Dialog Insight

Catalyseur d'intelligence marketing

Le langage DI#

Guide d'utilisation

19/4/2016

Canada • France • Russie

dialoginsight.com

Table des matières

La syntaxe de base 3
Les balises du DI# 3
Échappement depuis du HTML 3
Séparation des instructions 4
Les commentaires 4
Les types 5
Introduction 5
Extension de type 7
string 7
int 9
decimal 9
datetime 9
bool 10
timespan 10
Tableaux et collections 11
Introduction 11
Déclaration 11
Instanciation 12
Les opérateurs 13
Les opérateurs de comparaisons 13
Opérateurs simple 13
Opérateurs complexes 13
Les opérateurs arithmétiques 15
Les opérateurs de tableaux 15
Opération unique 15
Opération en lot 15
Portée des variables (scope) 16
Expressions de requête 17

Utilisation 17
Exemples 18
Notes importantes 20
Les structures de contrôle 21
if 21
else 22
if/else raccourci 22
while 23
foreach 23
Itération sur un tableau 23
Utilisation d'une expression complexe 24
Itération avec compteur 25
break 25
switch 25
continue 27
return 27
Les fonctions 29
Déclaration 29
Appel 30
Portée des variables 30

La syntaxe de base

Les balises du DI#

Lorsque notre compilateur traite du code, il cherche les balises d'ouverture et de fermeture suivantes **[[** et **]]** qui délimitent le code qu'il doit interpréter. Tout ce qui se trouve en dehors des balises ouvrantes / fermantes de DI# est ignoré. À l'intérieur de ces balises, il est possible de ne pas interpréter de code afin d'afficher un simple texte. Pour cela, il suffit d'utiliser la méthode **output.write(…);**
Notez qu'un raccourci existe afin d'afficher du texte lorsqu'il s'agit de la seule instruction à exécuter entre les crochets : **[[=… ;]]**

Exemple :

```
Le code suivant [[="texte";]] est un raccourci pour [[output.write("texte");]]
```

Échappement depuis du HTML

Tout ce qui se trouve en dehors d'une paire de balises ouvrantes/fermantes est ignoré par le compilateur, ce qui permet d'avoir du code DI# mixant les contenus. Ceci permet à DI# d'être contenu dans des documents HTML, pour créer par exemple des templates.

```
<p>Ceci sera ignoré par le compilateur et affiché dans le navigateur.</p>
[[="Alors que ceci sera analysé par le compilateur de DI";]]
<p>Ceci sera aussi ignoré par le compilateur et affiché au navigateur.</p>
```

Ou

```
<p>Ceci sera ignoré par le compilateur et affiché dans le navigateur.</p>
[[output.write("Alors que ceci sera analysé par le compilateur de DI");]]
<p>Ceci sera aussi ignoré par le compilateur et affiché au navigateur.</p>
```

Exemple : Échappement avancé en utilisant des conditions

```
[[if(a > b){]]
  Ceci sera affiché si l'expression est vraie.
[[} else {]]
  Sinon, ceci sera affiché.
[[}]]
```

Ou

```
[[
if(a > b)
{
    output.write("Ceci sera affiché si l'expression est vraie.");
}
else
{
    output.write("Sinon, ceci sera affiché.");
}
]]
```

Dans cet exemple, le compilateur va ignorer les blocs où la condition n'est pas remplie, même s'ils sont en dehors des balises ouvrantes/fermantes de DI#. Tout le code contenu dans la condition *else* ne sera donc jamais interprétée puisque l'interpréteur DI# va passer les blocs contenant ce qui n'est pas rempli par la condition.

Séparation des instructions

Comme en C#, en Perl ou en PHP, DI# requiert que les instructions soient terminées par un point-virgule à la fin de chaque instruction. Aucune balise fermante n'implique de fins d'instructions, le point-virgule est obligatoire.

Les commentaires

Le DI# supporte deux types de commentaires : Le commentaire d'une ligne, ou les commentaires multilignes :

- Les commentaires sur une seule ligne ne commentent que jusqu'à la fin de la ligne du bloc PHP courant. Ceci signifie que toute le code HTML après **//** sera ignorée jusqu'à la fin de la ligne.
- Les commentaires multilignes permettent quant à eux de commenter un bloc d'instruction en commençant par **/*** et en finissant par ***/**. Tout ce qui est inclus entre ces deux balises sera ignoré.

