# Les outils fournis par Visual Studio

Les causes des erreurs de logiques peuvent être particulièrement délicates à déceler. Visual Studio fournit pour cela un certain nombre d'outils permettant de contrôler l'état d'exécution de l'application, de placer des points d'arrêt ou de visualiser les données manipulées par l'application.

## 1. Contrôle de l'exécution

Pour pouvoir explorer le code à la recherche de la cause d'une erreur d'exécution ou de logique, il est important de pouvoir contrôler l'exécution d'une application dans des conditions permettant cette exploration. Visual Studio intègre cette possibilité au travers de l'utilisation d'un débogueur.

Dans Visual Studio, un projet peut être dans trois états distincts :

- En mode conception : le développement est en cours.
- En exécution : le code est compilé et l'application est lancée et attachée au débogueur intégré.
- En pause : le débogueur a stoppé l'exécution de l'application entre deux instructions.

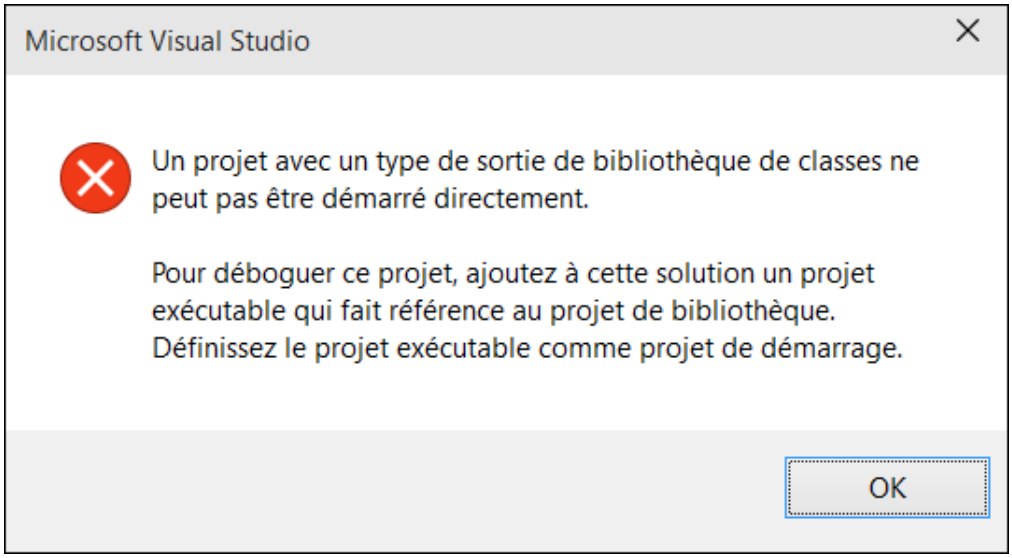
### a. Démarrage

Le lancement de l'application en mode débogage (Debug dans Visual Studio) est effectué en cliquant sur le bouton **Démarrer** de la barre d'outils.


Il est aussi possible de lancer le débogage en utilisant la touche [F5] du clavier.

La combinaison de touches [Ctrl][F5] lance l'application normalement, sans l'attacher au débogueur de Visual Studio. Le projet reste en mode conception, mais il est impossible de le compiler ou le lancer tant que l'application n'a pas été fermée.

Lorsque la solution contient plusieurs projets, le projet généré et lancé est celui qui a été défini comme projet de démarrage. Il est aussi nécessaire que ce projet génère un fichier exécutable. Dans le cas contraire, une erreur est affichée :




### b. Arrêt

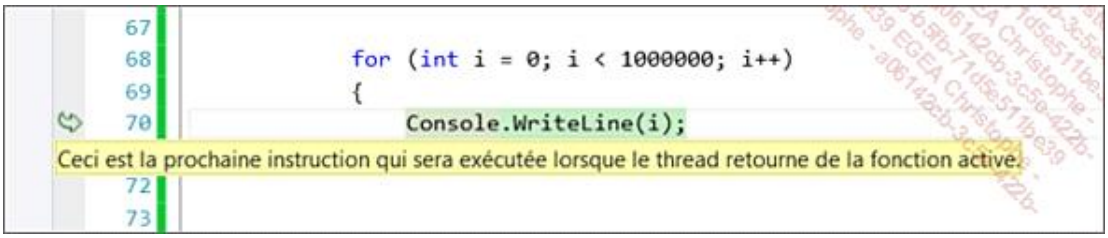
Lorsque le projet est en mode exécution ou pause, il est possible d'arrêter complètement son exécution afin de repasser en mode conception. Cette action est effectuée par un clic sur le bouton d'arrêt qui apparaît dans la barre d'outils en mode exécution ou pause.


L'infobulle placée sur ce bouton indique que la combinaison de touches [Shift][F5] arrête également l'exécution de l'application en mode débogage.

### c. Pause

Visual Studio permet de mettre en pause l'application en cours de débogage afin de permettre l'inspection du code exécuté. Cette action est disponible à l'aide du bouton **Pause** dans la barre d'outils.


Une fois l'application en pause, un repère est placé sur la prochaine ligne qui sera exécutée.




Cette méthode est relativement hasardeuse car il faut beaucoup de chance pour s'arrêter sur une ligne précise. Comme nous le verrons plus loin, l'utilisation de points d'arrêt est beaucoup plus efficace pour explorer le code en

cours d'exécution.

