Dialog Insight

Smart Marketing Catalyst

DI# language

User guide

19/4/2016

Table of contents

Basic syntax

The DI# tags

When the compiler processes code, it looks for these opening and closing tags **[[** and **]]** indicating the boundaries of the code that needs to be parsed. Everything outside the DI# opening and closing tags is ignored. Inside those tags, if you want to display simple text that should not to be interpreted as code, you can use the **output.write(…);** method. If there is only a single operation between the tags, you can use the **[[=… ;]]** shortcut.

Example :

```
The following code [[="text";]] is a shortcut for [[output.write("text");]]
```

Escaping from HTML

Everything outside the opening/closing tags is ignored by the compiler, which allows DI# code with mixed contents. DI# code can be inserted in HTML documents to create templates, among other things.

```
<p>This will be ignored by the compiler and displayed by the browser.</p>
[[="This will be analyzed by the DI# compiler.";]]
<p>This will also be ignored by the compiler and displayed by the
browser.</p>
```

or

```
<p>This will be ignored by the compiler and displayed by the browser.</p>
[[output.write("This will be analyzed by the DI# compiler.");]]
<p>This will also be ignored by the compiler and displayed by the
browser.</p>
```

Example : Advanced escaping using conditions

```
[[if(a > b){]]
  This will be displayed if the expression is true.
[[} else {]]
  Otherwise, this will displayed.
[[}]]
```

or

```
[[
if(a > b)
{
    output.write("This will be displayed if the expression is true.");
}
else
{
    output.write("Otherwise, this will be displayed.");
}
]]
```

In this example, the compiler will ignore blocks where the condition is not met, even if they're outside the DI# opening/closing tags. If the above condition is true, all the code contained in the *else* condition will not be evaluated since the DI# interpreter will skip the blocks where the condition is false.

Instructions delimiters

Like C#, Perl or PHP, the DI# language requires instructions to be terminated by a semicolon. The closing tags do not imply the end of a statement, the semicolon is mandatory.

Comments

DI# allows for two types of comments: single-line or multiline. Single-line comments can be an entire line or just the last part of one. Begin your comment with **//** and everything for there to the end of the line will be ignored by the compiler. Multiline comments can stretch over a large chunk of code. Begin your comment with **/*** and finish it with ***/** Everything in-between will be ignored by the compiler.

Variable types

Introduction

Declaration	Type	Definition
string	Characters string	A Unicode character string, each character is stored using 2 bytes.
int	Integer number	A 32-bit integer number between -2 147 483 648 and 2 147 483 647.
decimal	Decimal number	High precision floating point numbers (28-29 significant digits). Values range from $(-7.9x10^{28})/(100^{28})$ to $(7.9x10^{28})/(100^{28})$.
datetime	Date and time	Specifies a date and time with the following format: yyyy.MM.dd hh:mm:ss
bool	Boolean	A boolean value can be either TRUE or FALSE.
timespan	Time span	Contrary to some other languages, timespan represents an amount of time, whether it's days, hours, minutes or seconds. This type exists primarily to perform operations datetime variables. Example: `timespan t = date1 - date2 ;` Note: To directly assign a timespan variable, you must specify a value corresponding to a number of days (for example, 2.25 if you want to represent 2 days and 6 hours).
datasource	Data source	Any type of data listed in this grid, or any complex data type like arrays, lists, dictionaries, etc. (similar to the *var* data type in C#)

In order to be compatible with projects or relational tables that include nullable fields, some of the data types allow for null values. However, we strongly advise against using nullable types in your code, in order to reduce the risks of execution error.

Note: For certain data types (int, decimal, datetime) the DI# language allows values larger than what is supported by the database, meaning that it's possible to create a variable containing a value that cannot be stored in a database field of the same type as the variable. Refer to the projects and relational tables documentation to make sure of what the database limits are.

