

Завдання на тему «interface-и»

Варіантів нема, усім одне й те саме завдання.

Створіть в точності такий власний інтерфейс:

```
interface IMyNumber<T> where T : IMyNumber<T>
{
    T Add(T b);
    T Subtract(T b);
    T Multiply(T b);
    T Divide(T b);
}
```

Напишіть два різні класи, кожен з яких реалізує цей інтерфейс:

- **Frac** — подає звичайні дроби, тобто у вигляді чисельника і знаменника (обидва типу `System.Numerics.BigInteger`);
- **MyComplex** — подає комплексні числа, як дійсну та уявну частини (обидві типу `double`).

Точніше кажучи, **MyFrac** буде реалізовувати `IMyNumber<MyFrac>`, а **MyComplex** буде реалізовувати `IMyNumber<MyComplex>`.

Що робити, якщо `BigInteger` не компілюється? Підключити його, бо це робиться неочевидним способом (принаймні, у тих версіях C#, в яких пробував). Крім того, що треба писати `using System.Numerics;` у коді програми, треба ще підключити в середовищі відповідну скомпільовану бібліотеку: у `Solution Explorer` натиснути правою кнопкою миші на `References`, вибрати `Add Reference...`, перейти на tab `“.NET”`, вибрати `System.Numerics`. Якщо раптом це робиться якось простіше — буду вдячний тому/тій, хто покаже, як.

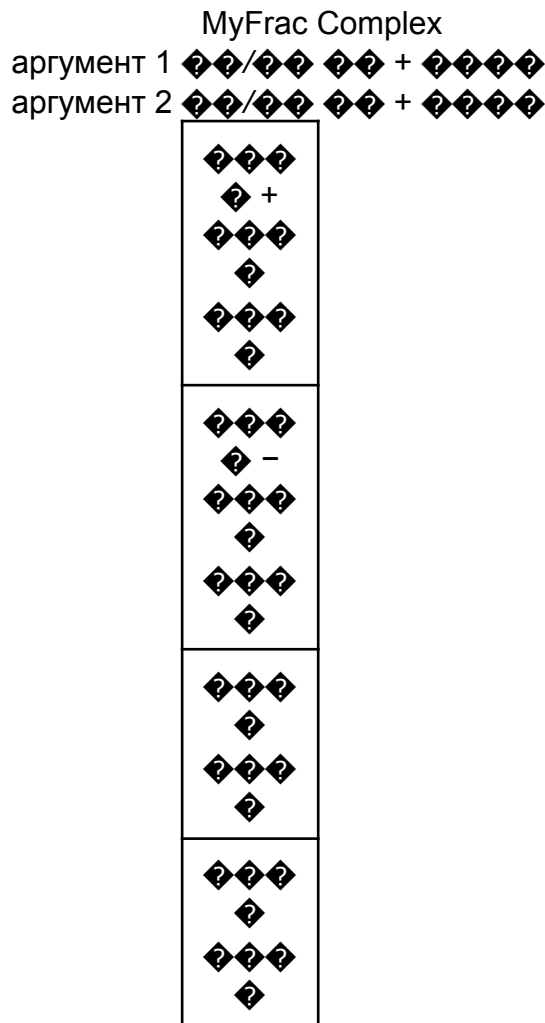
Також буду вдячний, якщо хтось проведе детальний аналіз можливості (чи можливо взагалі, і якщо можливо, то як) підключення `System.Numerics` при компіляції в `gmcs (monoC#)` під `Linux`: (а) з командного рядка; (б) з програми учасника на онлайн-перевірці, коли нема можливості вплинути на командний рядок (наприклад, на `informatics`).