### d. Reprise

Une fois l'exécution de l'application interrompue, plusieurs options sont disponibles. Il est tout d'abord possible de reprendre le cours normal de l'exécution en cliquant sur le bouton **Continuer** ou en appuyant sur la touche [F5]. Il est aussi possible de poursuivre en mode pas à pas afin de suivre le cheminement de l'exécution.

Trois modes de pas à pas existent :

- le pas à pas principal ([F10]),
- le pas à pas détaillé ([F11]),
- le pas à pas sortant ([Shift][F11]).

Le pas à pas principal et le pas à pas détaillé sont quasiment identiques. La différence entre ces deux modes est dans la manière de gérer les appels de procédures ou fonctions. Lorsque l'application est arrêtée sur une ligne contenant une exécution de méthode, le pas à pas détaillé va permettre de rentrer dans cet appel et d'explorer son code. Le mode pas à pas principal exécutera quant à lui l'appel de fonction en un seul bloc.

Le pas à pas sortant permet quant à lui l'exécution de la méthode courante en un bloc et repasse le projet en mode pause sur la ligne suivant l'appel de la méthode.

Une dernière solution est d'exécuter le code courant jusqu'à la position définie par le curseur d'édition. Pour cela, il faut cliquer sur une ligne puis utiliser la combinaison de touches [Ctrl][F10]. Ceci peut s'avérer très pratique pour sortir de l'exécution d'une boucle, par exemple.

## 2. Points d'arrêt

Les points d'arrêt sont des outils indispensables pour le débogage d'une application C#. Ils offrent la possibilité au développeur de déterminer l'emplacement d'une ou plusieurs interruptions d'exécution dans le code, ce qui permet des scénarios de débogage complexes, nécessitant un grand nombre de manipulations, mais sans devoir exécuter la totalité du code en mode pas à pas.

Les points d'arrêt peuvent être conditionnels, c'est-à-dire qu'il est possible de spécifier une condition qui doit être respectée afin que le débogueur interrompe l'exécution.

Les TracePoints sont très similaires aux points d'arrêt, mais ils permettent de choisir une action à effectuer lorsqu'ils sont atteints. Cette action peut être le passage en mode pause et/ou l'affichage d'un message permettant de tracer l'exécution.

Points d'arrêt et TracePoints sont représentés dans Visual Studio par des icônes indiquant leur nature et leur état. Le tableau ci-dessous regroupe ces différentes icônes. Pour chacun des quatre premiers éléments, l'icône est déclinée en deux versions, la première correspondant à l'état actif, et la seconde à l'état désactivé.

Représente un point d'arrêt normal.

Représente un point d'arrêt conditionnel.

Représente un TracePoint normal.

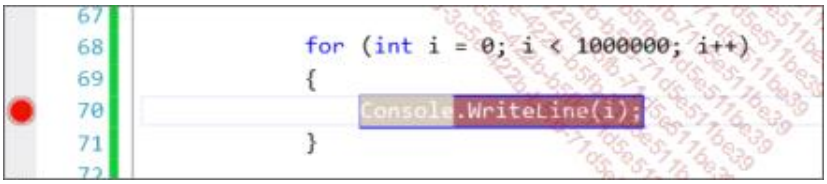
Représente un TracePoint avancé (avec une condition, un nombre de passages ou un filtre).

Point d'arrêt ou TracePoint désactivé à cause d'une erreur dans une condition.

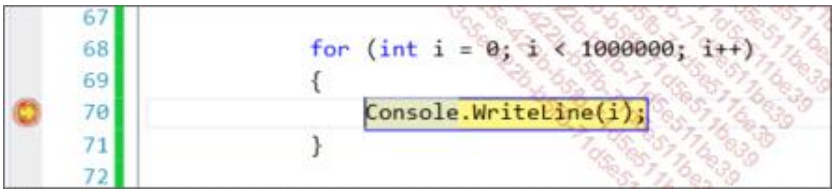
Point d'arrêt ou TracePoint désactivé à cause d'un problème temporaire.

## Utilisation d'un point d'arrêt

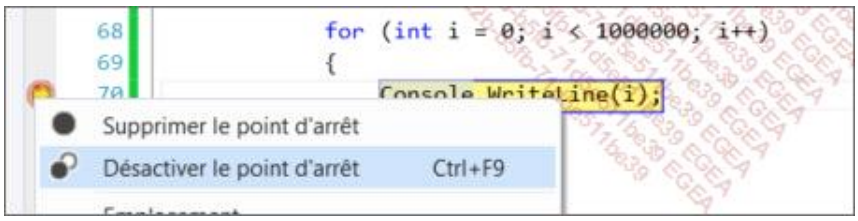
Pour placer un point d'arrêt, il suffit de placer le curseur d'édition sur une ligne de code et d'appuyer sur la touche [F9]. La ligne correspondante est surlignée en rouge et l'icône correspondant au point d'arrêt est affichée dans la marge de la fenêtre d'édition du code. Il est aussi possible de cliquer directement dans cette même marge pour définir un point d'arrêt.



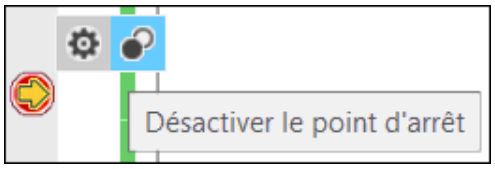

Lorsqu'un point d'arrêt est atteint lors du débogage, Visual Studio repasse au premier plan et la portion de code comprenant le point d'arrêt est affichée dans la fenêtre d'édition de code. La couleur de surlignage de la ligne passe du rouge au jaune. À ce stade, l'instruction n'a pas encore été exécutée, il est donc possible d'utiliser le pas à pas ou le pas à pas détaillé pour visualiser le comportement de cette instruction.




