Les opérateurs

Les opérateurs sont des mots-clés du langage permettant l'exécution de traitements sur des variables, valeurs de retour de fonctions ou valeurs littérales (les opérandes).

La combinaison d'opérateurs et d'opérandes constitue une expression qui sera évaluée à l'exécution et retournera un résultat dépendant des différents éléments qui la composent.

Nous avons déjà vu l'opérateur d'affectation = qui permet d'assigner une valeur à une variable. Celle-ci peut être une variable, une valeur littérale, le résultat d'une fonction ou le résultat d'une expression.

Les différents autres opérateurs peuvent être classés en six familles relatives au type de traitement qu'ils permettent d'effectuer.

1. Les opérateurs d'accès

Le langage C# définit plusieurs modes d'accès aux données des objets qu'il permet de manipuler. Chacun de ces modes utilise un opérateur particulier.

a. Accès simple : . (point)

C# utilise le symbole . (point) pour permettre l'accès aux membres d'un objet, ou éventuellement d'un type.

```
//Ici, on utilise l'opérateur . pour accéder
//au membre "Nom" de la variable "personne"

string nomEleve = personne.Nom;
```

b. Accès indexé : [ ]

L'opérateur d'accès indexé offre la possibilité d'accéder à une valeur contenue dans une variable en fournissant un index. L'utilisation de variables de type tableau implique généralement l'utilisation de cet opérateur pour la lecture ou la modification des données qu'il contient.

```
int[] tableau = new int[10];
tableau[0] = 123 ;
```

c. Accès avec nullité conditionnelle : ?

Avec C#, l'accès à un membre d'un objet dont la valeur est `null` déclenche une erreur d'exécution.

```
//Si la variable personne vaut null,
//une erreur d'exécution est déclenchée

string nomEleve = personne.Nom;
```

Il arrive donc souvent que le code accédant à un membre « à risque » soit encapsulé dans un bloc qui teste la

valeur de l'objet principal avant d'accéder à son membre.

Cette technique peut s'avérer lourde, notamment lorsqu'il est nécessaire d'imbriquer plusieurs tests pour accéder à une donnée. C# 6 apporte parmi ses nouveautés la possibilité d'utiliser l'opérateur ? pour simplifier l'écriture de ce type de code. Il peut en effet être combiné à l'opérateur d'accès simple ou l'opérateur d'accès indexé pour retourner la valeur `null` si l'objet « père » vaut `null`.

```
string nomEleve = personne?.Nom;
```

Dans cet exemple, si la variable `personne` est initialisée, la variable `nomEleve` contient la valeur de `personne.Nom`. En revanche, si la variable `personne` vaut `null`, aucune erreur d'exécution n'est levée et la variable `nomEleve` a pour valeur `null`.

2. Les opérateurs arithmétiques

Les opérateurs arithmétiques permettent d'effectuer des calculs sur leurs opérandes.

Opérateur	Nom de l'opération	Exemple	Résultat
+	Addition	1 + 2	3
-	Soustraction	1 - 2	-1
*	Multiplication	1 * 2	2
/	Division	1 / 2	0.5
%	Modulo (reste de la division entière)	1 % 2	1

Ces opérateurs sont principalement utilisés avec des opérandes numériques. Certains types modifient le comportement de ces opérateurs afin d'exécuter des traitements plus en accord avec leur logique. C'est notamment le cas du type `string` pour lequel l'opérateur d'addition permet d'effectuer une concaténation, c'est-à-dire de joindre deux chaînes de caractères.

3. Les opérateurs de comparaison

Les opérateurs de comparaison permettent de définir des expressions dont le résultat est une valeur booléenne. Ils sont principalement utilisés pour l'évaluation de conditions dans les structures de contrôle.

Opérateur	Nom de l'opération	Exemple	Résultat
==	Égalité	10 == 20	false
!=	Inégalité	"C#" != "VB.NET"	true
>	Supériorité	10 > 20	false
>=	Supériorité ou égalité	10 >= 20	false
<	Infériorité	10 < 20	true
<=	Infériorité ou égalité	10 <= 20	true
is	Comparaison de type	"Hello world" is string	true

4. Les opérateurs conditionnels

Deux opérateurs du langage simplifient l'écriture du code en permettant d'éliminer des blocs de code liés à des branchements conditionnels.

a. Opérateur ternaire : ? ... :

L'opérateur ternaire renvoie une valeur en fonction d'une expression booléenne. Il accepte pour cela trois opérandes qui sont, dans l'ordre :

- une expression booléenne,
- une expression dont la valeur est renvoyée si l'expression booléenne vaut `true`,
- une expression dont la valeur est renvoyée lorsque l'expression booléenne vaut `false`.

Le type de données renvoyé par les deux dernières expressions doit être identique. Dans le cas contraire, le compilateur génère une erreur de compilation.

Le format d'une expression utilisant l'opérateur ternaire est le suivant :

```
<expression booléenne> ? <expression à évaluer si true> :
<expression à évaluer si false>
```

L'assignation d'une chaîne de caractères en fonction d'un nombre peut être effectuée à l'aide de l'opérateur ternaire de la manière suivante :

```
int nbClients = 98;
string libelle = nbClients <= 100 ? nbClients.ToString() +
"clients" : "Plus de 100 clients";

Console.WriteLine(libelle);    //Affiche "98 clients"

nbClients = 120;
libelle = nbClients <= 100 ? nbClients.ToString() + "clients" :
"Plus de 100 clients";

Console.WriteLine(libelle);    //Affiche "Plus de 100 clients"
```

b. Opérateur de fusion de valeur nulle : ??