(Навіщо тут навороти вигляду `IMyNumber<T> where T : IMyNumber<T>`? Чом би не зробити по-простому `interface IMyNumber` з методами `IMyNumber add(IMyNumber that)`; та аналогічними? Ми вводимо поняття (класи) різних чисел, і додавати/віднімати/множити/ділити можна різні числа. Але, мабуть, все ж дроби з дробами, а комплексні з комплексними. Виклик `f.add(c)` (де `f` є примірником `MyFrac`, а `c` — примірником `MyComplex`) мабуть краще вважати помилковим і беззмстовним, ніж дозволяти мовчки отримувати в результаті казна-що. Якщо все ж виникне потреба скласти дріб із комплексним числом — мабуть, краще мати у класі `MyFrac` метод `double doubleValue()`, що дозволяє отримати приблизне дійсне значення (або аналогічну `property`, доступну лише для читання (`get`)), створити з того дійсного значення комплексне число, і додати вже два комплексні числ'а.

Аналогічна ситуація: колись вважалося правильним мати `interface IComparable` без `generic`-параметра і реалізовувати його шляхом написання методу `public int CompareTo(Object that)`. Тільки коли реалізовували, наприклад, `class MyFrac : IComparable`, пишучи тіло методу `public int CompareTo(Object that)` так, щоб все одно твердо розраховувати на те, що той `that` насправді не довільний `Object`, а саме `MyFrac`, і кінець кінцем ніщо не вберігало від ситуації, коли методу `CompareTo` примірника (об'єкту) класу `MyFrac` помилково передають як параметр примірник (об'єкт) якогось зовсім іншого, неможливого для порівняння з дробом класу (наприклад, `Student`, який містить прізвище, ім'я, групу та оцінки за минулий семестр). У більшості випадків це завершувалося `exception-ом` і або помилкою часу виконання, або незрозумілим пропуском порівняння (що могло призводити, наприклад, до того, що почали сортувати, але кинули на півдорозі).

Нині вважається правильнішим, щоб класи реалізовували інтерфейс `IComparable<...>`, параметризований собою. Внаслідок чого, `class MyFrac : IComparable<MyFrac>` реалізує `public int CompareTo(MyFrac that)`, а `class Student : IComparable<Student>` реалізує `public int CompareTo(Student that)`. При такому підході, код, який намагається визначати «хто більший — студент Коваленко Андрій чи дріб 42/17?» просто не скомпілюється. Що у переважній більшості випадків краще, ніж ситуація з попереднього абзацу.

Для реалізації власне арифметичних операцій треба знати (або подивитися) формули, за якими вони відбуваються. Повторимо їх (у математичному вигляді) тут:



Add($\frac{??}{??} + \frac{??}{??}$) + ($\frac{??}{??} + \frac{??}{??}$) $\frac{??}{??}$

Subtract($\frac{??}{??} - \frac{??}{??}$) + ($\frac{??}{??} - \frac{??}{??}$) $\frac{??}{??}$

Multiply($\frac{??}{??} \frac{??}{??} - \frac{??}{??} \frac{??}{??}$) + ($\frac{??}{??} \frac{??}{??} + \frac{??}{??} \frac{??}{??}$) $\frac{??}{??}$

Divide $\frac{??}{??} \frac{??}{??} + \frac{??}{??} \frac{??}{??}$


$$\frac{??^2 + ??^2 + \frac{??}{??} \frac{??}{??} - \frac{??}{??} \frac{??}{??}}{\frac{??}{??}^2 + \frac{??}{??}^2 \frac{??}{??}}$$

Само собою, усі ці формули треба перетворити, щоб кожна з функцій працювала з полями (або пом та denom для MyFrac, або re та im для MyComplex) двох однотипних об'єктів (this та отриманий параметром that) — і ні в якому разі не намагалася взяти $\frac{??}{??}$, $\frac{??}{??}$, $\frac{??}{??}$, $\frac{??}{??}$ з якихось інших джерел! Власне, наведемо готові реалізації 2-х з цих 8-ми формул.