Example:

```
string a = "text";
int b = 1;
decimal c = 1.234;
datetime d1 = 2016.01.01 12:34:56;
datetime d2 = 2016.01.01;
bool e = true; // or false or null
timespan g = 2.25; // 2 days and 6 hours
datasource h = {1, 2, 3};
```

Type extensions

The DI# language implements several methods allowing for easy manipulation of the basic data types (string, int, etc.).

string

Property or method	Returned type	Description
Length	int	Returns the number of characters in a string object.
Capitalize(…)	string	`MyText.Capitalize()` Puts the 1st letter of the string in uppercase. `MyText.Capitalize(true)` Puts the 1st letter of every word of the string in uppercase.
IndexOf(…)	int	`MyText.IndexOf("find")` Returns the position of the 1st occurrence of the string passed as a parameter within the searched string.
LastIndexOf(…)	int	`MyText.LastIndexOf("find")` Returns the position of the last occurrence of the string passed as a parameter within the searched string.
Left(n)	string	Returns the first *n* characters from the left of the string. Equivalent to `MyText.Substring(0,n)`
Right(n)	string	Returns the first *n* characters from the right of the string. Equivalent to `MyText.Substring(MyText.Length-n,n)`
Replace(…)	string	`MyText.Replace("find","replace")` Returns a new string object where all occurrences of the 1st parameter string have been replaced with the 2nd parameter string.

Substring(…)	string	`MyText.Substring(start,length)` Extracts part of the original string, starting at the 1st parameter index, for the number of characters specified in the 2nd parameter. `MyText.Substring(start)` Extracts the characters from the original string, starting at a specific position, to the end of the string.
ToLower()	string	Converts all characters from a string to lowercase.
ToUpper()	string	Converts all characters from a string to uppercase.
Trim()	string	Removes all leading and trailing spaces from the string.

Examples:

```
[[
string MyText = "this is a CHARACTER string  ";

MyText.Length;
// Returns 28
MyText.Capitalize(true);
// Returns "This Is A CHARACTER String  "
MyText.Capitalize(); // or MyText Capitalize(false);
// Returns "This is a CHARACTER string  "
MyText.IndexOf("s");
// Returns 3
MyText.LastIndexOf("s");
// Returns 20
MyText.Left(12);
// Returns "this is a CH"
MyText.Right(12);
// Returns "TER string  "
MyText.Replace("CHARACTER", "test");
// Returns "this is a test string  "
MyText.Substring(12);
// Returns "TER string  "
MyText.Substring(12, 3);
// Returns "TER"
MyText.ToLower();
// Returns "this is a character string  "
MyText.ToUpper();
// Returns "THIS IS A CHARACTER STRING  "
MyText.Trim();
// Returns "this is a CHARACTER string"
]]
```

Note that based on the data type of the value returned by these methods and properties, you can chain other methods and properties. For example :

```
[[
string MyText = "this is a CHARACTER string  ";
MyText.Trim().Replace("CHARACTER ", "test").Capitalize();
// Returns "This is a test string"
]]
```

int

Property or method	Returned type	Description
ToString()	string	Returns the value of the object as a character string.

decimal

Property or method	Returned type	Description
ToString()	string	Returns the value of the object as a character string.

