# Annotated example programs

(Annotate as many of the standard constructs as possible, using arrows).

# List of keywords & their meanings

as – see Function definition.

abstract – see Inheritance

case – see switch…case

catch – see try … catch

class – see Class definition

constant – see Constant definition

constructor – see Constructor

curry – see Functional programming

default – see switch…case

else – see if…then…else

end – used (followed by another keyword) at the end of several programming constructs

enumeration – see Enumeration

for – see for…loop and for…in loop

function – see Function definition

if – see if…then…else

immutable – see Immutable class

in – see for…in loop

inherits – see Inheritance

lambda – see Function definition.

let – see Functional programming

main – see Main routine

new – see Instantiation

partial – see Functional programming

private – see Private members

procedure – Procedure definition

property – see Property

repeat – see repeat…until loop

return – see Function definition

self – see Object-oriented programming

step - see for…loop

switch – see switch…case

then – see if…then…else

throw – see Exceptions

to – see for…loop

try – see try … catch

until – see repeat…until loop

var – see Variable definition

while – see while loop

with – see Instantiation

# Project structure

## Single file program

Smaller Elan projects typically consist of a single file of code. The file name may be any valid Windows file name but to be recognised by the Elan compiler, it must end with the file extension .elan – for example program.elan

The first line of *every* Elan file *must* be a Comment – though it *may* be an empty comment line (i.e. just the # symbol). If you type other code on the first line, the Elan IDE will automatically correct this by inserting a new empty comment line at the top of the file.

To be an executable program the file *must* contain a single Main routine.

It *may* additionally contain other Top-level constructs.

## Working with multiple files

For larger projects the code may be split into multiple files. The filenames may be chosen by the programmer – the file names have no significance to the compiler, provided each ends with the file extension .elan.

Each code file may contain any mixture of the Top-level constructs, with the constraint that:

* Only one file in the project may contain a Main routine.
* Each top-level construct must be complete within a single file (you may not, for example, split a single procedure definition across multiple files).

Code in any file may reference constructs defined in another file within the project – without the need for an import or equivalent statement.

## Comment

A comment begins with # (the 'hash' symbol). This may be placed at the start of the line, or after other code, for example:

# This is a single line comment

# For a multi-line comment,  
# each line must begin with #

var count = 0 # Comment may follow other code

# Top-level constructs

Top-level constructs are forms of code that can be written within a [file](#_Project_structure). All other code constructs must be written *inside* one of these top-level constructs. The complete list of top-level constructs is:

* Main routine
* Procedure definition
* Function definition
* Constant definition
* Class definition
* Enumeration definition

## Main routine

For a program to be executable there must be a main routine - and only one. This is the first code that is called when the program is run.

The main routine is defined by :

* the main keyword on its own line
* a Statement block
* end main on its own line

## Procedure definition

A procedure is one of the two forms of Subroutine.

A procedure is defined by:

* the procedure keyword followed by:
  + The procedure's *name*, which must follow the rules of any Identifier.
  + Round brackets, either empty or containing one or more comma-separated parameter definitions
* end procedure on its own line

## Function definition

A function is one of the two forms of Subroutine.

A function is defined by:

* the function keyword followed by:
  + The function's name, which must follow the rules of any Identifier.
  + Round brackets, either empty or containing one or more comma-separated parameter definitions
  + The keyword as followed by the Type of the value returned by the function
* a Statement block
* end function on its own line

Note that a function definition may alternatively be written using Expression syntax – the latter being more commonly adopted in the context of Functional programming. To differentiate the two, the form of function definition described above is sometimes referred to as a 'statement-bodied function'.

## Constant definition

A constant is a named value, where the value is fixed at compile time and may not be altered. As top-level constructs, all constants are 'global' - visible everywhere. (This in contrast to Variables, which are always scoped within the main routine or a subroutine.) A constant is defined by:

* The constant keyword followed by:
  + The name, which must follow the rules of any Identifier.
  + A literal value. This may be for any or the Standard value types or any of the Standard data structures *except for* Array (because an Array is a Mutable type*).*

Note that the Standard Library defines a small number of constants, such as pi and newline.