Les types

Introduction

Déclaration	Type	Définition
string	Chaine de caractères	Chaine de caractères Unicode, où un caractère est stocké sur 2 octets.
int	Nombre entier	Nombre entier 32 bits situé entre -2 147 483 648 et 2 147 483 647.
decimal	Nombres décimaux	Désigne tous les nombres à virgule flottante d'une très grande précision (28-29 chiffres significatifs). Le range se décrit techniquement comme suit : (-7.9 x 10^{28} to 7.9 x 10^{28}) / (100 ^{28})
datetime	Date et heure	Représente la date et l'heure au format suivant : yyyy.MM.dd hh:mm:ss
bool	Booléen	Un booléen représente une valeur de vérité. Il peut valoir TRUE ou FALSE.
timespan		Contrairement à d'autres langages, timespan représente une durée de temps, soit un nombre de Jours, Heures, Minutes et Secondes. Ce type existe principalement pour faire des opérations sur le type « datetime ». Exemple : `timespan t = date1 – date2 ;` Note : pour assigner une directement une valeur, il faut noter qu'on doit fournir une valeur décimale qui représente un nombre de jours (par exemple 2.25 = 2 jours 6 heures)
datasource	Source de données	*Datasource* représente n'importe quel type de données : ceux listés dans ce tableau, ainsi que les types dits «complexes» tels que les tableaux, les collections, les listes, les dictionnaires, etc. (équivalent à *var* en C# par exemple)

Par souci de compatibilité avec les champs nullable du projet ou des tables relationnelles, certains de ces types supportent d'être assignés à Null. Toutefois, il est fortement déconseillé d'instancier ou d'utiliser des types nullable dans le code, afin de limiter les risques d'échecs d'exécution.

Note : Pour certains types (int, decimal, datetime), le langage DI# supporte des limites souvent plus grandes que celles des bases de données, il serait donc possible de créer une variable avec une valeur qui ne pourra pas être enregistrée dans la base de données par la suite. Consultez la documentation des projets / tables relationnelle pour connaitre les limites de la base de données.

Exemple :

```
string a = "texte";

int b = 1;

decimal c = 1.234;

datetime d1 = 2016.01.01 12:34:56;

datetime d2 = 2016.01.01;

bool e = true; // ou false ou null

timespan g = 2.25; // 2 jours 6 heures

datasource h = {1, 2, 3};
```

Extension de type

Le langage DI# implémente plusieurs méthodes qui permettent de manipuler facilement les types de bases (string, int, etc.).

string

Propriétés ou méthodes	Type de retour	Description
Length	int	Obtient le nombre de caractères de l'objet String actuel.
Capitalize	string	Permet de mettre la première lettre de la chaine de caractère en majuscule. En passant *True* en paramètre, cela met en majuscule la première lettre de tous les mots de la chaine de caractère.
IndexOf	int	Retourne la position à laquelle la première occurrence de la chaine de caractère passé en paramètre existe.
LastIndexOf	int	Retourne la position à laquelle la dernière occurrence de la chaine de caractère passé en paramètre existe.
Left	string	Retourne autant de caractère depuis le début de l'objet string cible que le nombre entier passé en paramètre. `MonTexte.Left(3) ;` équivaut à `MonTexte.Substring(0, 3) ;`
Right	string	Retourne autant de caractère depuis la fin de l'objet string cible que le nombre entier passé en paramètre. `MonTexte.Left(3) ;` équivaut à `MonTexte.Substring(MonTexte.Length - 3, 3) ;`
Replace	string	Retourne un nouvel objet string dont toutes les occurrences de la chaine de caractères passés en premier paramètre sont remplacées par la chaine de caractères passés en second paramètre.
Substring	string	Récupère une sous-chaîne de l'objet string ciblé. La sous-chaîne commence à une position de caractère spécifiée (si non spécifié, commence au début) et sa longueur est définie.
ToString	string	Retourne la valeur de l'objet en chaine de caractères.
ToLower	string	Retourne la chaine de caractère ciblé ou tous les caractères sont en minuscule.
ToUpper	string	Retourne la chaine de caractère ciblé ou tous les caractères sont en majuscule.
Trim	string	Retourne la chaine de caractère ciblé sans aucun espace avant le premier et après le dernier caractère.

