**DKL**

**[Devkron Language]**

**Metalenguaje dinámico declarativo con sintaxis variable.**

Versión 1.0

**Aviso legal**

El contenido de este documento describe y hace referencia a una tecnología propiedad de Induxsoft®.

Esta documentación se proporciona “tal cual”, sin garantías de ningún tipo, expresa o implícita, incluyendo, pero no limitando a garantías de comercialización, idoneidad para un propósito particular y no infracción. En ningún caso el propietario [Induxsoft®] o los autores del mismo serán responsables de ninguna reclamación, daños u otras responsabilidades, ya sean en un litigio, agravio o de otro modo, que surja de o en conexión con el uso que se dé a la información que aquí se proporciona.

**Copyright ® Induxsoft ® México 2019-2021**

Induxsoft® es una marca registrada y concesionada a Induxsoft Data Services S de RL de CV

Los demás nombres y marcas comerciales a los que se podría hacer referencia en este documento son propiedad de sus respectivos titulares.

Contenido

[Capítulo I – Introducción 5](#_Toc77847461)

[Primero pasos 5](#_Toc77847462)

[Ejecución de programas 5](#_Toc77847463)

[Archivo dkli.dkh 6](#_Toc77847464)

[Inclusión de bibliotecas de funciones 6](#_Toc77847465)

[Generar XML 7](#_Toc77847466)

[Símbolos primitivos 7](#_Toc77847467)

[Identificadores 7](#_Toc77847468)

[Cadenas 7](#_Toc77847469)

[Agrupadores 8](#_Toc77847470)

[Comentarios 8](#_Toc77847471)

[Elementos y atributos 8](#_Toc77847472)

[Elemento 8](#_Toc77847473)

[Atributos 8](#_Toc77847474)

[Bloques de texto 9](#_Toc77847475)

[Meta atributos 10](#_Toc77847476)

[Directivas 10](#_Toc77847477)

[Inclusión de archivos de código 10](#_Toc77847478)

[Definición de meta-atributos 11](#_Toc77847479)

[Definición de plantillas de elementos 11](#_Toc77847480)

[Definición de patrones de sintaxis 13](#_Toc77847481)

[Contexto declarativo e imperativo 13](#_Toc77847482)

[Capítulo II – Lenguaje imperativo 15](#_Toc77847483)

[Variables 15](#_Toc77847484)

[Tipos de datos 15](#_Toc77847485)

[Declaración de variables 16](#_Toc77847486)

[Atributos de variables 16](#_Toc77847487)

[Plantillas de variables 17](#_Toc77847488)

[Ámbito de las variables 17](#_Toc77847489)

[Expresiones 17](#_Toc77847490)

[Constantes numéricas 18](#_Toc77847491)

[Constantes de cadena 18](#_Toc77847492)

[Llamadas a funciones 18](#_Toc77847493)

[Operadores 18](#_Toc77847494)

[Aritméticos 18](#_Toc77847495)

[Lógicos 18](#_Toc77847496)

[Prioridad de operadores 19](#_Toc77847497)

[Concatenación de cadenas 19](#_Toc77847498)

[Sentencias 19](#_Toc77847499)

[Control de flujo 19](#_Toc77847500)

[Ciclos 23](#_Toc77847501)

[Inclusión 24](#_Toc77847502)

[Asignación 24](#_Toc77847503)

[Estructuras (o registros) de datos 25](#_Toc77847504)

[Excepciones 28](#_Toc77847505)

[Organización y estructura del programa 28](#_Toc77847506)

[Ámbitos 28](#_Toc77847507)

[Módulos 29](#_Toc77847508)

[Funciones definidas por el programador 31](#_Toc77847509)

[Temas avanzados 32](#_Toc77847510)

[Referencias a funciones 32](#_Toc77847511)

[Funciones abiertas 32](#_Toc77847512)

[Capítulo III – Funciones incorporadas 33](#_Toc77847513)

[Interoperabilidad .Net 33](#_Toc77847514)

[Depuración 35](#_Toc77847515)

[Manejo de errores 36](#_Toc77847516)

[Sistema 36](#_Toc77847517)

[Consola 38](#_Toc77847518)

[Matemáticas 39](#_Toc77847519)

[Lógicas 40](#_Toc77847520)

[Conversiones 41](#_Toc77847521)

[Fecha y hora 41](#_Toc77847522)

# Capítulo I – Introducción

El lenguaje Devkron, o simplemente DKL [Devkron Language] es declarativo; completamente traducible a XML; pero también imperativo y procedural.

DKL es un lenguaje interpretado cuya sintaxis se asemeja mucho al lenguaje C, por lo que el código es familiar para prácticamente cualquier programador, haciendo su aprendizaje y utilización más sencilla y natural.

Además, incluye directivas de auto-definición de sintaxis que facilitan la codificación y elevan la productividad del desarrollador al permitirle ir adaptando el lenguaje a sus necesidades y preferencias.

DKL es multi-propósito y generalmente se emplea para construir sitios y servicios Web, programar procesos por lotes, generar documentos XML a partir de bases de datos relacionales, etc.

## Primero pasos

### Ejecución de programas

devkron.exe es el ejecutable por línea de comando del intérprete del lenguaje.

Suponga que tiene el siguiente código en c:\mis programas\hola.dkl

#include "dkli.dkh"

#!

program "hola"

{

    do print ("Hola mundo")

}

Para ejecutarlo deberá escribir:

devkron.exe “c:\mis programas\hola.dkl”

Hágalo y vea el resultado

#### Parámetros de línea de comando de devkron.exe

Todos son opcionales, el único requerido es el primer argumento que debe ser la ruta y nombre de un programa DKL.

devkron.exe archivo.dkl [parámetros]

archivo.dkl es la ruta y nombre de archivo del programa a ejecutar.

|  |  |
| --- | --- |
| -id:valor | Establece un identificador para la ejecución, si no se indica, Devkron genera un GUID automáticamente |
| -stepbystep | Realiza una ejecución paso a paso del código, puede ser útil para depurar |
| -debugfile:archivo | Indica el archivo sobre el que únicamente se aplicará la ejecución paso a paso, si no se establece, se aplicará a todos. Este parámetro trabaja junto con –stepbystep |
| -parse | Indica al intérprete que omita la ejecución imperativa proporcionando únicamente la salida XML del archivo DKL |
| -output:archivo | Indica el archivo de salida para el resultado del árbol XML producido por el programa DKL |
| -hide-log | Oculta la información de ejecución de la salida de consola |
| -hide-critical | Oculta los errores de la salida de consola |
| -hide-output | Oculta la salida por consola del árbol XML resultante de la ejecución del programa DKL |
| -ninja | Oculta toda salida por consola |
| -encoding:codificacion | Establece la codificación de la salida, toma alguno de los siguientes valores: default, utf8,utf8bom, utf7, utf32, Unicode, ascii, not-set. Si no se indica, se asume la el valor ‘default’ que toma la codificación configurada para la consola y utf8bom para la salida de archivo. |

Ejemplo:

devkron.exe “c:\mis programas\programa.dkl” “–output:c:\mis salida\programa.xml” –ninja

Guarda la salida XML del programa.dkl en el archivo programa.xml sin mostrar ningún dato por consola.