## Class definition

## Enumeration definition

An enumeration is a type where each instance must be one of a fixed set of named values. An enumeration is defined by:

* The enumeration keyword followed immediately by:
  + The name, which must follow the rules for any Type name.
* Starting on a new line, a list of comma-separated identifiers (each following the rules for any Identifier) representing the different allowed values.
* end enumeration on its own line.

For example:

enumeration PointsOfCompass

north, northeast, east, southeast, south, southwest, west, northwest

end enumeration

# Statement block

The statement block may include any of the following kinds of statement, with each such statement starting on a new line.

### Variable definition

A variable is a named value like a constant, but unlike a constant, it may be subsequently re-assigned to different value *of the same type.*

var message = "hello"

var x = 1 + 1

* A variable may only be defined within a Routine, and its scope is limited to that routine.
* The name defined for the variable must follow the rules for any Identifier.
* The variable *must* be initialised with a value, which may be specified as a *literal*, or calculated from an Expression.
* Elan uses Static typing. The type of a variable is determined by the type of the initial value assigned to the variable.

### Variable assignment

* An assignment puts a new value into an *existing* variable, element in an array, or property of an object.
* The new value may be defined as a literal, or by an Expression.
* However, any new value assigned must be of the same type as the variable, i.e. of the same type as the value with which the variable was initialised.
* The new value may be defined by a literal or by an expression – the latter may make reference to other variables or constants in scope and/or to the *current* value of the variable.

Examples: cover all cases mentioned in first bullet

var x = 3 # definition  
x = 4 # assignment  
x = x + 1 # assigns to x a value one larger than its current value

## Variable scoping

## Control-flow statement

Elan supports six forms of control-flow statement, each of which runs over more than one line of code, and may incorporate one or more Statement blocks. The control flow statement determines either how many times a single block is executed, or which of more than one block is executed depending upon the value held by one or more variables.

### if…then…else

This is the most common form of 'selection' in procedural programming

### switch…case

### for…loop

### for…in loop

### while loop

### repeat…until loop

### try … catch

## Procedure call

* Because a procedure does not return a value, it *cannot be called within an expression*.
* A *freestanding* procedure call consists just of the name of the procedure always followed by round brackets. If the procedure defines parameters, then the call must include an argument (each defined as a literal value or an expression) corresponding to each required parameter. Arguments are placed within the round brackets and – if there is more than one – separated by commas. Examples:
* A proceduremay also be called using Dot-syntax. Examples:

## System call

* A system call is

## Exception

# Expressions

An expression is a single line of code (though it may be wrapped over more than one line on screen or paper) made up of multiple elements that, when evaluated, produces a value.

An expression may incorporate any or all of the following elements:

* Literal value
* Reference to a variable or constant
* Arithmetic operators and brackets
* Function call

Examples:

* 3 + 4
* 3 \* x ^ 2 – 2 \* x + k
* max(a, b) / min(c, d)
* firstName.ToUpper()+" " + lastName.ToUpper()

An expression may not contain procedure call or any system call

# List System Calls

## Input/Output

## Other System Calls

# Standard types

## Standard value types

## Standard data structures

### String

### List

### Array

### Tuple

### Dictionary

# Enumeration

# Object-oriented programming

## Class definition

### Constructor

### Property

### Method

### Private members

## Instantiation

## Inheritance

## Immutable class

# Functional programming

## Expression syntax

# Standard Library

(Includes functions, constants and some types)

# Definitions of terms

## Identifier

The term identifier in Elan refers to any of the following:

* Variable name
* Constant name
* Procedure name
* Parameter name
* Function name

All Elan identifiers must commence with a lower-case letter, which may be followed by any combination of lower-case or upper-case letters, numeric digits, and the underscore symbol \_. All identifiers visible within a given Scope must be unique.

Identifiers must not be identical to any Keyword– even if using different case. However, a keyword may be incorporated within a larger identifier.

Any identifier defined, or subsequently referenced, using the incorrect case will be auto-corrected by the Elan IDE.

## Type

The word *type* refers to any of the Standard types provided with the Elan language, or to any user-defined type, for which there will be a Class definition.

## Type name

Type names in Elan always start with an upper-case letter, followed by any combination of lower-case or upper-case letters, numeric digits, and the underscore symbol. This applies both to Standard types that come with the language and Classes. A recognised type name entered in the wrong case will be auto-corrected by the Elan IDE.