Exemples :

```
[[
string MonTexte = "ceci EST une chaîne de caractère  ";

MonTexte.Length;
// Retourne le nombre 34
```

```
MonTexte.Capitalize(true);
// Retourne Ceci EST Une Chaîne De Caractère

MonTexte.Capitalize(); // ou MonTexte.Capitalize(false);
// Retourne la chaine : "Ceci EST une chaîne de caractère  "

MonTexte.IndexOf("a");
// Retourne le nombre 15

MonTexte.LastIndexOf("a");
// Retourne le nombre 26

MonTexte.Left(12);
// Retourne la chaine : "ceci EST une"

MonTexte.Right(12);
// Retourne la chaîne : "caractère"

MonTexte.Replace("une chaîne", "un test");
// Retourne la chaîne : "ceci EST un test de caractère  "

MonTexte.Substring(12);
// Retourne la chaîne : "chaîne de caractère "

MonTexte.Substring(12, 3);
// Retourne la chaîne : " ch"

MonTexte.ToLower();
// Retourne la chaîne : "ceci est une chaîne de caractère  "

MonTexte.ToUpper();
// Retourne la chaîne : "CECI EST UNE CHAÎNE DE CARACTÈRE  "

MonTexte.Trim();
// Retourne "ceci EST une chaîne de caractère"

]]
```

Notez que les valeurs retournées par ces méthodes et propriétés sont des valeurs du type approprié et que vous pouvez donc enchainer des appels de méthodes ou accès aux propriétés du type retourné, par exemple :

```
[[
string MonTexte = "ceci EST une chaîne de caractère  ";
MonTexte.Trim().Replace("une chaîne", "un test").Capitalize();
// Retourne « Ceci EST un test de caractère   »

]]
```


int

Propriétés ou méthodes	Type de retour	Description
ToString()	string	Retourne la valeur de l'objet en chaine de caractères

decimal

Propriétés ou méthodes	Type de retour	Description
ToString()	string	Retourne la valeur de l'objet en chaine de caractères

datetime

Propriétés ou méthodes	Type de retour	Description
AddYears	DateTime	Ajoute aux années de la date ciblée le nombre passé en paramètre.
AddMonths	DateTime	Ajoute aux mois de la date ciblée le nombre passé en paramètre.
AddDays	DateTime	Ajoute aux jours de la date ciblée le nombre passé en paramètre.
AddHours	DateTime	Ajoute aux heures de la date ciblée le nombre passé en paramètre.
AddMinutes	DateTime	Ajoute aux minutes de la date ciblée le nombre passé en paramètre.
AddSeconds	DateTime	Ajoute aux secondes de la date ciblée le nombre passé en paramètre.
DateDiff	Timespan	Soustrait la date passée en paramètre à la date ciblée et retourne le résultat au format timespan.
isDST	bool	Retourne vrai si la date ciblée est dans l'heure avancée d'été, basé sur la culture du contexte local.
ToString	string	Retourne la valeur de la date ciblée sous forme d'une chaine de caractères. Ce résultat peut être formaté dans un format précis en passant un modèle de date en paramètre. Voir : https://msdn.microsoft.com/fr-fr/library/az4se3k1(v=vs.110).aspx
Date	DateTime	Retourne une nouvelle date à partir de l'année, le mois et le jour de la date ciblé. Toutes les autres propriétés comme les heures, minutes, secondes, etc. sont ignorées.
Year	int	Retourne l'année de la date ciblée sous forme d'un nombre entier.
Month	int	Retourne le mois de la date ciblée sous forme d'un nombre entier de 1 à 12.
Day	int	Retourne le jour du mois de la date ciblée sous forme d'un nombre entier de 1 à 31.
DayOfYear	int	Retourne le jour de l'année à laquelle correspond la date ciblée sous forme d'un nombre entier de 1 à 365.
DayOfWeek	int	Retourne le jour de la semaine à laquelle correspond la date ciblée sous forme d'un nombre entier de 1 à 7.
Hour	int	Retourne le nombre d'heures de la date ciblée sous forme d'un nombre entier de 0 à 23.
Minute	int	Retourne le nombre de minutes de la date ciblée sous forme d'un nombre entier de 0 à 59.