Додавання комплексних чисел ($\frac{??}{??} + \frac{??}{??}$) + ($\frac{??}{??} + \frac{??}{??}$) $\frac{??}{??}$:


```
public MyComplex add(MyComplex that) {
    return new MyComplex(this.re + that.re, this.im + that.im);
}
```

Додавання дробів

 — спосіб 1:

```
public MyFrac Add(MyFrac that)
{
    return new MyFrac(BigInteger.Add(BigInteger.Multiply(this.nom, that.denom),
                                      BigInteger.Multiply(this.denom, that.nom)),
                      BigInteger.Multiply(this.denom, that.denom));
}
```

Додавання дробів

 — спосіб 2 (протилежна крайність: якщо у способі 1 все зіплене в один довгелезний вираз, то тут навпаки аж надто дрібно покрито з аж надто великою кількістю проміжних змінних; рекомендується шукати якийсь розумний компроміс десь посередині між цими варіантами):

```
public MyFrac Add(MyFrac that)
{
    BigInteger a = this.nom;
    BigInteger b = this.denom;
    BigInteger c = that.nom;
    BigInteger d = that.denom;
    BigInteger ad = BigInteger.Multiply(a, d);
    BigInteger bc = BigInteger.Multiply(b, c);
    BigInteger ad_plus_bc = BigInteger.Add(ad, bc);
    BigInteger bd = BigInteger.Multiply(b, d);
    return new MyFrac(ad_plus_bc, bd);
}
```

Насправді працює також і третій варіант:

```
MyFrac Add(MyFrac that)
{
    return new MyFrac(this.nom * that.denom + that.nom * this.denom, this.denom * that.denom);
}
```

2

Він в цілому зручніший, і не заборонений; перші два, насправді, були показані більше для того, щоб показати, що методами Add, Subtract, Multiply та Divide (саме як м'єтдами) теж можна користуватися (і, наприклад, у аналогічному класі BigInteger мови Java це єдиний спосіб).

Виникає питання: раз у бібліотечному типі BigInteger зробили арифметичні операції через звичайні значки +, -, *, /, то чи можемо ми зробити так само — зробити (чи то додатково до вказаних на самому початку завдання довгих методів, чи то замість них) короткі перевантажені операції для нашого IMyNumber?

Наскільки вдалося з'ясувати, частково так і частково ні. Для інтерфейсу IMyNumber, начебто, ні (наприклад, за адресою <https://stackoverflow.com/questions/5066281/can-i-overload-an-operator-on-an-interface> стверджують: “No, you can't. Overloading . . . requires static methods in one of the types you use, and an interface can't contain those. Extension methods can't help either.”). Разом з тим, у сам'их класах MyFrac та MyComplex абсолютно ніщо не заважає реалізовувати методи у стилі

```
static MyFrac operator + (MyFrac f1, MyFrac f2)
{
    return f1.Add(f2);
}
```

і таким чином надати тому користувачеві, який потім буде використовувати цю бібліотеку, зручніший синтаксис перевантажених операторів. Тільки от доводиться або кожен арифметичну дію писати окремо в MyComplex і окремо в MyFrac (що погано, бо це дублювання коду), або створювати, крім інтерфейсу, також і абстрактний клас-предок (що погано, бо наслідування за інтерфейсами множинне, а за класами одинарне, тож наявність класу-предка унеможливорює успадкування від чогось іншого, навіть якщо воно потім виявиться потрібніше.)

Зробіть, щоб при діленні на ноль метод Divide кожного з класів створював («кидав», throw) System.DivideByZeroException.

Крім того, зробіть, щоб кожен з класів мав кілька конструкторів. Точний перелік визначити самостійно, але серед них обов'язково public MyFrac(BigInteger nom, BigInteger denom) та public MyComplex(double re, double im) — вони і використовуються у реалізаціях арифметичних дій, і досить зручні для тих користувачів, які потім використовуватимуть цю бібліотеку. Ще ставиться вимога мати окремий конструктор public MyFrac(int nom, int denom) — він робить набагато зручнішим створення дробів з маленькими чисельником і знаменником. При цьому нема нагальної потреби створювати окремий конструктор public MyComplex(int re, int im), бо тут при підстановці цілих чисел буде успішно викликатися конструктор від double-ів (правда, це лише доки нема інших конструкторів з двома числовими параметрами).

Крім того, кожен з цих класів повинен містити override public String ToString(), щоб забезпечити осмислене виведення відповідних чисел.