datetime

Property or method	Returned type	Description
AddYears(n)	DateTime	Adds *n* years to the original date.
AddMonths(n)	DateTime	Adds *n* months to the original date.
AddDays(n)	DateTime	Adds *n* days to the original date.
AddHours(n)	DateTime	Adds *n* hours to the original date.
AddMinutes(n)	DateTime	Adds *n* minutes to the original date.
AddSeconds(n)	DateTime	Adds *n* seconds to the original date.
DateDiff(d)	Timespan	Subtracts the *d* date from the original date and returns a timespan.
isDST()	bool	Returns true if the original date is using Daylight Savings Time, based on the culture of the execution context.
ToString(…)	string	Returns the value of the object as a character string. The result can be formatted using customization parameters. For more details: https://msdn.microsoft.com/en-us/library/az4se3k1(v=vs.110).aspx
Date	DateTime	Returns a new date composed only of the year/month/day of the original date. Hours/minutes/seconds are ignored.
Year	int	Returns the year part of the original date.
Month	int	Returns the month number (1 to 12) of the original date.
Day	int	Returns the day of the month (1 to 31) of the original date.
DayOfYear	int	Returns the day of the year (1 to 365) of the original date.
DayOfWeek	int	Returns the day of the week (Sunday = 1, Saturday = 7) of the original date.
Hour	int	Returns the hour part (0 to 23) of the original date.
Minute	int	Returns the minute part (0 to 59) of the original date.
Second	int	Returns the second part (0 to 59) of the original date.
Ticks	int	Returns the number of ticks elapsed between January 1st, 0001 and the original date. 1 second = 10 000 000 ticks.

```
[[
datetime MyDate = 2016.01.01 12:34:56;
MyDate.AddYears(3);    // Returns 2019.01.01 12:34:56
MyDate.AddMonths(3);   // Returns 2016.04.01 12:34:56
MyDate.AddDays(3);     // Returns 2016.01.04 12:34:56
MyDate.AddHours(3);    // Returns 2016.01.01 15:34:56
MyDate.AddMinutes(3);  // Returns 2016.01.01 12:37:56
MyDate.AddSeconds(3);  // Returns 2016.01.01 12:34:59
MyDate.DateDiff("2014.03.03").Days; // Returns 669
MyDate.isDST();        // Returns False
MyDate.ToString();     // Returns "2016.01.01 12:34:56"
MyDate.ToString("d/M/yyyy HH:mm:ss"); // Returns "1-1-2016 12:34:56"
MyDate.ToString("F"); // Returns "1 January 2016 12:34:56"
MyDate.ToString("ddd, dd MMM yyyy HH':'mm':'ss 'GMT' ");
                      // Returns "Friday., 01 January. 2016 12:34:56 GMT"
MyDate.Date;          // Returns 2016.01.01 00:00:00
MyDate.Day;           // Returns 1
MyDate.DayOfWeek;     // Returns 5
MyDate.DayOfYear;     // Returns 1
MyDate.Hour;          // Returns 12
MyDate.Minute;        // Returns 34
MyDate.Month;         // Returns 1
MyDate.Second;        // Returns 56
MyDate.Ticks;         // Returns 635872484960000000
MyDate.Year;          // Returns 2016
]]
```

bool

Property or method	Returned type	Description
ToString()	string	Returns the character string "True" or "False".

timespan

Property or method	Returned type	Description
ToString()	string	Returns the value of the object as a character string.

Arrays and collections

Introduction

A DI# array is a container for one or more variables, associating keys with values in an ordered way. It can be used for simple lists, dictionaries, or more complex purposes like hash tables, queues and more. The value of an array element can be a single variable, a list or a multidimensional array. The data type of each element in an array can be fixed or dynamic (the same way *List<datatype>* works in any .NET language).

Declaration

A typical array declaration in DI# looks like this: `{ value1, value2, value3 }`

Example :
```
datasource myArray = { 1, 2, 3, "abc", "def" };
```

You can define a collection of named elements like this:
```
{ fieldName1 : value1, fieldName2 : value2, fieldName3 : value3 }
```

Example :
```
datasource myDictionary = { FirstName:"John", Name:"Smith", Age:30 };
```

An element's value is an *expression*, which itself can contain other *expressions* like in this example:

```
datasource myArray =
{
    { FirstName: "Jean", Name: "Tremblay", Age : 40 },
    { FirstName: "John", Name: "Smith", Age : 30 }
};
```

An array can be declared with a fixed length (`int a[10];`) or with a variable length (`string b[];`).

Examples :
```
string a[] = {"text1", "text2", "text3"};
int b[] = {1, 2, 3};
decimal c[] = {1.111, 2.222, 3.333};
datetime d[] = {2016.01.01 01:01:01, 2016.02.02 02:02:02};
bool e[] = {true, false, null};
timespan g[] = {1472500, 1472500};
datasource h[] = {123, "text", 1.111, 2016.01.01 01:01:01, true};
```

Once an array is declared, it can only accept values of the type specified. For example, an array declared `int a[10]` will only accept integer values.