Second	int	Retourne le nombre de secondes de la date ciblée sous forme d'un nombre entier de 0 à 59.
Ticks	int	Retourne le nombre de graduations représentant la date et l'heure de cette instance. Une graduation représente 100 nanosecondes, ou un dix-millionième de seconde. Il existe 10 000 graduations dans une milliseconde, ou les cycles de 10 millions en une seconde.

```
[[
datetime MyDate = 2016.01.01 12:34:56;

MyDate.AddYears(3); // Résultat : 2019-01-01 12:34:56
MyDate.AddMonths(3); // Résultat : 2016-04-01 12:34:56
MyDate.AddDays(3); // Résultat : 2016-01-04 12:34:56
MyDate.AddHours(3); // Résultat : 2016-01-01 15:34:56
MyDate.AddMinutes(3); // Résultat : 2016-01-01 12:37:56
MyDate.AddSeconds(3); // Résultat : 2016-01-01 12:34:59

MyDate.DateDiff("2014.03.03").Days; // Affiche 669
MyDate.isDST(); // Affiche False
MyDate.ToString(); // Affiche : 2016.01.01 12:34:56
MyDate.ToString("d/M/yyyy HH:mm:ss"); // Résultat : 1-1-2016 12:34:56
MyDate.ToString("F"); // Résultat : 1 janvier 2016 12:34:56
MyDate.ToString("ddd, dd MMM yyyy HH':'mm':'ss 'GMT' ");
// Résultat : ven., 01 janv. 2016 12:34:56 GMT

MyDate.Date; // Résultat : 2016-01-01 00:00:00
MyDate.Day; // Résultat : 1
MyDate.DayOfWeek; // Résultat : 5
MyDate.DayOfYear; // Résultat : 1
MyDate.Hour; // Résultat : 12
MyDate.Minute; // Résultat : 34
MyDate.Month; // Résultat : 1
MyDate.Second; // Résultat : 56
MyDate.Ticks; // Résultat : 635872484960000000
MyDate.Year; // Résultat : 2016
]]
```

bool

Propriétés ou méthodes	Type de retour	Description
ToString()	string	Retourne la valeur de l'objet en chaine de caractères : «True» ou «False»

timespan

Propriétés ou méthodes	Type de retour	Description
ToString()	string	Retourne la valeur de l'objet en chaine de caractères

Tableaux et collections

Introduction

Un tableau en DI# est en fait une carte ordonnée. Une carte est un type qui associe des valeurs à des clés. Ce type est optimisé pour différentes utilisations ; il peut être considéré comme un tableau, une liste, une table de hashage, un dictionnaire, une collection, une pile, une file d'attente et probablement plus. On peut avoir, comme valeur d'un tableau, d'autres tableaux, multidimensionnels ou non. Autrement dit, on peut instancier un tableau de n'importe quel datatype qui serait soit fixe (voir les types listés ci-dessus), soit dynamique (équivalent à *List<datatype>* en .NET).

Déclaration

On peut définir un tableau en DI# comme ceci :

```
{ value1, value2, value3 }
```

Exemple :

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

On peut définir une collection d'éléments ainsi :

```
{ fieldName1 : value1, fieldName2 : value2, fieldName3 : value3 }
```

Exemple :

```
datasource myDictionary  = { Prenom : "John", Nom : "Smith", Age : 30 };
```

Les valeurs listées sont des *Expressions*, dont la valeur peut elle-même contenir une expression. On peut combiner les deux comme dans cet exemple :

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

Il existe de façon d'instancier un tableau :

- soit en créer un tableau de long fixe : `int a[10];`
- soit en créant un tableau de longueur variable : `string b[] ;`

Exemples :

```
string a[] = {"texte1", "texte2", "texte3"};

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, 2016.03.03
03:03:03};

bool e[] = {true, false, null};

timespan g[] = {1472500, 1472500};

datasource h[] = {123, "texte1", 1.111, 2016.01.01 01:01:01, true};
```

Instanciation

On peut assigner des valeurs initiales en déclarant l'array :

```
int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
string b[] = { "texte 1", "test", "abc" };
```

Lorsqu'une variable est déclarée de type tableau, cette dernière ne peut accepter que des valeurs du type de données du tableau. Exemple : `int a[10]` n'acceptera que des int.

Les opérateurs

Les opérateurs de comparaisons

Opérateurs simple

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

Exemple	Nom	Résultat
a == b	Egal	TRUE si a est égal à b après le transtypage.
a != b	Différent	TRUE si a est différent de b après le transtypage.
a < b	Plus petit que	TRUE si a est strictement plus petit que b.
a <= b	Plus petit ou égale à	TRUE si a est plus petit ou égale à b.
a > b	Plus grand que	TRUE si a est strictement plus grand que b.
a >= b	Plus grand ou égale à	TRUE si a est plus grand ou égale à b.

Contrairement à certains langages comme PHP, ces opérateurs ne peuvent être utilisés sur des types textuels. Ainsi, le DI# ne fait donc pas de comparaisons du style « ordre alphabétique » ou « if ("abc" < "bcd")» serait vrai par exemple.