### Archivo dkli.dkh

El archivo dkli.dkh contiene los patrones de sintaxis para el contexto imperativo, por eso es necesario que se incluya siempre al principio de los programas que harán uso de este contexto.

### Inclusión de bibliotecas de funciones

Existen varias bibliotecas de funciones que pueden usarse en programas imperativos de DKL.

Las inclusiones de bibliotecas deben realizarse dentro del nodo raíz del árbol, como en el ejemplo siguiente:

#include "dkli.dkh"

#!

program "ejemplo"

{

    #include "functions.dkh"

    #include "serialize.dkh"

Por convención, se usa la extensión dkh para los archivos de las bibliotecas de funciones aunque son código dkl también.

### Generar XML

Como ya se ha mencionado, DKL puede generar XML, pruebe el siguiente código también y observe los resultados, más adelante comprenderá mejor la razón.

(Guarde el código en un archivo .dkl y ejecútelo a través de devkron.exe)

Personas(cantidad="5")

{

    Persona(nombre="Juan")

    Persona(nombre="Pedro")

    Persona(nombre="Camila")

}

## Símbolos primitivos

### Identificadores

Letras (a...z | A...Z), números (0...9), el carácter @, el \_ (guion bajo) y el. punto.

Deben iniciar con una letra, el carácter @ o el carácter \_, pueden contener puntos a partir del segundo carácter.

Los siguientes son ejemplos de identificadores válidos:

Identificador12

\_Identificdor12

@identificador1.1\_1a

@\_identi.ficad.o1r

### Cadenas

Las cadenas deben estar encerradas entre comillas simples o dobles, además se admiten las siguientes secuencias de escape (al estilo del lenguaje C):

|  |  |
| --- | --- |
| Secuencia de escape | Carácter |
| \\ | \ |
| \' | ' |
| \" | " |
| \r | Retorno de carro [13] |
| \n | Avance de línea [10] |
| \t | Tabulador [9] |

Ejemplos de cadenas válidas:

"Algún 'texto' 1"

'Algún texto\r\n"entre comillado dentro de comillas simples"'

### Agrupadores

Permiten delimitar un bloque o una parte de las declaraciones

* ( ) Declaración de atributos
* { } Cuerpo de la declaración

### Comentarios

Se utiliza la misma sintaxis que en el lenguaje C.

//Comentario hasta el final de la línea

/\* Texto de comentario que puede

incluir varias líneas \*/

## Elementos y atributos

Para describir la sintaxis de esta sección se utiliza la siguiente convención:

**opción1 | opción2** El carácter pipe | indica que se usa la opción1 o la opción2, pero obligatoriamente alguna de ellas.

[**Sintaxis opcional]** Las sintaxis indicadas entre corchetes [] indica que puede incluirse o no (es absolutamente opcional).

### Elemento

***cadena*** | ***identificador*** [ **(*atributos*)]** [ **{declaraciones *o sentencias***}]

### Atributos

Lista clave y valor separado por un espacio en blanco en la siguiente forma:

***cadena*** | ***identificador*** **=** ***cadena*** | ***Identificador***

Los elementos vacíos (que no tienen atributos ni elementos internos) deben declararse con las dos llaves {}.

Los elementos que tienen atributos, pero no contienen otros elementos no necesitan las dos llaves.

Ejemplos:

|  |  |
| --- | --- |
| DKL | XML |
| v { } //Elemento vacío | <v/> |
| E (a1="valor1" a2="valor2") | <E a1 = "valor1" a2 = "valor2" /> |
| P { h1(a1="valor1") } | <P >  <h1 a1 = "valor1" />  </P> |
| v | INCORRECTO |
| v() | INCORRECTO |
| v() { h1(a1="valor1") } | INCORRECTO |

En DKL se pueden usar identificadores sin declarar o cadenas como nombres de elementos o atributos. La regla es que, si el identificador no ha sido declarado previamente con una sentencia #def, se traducirá tal cual en XML.

|  |  |
| --- | --- |
| DKL | XML |
| "prefijo:Persona" ( nombre="Juan" "prefijo:apellido"="López" ) | <prefijo:Persona nombre = "Juan" prefijo:apellido = "López" /> |
| Mascota ("nombre"="Firulais" "prefijo:Propietario"="Juan López") | <Mascota nombre = "Firulais" prefijo:Propietario = "Juan López" /> |
| Personas(cantidad="5")  {  Persona(nombre="Juan")  Persona(nombre="Pedro")  Persona(nombre="Camila")  } | <Personas cantidad = "5" >  <Persona nombre = "Juan" />  <Persona nombre = "Pedro" />  <Persona nombre = "Camila" />  </Personas> |

### Bloques de texto

Un elemento puede contener además de otros elementos, texto o bloques CDATA.

Puede incluir texto entre comillas dentro del cuerpo de los elementos, en cuyo caso al convertirse a XML se realizará el “escapado” de caracteres.

Por ejemplo:

Datos { "Algún texto > que otro texto" }

Produce

<Datos> Algún texto &gt; que otro texto </Datos>

Recuerde que puede usar las secuencias de escape de C (\n\r,...) en las cadenas y estas se traducirán en la salida XML según corresponda.

En algunos casos puede que necesite evitar el escapado de caracteres de XML, por lo que tiene dos alternativas: declara el texto como bloque CDATA o bien explícitamente evitar el escapado.

El carácter! al inicio de una cadena indica que se produzca un CDATA

Datos {!"Algún texto > que otro texto"}

Produce

<Datos> <![CDATA[Algún texto > que otro texto]]> </Datos>

Por su parte, el carácter $ precediendo a la cadena coloca el texto sin escapado.

Datos {$"Algún texto > que otro texto" }

Produce

<Datos> Algún texto > que otro texto</Datos>

La omisión del escapado XML puede resultar útil si está produciendo HTML, por ejemplo.

### Meta atributos

Los meta-atributos son la única información que no se puede mapear a o desde XML.

Se utilizan para asociar información a un nodo (elemento) en particular que sea usada por los motores en tiempo de ejecución de Devkron.