Operators

Comparison operators

Simple operators

- ==
- !=
- <
- <=
- >
- >=

Example	Description	Result
a == b	Equal to	TRUE if a equals b
a != b	Different than	TRUE if a is not equal to b
a < b	Lesser than	TRUE if a is lesser than b but not equal
a <= b	Lesser or equal to	TRUE if a is lesser or equal to b
a > b	Greater than	TRUE if a is greater than b but not equal
a >= b	Greater or equal to	TRUE if a is greater or equal to b

Contrary to some languages like PHP, these operators cannot be used on string variables. DI# does not support "alphabetical order" comparisons like *"abc"<"bcd"* that other languages allow.

Complex operators

LIKE (and its opposite *NOT LIKE*) evaluates a character string based on pattern matching, with the % symbol being used as a wildcard that replaces any number of consecutive characters.

- *LIKE "a%"* will return TRUE if the string value begins with "a".
- *LIKE "%a"* will return TRUE if the string value ends with "a".
- *LIKE "%a%"* will return TRUE if the string contains "a" anywhere in it, including at the beginning or at the end.
- *LIKE "th%s"* will return TRUE if the string starts with "th" and ends with "s", no matter how many characters are between. "this", "thus", "thanks" would all fit.
- *LIKE "abc"* with no wildcard (%) has the same effect as the equal operator (==).

CONTAINS (and its opposite *NOT CONTAINS*) searches inside a string value for the occurrence of a specific character string.

- *CONTAINS "abc"* will return TRUE if the string value contains "abc" somewhere inside the string, including at the beginning or at the end. It has the same effect as *LIKE "%abc%"*.
- *CONTAINS* only applies to character strings. It cannot be used to determine whether an array or a list contain a specific value.

IS NULL (and its opposite *IS NOT NULL*) tests if a variable contains a value or not. Note that in the case of a string variable, an empty value ("") is NOT the same as a NULL value. A NULL value is when a variable has not been assigned a value at all.

Examples:

```
string a = "this is a test";
output.write(a contains "this" ? true : false);  // Returns True
output.write(a like "this" ? true : false);      // Returns False
output.write(a like "this%" ? true : false);     // Returns True
output.write(a is null ? true : false);          // Returns False
```

Mathematical operators

Supported operators:

- Division : **/**
- Multiplication : *****
- Modulo : **%**
- Addition : **+**
- Subtraction : **-**

Example :

Example	Name	Result
a + b	Addition	Sum of a and b
a - b	Subtraction	Remainder of b taken away from a
a * b	Multiplication	a multiplied by b
a / b	Division	a divided by b
a % b	Modulo	Take b away from a as many times as possible, the modulo is the remainder (will always be an integer number lesser than b).

Note: The negation operator is not supported when applied to variables. The *-b* expression will not compile, try multiplying by -1 instead (*b * -1*). Because of this, a subtraction written *a-b* will not compile unless you leave a space after the minus sign (*a - b* will compile just fine).

Array operators

Unitary operators

- Set the value of an item at a specific position of an array:

```
x[0] = 123; // the value of the 1st item (index 0) is now 123
```

- Add an item at the end of a dynamic-size array:

```
x += "test"; // add an item "test" at the end of an array of strings
```

- Delete a value from a dynamic-size array:

```
x -= "test"; // all items with the value "test" will be removed
```

Batch operators

- Assign multiple values with a single instruction:
```
x = { "blue", "red", "green" };
// x now contains 3 items - "blue", "red" and "green"
```
- Add multiple items at the end of a dynamic-size array:
```
x += { "orange", "pink", "yellow" }; // x now contains 6 items
// "blue", "red", "green", "orange", "pink" and "yellow"
```
- Delete multiple values from a dynamic-size array:
```
x -= { "blue", "pink"};
// all items with the value "blue" or "pink" will be removed
```

All the above examples assume that you are using values of the correct data type for the array being modified.