Opérateurs complexes

- LIKE (ou son contraire NOTE LIKE) :

L'opérateur LIKE permet d'effectuer une recherche sur un modèle particulier. Il est par exemple possible de rechercher si valeur d'une chaine de caractère commence par telle ou telle lettre.

- LIKE "%a" : le caractère « % » est un caractère joker qui remplace tous les autres caractères. Ainsi, ce modèle permet de rechercher toutes les chaines de caractère qui se termine par un « a ».
- LIKE "a%" : ce modèle permet de rechercher toutes les lignes de « colonne » qui commence par un « a ».

- LIKE "%a%" : ce modèle est utilisé pour rechercher tous les enregistrements qui utilisent le caractère « a ».
- LIKE "pa%on" : ce modèle permet de rechercher les chaines qui commence par « pa » et qui se terminent par « on », comme « pantalon » ou « pardon ».

Note : Utiliser l'opérateur LIKE sans modèle sur une chaine de caractère produit le même effet que d'utiliser l'opérateur égal : ==

➢ CONTAINS (ou son contraire NOT CONTAINS) :

L'opérateur CONTAINS permet d'effectuer une recherche sur un modèle particulier. Il est par exemple possible de rechercher si valeur est contenu dans une chaine de caractère.
Note : Utiliser l'opérateur CONTAINS sur une chaine de caractère produit le même effet que d'utiliser l'opérateur LIKE avec ce modèle "%a%".

Les opérateurs LIKE et CONTAINS ne s'appliquent que sur des chaines de caractères. Autrement dit, il n'est pas possible d'utiliser CONTAINS pour savoir si une valeur est contenue dans un tableau par exemple.

➢ IS NULL (ou son contraire IS NOT NULL) :

L'opérateur IS NULL permet de savoir si la variable contient une valeur.

Exemples :

```
string a = "ceci est un test";

output.write(a contains "ceci" ? true : false);
// Affiche True

output.write(a like "ceci" ? true : false);
// Affiche False

output.write(a like"ceci%" ? true : false);
// Affiche True

output.write(a is null ? true : false);
// Affiche False
```

Les opérateurs arithmétiques

Opérateurs supportés :

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

Exemple :

Exemple	Nom	Résultat
a + b	Addition	Somme de a et b.
a - b	Soustraction	Différence de a et b.
a * b	Multiplication	Produit de a et b.
a / b	Division	Quotient de a et b.
a % b	Modulo	Reste de a divisé par b.

Note : la négation d'une valeur n'est pas supportée. Ainsi l'expression «-b» ne compilera pas. Ainsi une soustraction écrite ainsi «`a-b`» ne compilera pas également. Il est donc important de toujours laisser un espace entre l'opérateur de soustraction et la valeur. L'expression «`a - b`» compilera

Les opérateurs de tableaux

Opération unique

- Ajouter un item à une position fixe :
```
x[0] = 123; // ajouter l'entier 123 à la position 0 du tableau
```

- Ajouter un item à la fin d'un tableau dynamique
```
x[] += "test"; // ajoute "test" à la fin du tableau
```

- Supprimer un item d'un tableau dynamique
```
x[] -= "test"; // efface toutes les instances égales à "test" du tableau
```

Opération en lot

- Assigner plusieurs valeurs en une seule instruction :

- ```
  x = { "blue", "red", "green" };
  // x contient maintenant blue, red et green
  ```
  Attention, cette instanciation ne fonctionnera que si toutes valeurs passées sont bien du même type que le tableau, ici des chaînes de caractères.

- Ajouter des éléments à la fin d'un tableau dynamique:
  ```
  x[] += { "blue", "pink", "blue" };
  // x contient maintenant blue, red, green, blue, pink et blue
  ```

- Effacer toutes les instances d'une ou plusieurs valeurs données d'un tableau dynamique:
  ```
  x[] -= { "blue", "pink"};
  // x contient maintenant red et green
  ```

Portée des variables (scope)

La portée d'une variable dépend du contexte dans lequel la variable est définie. Une variable définie à la racine du code aura une portée sur la totalité du script. Mais, une variable définie dans une fonction sera locale à la fonction. Autrement dit, toute variable créée à une portée limitée à sa propre fonction, ainsi qu'à ses fonctions sous-jacentes.