Meta-atributos predefinidos

|  |  |
| --- | --- |
| dkl/perform | Indica que el elemento debe ser interpretado imperativamente |
| dkl/hide-element | Indica al generador de la salida XML que el nodo y todos sus hijos deben omitirse (ocultarse) |
| dkli/default-root-type | Nombre predeterminado del nodo raíz en caso que tras la ejecución imperativa no se pueda determinar un solo nodo raíz. |
| meta-xml-declaration | Establece una cadena de texto que se inserta justo antes de la declaración del nodo raíz. |

## Directivas

### Inclusión de archivos de código

#### #include

#include "archivo"

#include "archivo" build

#include "archivo" with "plantilla"

#include UTF8 "archivo"

#include UTF8 "archivo" build

#include UTF8 "archivo" with "plantilla"

Archivo es la ubicación local o remota de un archivo que va a incorporarse en la posición donde se encuentre la sentencia.

Debido a que el parámetro archivo de la sentencia es una cadena y se usa la sintaxis de C, deberá usar doble \\ para indicar una \.

Ejemplos:

#include "c:\\archivo.dkl" //Correcto

#include "c:\archivo.dkl" //Incorrecto

#include "archivo.dkl" /\*Correcto, el archivo se busca en la misma ubicación que el programa que lo requiere o en las ubicaciones predeterminadas del entorno.\*/

#include "https://dev.induxsoft.net/archivo.dkl" //Correcto

### Definición de meta-atributos

#### #set

# set "meta-atributo" "valor"

Asignación de meta atributos al elemento (nodo) inmediato siguiente a la sentencia.

# set "AtributoAsociado" "valor"

E(a="5") { }

El elemento E tiene asociado un meta-atributo denominado AtributoAsociado.

### Definición de plantillas de elementos

#### #def

#def identificador [(atributos por orden)] "valor" [(atributos predeterminados)

Asignación de cadena a identificador

Es la forma más simple de uso de la palabra #def y funciona asignando un valor de cadena que es reemplazado durante la formación del XML. (Casi como una variable o macro)

DKL

# def J "Juan"

# def P "Pedro"

# def C "Camila"

# def n "nombre"

Personas(cantidad="5")

{

Persona(n=J)

Persona(n=P)

Persona(n=C)

}

XML

<Personas cantidad = "5" >

<Persona nombre = "Juan" />

<Persona nombre = "Pedro" />

<Persona nombre = "Camila" />

</Personas>

Combinando el uso de #def e #include

archivo1.dkl

# def J "Juan"

# def P "Pedro"

# def C "Camila"

# def n "nombre"

archivo2.dkl

#include "archivo1.dkl"

Personas(cantidad="5")

{

Persona(n=J)

Persona(n=P)

Persona(n=C)

}

XML

<Personas cantidad = "5" >

<Persona nombre = "Juan" />

<Persona nombre = "Pedro" />

<Persona nombre = "Camila" />

</Personas>

Si la sentencia #def sólo relaciona una cadena con un identificador puede usarse indistintamente como nombre de elemento, atributo y valor.

DKL

# def n "nombre"

n(n=n)

XML

<nombre nombre = "nombre" />

Atributos por orden. Es una lista de identificadores o cadenas separadas por comas que simplifican la asignación de valores de atributos en una declaración de elemento.

#### #redef

La sentencia #redef cambia la cadena asociada a una declaración #def previa.

El efecto de #redef es al código que le sigue únicamente.

### Definición de patrones de sintaxis

#### #pattern

#pattern Identificador cadena\_de\_patrón

Identificador es cualquiera definido previamente por una sentencia #def.

cadena\_de\_patron es una cadena que contiene los símbolos válidos para esa sintaxis en particular.

#def color (nombre) "prefijoXml:Color"

#pattern color "COLOR '$nombre'"

Permitiría el siguiente código:

COLOR azul

Que sería formado como:

<prefijoXml:Color nombre="azul"/>

Tabla de prefijos para símbolos en patrones:

|  |  |
| --- | --- |
| $ | Identificador (los admitidos en el lenguaje C) |
| & | Expresión (Expresión válida según sintaxis de C) |
| % | Macro o uso del definidor (reemplaza el nombre por lo que corresponda en una sentencia #def previa) |
| @ | Cadena (Cadena válida de C [Entre comillas dobles o simples]) |
| # | Número (Sintaxis de C para valores numéricos) |
| \* | Mezcla (Número, cadena o Identificador) |

(Si no se especifica un prefijo, se asume que es una expresión)

En la sintaxis de un patrón no se permiten las siguientes secuencias:

* Expresión Expresión
* Identificador Expresión
* PalabraReservada Expresión

### Contexto declarativo e imperativo

El contexto se refiere al modo como son procesados los elementos del árbol del programa DKL y son dos:

1. Contexto declarativo. Significa que los elementos son transformados a su equivalente XML tal como han sido expresados.
2. Contexto imperativo. Significa que los elementos serán considerados sentencias sujetas de interpretación al estilo de cualquier lenguaje imperativo, es decir se realizarán asignaciones de variables, llamadas a funciones, ciclos, etc.

Las directivas de selección de contexto sirven para establecerlos en el código del programa.

#### Conmutación de contexto

La directiva:

##

Conmuta entre contextos, si la secuencia de ejecución está en modo declarativo cambia a imperativo y viceversa.

#### Contexto imperativo

La directiva:

#!

Pasa a modo imperativo el contexto de ejecución para las líneas siguientes.

#### Contexto declarativo

La directiva:

#$

Pasa a modo declarativo el contexto de ejecución para las líneas siguientes.

#### Inyección de expresiones

En ocasiones requerirá que el valor o nombre de elementos o atributos, así como partes del texto contenido dentro de elementos, se genere dinámicamente por el programa.

Para hacerlo, use la siguiente sintaxis:

#<expresión : formato> Inserción de expresiones

Por ejemplo:

a(href="#<url>"){"Hipervínculo"} // coloca el valor de una variable denominada url en la posición de la marca.

Dentro de las marcas de inyección pueden escribirse expresiones completas.

Adicionalmente, el carácter : se utiliza para que al resultado de la expresión se le proporcione un formato específico (solo aplica cuando el resultado es numérico).

nodo {" El valor monetario es:#<monto \* 1.16 : $ #,#.00>"}

Genera un nodo con formato.

La marca de inyección únicamente puede usarse en:

* El nombre o el valor de atributos
* El nombre de elementos
* Bloques de texto dentro de elementos

# Capítulo II – Lenguaje imperativo

## Variables

### Tipos de datos

#### Tipos valor

DKL es un lenguaje de tipado estático, es decir que si una variable se declara de un tipo no podrá cambiarlo durante toda la vida del programa.