Pour désactiver le point d'arrêt, effectuez un clic droit sur l'icône du point d'arrêt pour ouvrir le menu contextuel puis sélectionnez **Désactiver le point d'arrêt**, ou utilisez la combinaison de touche [Ctrl][F9] lorsque le curseur d'édition est placé sur une ligne sur laquelle un point d'arrêt est défini.



Il peut également être désactivé en utilisant le menu rapide qui est affiché lorsque le curseur de la souris survole l'icône indiquant le point d'arrêt. Le second bouton de ce menu permet d'activer ou désactiver le point d'arrêt.

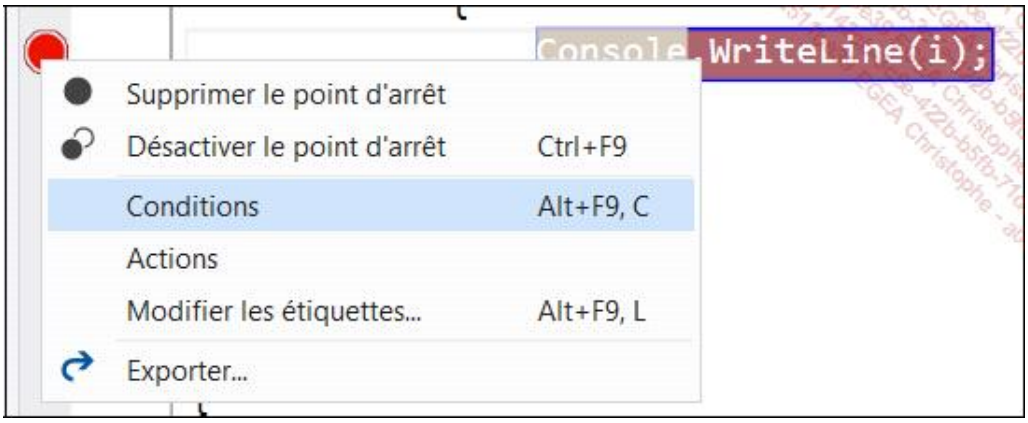



Un point d'arrêt est supprimé lorsque l'on clique sur l'icône qui lui est associée, ou à l'aide de l'option **Supprimer le point d'arrêt** du menu contextuel.

## Points d'arrêt conditionnels

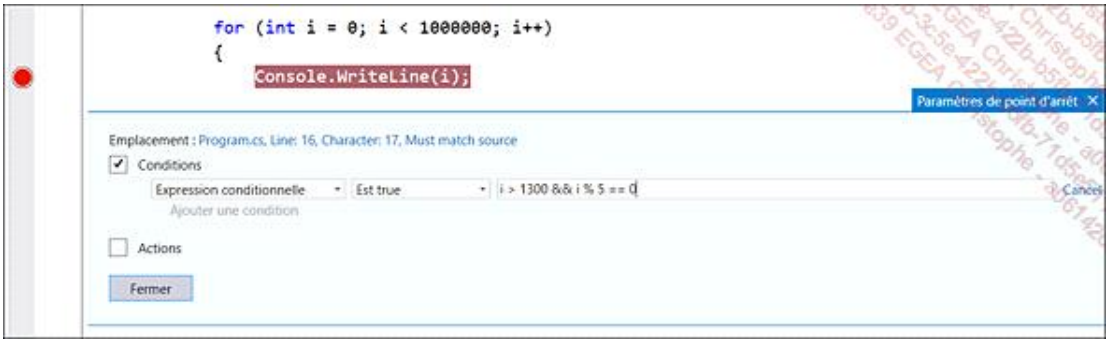
Il est possible de placer des conditions sur les points d'arrêt de manière à n'interrompre le flux d'exécution que dans des cas particuliers, lorsqu'un compteur de boucle atteint une valeur particulière par exemple.

Pour ajouter une condition à un point d'arrêt, ouvrez son menu contextuel à l'aide d'un clic droit sur l'icône, puis sélectionnez **Conditions**.




La fenêtre **Paramètres de point d'arrêt** intégrée à l'éditeur et permettant la modification des conditions s'ouvre alors. Il est possible d'y saisir un ou plusieurs prédicats C# évalués à chaque exécution de la ligne de code. Ceux-ci sont définis par trois éléments : un type, un mode d'évaluation et une expression.

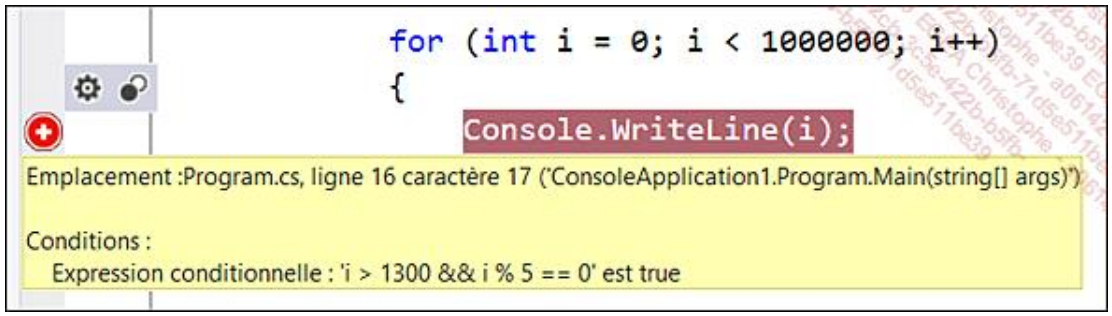
Par défaut, le prédicat attendu est de type **Expression conditionnelle** : c'est une expression C# renvoyant une valeur booléenne, comme une comparaison. Le mode d'évaluation initial est quant à lui **Est true**. Ces paramètres indiquent que l'exécution est interrompue lorsque le résultat de l'évaluation de l'expression vaut `true`.