Examples:

## Scope

Scope refers to the extent of the code within which something may be referenced Subroutines are defined at top-level, so their scope is said to be *global*. A constant may be defined at top-level and hence have a *global* scope, or defined within a Routine, in which case its scope is limited to that Routine. A parameter or variable is always defined within a Routine and scoped to that routine – Elan does *not* permit 'global variables'.

Within a scope, each defined identifier must be distinct from all others in that scope. However, the same identifier may be defined in different scopes to refer to different things (for example the identifier x may be used as variable name within two different routines without creating any conflict or confusion).

## Routine

*Routine* refers to a Main routine, or Subroutine.

## Subroutine

*Subroutine* refers to either a Procedure or a Function.

## Dot-syntax

Any function, and any procedure that defines

## Instance

## Class

## Method

A method is a procedure or function that is defined on a class.

A method is *always* called using Dot-syntax, where the dot follows a reference to an object (the reference might be a variable holding that instance, or it might be an expression that returns an object instance).

A method has direct access to the Members of the instance on which it is called.

## Member

Refers to a Property or a Method defined on a class.

## Parameter

One or more parameters form part of every Function definition, and optionally as part of a Procedure definition.

## Delimiters

The delimiters of any code construct in Elan are the first and last lines of the code construct, typically these both involve keywords and sometimes other code elements. Many pairs of delimiters take the form of a keyword at the start, and end followed by the same keyword at the end, for example: if … end if, or procedure … end procedure.

## Signature

The signature of a subroutine in Elan consists of the name of the subroutine, the parameters (if any) that it defines, and – if the subroutine is a function, the type that it returns. All this information is contained within the first line of the subroutine definition. The signature defines all that you need to know in order to call that subroutine from within other code, and if that subroutine has been written by another part – for example those in the Standard Library– the signature may be the only information about the subroutine available to the programmer. Strictly speaking Main routine technically has a signature, but it is always the same, and is never called by other code, so it is not usually of interest.

## Body

The body of a subroutine in Elan consists of all the code between the Signature and the ending delimiter. A general principle in programming is that of 'separating the interface from the implementation'. In the case of a subroutine, the Signature defines the interface and the body defines the implementation.

## Keyword

[complete list]

## Static typing

## Parameter

A parameter defines a value to be passed into a Procedure or Function and for use within it.

## Argument

Programmers sometimes use the terms *parameter* and *argument* interchangeably, the two words really represent two different perspectives on the same thing.

When a Function, or Procedure that defines parameters, is called, the calling code must supply arguments: values to be passed in to the a Function, or Procedure with the call. When the code inside the a Function, or Procedure is executed, those values will have been associated with the parameter names defined in the a Function, or Procedure's signature.

## Mutable

A mutable type is one where the value(s) held by an instance of that type may be altered. Of the standard types, only Array is mutable. The others are all *immutable* – meaning that the value held by an instance may not be changed. Functions or procedures that *appear* to change an immutable type, for instance adding to a list, or converting a string to upper case, in fact return a new instance of that type, based on the original one but with the specified differences.

# Code templates

red fields *must* be replaced by user code and may not be empty. Field name disappears as soon as you type anything, but reappears if the field is emptied.

amber code is optional. The field may be deleted, or will disappear once the cursor leaves the construct, but will reappear if the user cursors or clicks onto the location where it *was*.

All other code, including keywords, punctuation and any spaces, newlines, or tabs outside of a modifiable field, are *unmodifiable*.

Templates may be inserted by right-clicking in a suitable place within code selecting the template name, OR by starting to type the code itself – as soon as the starting code for a template is valid and unambiguous.

The user may navigate to next field with cursor key or tab. However, the user may also just continue to type and provided that the text matches the template code (allowing for variations in spaces) then the cursor will move through the template. In this way a user may always just type, or paste-in, correct code.

Can move between fields by tabbing, or by typing the defined (unmodifiable) code.

When you are editing a field (mandatory or optional), the code must conform to the grammar rule for that field.

# Templates for top-level constructs

## Constant

constant <name> **=** <value>

## Main

main  
 <statements>  
end main

Statement templates recognised or offered in within the <statements> field are any of those listed under Templates for statements.