Formalmente existen únicamente dos tipos valor: numéricos y cadenas.

Internamente los valores numéricos se almacenan en tipos System.Decimal de Net Framework o Net Core.

Los resultados de evaluaciones lógicas se consideran numéricos también, siendo 1 para resultados verdaderos y 0 para resultados falsos.

Los tipos de cadena internamente se almacenan como System.String de Net Framework o Net Core.

#### Tipos referencia

Las referencias son el mecanismo para acceder a objetos, internamente son el índice de una tabla de objetos que persiste durante todo el tiempo de ejecución del programa.

DKL utiliza el método de conteo de referencias para administrar la memoria asignada.

Las variables de referencia deben declararse con la palabra reservada ref.

Una variable de referencia es una variable numérica marcada como tal, por lo tanto, es posible asignar el contenido de variables de referencia a variables numéricas y viceversa sin que ocurra un error de tipos en tiempo de ejecución, sin embargo esto no debe hacerse bajo ninguna circunstancia.

Es importante considerar que, si no existe ninguna referencia apuntando a un objeto en la tabla de objetos interna, se eliminará de la memoria.

Siempre que utilice objetos, use variables de referencia.

Ejemplo:

ref objeto=\_new(“System.StringBuilder”) // Instancia y aloja la referencia a un StringBuilder de Net Framework o Net core.

### Declaración de variables

Las variables numéricas y de cadena para declararse únicamente deben asignarse por primera vez.

Ejemplos:

a=1 //declara una variable numérica con valor inicial

b="hola mundo" // declara una variable de cadena con valor inicial

Adicionalmente, las variables pueden tener atributos de validación que permiten controlar y prevenir los datos que almacenan.

### Atributos de variables

Para variables numéricas puede establecer los atributos: min, max indican los valores de Frontera de la variable numérica, round la cantidad de decimales a la que se redondea automáticamente y format la presentación con que se muestra como cadena, se usan los mismos caracteres de control de C#.

Para variables de cadena puede establecer los atributos: min y max (que sirven para controlar la cantidad de caracteres que puede contener la variable únicamente).

Ejemplos

a=1 //Declaración de variable numérica con todos los atributos.

{

set min:0

set max:100

set round: 2

set format "$ #,#.00"

}

La declaración anterior indica que la variable no puede almacenar valores por debajo de cero ni por encima de 100 y que todos serán redondeados a dos decimales, adicionalmente en el contexto declarativo se insertarán con el formato de dinero.

b="hola mundo" //Declaración de variable de cadena con atributos

{

set min: 1

set max:40

}

La declaración anterior define que la variable no puede contener cadenas con menos de 1 carácter ni con más de 40 de extensión.

### Plantillas de variables

Los atributos de una variable pueden heredarse a otra en su declaración.

b=1

{

set min: 100

}

a[b] //Declara una variable a que hereda de b un valor inicial de 1 y un mínimo de 100.

a[b] =50 // Igual que la declaración anterior, salvo porque el valor inicial es establecido a 50

a[b] //Hereda los atributos de b y adiciona un máximo de 200

{

set max:200

}

### Ámbito de las variables

Durante la ejecución de un programa Devkron las variables pueden ser globales o locales.

Las variables globales están disponibles en todos los ámbitos, mientras que las locales únicamente están disponibles dentro del ámbito que las declaró.

Las variables globales se declaran igual que las locales, la única diferencia es que el nombre de las variables globales siempre debe iniciar con el carácter @.

Ejemplos:

@variable\_global=0

variable\_local=0

## Expresiones

Las expresiones en DKL siguen la convención:

Término Operador Término

En donde Término es una constante numérica, una constante de cadena, una variable, una llamada a función o una expresión encerrada entre paréntesis.

Por su parte, el operador puede ser aritmético o lógico según se explica más adelante.

A+5 es una expresión válida

a<b || c>d también es una expresión válida.

### Constantes numéricas

Inician con un número, un signo menos (negativo) o un punto, los caracteres subsecuentes pueden ser solo números, el punto decimal (carácter punto) es flotante.

* 10
* 10.58
* 0.58
* .69
* -5
* -.25
* -0.58

### Constantes de cadena

Están delimitadas por comillas dobles y pueden contener cualquier carácter imprimible (no se admiten secuencias de control o escape).

* “Hola mundo”
* “Esto es una cadena de caracteres”

### Llamadas a funciones

Siguen las mismas reglas para las variables, pero deben ser seguidas de una pareja de paréntesis que encierre los parámetros existan o no. Si hubiese parámetros, deberán delimitarse con una coma

* + F()
  + Funcion(10,25.5,”hola”)

Los identificadores (nombres) de funciones no distinguen entre mayúsculas y minúsculas.

funcion1(10) es la misma función que FunCION(10).

## Operadores

### Aritméticos

* + Suma
* – Resta
* \* Multiplicación
* / División

### Lógicos

* < Menor qué
* > Mayor qué
* <= Menor o igual qué
* >= Mayor o igual qué
* == Igual
* != Desigualdad

### Prioridad de operadores

Las operaciones aritmético lógicas se realizan según el siguiente orden en una expresión:

1. Operaciones lógicas (<>!=<=…)
2. Multiplicaciones (\*) y divisiones (/)
3. Sumas (+) y restas (-)

Para modificar esta prioridad deben utilizarse agrupadores, es decir parejas de paréntesis ().

**Ejemplo**

4+5\*3 es igual a 19

(4+5)\*3 es igual 27

### Concatenación de cadenas

La concatenación de cadenas se realiza mediante el operador +, por lo que la expresión:

“hola ”+”mundo” da como resultado la cadena “holamundo”

## Sentencias

Las sentencias son por línea y no requieren marcador de terminación, en otras palabras: No se utiliza el ; ni ningún otro carácter de delimitación.

Las sentencias son sensibles a mayúsculas y minúsculas.

Todas las palabras reservadas de DKL como sentencias están en minúsculas.

### Control de flujo

Permiten la ejecución condicional de secciones de código.

