Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет

телекоммуникаций и информатики»

Кафедра ПМиК

Современные технологии программирования 2

Универсальный калькулятор

Выполнил: студент 4 курса

Ф. ИВТ, группа: ИП-711

Мартасов И. О.

Проверил: ассистент кафедры ПМиК

Агалаков Антон Александрович

Новосибирск, 2021

**Содержание**

1. Задание.
   1. Практическая работа. Иерархия классов Число.
   2. Практическая работа. Иерархия классов Редактор
   3. Практическая работа. Абстрактный тип данных (ADT) Память
   4. Практическая работа. Абстрактный тип данных (ADT) Процессор
   5. Практическая работа. Управление калькулятором.
   6. Практическая работа. Интерфейс.
2. Текст программы.

**Задание**

1. Разработайте Универсальный калькулятор с интерфейсом в стиле Windows, который позволил бы вычислять выражения с р-ичными числами, простыми дробями, комплексными числами.
2. Калькулятор необходимо снабдить системой справочной.

**Спецификация**

1. Калькулятор должен обеспечить вычисление выражений в одном из режимов:
   * р-ичные числа,
   * комплексные числа,
   * рациональными дроби.
2. Остальные требования: работа с памятью, работа с буфером обмена, прецеденты использования те же, что и для калькуляторов р-ичных чисел, простых дробей и комплексных чисел.

**Практическая работа**

**Иерархия классов Число**

**Задание**

1. Разработать и реализовать иерархию классов «Число», обеспечивающую выполнение операций над р-ичными числами, простыми дробями и комплексными числами.

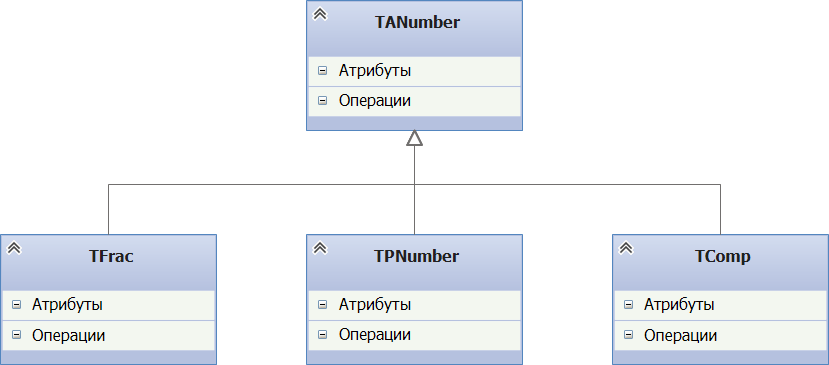
На унифицированном языке моделирования UML (Unified Modeling Language) диаграмма класса TАNumber (Число) выглядит следующим образом:

|  |
| --- |
| **TANumber (Число)** |
| строка: String (свойство) |
| числоЕстьНоль(B: TANumber): Boolean; virtual; |
| копировать: TANumber; ; virtual; |
| сложить (B: TANumber): TANumber; virtual; |
| вычесть (B: TANumber): TANumber; virtual; |
| перемножить (B: TANumber): TANumber; virtual; |
| поделить (B: TANumber): TANumber; virtual; |
| равенствоЧисел (B: TANumber): Boolean; virtual; |
| квадрат: TANumber; virtual; |
| обратное: TANumber; virtual; |
| читатьЧислоВформатеСтроки: String; virtual; (метод свойства) |
| писатьЧислоВформатеСтроки(a: String) ; virtual; (метод свойства) |
| Обязанность:  выполнение арифметических операций над действительными числами, простыми дробями и комплексными числами |

