# 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 uses of these parse methods can be avoided using validated input methods.

## 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

round

toPrecision ?? maybe get rid of this now we have round

range

## Maths functions

pi -returns the constant value 3.141592653589793

Each of the following functions takes a single argument of type Float

abs - returns the absolute value of the input.

acos - returns the arccosine of the input.

asin - returns the arcsine of the input.

atan - returns the arctangent of the input.

cos - returns the cosine of the input.

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

logE - returns the natural logarithm of the input.

log10 - returns the base-10 logarithm of the input.

log2 - returns the base-2 logarithm of the input.

sin - returns the sine of the input.

sqrt - returns the positive square root of the input.

tan - returns the tangent of the input.

Examples of the maths functions being used:

A screenshot of a computer code

Description automatically generated

## Higher order functions

filter

map

reduce

max

maxBy

min

minBy

any

contains

size

# 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

initialiseAsArray and initialise2DAsArray

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

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

asIter

## Count TODO difference between length and count ?

length

head

# Console

## See

print

printTab

clearConsole

## Input

# Graphics

About Graphics

See

clearGraphics

See also pause

initialisedGraphics

# Reading & writing data files

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