#### if -else if – else

La sentencia if de Devkron es similar a la del lenguaje C, con la excepción de que no requiere paréntesis para delimitar la expresión a evaluar.

if Expresión

{

//Código a ejecutar si Expresión es verdadera

}

else if Expresión2

{

//Código a ejecutar si Expresión es falsa, pero Expresión2 verdadera

}

else

{

//Código a ejecutar si ninguna expresión es verdadera

}

#### switch – case

Permite ejecutar secciones de código con base en valores que coinciden con una expresión principal, su estructura es la siguiente:

switch Expresión

{

case ExpresiónOp1

{

//Este código se ejecuta si el resultado de ExpresiónOp1==Expresión

}

case ExpresiónOp2

{

//Este código se ejecuta si el resultado de ExpresiónOp2==Expresión

return //Esta sentencia obliga a salir del bloque switch completo

}

default

{

/\*Este código se ejecuta si Expresión no coincide con ExpresiónOp1 ni con ExpresiónOp2\*/

return

}

/\*Este código se ejecuta si Expresión==ExpresiónOp1 o a cualquier otro case que no contenga una sentencia break en su bloque \*/

}

#### return

La sentencia return en el ámbito principal de ejecución termina el programa.

La sentencia return en el ámbito del cuerpo de una función definida por el programador retorna el valor indicado (si se ha especificado) a la expresión que la ha llamado.

funcion1::

{

return 10

}

funcion2::

{

return

}

program

{

a=funcion1() //La variable a será asignada con el valor 10

return //Termina el programa

}

Si se ejecuta return dentro del cuerpo de una función sin especificar el valor de retorno, la función devolverá cero (numérico).

#### break

Aborta un ciclo o la ejecución de un bloque switch

#### do y call

Permiten llamar a una función sin asignar su valor a ninguna variable.

do y call son equivalentes (de hecho call es un ‘alias’ de do).

ejemplo:

do funcion1(“hola”) //Llama a la función indicada

En DKL no se puede llamar a las funciones más que dentro de expresiones, sentencias de asignación o mediante sentencias do/call.

### Ciclos

#### while

Itera el bloque de código asociado mientras la Expresión sea verdadera.

while Expresión

{

//Código

}

#### repeat

Repite el bloque de código asociado las veces que la expresión indique

veces=5

repeat veces+2

{

//Este código se repita 7 veces

}

#### for

Realiza la iteración de un bloque de código asociado conforme se haya especificado, su sintaxis es muy similar a la del lenguaje C, pero no igual.

Su estructura es la siguiente:

for asignación\_inicial; condición; incremento

asignación\_inicial y condición son requeridos, el incremento es opcional y en caso de no indicarse se asumirá de uno en uno.

for a=1; a<10; a=a+2

{

//Este código se ejecutará 4 veces, siendo los valores de a=1, 3, 7 y 9

}

### Inclusión

#### include

Inserta el código de un archivo DKL indicado en la posición de la sentencia include si la ejecución pasa por ella.

A diferencia de la directiva #include; que siempre realiza la inclusión del código; la sentencia include solo lo hace si es ejecutada.

Ejemplo:

#include “archivo1.dkl”

if expresión

{

#include “arhivoX.dkl”

include “archivo2.dkl”

}

archivo1.dkl y archivoX.dkl siempre serán insertados en donde se encuentran sus respectivas directivas, no obstante archivo2.dkl solo se incluirá si expresión es verdadera.

### Asignación

#### Asignación de variables de tipo valor

La asignación de variables de tipo valor (numéricas o de cadena) se realiza con la siguiente sintaxis:

variable = valor

El carácter = es el operador de asignación, no debe confundirse con el operador de igualdad ==

#### Asignación de texto a variables

Para clarificar mejor la asignación de grandes cantidades de texto a variables, puede usarse la sentencia text con la siguiente sintaxis:

text variable

{

“texto a asignar a la variable”

}

#### Asignación de variables de referencia

Las variables de referencia siempre deben asignarse con la sentencia ref

La sintaxis es:

ref variable=expresión\_que\_devuelve\_una\_referencia

ref variable: expresión\_que\_devuelve\_una\_referencia

El operador = en una sentencia ref liberará la variable a asignar si ya estaba asignada y creará una nueva entrada en la tabla interna del intérprete de referencias de objetos, los recursos asociados al objeto serán limpiados automáticamente si no hay otra variable que contenga la misma referencia.

El operador : en la sentencia ref forzará la reutilización de los mismos recursos de la tabla de objetos de DKL para alojar ahí la nueva referencia, lo que ocasiona que si otras variables apuntan a la misma referencia también se verán afectadas.

Si la variable no estaba siendo usada con anterioridad, es indistinto el uso de = o : en ref.

### Estructuras (o registros) de datos

Una estructura o registro de DKL es un tipo referencia que apunta hacia un objeto de datos complejo.

La sentencia new crea un registro y lo referencia a una variable.

(Internamente instancia la clase System.Collections.Generic.Dictionary<string,object> insensible a mayúsculas o minúsculas, por lo que cualquier objeto de .Net puede ser tratado como registro si implementa la interfaz System.Collections.Generic.IDictionary<string,object>.)

Por su parte, la sentencia using crea si no existe o reutiliza uno previamente creado.

new r

{

@”campo1”:”datos del campo1”

}

using r

{

@”campo2”:”datos del campo2”

}

El bloque de código anterior crea y luego complementa una estructura referenciada por la variable r que expresada en JSON sería así:

{ “campo1”:”datos del campo1”, “campo2”:”datos del campo2” }

Miembros (campos) de las estructuras (o registros)

Los miembros siempre se indican empezando con el carácter @, seguido de una cadena entre comillas (dobles o simples) que corresponde al nombre.

Los miembros que contienen datos de tipo valor (números o cadenas) se asignan con la sintaxis:

@”nombre del campo” : “valor de cadena”

@”nombre del campo” : 9651 //Valor numérico

@”nombre del campo” : 8+f() // Expresión

En el caso de que los miembros hagan referencia a un objeto, se utiliza el carácter \* al final del nombre como indicador.

@”miembro referencia de objeto\*”: variable\_referencia

#### Miembros que son estructuras

Una campo puede contener a su vez una estructura completa, declaración que se realiza con las sentencias new member o member, que crean o bien crean o re-usan una estructura miembro.

new r

{

new member @”r.1”

{

@”campo\_r.1.1” : ”valor”

}

}

using r

{

member @”r.1”

{

@”campo\_r.1.1” : ”valor11”

@”campo\_r.1.2” : ”valor12”

}

new member @”r.2”

{

@”campo\_r.2.1” : ”valor”

}

}

Esto sería representado en JSON como:

{

"r.1":{

"campo\_r.1.1":"valor11",

"campo\_r.1.2":"valor12"

},

"r.2":{

"campo\_r.2.1":"valor"

}

}

#### Asignación de campos de registros a variables y viceversa

Asignar valores a un campo de un registro (o estructura), al igual que obtener su valor y asignarlo a una variable, se realiza con la siguiente sintaxis:

using cliente

{

@”nombre”: “Juan”

member @”conyuge”