Exemple :

```
[[
int x = 123; // portée globale

string MyFunction(string value)
{
    int y = 456; // portée locale
    x = x + 1;// utilisation possible d'une variable globale dans la fonction
}
]]
```

Le contexte est quant à lui délimité par **{** et **}**. Ainsi, la déclaration d'une variable à l'intérieur de ces accolades, en fera une variable locale.
```

Expressions de requête

Les expressions de requête peuvent être utilisées pour interroger et transformer des données à partir de n'importe quelle source de données complexe. Ces requêtes servent à manipuler, trier ou filtrer un tableau (ou dictionnaire, liste, etc.).

L'objectif est, à partir d'un tableau, de créer une nouvelle source de données qui :

- Filtre optionnellement le tableau original (clauses "Where")
- Trie optionnellement le tableau original (clauses "Order by")
- Extrait une propriété spécifique des items du tableau original
- Dédoublonne optionnellement les résultats

Utilisation :

Élément de requêtes	description
Select	Permet de sélectionner les valeurs à sélectionner parmi la source de données Il est possible d'utiliser l'option **distinct** afin de ne sélectionner que les valeurs distinctes parmi la source de données.
From	Permet de choisir sur quelle source de données la requête sera appliquée
Where	Permet de restreindre les données sélectionnées avec une clause d'inclusion ou d'exclusion.
Order by	Permet de trier les données selon un ou plusieurs champs par ordre ascendant ou descendant. Mots-clés supportés : – asc – ascending – desc – descending

Concrètement, les expressions de requêtes s'utilise comme ceci :

```
select identifier.propriete3
from identifier in expression
where identifier.propriete1 = valeur
order by (identifier.propriete2 asc)
```

Avec :

- Expression : la structure dans laquelle on veut récupérer un élément (un tableau, une collection, etc.)
- Identifier : le nom de la variable qui va récupérer l'élément du tableau sélectionné par la clause Where

Exemples

Supposons la source de données suivante :

```
[[
      datasource Vehicules =
      {
          { Marque: "Toyota", Modele: "Sienna", Annee: 2015},
          { Marque: "Ford", Modele: "Fusion", Annee: 2013},
          { Marque: "Chevrolet ", Modele: "Equinox", Annee: 2005 },
          { Marque: "Hyundai", Modele: "Accent", Annee: 2016 },
          { Marque: "Mazda", Modele: "CX-5", Annee: 2015 },
          { Marque: "Hyundai", Modele: "Accent", Annee: 2010 },

      };
]]
```

- Pour sélectionner toutes les marques de tous les véhicules :

```
[[
datasource Marques =   select voiture.Marque
                       from voiture in Vehicules;
/*
le tableau Marques contient :
Toyota
Ford
Chevrolet
Hyundai
Mazda
Hyundai
*/
]]
```

- Pour sélectionner toutes les marques uniques (sans doublon) de tous les véhicules :

```
[[
datasource Marques =   select distinct voiture.Marque
                       from voiture in Vehicules;
/*
le tableau Marques contient :
Toyota
Ford
Chevrolet
```

```
Hyundai
Mazda
*/
]]
```

- Pour sélectionner toutes les marques uniques (sans doublon) de tous les véhicules ordonnés du modèle le plus récent au plus ancien :

```
[[
datasource Marques =   select distinct voiture.Marque
                       from voiture in Vehicules
                       order by (voiture.Annee desc);
/*
le tableau Marques contient :
Hyundai
Mazda
Toyota
Ford
Chevrolet
*/
]]
```

- Pour sélectionner toutes les marques uniques (sans doublon) de tous les véhicules ordonnés du modèle le plus récent au plus ancien, en excluant les modèles avant 2014 :

```
[[
datasource Marques =   select distinct voiture.Marque
from voiture in Vehicules
where (voiture.Annee > 2014)
order by (voiture.Annee desc);
/*
le tableau Marques contient :
Hyundai
Mazda
Toyota
*/
]]
```

Note : il est possible de cloner un tableau en sélectionnant tous items de ce dernier :

```
[[
datasource VehiculesCopie = select voiture from voiture in Vehicules;
]]
```

Notes importantes

Bien que ces fonctionnalités de requête et recherche de données dans les sources disponibles, bien que très puissantes, imposent des pénalités de performance importantes.

Si utilisées de manière incorrecte dans un message, par exemple, ces requêtes peuvent avoir un impact important sur la vitesse de production des messages – considérez par exemple que toutes requête faite dans un message transmis à un million de contact devra être traitée un million de fois.

N'hésitez pas à communiquer avec notre équipe de support ou nos analystes pour des conseils lors de la création de messages nécessitant ce degré de complexité.

Les structures de contrôle

if

L'instruction if est une des plus importantes instructions de tous les langages. Elle permet l'exécution conditionnelle d'une partie de code. Les fonctionnalités de l'instruction if sont les mêmes qu'en C :