Ici, la condition indique que le code ne doit s'interrompre que lorsque la variable `i` a une valeur supérieure à 1300 et

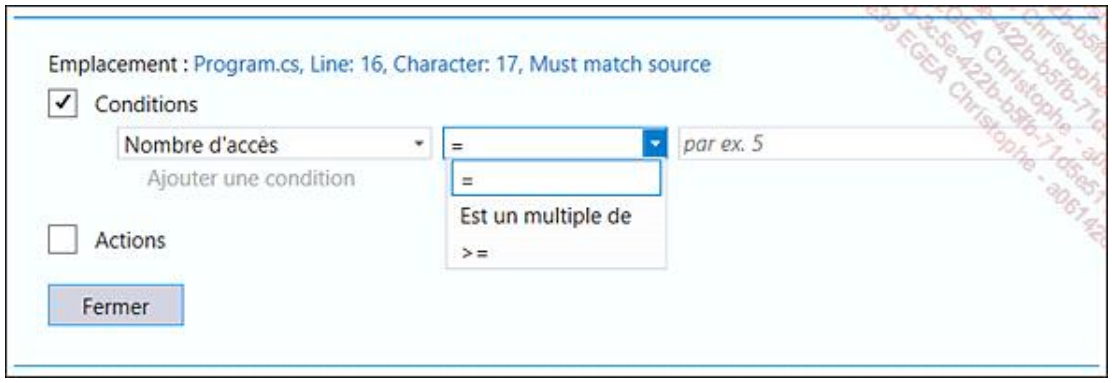
que cette valeur est un multiple de 5.

L'icône représentant le point d'arrêt change d'aspect, et l'infobulle correspondante présente la condition spécifiée.

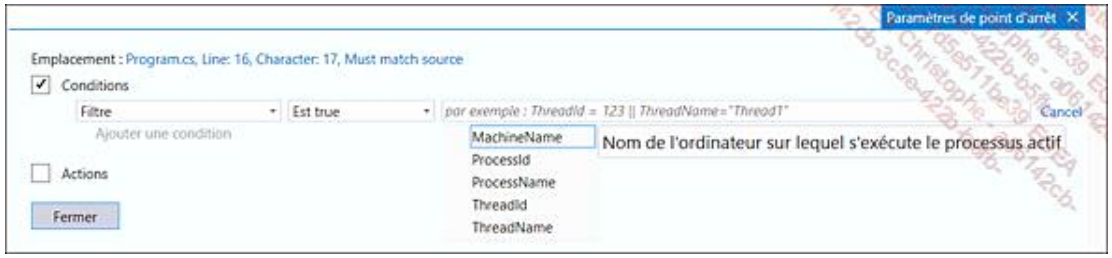



Le changement du mode d'évaluation par la valeur **en cas de modification** provoque une modification du comportement du point d'arrêt : l'exécution n'est interrompue que lorsque la valeur de retour de l'évaluation du prédicat est différente de l'évaluation précédente. Dans le cas décrit ici, l'évaluation renvoie `false` jusqu'à ce que la valeur de `i` soit 1304. Lorsqu'elle passe à 1305, l'évaluation de la condition renvoie `true` : le code est interrompu. Lorsque `i` est égal à 1306, l'évaluation de la condition vaut `false` : l'exécution est de nouveau interrompue. Mais à l'itération suivante, `i` vaut 1307, le prédicat a pour valeur `false` : le point d'arrêt n'est pas atteint.

Les conditions peuvent également évaluer des prédicats relatifs au nombre d'accès effectués à un point d'arrêt. Celui-ci ne s'active que lorsqu'il a été atteint un certain nombre de fois.