{

@”nombre”:”Lupita”

}

}

n : cliente<”nombre”> // n es asignada con el valor “juan”

nc : cliente<” conyuge/nombre”> // nc es asignada con el valor “lupita”

cliente<”nombre”>:”pedro” //Ahora el miembro nombre de cliente contiene “pedro”

cliente<”conyuge/nombre”>:”sandra” //Ahora el miembro nombre del miembro conyuge de cliente contiene “sandra”

Esta sintaxis de asignación permite que la expresión de ruta de los miembros sea formada dinámicamente, por ejemplo:

campo=”conyuge/nombre”

cliente<campo>:”maria ” //Asigna “maria” al campo nombre del miembro conyuge de cliente

Puede asignar variables de referencia a campos usando el carácter \* al final de la ruta del miembro:

ref objeto=algun\_objeto

registro<”campo\*”>:objeto

Así también, puede obtener una referencia desde un registro con la sintaxis precediendo la ruta del campo con el carácter &:

objeto2:registro<”&campo\_referencia”>

Observe que aunque se está asignando una variable de tipo referencia, no se requiere la sentencia ref en este caso.

### Excepciones

La sentencia exception permite establecer un bloque de código al cuál se conducirá la ejecución del programa en caso de ocurrir una condición de excepción (error).

El bloque exception que recibe la ejecución es siempre el más cercano a donde se produce el error.

exception

{

//Código a ejecutar en caso de error

}

## Organización y estructura del programa

### Ámbitos

Todos los programas en DKL tienen 3 ámbitos:

* Global. Las variables declaradas como globales están disponibles en cualquier ámbito
* Principal. El ámbito principal del programa consiste en el código fuera de las funciones definidas por el programador.
* De función. El correspondiente al bloque de código de una función definida por el programador en particular.

#include “dkli.dkh”

#!

program

{

funcion1::

{

//Ámbito de función

a=3 //Variable local al ámbito de la función

}

//Ámbito principal

@a=1 //Variable global

a =2 //Variable local al ámbito principal

}

### Módulos

Todo archivo DKL es un árbol y debe tener un por tanto un elemento raíz.

Si se escribe un programa imperativo deberá incluirse el cuerpo de código dentro de un elemento program o module.

Los elementos program y module no tienen más propósito que servir de nodo raíz y contenedor del ámbito principal y son totalmente equivalentes.

program “mi programa”

{

//Código aquí

}

es equivalente a:

module “mi programa”

{

//Código aquí

}

También es válido:

module “modulo raíz”

{

funcion1::

{

//Este es el ámbito de una función

}

//Ámbito principal

module “modulo interno”

{

//También aquí es el ámbito principal

}

}

La cadena de texto que sique a la declaración de module o program es opcional.

Cuando se realizan inclusiones (#include o include) el código se inserta en la ubicación de la sentencia o directiva.

El código de un bloque module o program pertenece al ámbito en que se encuentra, DKL no tiene ámbitos aislados para estos tipos de bloques.

### Funciones definidas por el programador

La declaración de funciones se realiza de la siguiente manera:

nombre\_funcion :: arg1, arg2, ...argn

{

...

}

Ejemplos:

sumaAyB :: a, b

{

return a+b

}

funcion\_sin\_argumentos::

{ }

Las funciones definidas por el programador tienen su propio ámbito cada una, por lo que las variables locales declaradas existen únicamente durante la ejecución de la función a la que pertenecen.

No se permite declarar funciones dentro de funciones.

Todas las funciones devuelven un valor, si no se especifica con la sentencia return, la función devuelve cero.

El número de parámetros definidos para la función es fijo.

Los parámetros que deben ser tratados como tipos referencia, deben ser indicados con el carácter & precediendo al nombre:

funcion1:: parámetro\_valor, &parametro\_referencia

{

}

Las funciones deben ser declaradas antes del código que las llame, por lo que el siguiente código producirá un error de “función no encontrada”:

a=f1()+5

f1::

{

return 3

}

El código debería ser como sigue:

f1::

{

return 3

}

a=f1()+5

## Temas avanzados

### Referencias a funciones

DKL ofrece la sentencia point para asignar a una variable la referencia a una función definida por el programador.

point variable to funcion

donde

variable obtendrá una referencia a la función indicada en función.

La llamada a una función se realiza con la sentencia that que tiene la siguiente sintaxis

that vf

Llama a la función referenciada por la variable vf

that vf(argumentos)

Llama a la función referenciada por la variable vf con los argumentos indicados

that vf(argumentos)->vr

Llama a la función referenciada por la variable vf con los argumentos indicados y asigna el valor de retorno a la variable vr

### Funciones abiertas

Una función abierta en Devkron es aquella que permite la ejecución de un bloque de código externo en una parte específica de su código interno.

#include "dkli.dkh"

#!

module "función abierta"

{

    iterar::max

    {

        expose index

        for index=0;index<max

        {

            that

        }

        return "Fin"

    }

    go iterar (index:i) with(10)->r

    {

        do print(str(i))

    }

    do print(r)

}

La sentencia that en el cuerpo de una función será reemplazada por el código de un bloque go que la llama.

Una función abierta puede exponer variables internas (locales) como de solo lectura al bloque insertado por go enumerándolas con la sentencia expose.

En el ejemplo, la función iterar, expone la variable interna index.

En el bloque go que hace la llamada, el código insertado hace uso de index con el nombre i

La sentencia go por su parte, la siguiente sintaxis:

go funcion [(variable\_expuesta:alias,...) [with (parámetros)] [->variable\_resultado]

Entre corchetes las partes que son opcionales.

El resultado devuelto por la función, en el ejemplo se asigna a la variable r

# Capítulo III – Funciones incorporadas

Las funciones incorporadas en el lenguaje no requieren que se incluya ninguna biblioteca porque están programadas en el núcleo del intérprete devkron.exe.

A diferencia de las funciones definidas por el programador o las que se definen en bibliotecas, las funciones incorporadas son las únicas que tienen en algunos casos, parámetros variables.