Variable scope

The scope of a variable depends on the context in which it has been declared. A variable defined at the code root will be visible for the entire script, and a variable declared in a function will be limited to that function. You can think of a variable's context as being delimited by **{** and **}**. Any variable declared inside curly brackets has a local scope.

Example :
```
[[
int x = 123;      // global scope

string MyFunction(string value)
{
    int y = 456; // local scope
    x = x + 1;   // global variables can be accessed in a function
}
]]
```

Query expressions

Query expressions can be used to fetch and transform information from any complex data source. These queries can retrieve, sort and filter an array (dictionary, list, etc.). Their goal is, starting from an array, to create a new data source that:

- Filters the original array (*WHERE* clause)
- Sorts the original array (*ORDER BY* clause)
- Extracts a specific property from items in the original array
- Deduplicates items (*DISTINCT* option)

Usage

Query element	Description
Select	Specifies the values to extract from the data source. Use the *distinct* option if you only wish to return unique values.
From	Identifies the data source to which the query is applied.
Where	Allows the filtering of results using inclusion or exclusion rules.
Order by	Sorts the data according to one or more fields, in ascending or descending order. Supported keywords: *asc, ascending, desc, descending.*

All these elements come together like this:

```
select identifier.property1
from identifier in expression
where identifier.property2 = value
order by (identifier.property3 asc)
```

`expression`: the structure from which the data will be extracted (array, collection, etc.)
`identifier`: the name of the variable that will be used to refer to the data source (in the `select`, `where` and `order by` clauses).

Examples

Let's suppose the following data source:

```
[[
datasource Vehicles =
{
    { Brand: "Toyota", Model: "Sienna", Year: 2015},
    { Brand: "Ford", Model: "Fusion", Year: 2013},
    { Brand: "Chevrolet ", Model: "Equinox", Year: 2005 },
    { Brand: "Hyundai", Model: "Accent", Year: 2016 },
    { Brand: "Mazda", Model: "CX-5", Year: 2015 },
    { Brand: "Hyundai", Model: "Accent", Year: 2010 },
};
]]
```

To select the brands of all the vehicles:

```
[[
datasource Brands = select car.Brand from car in Vehicles;
]]
```

The resulting `Brands` object would contain:

```
Toyota
Ford
Chevrolet
Hyundai
Mazda
Hyundai
```

To select the unique brands (without duplicates) of all the vehicles:

```
[[
datasource Brands = select distinct car.Brand from car in Vehicles;
]]
```

The resulting `Brands` object would contain:

```
Toyota
Ford
Chevrolet
Hyundai
Mazda
```

To select the unique brands of all the vehicles, from the most recent to the oldest:

```
[[
datasource Brands = select distinct car.Brand from car in Vehicles
order by (car.Year desc);
]]
```

The resulting `Brands` object would contain:

```
Hyundai
Mazda
Toyota
Ford
Chevrolet
```

To select the unique brands of all the vehicles, from the most recent to the oldest, excluding those from before 2014 :

```
[[
datasource Brands = select distinct car.Brand from car in Vehicles
where (car.year > 2014) order by (car.Year desc);
]]
```

The resulting `Brands` object would contain:

```
Hyundai
Mazda
Toyota
```

Note that it's possible to clone an array by selecting all its items:

```
[[
datasource VehiclesCopy = select car from car in Vehicles;
]]
```

Important notes

Although they are powerful, the data source query features come at a huge performance cost. If used incorrectly in a message, these queries can have an important impact on the speed at which messages are prepared. A single query placed in a message sent to a million contacts will have to be processed a million times. Don't hesitate to ask our support team or our analysts for advice when creating messages requiring this level of complexity.