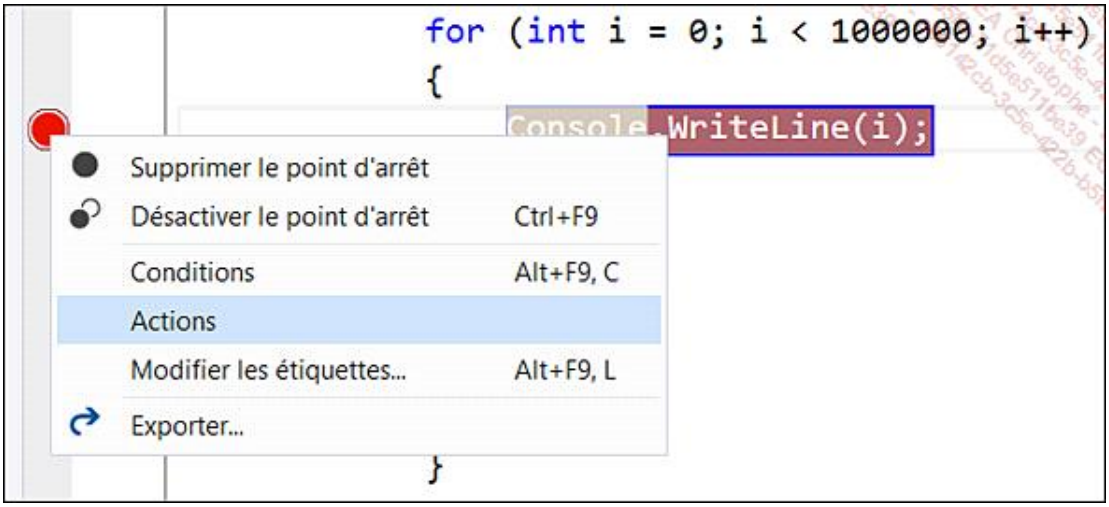



Un troisième type de prédicat est disponible : **Filtre**. Il offre des possibilités avancées de contrôle des points d'arrêt. L'utilisation de ce type de condition permet de spécifier que le point d'arrêt ne doit être activé que pour une machine particulière ou un thread particulier, notamment.

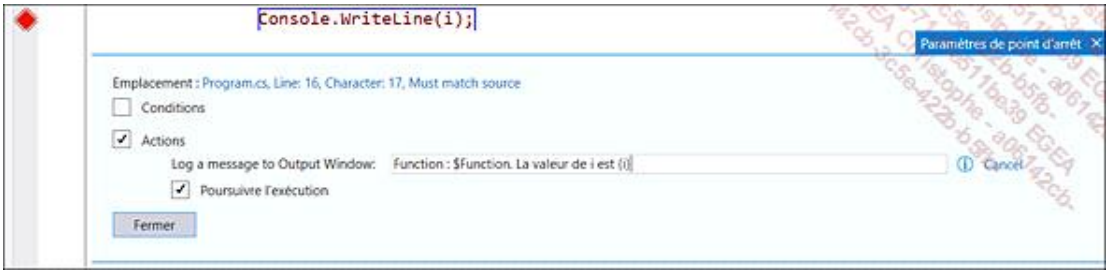



## TracePoints

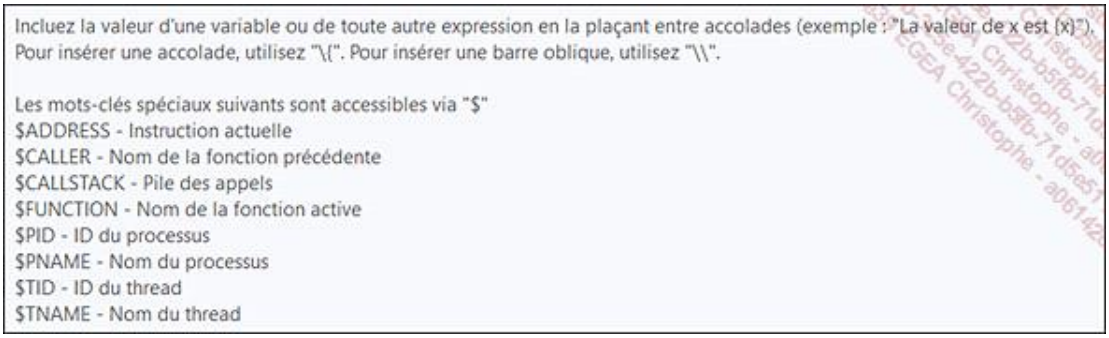
Les TracePoints peuvent être créés à l'aide de l'option **Actions** du menu contextuel d'un point d'arrêt.




L'utilisation de cette option ouvre elle aussi la fenêtre **Paramètres de point d'arrêt**. Dans cette configuration, elle permet de définir un message à afficher dans la fenêtre de sortie de Visual Studio à l'aide d'une macro. Pour que le point d'arrêt devienne vraiment un TracePoint, il faut que l'option **Continuer l'exécution** soit cochée.




Les différents mots-clés utilisables dans les macros sont visualisables en survolant l'icône **Information** située sur la droite du champ éditable.




Si une ou plusieurs conditions sont appliquées au TracePoint, celui-ci n'est activé que lorsque les conditions sont vérifiées. En conservant la condition créée plus haut, le début du résultat dans la fenêtre de sortie est le suivant :

```
Function: ConsoleApplication2.Program.Main(string[]), La valeur
de i est 1305
Function: ConsoleApplication2.Program.Main(string[]), La valeur
de i est 1310
Function: ConsoleApplication2.Program.Main(string[]), La valeur
de i est 1315
```

```
Function: ConsoleApplication2.Program.Main(string[]), La valeur
de i est 1320
```