## Interoperabilidad .Net

Permiten extender las características del lenguaje mediante la interoperabilidad con la plataforma .Net (.Net Framework o .Net Core)

|  |  |
| --- | --- |
| \_import\_library  Carga un ensamblado y devuelve una referencia que apunta hacia él y que puede ser usada por otras funciones de interoperabilidad.  \_import\_library(nombre\_ensamblado) | nombre\_ensamblado – Cadena con el nombre cualificado o de archivo de un ensamblado .Net |
| \_new  Devuelve una referencia a una instancia de un objeto de una clase de .Net  \_new(tipo)  \_new(ensamblado, tipo)  \_new(tipo, argumentos)  \_new(ensamblado, tipo, argumentos) | tipo – Cadena con el nombre completo de una clase de .Net  ensamblado – Referencia a un ensamblado .Net cargado a través de la función \_import\_library. Si se omite, se asume que la clase pertenece a un ensamblado cargado intrínsecamente con el intérprete.  argumentos – Es una lista de parámetros que serán pasados al constructor de la clase. |
| \_map\_function  Mapea un nombre de función disponible en Devkron hacia un método estático de una clase en un ensamblado .Net  \_map\_function(ensamblado, tipo,método, función, argumentos, retorno) | ensamblado – Referencia a un ensamblado .Net cargado a través de la función \_import\_library. Si es -1 se asume que el ensamblado está intrínsecamente cargado por el intérprete.  tipo – Cadena con el nombre completo de una clase de .Net que pertenece al ensamblado indicado.  método – Cadena con el nombre del método estático de la clase indicada por tipo.  función – Cadena con el nombre que tendrá la función mapeada al método.  argumentos – Cadena que indica los tipos de .Net que corresponden a la firma del método, deben delimitarse con comas.  retorno – Cadena que indica el tipo del resultado, se admiten los siguientes: void– El método no devuelve nada, $-El método devuelve una cadena, #-El método devuelve un valor numérico o \*-El método devuelve un objeto y se tratará como referencia. |
| \_set  Establece el valor de una propiedad de un objeto.  \_set(referencia, propiedad, valor) | referencia – Una referencia al objeto al que pertenece la propiedad  propiedad – Una cadena con el nombre de la propiedad a establecer. Si termina con el carácter \* se asume que el parámetro valor es una referencia a otro objeto.  valor-Una cadena, número o referencia que será asignada a la propiedad. |
| \_get  Obtiene el valor de una propiedad de un objeto.  \_get(referencia, propiedad) | referencia – Una referencia al objeto al que pertenece la propiedad  propiedad – Una cadena con el nombre de la propiedad a establecer. El nombre de la propiedad debe estar precedido por alguno de los siguientes caracteres de especificación de tipo: $ - El retorno se asume una cadena, #- El retorno se asume un número o & - El retorno se asume una referencia a un objeto. |
| \_invoke  Invoca a un método de instancia que haya sido mapeado por \_map\_function y devuelve lo que corresponda.  \_invoke(referencia, función\_mapeada, argumentos...) | referencia – Es una referencia al objeto (instancia de clase) al que se invocará el método.  función\_mapeada – Es una cadena con el nombre de la función que previamente se mapeo con \_map\_function.  argumentos – Es una lista de argumentos que coinciden con los de la función mapeada. |
| \_gettype  Devuelve una referencia a un objeto System.Type que corresponde al tipo indicado.  \_gettype(tipo)  \_gettype(ensamblado, tipo) | ensamblado – Referencia a un ensamblado cargado con \_load\_library, si se omite se asume un ensamblado cargado intrínsecamente por el intérprete.  tipo – Cadena con el nombre completo del tipo (clase) del cual se desea obtener una referencia a su tipo. |

## Depuración

Permiten conocer el estado interno del intérprete en tiempo de ejecución

|  |  |
| --- | --- |
| dump\_vars  Devuelve una cadena con la lista de variables locales inicializadas.  dump\_vars() | Sin argumentos |
| dump\_gvars  Devuelve una cadena con la lista de variables globales inicializadas. | Sin argumentos |
| dump\_objects  Devuelve una cadena con información de la tabla de referencias a objetos.  dump\_objects() | Sin argumentos |

## Manejo de errores

Permiten obtener información acerca de errores o excepciones ocurridas en tiempo de ejecución.

|  |  |
| --- | --- |
| last\_error  Devuelve una cadena con el último mensaje de error ocurrido.  last\_error() | Sin argumentos |
| last\_error\_code  Devuelve el número (código numérico) del último error ocurrido.  last\_error\_code() | Sin argumentos |
| rise\_error  Desencadena una excepción (error)  rise\_error(código, mensaje) | código – Número (código) del error  mensaje – Cadena con el mensaje de error |
| clear\_exceptions  Limpia la pila de excepciones del intérprete  clear\_exceptions() | Sin argumentos |
| pop\_exception  Devuelve una referencia a la última excepción y la saca de la pila del intérprete.  pop\_exception() | Sin argumentos |

## Sistema

|  |  |
| --- | --- |
| runscript  Ejecuta un programa DKL y devuelve el árbol de resultados como una cadena.  runscript(programa, nombre\_variable, referencia) | programa – Cadena con la ruta y nombre del archivo dkl a ejecutal  nombre\_variable – Cadena con el nombre de la variable global (sin incluir el carácter @) que apuntará a la referencia indicada.  referencia – Referencia a un objeto que será pasado al programa que se ejecutará con el nombre de variable indicado |
| eval  Evalúa la expresión DKL válida y devuelve el resultado  eval(expresión) | expresión – Es una cadena que contiene una expresión a ser evaluada. |
| fxml  Devuelve la salida XML del árbol de resultados al momento de la llamada como una cadena de texto y limpia el buffer interno del intérprete.  fxml()  fxml(argumentos) | argumentos – Es una lista de argumentos variable cuyas expresiones serán evaluadas antes de devolver el resultado. |
| ftext  Devuelve el texto con las marcas de inyección de expresiones evaluadas.  ftext(texto) | texto – Cadena que contiene inyecciones de expresiones. |
| isset  Devuelve cierto si la variable indicada ha sido declarada.  isset(nombre\_variable) | nombre\_variable – Es una cadena con el nombre de una variable. |
| isnull  Devuelve cierto si la referencia indicada apunta hacia un objeto null.  isnull(referencia) | referencia – Referencia a un objeto |
| ifnull  Devuelve el segundo argumento si el primero es una referencia null o el mismo primero si no lo es.  ifnull(referencia, ref\_default) | referencia – Es una referencia a un objeto que será evaluada  ref\_default – Es una referencia a un objeto que será devuelto en cado de que referencia sea null |
| \_null  Devuelve una nueva referencia hacia un objeto null.  \_null() | Sin argumentos |
| \_free  Libera los recursos asociados a una referencia  \_free(referencia) | referencia – Una variable de referencia que apunta a un objeto |