```
if (expression)
   commande ;
```

ou

```
if (expression)
{
   Commandes 1 ;
   Commandes 2 ;
}
```

L'instruction if converti son expression en sa valeur booléenne. Si l'expression vaut TRUE, DI# exécutera l'instruction et si elle vaut FALSE, l'instruction sera ignorée.

L'exemple suivant affiche la phrase « *a est plus grand que b* » si a est plus grand que b :

```
[[
int a = 1;
int b = 2;
if (a > b)
  output.write("a est plus grand que b");
]]
```

Vous pouvez imbriquer indéfiniment des instructions if dans d'autres instructions if, ce qui permet une grande flexibilité dans l'exécution d'une partie de code suivant un grand nombre de conditions.

else

Souvent, vous voulez exécuter une instruction si une condition est remplie, et une autre instruction si cette condition n'est pas remplie. C'est à cela que sert else.
else fonctionne après un if et exécute les instructions correspondantes au cas où l'expression du if est FALSE. Dans l'exemple suivant, ce bout de code affiche « *a est plus grand que b* » si la variable a est plus grande que la variable b, et « *a est plus petit que b* » dans le cas contraire :

```
[[
int a = 1;
int b = 2;
if (a > b) {
  output.write("a est plus grand que b");
} else {
  output.write("a est plus petit que b");
}
]]
```

if/else raccourci

Comme en C#, il est possible d'utiliser un raccourci à l'instruction if/else en utilisant l'opérateur conditionnel **? :** tel que :

```
(condition) ? (expression_si_vrai) : (expression_si_faux);
```

Exemple :

```
[[
int a = 1;
int b = 2;
output.write((a > b) ? "a est plus grand que b" : "a est plus petit que b");
]]
```

Note : Dû à des limitations du compilateur DI#, assurez-vous de placer la condition entre parenthèses lorsque vous utilisez cette forme.

while

La boucle while est le moyen le plus simple d'implémenter une boucle en DI#. Cette boucle se comporte de la même manière qu'en C. L'exemple le plus simple d'une boucle while est le suivant :

```
while (expression)
   commande ;
```

ou

```
while (expression)
{
   Commandes 1 ;
   Commandes 2 ;
}
```

L'exemple suivant affiche tous les nombres de 1 jusqu'à 10 :

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

Note : Portez attention à la gestion de vos variables qui contrôlent la boucle pour ne pas créer une boucle sans fin. Par mesure de protection, l'exécution des boucles *while* en DI# prennent fin automatiquement dès que 250 itérations sont atteintes.

foreach

Itération sur un tableau

La structure de langage foreach fournit une façon simple de parcourir des tableaux.

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

ou

```
foreach (identifier in expression)
{
    commande 1 ;
    commande 2 ;
}
```

Dans cette structure, «identifier» est le nom que l'on choisit pour la qui itère les valeurs du tableau retourné par l' «expression». Cette variable doit être du même type que les valeurs du tableau.

L'exemple suivant montre comment afficher tous les éléments d'un simple tableau de nombres entiers :

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

Utilisation d'une expression complexe

Puisqu'un *foreach* permet d'itérer n'importe quel tableau issu d'une expression, il est possible d'utiliser des expressions complexes directement dans la structure du *foreach*, comme par exemple une expression de requête (voir leurs utilisations dans la partie précédente).

Exemple :

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

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

Itération avec compteur

Comme dans de nombreux langages, il est possible d'entretenir un compteur pendant que l'on itère les valeurs de l'expression :

```
foreach (compteur => identifier in expression)
    commande ;
```

Dans cette structure, «compteur» est le nom d'une variable qui est un nombre entier initialisé à 0 qui s'incrémentera à chaque itération de l'expression. Ceci permet d'avoir en tout temps

```
[[
int myArray[] = { 1, 2, 3, 4, 5};
foreach(itemOrder => number in myArray)
{
    output.write("position : " + itemOrder);
    output.write("valeur : " + number);
}
]]
```

Note : tout comme pour la boucle *while*, une boucle *foreach* s'interrompt automatiquement après 250 itérations.

break

L'instruction break permet de sortir d'une structure itérative foreach ou while. L'exemple suivant permet de «casser» la boucle du foreach afin de n'afficher que les 3 premiers nombres :

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

switch

L'instruction switch équivaut à une série d'instructions if. En de nombreuses occasions, vous aurez besoin de comparer la même variable (ou expression) avec un grand

nombre de valeurs différentes, et d'exécuter différentes parties de code suivant la valeur à laquelle elle est égale. C'est exactement à cela que sert l'instruction switch.

Utilisation :