## Comment

#

Note: hash is followed by space (not editable)

## Procedure definition

procedure <name>(<parameter definitions>)  
 <statements>  
end procedure

Statement templates recognised or offered in within the <statements> field are any of those listed under Templates for statements.

## Function definition (statement bodied)

function <name>(<parameter definitions>) as <return type>  
 <statements>  
 return <value or expression>  
end function

Statement templates recognised or offered in within the <statements> field are any of those listed under Templates for statements, except Procedure (or system call). This rule also applies to any statement blocks within sub-constructs when inside a function.

function <name>(<parameter definitions>) as <return type> **->** <expression>

Statement templates recognised or offered in within the <statements> field are any of those listed under Templates for statements, except Procedure (or system call). This rule also applies to any statement blocks within sub-constructs when inside a function.

## Function definition (expression syntax)

function <name>(<parameter definitions>) as <return type> **->** <expression>

Statement templates recognised or offered in within the <statements> field are any of those listed under Templates for statements, except Procedure (or system call). This rule also applies to any statement blocks within sub-constructs when inside a function.

## Enumeration

enumeration <name>  
 <comma-separated list of identifiers>  
end enumeration

## Class

<abstract><immutable> class <name> <inherits …>  
 <constructor, properties, methods (functions or procedures)>  
end class

Statement templates recognised or offered in within the <statements> are: Property, Constructor, Procedure, Function, and Comment

# Templates for statements

Note that Comment is also a valid template in the context of a statement block.

## Variable definition

var <variable name> **=** <value or expression>

Note: variable name may here be:

* A tuple decomposition
* A list decomposition

## Variable assignment

<variable name> **=** <value or expression>

Note: variable name may here be:

* Indexed
* Be a property qualified by dot syntax e.g. cust.address.town

## If … then statement

if <condition> then  
 <statements, else if clause, else clause>  
end if

## Switch … case statement

switch <variable>  
 case <value>  
 <statements>  
 <further case clauses>  
 <default clause>>  
end switch

## For loop

for <variable name> = <value> to <value> <step <value>>   
 <statements>  
end for

## For…In loop

for <variable name> in <list, array, string or other iterable>  
 <statements>  
end for

## While loop

while <condition>  
 <statements>  
end while

## Repeat loop

repeat   
 <statements>  
until <condition>

## Try…catch

try   
 <statements>  
catch <variable name> <Exception or specific sub-class of Exception)  
 <statements>  
end try

## Procedure (or system call)

<variable.><name>(<arguments>)

## Throw exception

throw new <Exception, or specific type of Exception>**(**<message>**)**

# Lower level templates

## Property

<private> property <name> as <Type>

Available only within Class template.

## Constructor

constructor**(**<parameter definitions>**)**  
 <statements>  
end constructor

Available only within Class template.

## Case

case <value>  
 <statements>

Available only within Switch…case statement template.

## Default

default  
 <statements>

Available only within Switch…case statement template.

## Else

else  
 <statements>

Available only within If … then statement template.

## Else if

else if <condition>  
 <statements>

Available only within If … then statement template.

## Parameter definition

<name> <Type> <, <parameter definition>

Available only within Procedure definition, Function definition, and Constructor templates.

## Call

foo(<firstName (String value or expression)>, <>, <>)

Available only within context of a statement, or an expression, triggered by recognition of the call. The arguments correspond to the parameters defined by foo.

By default the argument fields are collapsed to angle brackets as shown, but the name and type requirements are shown when the cursor is on them.

## Quotes, and brackets

When you type a single or double quotation mark, or any of the opening bracket marks, the closing mark is added. The pair of delimiters and contents between them are treated as a construct. May be selected and/or moved, converted to another form, and delimiters may be removed or a further one added.

Single quotes may contain only a single character.

The context for the quotes template is a value or expression, similarly for brackets.

# Editing templated code

If the user cursors or clicks to any unmodifiable code within the template, the whole template is highlighted (including all editable fields).

If the user (by mouse or keyboard) selects code that spans (wholly or partially) across more than one construct, then *the whole* of each touched construct is selected.

Right-clicking or hitting ? then pops up a menu for changes, including:

Delete whole <construct name>

Delete <construct name> copying the statements to clipboard

Move (Then move cursor, right click and select *Move selected code here*).

Copy to clipboard

Change to <construct name> (e.g. changing between loop types)

Other specifics.