## Consola

|  |  |
| --- | --- |
| print  Imprime en la consola un texto y agrega una secuencia de fin de línea  print(texto) | texto – Cadena de texto a imprimir en la consola |
| input  Lee una secuencia de caracteres de la consola hasta que se recibe un Enter y la devuelve como una cadena  input()  input(texto) | texto – Cadena que se imprime antes de esperar por la entrada de consola |
| stdin  Lee toda la entrada de consola (hasta el final del buffer) y la devuelve como una cadena.  stdin() | Sin argumentos |
| write  Escribe una cadena de texto en la consola sin la secuencia de fin de línea.  write(texto) | texto – Cadena de texto a imprimir |
| write\_utf8  Escribe una cadena de texto en la consola con codificación utf8 independientemente de la establecida en la configuración del sistema.  write\_utf8(texto) | texto – Cadena de texto a imprimir |
| write\_bytes  Escribe los bytes directamente a la consola.  write\_bytes(bytes) | bytes – Referencia a un array de bytes de .Net |

## Matemáticas

|  |  |
| --- | --- |
| sqr  Devuelve la raíz cuadrada de un número  sqr(número) | número – Número |
| abs  Devuelve el valor absoluto de un número  abs(número) | número – Número |
| round  Devuelve un número redondeado a la cantidad de decimales indicado  round(número, decimales) | número – Número a redondear  decimales – Número que indica la cantidad de decimales a los que se redondeará. |
| rnd  Devuelve un número aleatorio  rnd(min, max) | min – Número límite inferior del rango de números aleatorios  max – Número límite superior del rango de números aleatorios |
| exp  Devuelve un número elevado a una potencia  exp(número, potencia) | número – Número que será elevado  potencia – Número de la potencia a la cual elevar |
| sen  Devuelve el seno de un ángulo expresado en radianes  sen(ángulo) | ángulo – Número que expresa un ángulo en radianes |
| cos  Devuelve el coseno de un ángulo expresado en radianes  cos(ángulo) | ángulo – Número que expresa un ángulo en radianes |
| tan  Devuelve la tangente de un ángulo expresado en radianes  tan(ángulo) | ángulo – Número que expresa un ángulo en radianes |
| mod  Devuelve el residuo de una división  mod(número, denominador) | número – Número a dividir  denominador – Número denominador de la división |
| div  Devuelve el resultado de una división o cero si el dividendo o el divisor son cero sin generar un error de división por cero.  div(número, denominador) | número – Número a dividir  denominador – Número denominador de la división |
| divlng  Devuelve la parte entera de una división considerando al dividendo y divisor como enteros  cociente(número, denominador) | número – Número a dividir  denominador – Número denominador de la división |
| log  Devuelve el logaritmo de un número  log(número, base) | número - Valor numérico del que se calculara el logaritmo  base - Valor numérico base al cual se elevará el logaritmo |
| trun  Devuelve la parte entera de un número  trun(número) | número – Número |
| dec  Devuelve la parte fraccionaria de un número  dec(número) | número – Número |

## Lógicas

|  |  |
| --- | --- |
| not  Devuelve el resultado de la operación lógica NOT sobre el valor booleano indicado  not(número) | número – Número que se considera cierto si es diferente de cero y falso en cualquier otro caso. |
| or  Devuelve el resultado de la operación lógica OR sobre dos valores booleanos dados  or (valor1, valor2) | valor1 – Número que se considera cierto si es diferente de cero y falso en cualquier otro caso.  valor2 – Número que se considera cierto si es diferente de cero y falso en cualquier otro caso. |
| and  Devuelve el resultado de la operación lógica AND sobre dos valores booleanos dados  and(valor1,valor2) | valor1 – Número que se considera cierto si es diferente de cero y falso en cualquier otro caso.  valor2 – Número que se considera cierto si es diferente de cero y falso en cualquier otro caso. |
| if  Devuelve el valor numérico del segundo argumento de la función si el resultado de la evaluación del primer argumento es cierto o el valor numérico del tercer argumento en caso contrario.  if(expresión, cierto, falso) | expresión – Número que será evaluado  cierto – Número  falso – Número |
| ifstr  Devuelve el valor de cadena del segundo argumento de la función si el resultado de la evaluación del primer argumento es cierto o el valor de cadena del tercer argumento en caso contrario.  ifstr(expresión, cierto, falso) | expresión – Número que será evaluado  cierto – Cadena  falso – Cadena |

## Conversiones

|  |  |
| --- | --- |
| str  Devuelve una cadena que representa al número dado.  str(número) | número – Número que será convertido en cadena |
| val  Devuelve un número que equivale al expresado como cadena dada.  val(cadena) | cadena- Cadena que será convertida a número |

## Fecha y hora

DKL ofrece un mecanismo sencillo para manejar valores de fecha y hora que consiste en tratar los valores como números decimales, en los que la parte entera representa a la fecha y la parte decimal a la hora. No hay representación de zona horaria.

Por ejemplo:

La fecha 23 de Noviembre del año 2023 se representa como el número 20231123

La fecha y hora 5 de Septiembre de 2007 a las 20:35:23 sería 20070905.203523

|  |  |
| --- | --- |
| ndatetime.fromstr  Devuelve el número que corresponde a la fecha indicada como cadena con el formato señalado.  ndatetime.fromstr(fecha, formato) | fecha – Una cadena que representa una fecha  formato – Un formato de fecha con la sintaxis usada por C# |
| ndatetime.tostr  Devuelve una cadena que representa la fecha representada por un número con el formato indicado  ndatetime.tostr(fecha, formato) | fecha – Un número que representa una fecha  formato – El formato con el que se representará la cadena de fecha (sintaxis de C#) |
| ndatetime.diff  Devuelve la diferencia (días, semanas, meses o años) entre dos fechas expresadas como números.  ndatetime.diff(fecha1, fecha2, intervalo) | fecha1- Número que representa una fecha  fecha2- Número que representa una fecha  intervalo – Cadena con alguno de los siguientes caracteres: d w m y (días, semanas, meses, años) |
| ndatetime.add  Devuelve un número que representa una fecha a la que se ha adicionado el intervalo indicado.  ndatetime.add(fecha, num, intervalo) | fecha – Un número que representa una fecha  num – Número que indica la cantidad a sumar en el intervalo indicado  intervalo – Cadena con alguno de los siguientes caracteres: d w m y (días, semanas, meses, años) |
| ndatetime.now  Devuelve un número que representa la fecha y hora actual del sistema  ndatetime.now() | Sin argumentos |
| ndatetime.year  Devuelve el año de una fecha representada como número  ndatetime.year(fecha) | fecha – Un número que representa una fecha |
| ndatetime.month  Devuelve el mes de una fecha representada como número  ndatetime.month(fecha) | fecha – Un número que representa una fecha |
| ndatetime.day  Devuelve el día de una fecha representada como número  ndatetime.day(fecha) | fecha – Un número que representa una fecha |