Flow control

if

The *if* instruction is one of the most important keywords in any programming language. It allows the conditional execution of a code block. In DI#, *if* works the same way as in most languages:

```
if (expression)
{
    command1;
    command2;
}
```

or

```
if (expression)
    command;
```

`expression` must return a boolean value. If it evaluates to TRUE the associated code block will be executed, if it evaluates to FALSE the commands will be ignored. In the example below, the sentence *"a is larger than b"* will be shown only if *a* is larger than *b*.

```
[[
int a = 1;
int b = 2;
if (a > b)
  output.write("a is larger than b");
]]
```

You can insert *if* statements inside other *if* statements, allowing for great flexibility when it comes to selecting code execution based on a large number of parameters.

else

You will often need to execute some commands if a particular condition is met, and some other commands if it isn't. The *else* keyword is used after an *if* and provides a code block to be executed when the `expression` evaluates to FALSE. In the example below the sentence *"a is larger than b"* will be shown only if *a* is larger than *b*, and the sentence *"a is smaller or equal to b"* if it isn't.

```
[[
int a = 1;
int b = 2;
if (a > b)
{
  output.write("a is larger than b");
} else {
  output.write("a is smaller or equal to b");
}
]]
```

if/else shorthand

Like in C#, it is possible to abbreviate an *if-else* statement by using the ?: operator.

```
(condition) ? (expression_if_true) : (expression_if_false);
```

Example :

```
[[
int a = 1;
int b = 2;
output.write((a > b) ? "a is larger than b" : "no it's not");
]]
```

Note: Due to limitation in the DI# compiler, make sure that you place the `condition` between parentheses when using this operator.

while

The *while* instruction is the simplest way to implement a code loop in DI#. This command behaves the same way as in C.

```
while (expression)
{
    command1;
    command2;
}
```

or

```
while (expression)
    command;
```

The following example displays the numbers 1 through 10:

```
[[
int a = 1;
while (a <= 10)
{
    output.write(a);
    a += 1;
}
]]
```

Note: Pay particular attention to the management of the variable on which the `condition` is based, to avoid creating endless loops. As a safety measure, DI# *while* statements are limited to 250 iterations.

foreach

Array iterator

The *foreach* statement provides a simple way of going through elements of an array.

```
foreach (identifier in expression)
{
    command1;
    command2;
}
```

or

```
foreach (identifier in expression)
    command;
```

`identifier` is the chosen name for the variable holding the value of each individual item contained in `expression` as we go through the loops. This variable will be of the same data type as the values in the array. The example below shows how to display all the elements of a simple array of integer numbers:

```
[[
int myArray[] = { 1, 2, 3, 4, 5};
foreach(number in myArray)
{
    output.write(number);
}
]]
```

Using a complex expression

Since a *foreach* allows iterating through any array resulting from the evaluation of an expression, it's possible to use complex expressions directly in the *foreach* statement:

```
[[
datasource myArray =
{
    { FirstName : "Jean", LastName : "Tremblay", Age : 40 },
    { FirstName : "John", LastName : "Smith", Age : 30 }
};

foreach(Namevalue in
    (select item.LastName
     from item in myArray
     order by (item.Age)))
{
    output.write("Last name: " + Namevalue);
}
]]
```

Iterating with a counter

As it's the case in several programming languages, it is possible to maintain a counter while going through the *foreach* loop:

```
foreach (counter => identifier in expression)
    command ;
```

`counter` is the name of a variable that will contain an integer number starting at 0 that will be incremented by 1 each time a new `identifier` is fetched from `expression`.

```
[[
int myArray[] = { 1, 3, 5, 7, 9};
foreach(itemOrder => number in myArray)
{
    output.write("position : " + itemOrder);
    output.write("value : " + number);
}
]]
```

Note: Like the *while* statement, a *foreach* loop is limited to 250 iterations.

break

The *break* command is used to terminate a *foreach* or *while* loop before all its elements have been processed. The example below stops the execution of the foreach after the number 3 has been processed:

```
[[
int myArray[] = { 1, 2, 3, 4, 5};
foreach(number in myArray)
{
    output.write(number);
    if (number == 3)
        break;
}
]]
```

switch

The *switch* statement is the equivalent to a series of *if* instructions based on the same expression. In some situations, you will need to test the value of a variable and provide several code blocks to be executed according to the possible values of that variable. The *switch* statement is built specifically for that purpose.

```
switch (expression)
{
  case value1:
  {
    command1;
    command2;
  }
  case value2:
  {
    command3;
    command4;
  }
}
```

or

```
switch (expression)
{
  case value1:
    command1;
  case value2:
    command2;
}
```