## 3. Visualiser le contenu des variables

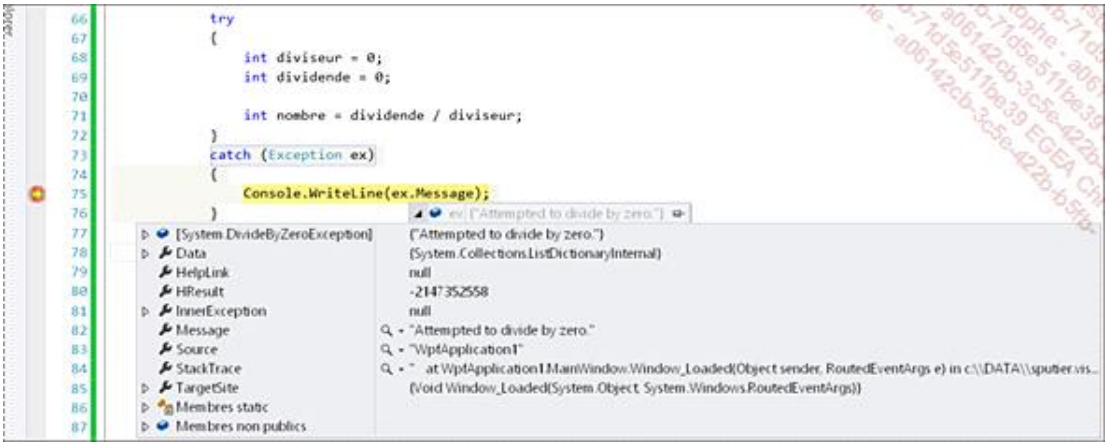
Le débogueur permet au développeur de suivre le fonctionnement de son application, instruction par instruction si nécessaire. Il est tout aussi important qu'il puisse inspecter le résultat d'une opération ou la valeur des paramètres passés à une méthode afin de comprendre la raison d'un résultat inattendu ou d'une erreur d'exécution.

Visual Studio fournit différents outils utilisables lorsque le projet est en mode pause pour accéder à ces valeurs.

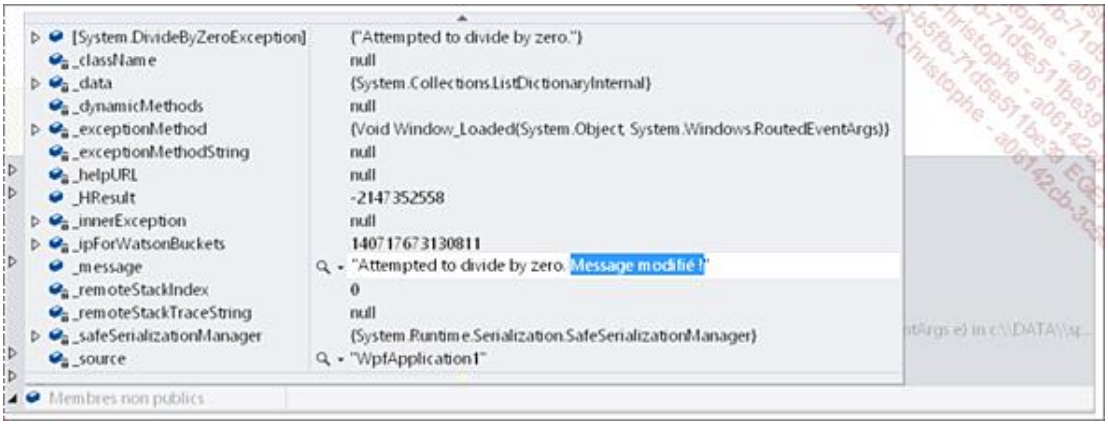
### a. DataTips

Les DataTips constituent le premier moyen d'accès au contenu de variables par leur simplicité et leur accessibilité. Il suffit en effet de placer le curseur de la souris sur un nom de variable accessible par le code sur lequel le débogueur est arrêté pour voir apparaître une petite fenêtre contenant la valeur de la variable.

Si la variable est d'un type complexe, cette fenêtre propose un bouton sous forme de triangle permettant d'inspecter ses propriétés.




Il est également possible de modifier les valeurs des variables affichées, ce qui peut s'avérer pratique pour valider un comportement.



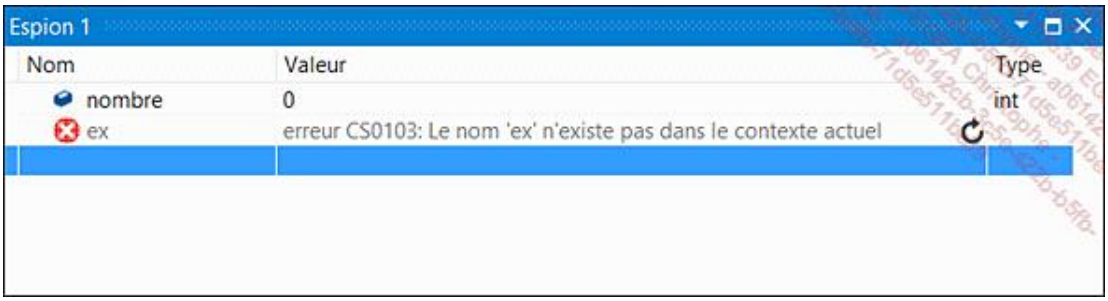

Les DataTips disparaissent automatiquement lorsque le curseur de la souris sort de leur surface.

### b. Fenêtres Espion

Les fenêtres **Espion 1** à **4** sont accessibles par le menu **Déboguer - Fenêtres - Espion** de Visual Studio. Ces fenêtres ont pour objectif de permettre la visualisation rapide et simultanée des valeurs de plusieurs variables.

Pour ajouter un espion pour une variable, il faut effectuer un clic droit sur un nom de variable et sélectionner l'option **Ajouter un espion**. Il est aussi possible d'ajouter une variable ou expression en saisissant son nom dans la colonne **Nom**.

Les valeurs des différentes expressions sont évaluées tout au long du débogage de l'application. Si l'une d'elles ne peut pas être évaluée, l'icône précédant l'exception montre une erreur, et la colonne **Valeur** affiche un message indiquant que l'expression n'existe pas dans le contexte actuel.