```
Switch (expression)          ou
{
      case valeur :
            commande;
}
```

```
Switch (expression)
{
      case valeur :
      {
            commande 1;
            commande 2;
      }
}
```

Les deux exemples suivants sont deux manières différentes d'écrire la même chose, l'une en utilisant une série de *if*, et l'autre en utilisant l'instruction *switch* :

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

```
switch (i)
{
    case 0:
        output.write("i égal 0");
    case 1:
        output.write("i égal 1");
    case 2:
```

```
    {
        output.write("i égal 2");
        output.write("i égal 2");

    }
}
]]
```

Contrairement à d'autres langages, il n'est pas nécessaire d'utiliser l'instruction *break* dans un *switch* de DI#. En effet dès qu'un des cas est respecté, ses instructions seront exécutées et le compilateur sortira immédiatement du switch. Notez qu'il n'y a également pas de *default* dans le DI#, ainsi si aucun cas n'est respecté, le switch n'exécutera aucune instruction.

continue

L'instruction continue est utilisée dans une boucle afin d'éluder les instructions de l'itération courante et de continuer l'exécution à la condition de l'évaluation et donc, de commencer la prochaine itération. L'exemple ci-dessous affiche tous les nombres entiers contenus dans le tableau à l'exception du nombre 3 car l'instruction continue aura pour conséquence de passer à l'instruction suivante, avant d'afficher le nombre 3 :

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

Un *continue* arrête donc immédiatement les instructions de la boucle et passe à la prochaine itération.

return

L'instruction *return* retourne le contrôle du programme au module appelant. L'exécution reprend alors à l'endroit de l'invocation du module.

- Si appelée depuis une fonction, la commande return termine immédiatement la fonction, et retourne l'argument qui lui est passé.
- Si appelée depuis l'environnement global, l'exécution du script est interrompue.

Return accepte une valeur de retour optionnel. Ceci permet d'assigner à une variable le résultat de ce qui a mis fin au script.

Dans l'exemple suivant l'instruction *return*, en plus d'arrêter le l'exécution de la fonction *checkNumber* retourne une valeur qui peut être assigné directement à une variable :

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

string result = checkNumber(1);

output.write(result);
// Affiche "x égal 1"
]]
```

Note : si l'instruction *return* est utilisée à la racine du script, ceci stop l'exécution complète du script et la valeur retournée par *return* peut dans certain cas remplacer ce que le script aurait dû afficher.

Les fonctions

Déclaration

Une fonction DI# se déclare de la même façon qu'en C. C'est-à-dire :

```
[Type] [Nom] ([Paramètre1], [Paramètre2], etc.)
{
      Instructions…
}
```

Les fonctions doivent être définies à la racine du template, c'est-à-dire qu'elles ne peuvent pas être définies à l'intérieur d'un scope.

- Exemple d'une déclaration classique, avec valeur de retour définit :

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

- Exemple d'une déclaration avec une valeur par défaut :

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

Note :

- Une fonction peut retourner n'importe quel type (string, int, decimal, etc.). Dans ce cas, la fonction doit nécessairement retourner une valeur. Cette valeur doit être obligatoirement du même type que le retour attendu par la fonction.
- Une fonction peut aussi être déclarée de type void. Dans ce cas, la fonction est utilisable comme simple instruction. L'exécution de cette fonction doit se terminer par un « return » (sans valeur de retour) ou bien atteindre la fin de la fonction.

Appel

- Une fonction s'appelle comme dans tous les autres langages :
  nom(params) ou *nom()*

  Une fonction peut être utilisée comme statement/commande dans le script, par exemple :
  ```
  OutputWithTags("mon texte", "i");
  ```

- Les fonctions qui retournent une valeur (i.e. toutes les fonctions dont le type n'est pas void) peuvent aussi être utilisées en tant qu'expression ou valeur :
  ```
  int x = addNumbers(1,1);
  ```

Portée des variables

Comme dans de nombreux langages, les variables créées à l'intérieur d'une fonction (les variables locales) ne sont pas visibles à l'extérieur de cette fonction. Les variables créées à l'extérieur de cette fonction ne quant à elles, pas accessible de l'intérieur de cette fonction.

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

Courriel : info@dialoginsight.com
Site Web : www.dialoginsight.com
Blogue : academie.dialoginsight.com

@DialogInsight
Dialog Insight