Both examples below are ways of achieving the same effect, one with a series of *if* statements, the other with a *switch*:

```
[[
int i = 0;
if (i == 0)
    output.write("i equals 0");
if (i == 1)
    output.write("i equals 1");
if (i == 2)
    output.write("i equals 2");


switch (i)
{
    case 0:
        output.write("i equals 0");
    case 1:
        output.write("i equals 1");
    case 2:
    {
        output.write("i equals 2");
        output.write("for real!");
    }
}
]]
```

Contrary to other programming languages, it is not necessary in DI# to use the *break* instruction to end the various *case* blocks inside your *switch*. Once a *case* is reached, the commands it contains will be executed and the compiler will exit the *switch* statement. There is also no *default* statement in the DI# version of *switch*, if no *case* match the branching expression, no code will be executed.

continue

The *continue* command is used inside a *foreach* or *while*, and forces that loop to begin processing the next item immediately without executing any of the remaining commands for the current iteration. The example below will display every number contained in the array except for 3, since the *continue* keyword will trigger the beginning of the next loop immediately, skipping the `output.write` command.

```
[[
Datasource myArray[] = { 1, 2, 3, 4, 5};
foreach(number in myArray)
{
    if (number == 3)
        continue;
    output.write(number);
}
]]
```

return

The *return* command forces the current module to exit and hands back control to the part of the program that had called it, resuming code execution at the next line of the calling module. If *return* is called from inside a function it terminates that function immediately. If *return* is called from the main program, code execution is terminated completely.

Return accepts a single optional parameter that, if present, becomes the return value of the function being terminated. In the example below, *return* stops the execution of `checkNumber` and returns a value to the main program that gets stored in the `result` variable.

```
[[
string checkNumber(int x)
{
    if (x == 0)
        return "x equals 0";
    if (x == 1)
        return "x equals 1";
    if (x == 2)
        return "x equals 2";
}

string result = checkNumber(1);
output.write(result);
// displays "x equals 1"
]]
```

Functions

Declaration

Function declaration in DI# follows the same basic structure as in C.

```
[Type] [Name] ([Parameter1], [Parameter2], etc.)
{
      Commands;
}
```

Functions must be defined at the root code level. They cannot be defined inside another function or inside any curly-bracket-delimited scope ({ }).

Example of a simple function declaration, with 2 integer parameters and an integer return value:

```
int addNumbers (int x, int y)
{
    return x + y;
}
```

Example of function declaration with a default parameter value:

```
void OutputWithHTMLTag (string value, string Tag = "B")
{
    if (value is null)
        return;
    Output.write("<" + Tag + ">" + value + "</" + Tag + ">");
}
```

Note: A function can return any data type (*string*, *int*, *decimal*, etc.), but if your function declaration specifies a return type the function itself HAS to return a value of the specified data type no matter how it terminates its execution.

The return type of a function can be *void*, in which case a simple return statement without any parameters is sufficient to end the function's execution, or reaching the end of the function's commands.

Calling

A function get called like any other programming language:

functionName(params) or *functionName ()*

A *void* function (without a return value) can be used as a simple statement inside a function or any code block:

```
OutputWithTags("my text", "i");
```

Functions with a return value of any type other than *void* can be used as expressions or values:

```
int x = addNumbers(1,1);
if (addNumbers(x,y) > z) continue;
```

Variable scope

Like it has been mentioned before, variables declared inside functions are visible only for that scope. Global variables declared at the root code level are visible everywhere.

Contact
Canada : 1 866 529-6214
France : 01 84 88 40 66
Russia : +7 (495) 226-04-11

Email : info@dialoginsight.com
Web site : www.dialoginsight.com
Blog : academie.dialoginsight.com

@DialogInsight
Dialog Insight