В якості перевірки написаних Вами класів, запустіть проект через таку точку входу (копіювати можна, перенабирати не обов'язково):

```
static void testAPlusBSquare<T>(T a, T b) where T : IMyNumber<T>
{
    Console.WriteLine("=== Starting testing (a+b)^2=a^2+2ab+b^2 with a = " + a + ", b = " + b + " ==="); T aPlusB =
    a.Add(b);
    Console.WriteLine("a = " + a);
    Console.WriteLine("b = " + b);
    Console.WriteLine("(a + b) = " + aPlusB);
    Console.WriteLine("(a+b)^2 = " + aPlusB.Multiply(aPlusB));
    Console.WriteLine(" = = = ");
    T curr = a.Multiply(a);
    Console.WriteLine("a^2 = " + curr);
    T wholeRightPart = curr;
    curr = a.Multiply(b); // ab
    curr = curr.Add(curr); // ab + ab = 2ab
    // I'm not sure how to create constant factor "2" in more elegant way,
    // without knowing how IMyNumber is implemented
    Console.WriteLine("2*a*b = " + curr);
```

3

```
wholeRightPart = wholeRightPart.Add(curr);
curr = b.Multiply(b);
Console.WriteLine("b^2 = " + curr);
wholeRightPart = wholeRightPart.Add(curr);
Console.WriteLine("a^2+2ab+b^2 = " + wholeRightPart);
Console.WriteLine("=== Finishing testing (a+b)^2=a^2+2ab+b^2 with a = " + a + ", b = " + b + " ==="); }
```

```
static void Main(string[] args)
{
    testAPlusBSquare(new MyFrac(1, 3), new MyFrac(1, 6));
    testAPlusBSquare(new MyComplex(1, 3), new MyComplex(1, 6));
    Console.ReadKey();
}
```

Результати повинні бути такими:

```
=== Starting testing (a+b)^2=a^2+2ab+b^2 with a = 1/3, b = 1/6 ===
a = 1/3
b = 1/6
(a + b) = 1/2
```

```

(a+b)^2 = 1/4
==
a^2 = 1/9
2*a*b = 1/9
b^2 = 1/36
a^2+2ab+b^2 = 1/4
=== Finishing testing (a+b)^2=a^2+2ab+b^2 with a = 1/3, b = 1/6 ===
=== Starting testing (a+b)^2=a^2+2ab+b^2 with a = 1+3i, b = 1+6i ===
a = 1+3i
b = 1+6i
(a + b) = 2+9i
(a+b)^2 = -77+36i
==
a^2 = -8+6i
2*a*b = -34+18i
b^2 = -35+12i
a^2+2ab+b^2 = -77+36i
=== Finishing testing (a+b)^2=a^2+2ab+b^2 with a = 1+3i, b = 1+6i ===

```

Цей метод testAPlusBSquare хоч якось перевіряє лише додавання та множення; допишіть свій аналогічний метод testSquaresDifference, який аналогічним чином обчислюватиме та виводитиме $(a^2 - b^2)$ та $(a^2 - b^2)^2$.

$a^2 + b^2$ (котрі теж повинні давати однаковий результат), і таким чином хоч якось перевірте правильність віднімання, ділення та виключення ділення на 0. Крім того, допишіть, щоб MyFrac реалізовував також інтерфейс IComparable<MyFrac>, і допишіть у метод Main код, який це якось використовуватиме (наприклад, сортуватиме шляхом виклику бібліотечного методу сортування). Реалізовувати інтерфейс IComparable<MyComplex> у MyComplex не треба, з таких міркувань: «розширити порядок дійсних чисел, включивши в нього всі комплексні числа і при цьому зберігши звичайні властивості порядку, неможливо».

Перевіряти правильність с'яме через використання вказаної точки входу — не пропозиція, а вимога. Одна з причин цієї вимоги — це ще один приклад поліморфізму (на рівні інтерфейсів), і з ним треба ознайомитись. З іншого боку, такий спосіб перевірки правильності не є ні сучасним, ні зручним. В цій задачі досить-таки доречні т. зв. unit-тести. Оскільки щодо них тут так досі й не написано нічого конкретного, вони вважатимуться дрібним бонусним завданням на тему цієї задачі: можуть принести додаткові бали, але не звільняють від необхідності зробити також і наведену тут перевірку правильності.