isBefore

isAfter

isBeforeOrSameAs

isAfterOrSameAs

## On ArrayList

add

insert

remove

removeFirst

removeAll

asImmutableList

An ArrayList can serve the role of an array (existing elements can be indexed), or a list (elements may be added). If you wish to use an array set up with an initial size (so that each element may be indexed without risk of an ‘out of range error’), use the initialiseAsArray dot-method, specifying the size required and the value with which to initialise each element – which must be compatible with the type of the ArrayList on which the method is called. Examples of use:

var a set to empty [Float]

call a.initialiseAsArray(100, 0)

set a[7] to 65.02

var board set to empty [[String]]

call board.initialiseAs2DArray(8, 8, "")

set board[3][4] to "W K"

Restrictions:

* Both methods may be used only on an ArrayList where the type of the elements is Int, Float, Boolean, String, or any type of enum. If you wish to initialise an array of any other type, you must write your own procedure to add the required number of elements to the empty ArrayList.

## On ImmutableList

get

getRange

with

withInsert

withRemove

withRemoveFirst

withRemoveAll

asArrayList

## On Dictionary

removeKey

keys

values

## On ImmutableDictionary

getKey

hasKey

withKey

withRemoveKey

## On Tuple

first

second

## On Graphics

**Functions**

### getChar

### putChar

### putText

### putForeground

### getForeground

### putBackground

### getBackground

### putAt

### getAt

**Procedures:**

### draw

### clearGraphics

### getKeystroke and getKeystrokeWithModifier

Example use where gr is an instance of type Graphics:

var k set to gr.getKeystroke()

* If the user has pressed a key that key will be returned as a String.
* If no key has been pressed, the method will return an empty string: "".
* If the key is a printable character, it will be returned as a single-character string, for example: "a", "X","3",":"
* Non-printable keys are returned as words, for example: "Space", "Backspace","Enter","ArrowDown"
* getKeystrokeWithModifier allows you to check whether the keystroke was ‘modified’ by, for example, the **Shift**, **Ctrl**, or **Alt** keys. The method returns a 2-tuple consisting of the key and the modifier (which may be an empty string). Learn how this works with this example:

### clearKeyBuffer

All keystrokes go into ‘queue’ called the keyboard ‘buffer’. If you are reading keys (see Reading keystrokes ) and you wish to prevent the user from adding keystrokes faster than they can be consumed, then:

call clearKeyBuffer()

## On instance of a class

### typeAndProperties

## On many different types

### asString

The asString method may be called on an instance of any type, to provide a string representation. When you use a print *statement* asString is called behind-the-scenes to generate a string that can be printed to the Console. However, if you are using the print & printTab methods, then you will commonly need to convert an instance into a string and the asString method is a convenient way to do this. This also applies if you are embedding a variable into a string, for example:

var a set to [3, 4, 5]  
print "The current list is {a.asString()}"

Note, however, that the specific format of the resulting string will vary considerable from type to type.

### length

The length method can be called:

* on a String, to determine the number of characters
* on an ArrayList or an ImmutableList, to determine the number of elements
* on a Dictionary, or an ImmutableDictionary to determine the number of key-value pairs

Example of use:

var s set to "Hello World!"  
print s.length()

### asIter

Many types of data structure are compatible with the type Iter (short for ‘iterable’) provided that their member-type is compatible. So for example, if a function requires an Iter<of Int> then you may pass in an immutable list of integers as an argumentWithin the realm of functional programming, the need *occasionally* arises desirable to be able to declare a and ensure that it is specifically of the type, say, Iter<of Int> rather than, say, {Int}. The following is an example of how to do that:

var myList set to { "apple", "orange", "pear"}  
var myIter set to myList.asIter()

# Reading & writing data files

Not yet implemented – but will be included in v1.0.0