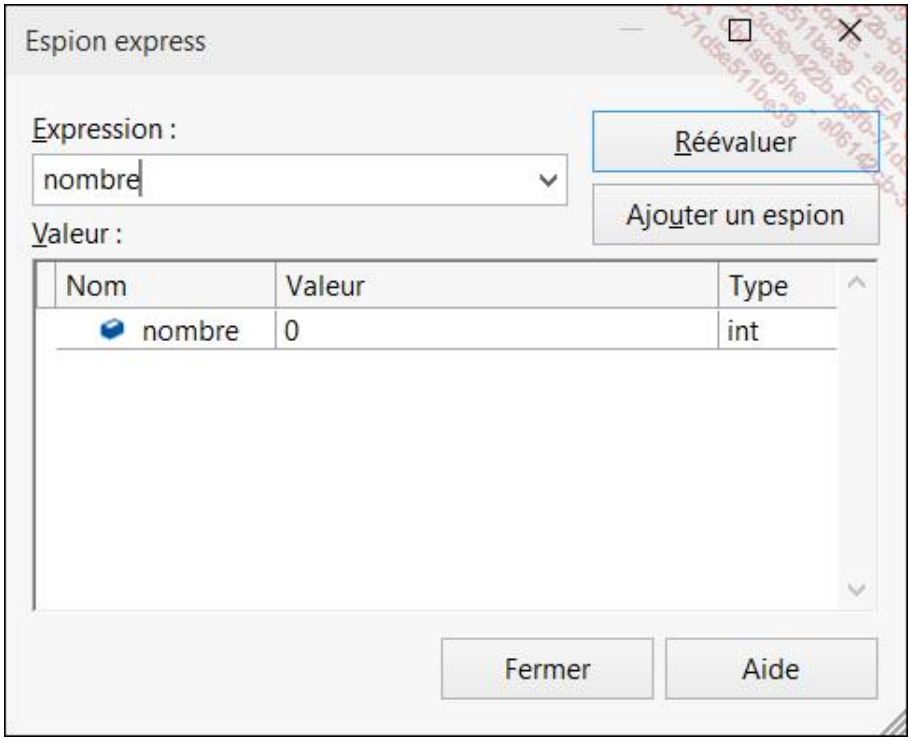

Tout comme dans un DataTip, il est possible de modifier la valeur d'une variable dans une fenêtre **Espion**.

La suppression d'un espion peut être effectuée en utilisant l'option **Supprimer un espion** avec un clic droit sur l'élément à supprimer, ou en utilisant la touche [Suppr].

### c. Fenêtre Espion express

La fenêtre **Espion express** utilise le même principe de fonctionnement que les fenêtres **Espion**. Elle ne permet en revanche la visualisation que d'une valeur à la fois, ce qui limite son intérêt par rapport à la fenêtre **Espion**. De plus, cette fenêtre est modale, ce qui implique que la fenêtre doit être fermée pour poursuivre le débogage.

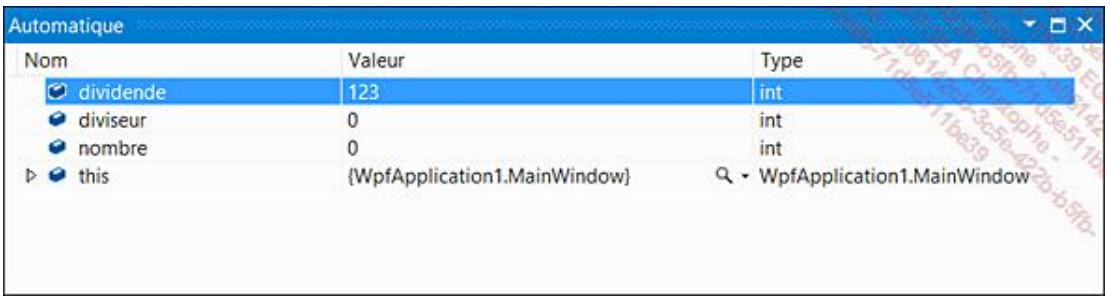
Elle est accessible via le menu **Déboguer - Espion express**, le raccourci-clavier [Ctrl] D, Q ou par l'utilisation de l'option **Espion express** lors du clic droit sur un nom de variable.




#### Fenêtre Automatique

La fenêtre **Automatique** présente une vision rapide des valeurs de variables utilisées dans l'instruction courante et l'instruction précédente. Elle est très similaire à une fenêtre **Espion**, mais ne permet pas l'ajout de valeurs à visualiser.

Elle est accessible par le menu **Déboguer - Automatique**, ou le raccourci-clavier [Ctrl] D, A.




Comme dans une fenêtre **Espion**, il est possible de modifier la valeur d'une variable en double-cliquant dans la colonne **Valeur**.

### d. Fenêtre Variables locales

Cette dernière fenêtre est identique à la fenêtre **Automatique**, mais les variables dont la valeur est affichée sont les variables locales de la procédure ou fonction en cours d'exécution.

Elle est accessible par le menu **Déboguer - Fenêtres - Variables locales**, ou par le raccourci [Ctrl] D, L.

## 4. Compilation conditionnelle

Bien qu'elle ne soit pas dédiée exclusivement au débogage, la compilation conditionnelle est un outil très intéressant pour suivre et diagnostiquer les problèmes, notamment en phase de développement.