Cet opérateur accepte deux opérandes et renvoie le premier des deux dont la valeur n'est pas `null`.

```
string ville = null;
string libelle = ville ?? "non définie";

Console.WriteLine(libelle);    //Affiche "non définie"

ville = "LYON";
libelle = ville ?? "non définie";

Console.WriteLine(libelle);    //Affiche "LYON"
```

5. Les opérateurs logiques

À l'exception de l'opérateur !, les opérations logiques permettent de combiner plusieurs expressions renvoyant un booléen, comme des expressions de comparaison.

a. Négation : !

L'opérateur Négation permet d'inverser une valeur booléenne. Cette valeur peut être représentée par un littéral, une variable ou une expression renvoyant une valeur booléenne.

```
bool booleen1 = true;
bool booleen2 = !booleen1;
//Arrivé ici, booleen2 vaut false

bool comparaison = !(booleen1 == booleen2);
// booleen1 == booleen2 renvoie false, donc comparaison vaut true
```

b. ET logique : &

L'opérateur ET logique permet de combiner deux expressions afin de déterminer si elles sont toutes deux vraies.

```
bool booleen1 = true;
bool booleen2 = false;

bool comparaison1 = booleen1 & booleen2;
// comparaison1 vaut false

booleen2 = true;
bool comparaison2 = booleen1 & booleen2;
// comparaison2 vaut true
```

c. OU logique : |

L'opérateur OU logique permet de combiner deux expressions afin de déterminer si au moins une des deux expressions est vraie.

```
bool booleen1 = true;
bool booleen2 = false;

bool comparaison = booleen1 | booleen2;
// comparaison vaut true
```

d. OU exclusif : ^

L'opérateur OU exclusif permet de combiner deux expressions afin de déterminer si une seule de ces expressions est vraie.

```
bool booleen1 = true;
bool booleen2 = false;

bool comparaison1 = booleen1 ^ booleen2;
// comparaison1 vaut true

booleen2 = true;
bool comparaison2  = booleen1 ^ booleen2;
// comparaison2 vaut false
```

e. ET conditionnel : &&

L'opérateur ET conditionnel permet de combiner deux expressions afin de déterminer si elles sont toutes deux vraies.

Le second opérande passé à cet opérateur ne sera évalué que si le premier opérande est évalué comme vrai.

```
bool booleen1 = false;
bool booleen2 = true;

bool comparaison1 = booleen1 && booleen2;
// comparaison1 vaut false, et booleen2 n'est pas évalué

booleen1 = true;
bool comparaison2 = booleen1 && booleen2;
// comparaison2 vaut true
```

f. OU conditionnel : ||

L'opérateur OU conditionnel permet de combiner deux expressions afin de déterminer si au moins une des deux est vraie.

Le second opérande passé à cet opérateur ne sera évalué que si le premier opérande est évalué comme faux.

```
bool booleen1 = true;
bool booleen2 = true;

bool comparaison1 = booleen1 || booleen2;
// comparaison1 vaut true, et booleen2 n'est pas évalué

booleen1 = false;
bool comparaison2 = booleen1 || booleen2;
// comparaison2 vaut true
```

6. Les opérateurs binaires

Les opérateurs binaires ne peuvent être appliqués que sur des types numériques entiers. Ils effectuent sur leurs

opérandes des opérations logiques au niveau des bits.

a. ET binaire : &

L'opérateur ET binaire effectue un ET logique sur chacun des bits des opérandes. Le résultat de l'expression est une valeur de type `int`. Effectuons l'opération suivante :

```
int resultat = 21 & 57;
```

	Opérande 1	Opérande 2	Résultat
Base 10	21	57	17
Base 2 (valeur binaire)	00010101	00111001	00010001

b. OU binaire : |

L'opérateur OU binaire effectue un OU logique sur chacun des bits des opérandes. Le résultat de l'expression est une valeur de type `int`. Effectuons l'opération suivante :

```
int resultat = 21 | 57;
```

	Opérande 1	Opérande 2	Résultat
Base 10	21	57	61
Base 2 (valeur binaire)	00010101	00111001	00111101

c. OU exclusif : ^

L'opérateur OU exclusif binaire effectue un OU exclusif sur chacun des bits des opérandes. Le résultat de l'expression est une valeur de type `int`.

	Opérande 1	Opérande 2	Résultat
Base 10	21	57	44
Base 2 (valeur binaire)	00010101	00111001	00101100

d. Négation : ~

L'opérateur négation binaire inverse la valeur de chacun des bits de son opérande. Effectuons l'opération suivante :

```
int resultat = ~21;
```

	Opérande	Résultat
Base 10	21	-22
Base 2 (valeur binaire)	00010101	11111111111111111111111111101010

L'opérateur de négation renvoie une valeur du même type que son opérande : ici, `int`. Ce type est codé sur 32

bits, ce qui explique la longueur du résultat renvoyé.

e. Décalage vers la droite : >>

Cet opérateur décale vers la droite les bits composant le premier opérande du nombre de positions spécifiées par le second opérande. Effectuons l'opération suivante :

```
int resultat = 21 >> 2;
```

	Opérande 1	Opérande 2	Résultat
Base 10	21	2	5
Base 2 (valeur binaire)	00010101		00000101

f. Décalage vers la gauche : <<

Cet opérateur décale vers la gauche les bits composant le premier opérande du nombre de positions spécifiées par le second opérande. Effectuons l'opération suivante :

```
int resultat = 21 << 2;
```

	Opérande 1	Opérande 2	Résultat
Base 10	21	2	84
Base 2 (valeur binaire)	00010101		01010100