演算子の定義

演算子のオーバーロード

実は、オーバーロードできるのはメソッドだけではない
C++ 同様に、C# 言語も演算子をオーバーロードして演算子の機能を拡張できる

演算子のオーバーロードは、言語自体を強化するものでもないし
また、演算子のオーバーロードがなくても問題なくプログラムを組むことができることから
この機能に対しては C++ のころから疑問視する声があった

この数年オブジェクト指向の中核となった Java 言語は
C++ の演算子のオーバーロードやテンプレートなど、多くの機能を削ったにもかかわらず
簡素でかつ強力な言語として世界中で注目されることとなったのは事実である

しかし、演算子のオーバーロードはオブジェクト指向の抽象化に貢献します
C# では、演算子のオーバーロードやインデクサなどを駆使して
直感的にソースを記述できる様に技術設計されているのです

まず、オーバーロードされた演算子は常に **public static** です
そして、メンバ宣言時に **operator** キーワードを指定します

public static result-type operator unary-operator (op-type operand)

これが、演算子のオーバーロードです
ご覧のように、演算子のオーバーロードはプロパティやインデクサ同様に
暗黙的に適切なメソッドを呼び出すための手段にすぎません

result-type には演算結果を返すための戻り値の型を指定します
技術的にはどのような型でもかまいませんが、常識として演算子が受ける引数に関連のある
その演算子が**本来持っている意味を尊重**した値を返すべきです

unary-operator には、オーバーロードする演算子を指定します
オーバーロードできる演算子は、次のものに限られています

単項演算子	2項演算子
+	+
-	-
!	*
~	/
++	%
--	&
true	|
false	^
	<<
	>>
	==
	!=
	<
	>
	<=
	>=

op-type はオペランドの型、operand は指定されたオペランドを受ける仮引数です
単項演算子の場合は一つですが、2項演算子の場合は2つの引数を受け取ります
重要なことですが代入演算子　**＝はオーバーロードできません**

さらに、従来の演算子をオーバーロードで隠蔽するということもできません
このことから、演算子のオーバーロードでは必ずクラス型を受け取る必要があります
2項演算子の場合は、オペランドのうち一方がクラス型でなければなりません

しかし、複合代入演算子は演算子をオーバーロードすると暗黙的にオーバーロードされます
例えば + 演算子をオーバーロードすると += 演算子もオーバーロードされるのです

```
class Output {
 public static Output operator << (Output cout , System.Object obj) {
  System.Console.WriteLine(obj.ToString());
  return cout;
 }
}

class Test {
 static void Main() {
  Output cout = new Output();
  cout <<= "Kitty on your lap";
 }
}
```

このプログラムは、C# 言語で C++ のコンソール出力を簡単に再現してみたものです

C++ では直感的にストリームを操作できるように演算子のオーバーロードが採用されていました

<< 演算子を Output クラスがオーバーロードしています
プログラム中で Outpu << Object という演算が見つかるとメソッドが呼び出されます
複合代入演算子は演算子をオーバーロードした時点で暗黙的にオーバーロードされます
そのため <<= は cout と文字列を評価して cout に戻り値を格納しています

ユーザー定義変換

むしろ、演算子のオーバーロードにおいて重要なのは**ユーザー定義変換**です
これは通常の演算子ではなく**キャスト演算子**をオーバーロードします

この機能を用いれば、文字列を独自のクラス型にキャストするということが可能になります
当然、与えられた文字列をどのように加工して独自のクラス型を生成するかという仕様は
クラスやユーザー定義変換の設計者が考えることです

通常、ユーザー定義変換を宣言するにも operator キーワードを使いますが
これ以外に、明示的な変換を意味する **explicit** キーワードと
暗黙的な変換を意味する **implicit** キーワードを用います

public static implicit operator conv-type-out (conv-type-in operand)
public static explicit operator conv-type-out (conv-type-in operand)

conv-type-out には変換後の型を
conv-type-in operand は変換される型をそれぞれ指定します
ユーザー定義変換が使用されると、この変換メソッドが自動的に呼び出されます
変換メソッドは、引数を受け取り、それを元に変換後の型を返します

ただし、受け取る型と変換後の型に継承関係があってはいけません
例えば、ユーザー定義変換は System.Object を受け取ることはできないのです

```
class Kitty {
 public string str;
 public Kitty(string str) {
  this.str = str;
 }
 public static explicit operator Kitty(string str) {
  System.Console.WriteLine("Kitty に変換");
  return new Kitty(str);
 }
}

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

Kitty クラスは明示的なユーザー定義変換を宣言しています
文字列を明示的に Kitty クラスに変換しようとした場合、このメソッドが呼び出されます
これを実行すれば、次のような結果がコンソールに出力されます

Kitty に変換
Kitty on your lap

正しく変換メソッドが呼び出され、予想した形で Kitty のインスタンスを生成できていますね
この機能も、メソッド呼び出すための手段に過ぎないことがわかると思います

この機能は、単純にメソッドの呼び出しを暗黙的にしている単純なものですが
それでも、キャストを用いた直感的な変換機構をクラスに持たせることができます
一般的なプログラム言語のように静的な StringToKitty() メソッドを作っても意味は同じです
StringToKitty() メソッドが String 型を受け取り Kitty を返すならば、それは変換メソッドと呼べます

しかし、さらに強力なのは C# が暗黙的なユーザー変換もサポートしている点です
暗黙的な変換には implicit キーワードを使ったユーザー変換を宣言します
ただし、暗黙的な変換しかないユーザー定義変換を明示的に行った場合は
エラーにはならず、暗黙的な変換メソッドが呼び出されます

```
class Kitty {
 public string str;
 public Kitty(string str) {
  this.str = str;
 }
 public static implicit operator Kitty(string str) {
  System.Console.WriteLine("Kitty に変換");
  return new Kitty(str);
 }
}

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