Le principe de la compilation conditionnelle est simple : on intègre ou non une portion de code à l'assembly généré en fonction d'une condition prédéfinie.

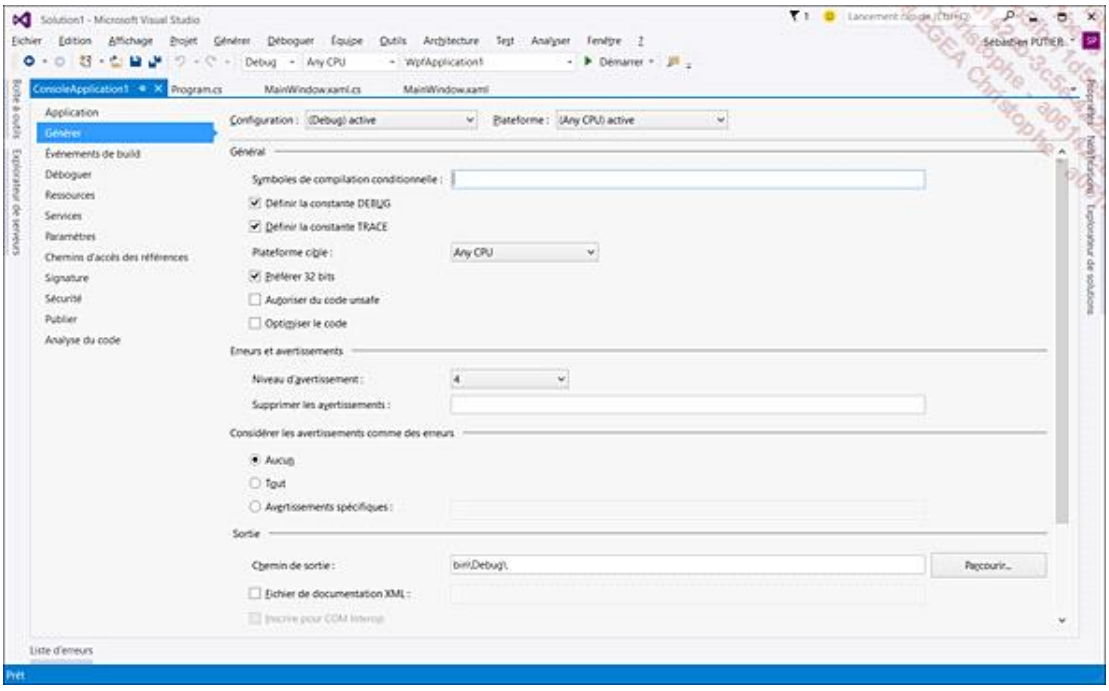
Cette condition porte obligatoirement sur l'existence ou non d'une ou plusieurs constantes de compilation.

Les constantes de compilation n'ont pas de valeur : elles existent ou n'existent pas. Elles peuvent être définies de deux façons différentes : globalement au niveau d'un projet, ou localement, dans le code.

## Constantes de compilation au niveau d'un projet

Pour définir de manière visuelle une constante de compilation au niveau du projet, il faut accéder à la fenêtre de propriétés d'un projet.

- Dans l'explorateur de solutions, faites un clic droit sur le projet concerné et sélectionnez le menu **Propriétés**.
- Une fois la fenêtre de propriétés ouvertes, sélectionnez le menu **Générer**.




La première partie de la section **Général** permet de définir des constantes personnalisées, ainsi que de définir les constantes DEBUG et TRACE qui sont utilisées par certaines classes du framework .NET.

Lorsque plusieurs constantes de compilation doivent être définies, elles doivent être séparées par le symbole ; (point-virgule).


Définir la constante SORTIE_CONSOLE a pour effet d'appliquer le paramètre **/define:SORTIE_CONSOLE** au projet

au moment de la compilation.

### Constantes de compilation locales

Il est possible de définir une constante de compilation pour un seul fichier, au niveau du code.

Pour cela, il est nécessaire d'utiliser la directive préprocesseur `#define` suivie du nom de la constante à définir. Un fichier peut contenir plusieurs directives `#define`, mais elles doivent toutes se situer en début de fichier, avant les autres instructions.

```
#define SORTIE_CONSOLE
```

Il peut parfois être nécessaire de ne définir une constante de compilation que sur une partie d'un fichier. Dans ce cas, il faut utiliser la directive `#undef` à l'endroit à partir duquel la constante ne doit plus être définie.

### Mise en œuvre de la compilation conditionnelle

La compilation conditionnelle permet d'inclure du code en fonction de l'existence d'une constante de compilation. Pour la mettre en œuvre, il est donc nécessaire de pouvoir vérifier cette existence.

Cette vérification se fait à l'aide de la directive de précompilation `#if ... #endif`. Cette instruction est semblable à l'instruction `if` : elle évalue un prédicat et inclut le code qu'elle contient si l'évaluation renvoie `true`.

Pour inclure du code en fonction de l'existence de la constante `SORTIE_CONSOLE` définie plus haut, il faut écrire :

```
#if SORTIE_CONSOLE
   Console.WriteLine("SORTIE_CONSOLE est définie");
#endif
```

Dans le cadre du débogage d'une application, il peut ainsi être intéressant de placer des instructions `#if` à certains endroits pour enregistrer un maximum d'informations pertinentes, notamment à l'intérieur de blocs `catch { }`.