空間の可視化

エイリアスディレクティブ

名前空間はクラスライブラリのカプセル化や識別子の問題を解決する最良の手段ですが
クラスの利用者は、名前空間をわざわざ入力しなければならず
頻繁に使う名前空間を何度も書く場合、それは効率の低下につながります

そこで、**using** キーワードを用います
このキーワードを用いれば、指定した名前空間を現在の名前空間で可視化することが可能です
これによって、多用する名前空間の修飾を省略することが可能です

using [alias =]class_or_namespace;

alias には、名前空間のエイリアスとなる識別子を指定します
class_or_namespace には、別名をつける名前空間、または型を指定します

alias を指定すれば、長い名前空間やクラス名に新しい名前を付けることができます
ある特定の名前空間の多用するクラスに簡単な別名をつければ
記述しなければならない修飾が減り、開発効率の向上を期待することができます
当然ですが、別名となる識別子は宣言された空間の別の識別子と衝突してはいけません

これを **using エイリアスディレクティブ** と呼びます
ただし、using ディレクティブは**名前空間のメンバ**です
名前空間、またはコンパイル単位の宣言空間にしか用いることができません

```
using cout = System.Console;

class Test {
 static void Main() {
  cout.WriteLine("Kitty on your lap");
 }
}
```

これまで、コンソールに文字列を出力する時は System.Console... と記述しました
ところが、すでに承知のとおりこれを何度も記述するのは大変面倒な作業です
そこで、using エイリアスディレクティブを用いて別名を命名します
エイリアスディレクティブは、名前空間や型に別の名前をつけることができるので
このプログラムでは、System.Console に cout という名前をつけました

using エイリアスディレクティブのスコープは、上のプログラムのように
コンパイル単位の宣言空間で宣言された場合、ファイル全体に共有されますが
名前空間のメンバとした場合、その名前空間の内部でのみ適応されます

```
namespace A {
 using cout = System.Console;
}

class Test {
 static void Main() {
  A.cout.WriteLine("Kitty on your lap"); //cout が不明
 }
}
```

別名 cout は名前空間 A のメンバですが、これは A の空間のための別名です
そのため、エイリアスディレクティブは他の空間からは常に不可視です
このプログラムをコンパイルすると Main() メソッドの cout でコンパイルエラーとなります
名前空間 A に cout という識別子は存在しないと判断されるためです

また、using エイリアスディレクティブは他のエイリアスの影響を受けません
using で指定する型や名前空間は、常に本名を強制されます
例えば、次のような指定はコンパイルエラーとなるでしょう

```
namespace A.B {}

namespace C {
 using Alias1 = A;
 using Alias2 = Alias1.B;
}
```

問題は、Alias2 で指定している Alias1 という識別子です
ここで指定する識別子だけは、他のエイリアスの影響を受けないため
Alias1 という識別子は存在しないと判断されてしまうのです

名前空間ディレクティブ

エイリアスディレクティブを用いる以外にも、便利な方法が存在します
それが、指定した名前空間を可視化する**名前空間ディレクティブ**です

using ディレクティブを用いる時 Alias を指定しないで名前空間だけを指示すると
コンパイラはディレクティブの宣言空間で不明な識別子を発見すると
using ディレクティブで指定した名前空間を検索するようになります
つまり、指定した名前空間をインポートし、その空間で可視化するということです

```
using System;
```

```
class Test {
 static void Main() {
  Console.WriteLine("Kitty on your lap");
 }
}
```

このプログラムは、using 名前空間ディレクティブを用いることによって
コンパイル単位で System 名前空間を可視化しています

これによって Main() メソッドを見てもわかるように
わざわざ、System 名前空間を修飾しなくても Console クラスが見えています
System 名前空間には、頻繁に使うクラスが多く宣言されているので
このように名前空間ディレクティブを用いることで、効率的にプログラムすることができます

名前空間ディレクティブの使用には、注意しなければならない点がいくつかあるのですが
重要なのは**型はインポートするが、空間はインポートしない**ことです
名前空間ディレクティブによって、クラスや構造体などは可視化されますが
可視化した名前空間に含まれる別の名前空間はその対象とはならないのです

```
namespace A.B {
 class Kitty {}
}

namespace C {
 using A;
 class Test {
  static void Main() {
   B.Kitty obj;
  }
 }
}
```

このプログラムは一見正しい様に見えますが、コンパイルできません
using A によって、名前空間 A はインポートされるため、A のメンバには直接アクセスできます
しかし、A にネストされた名前空間 B は名前空間ディレクティブではインポートされないため
Main() メソッドの B という名前空間名は発見されません

また、名前空間ディレクティブでインポートした名前空間の識別子と
using を宣言した名前空間の識別子が衝突した場合
エラーにはならず**カレントスペースの識別子で隠蔽**されます

```
namespace KittyScope {
 class Kitty {
  public override string ToString() {
   return "Kitty on your lap";
  }
 }
}

namespace MainScope {
 using KittyScope ;

 class Kitty {
  public override string ToString() {
   return "Silver Gene";
  }
 }

 class Test {
  static void Main() {
   System.Console.WriteLine(new Kitty());
   System.Console.WriteLine(new KittyScope.Kitty());
  }
 }
}
```

このプログラムを実行した結果、次のようになるでしょう

Silver Gene
Kitty on your lap

このプログラムでは、KittyScope と MainScope 名前空間の両方にKitty クラスが存在し
MainScope 内で KittyScope を名前空間ディレクティブで可視化します
すると、Kitty クラスの名前が競合し、KittyScope の Kitty クラスは隠蔽されます
そのため、Main() メソッドは結局 KittyScope の Kitty クラスにアクセスするために
名前空間を直接修飾しなければならないのです

---

using ディレクティブ

using [alias =]class_or_namespace;

名前空間ディレクティブによって名前空間を現在の空間にインポートします
または、エイリアスディレクティブによって空間や型の別名を定義します

alias - 名前空間、または型の別名となる識別子を指定します
class_or_namespace - 名前空間、または型を指定します