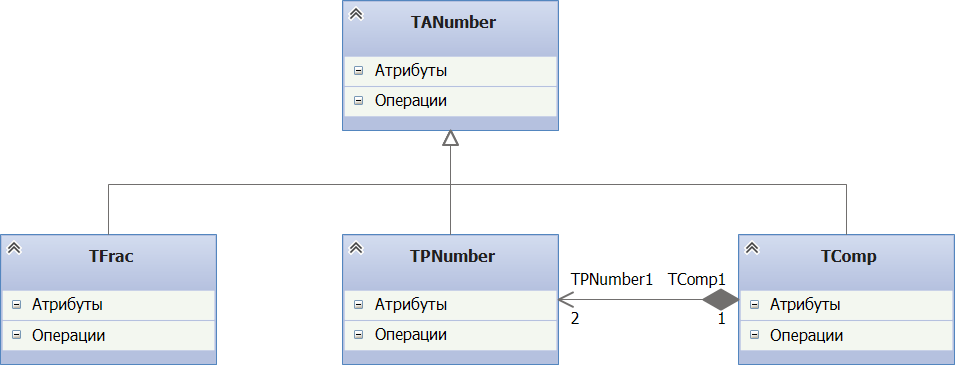
1. Класс должен отвечать за выполнение следующих операций над действительными числами, простыми дробями и комплексными числами:
   * сложение;
   * вычитание;
   * умножение;
   * деление;
   * эквивалентность двух чисел;
   * равенство числа нулю;
   * запись строкового представления числа;
   * чтение строкового представления числа;
   * создание копии числа
2. Иерархия классов «Число» представлена на (Рис. 1, Рис. 2. Рис. 3). Они отличаются тем, что первая иерархия не обеспечивает повторного использования кода, в отличие от второй. Здесь TANumber – число, TFrac

– постая дробь, TPNumber – р-ичное число, TComplex – комплексное число. Третья иерархия обеспечивает работу со всеми типами чисел в системе счисления с выбранным основанием.

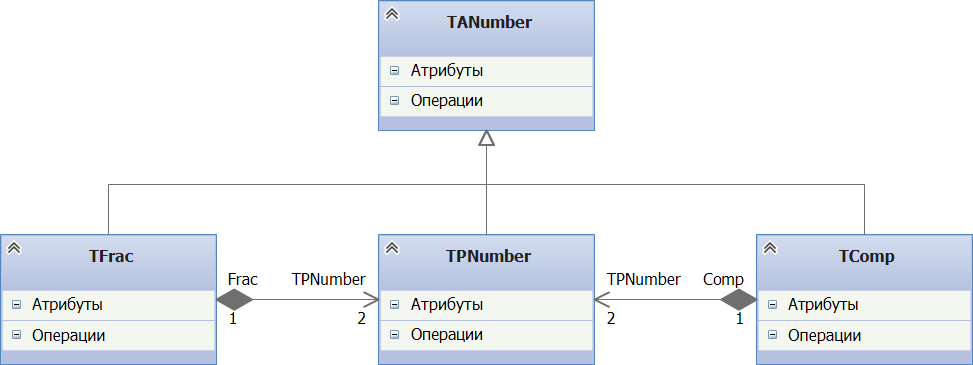
1. Протестировать каждый класс иерархии.



**Рис. 1. Иерархия классов «число» без повторного использования кода**



**Рис. 2. Иерархия классов «число» с повторным использованием кода**



**Рис. 3. Иерархия классов «число» для работы с числами в системе счисления с выбранным основанием**

**Практическая работа**

**Иерархия классов Редактор**

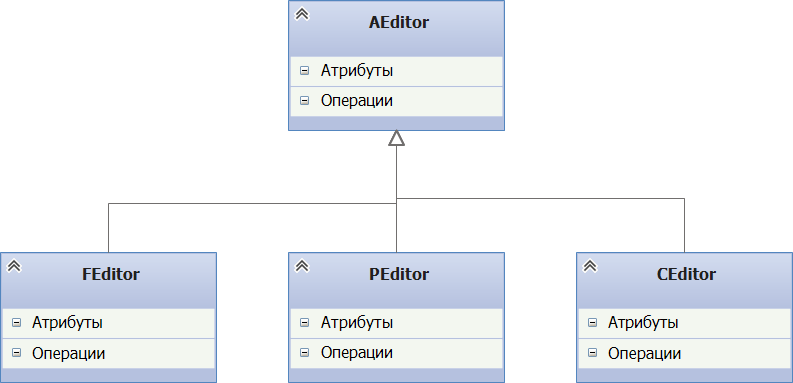
**Задание**

1. Разработать и реализовать иерархию классов «Редактор», наследующих от АEditor, которая обеспечивала бы возможность редактирования р-ичных чисел, простых дробей и комплексных чисел.

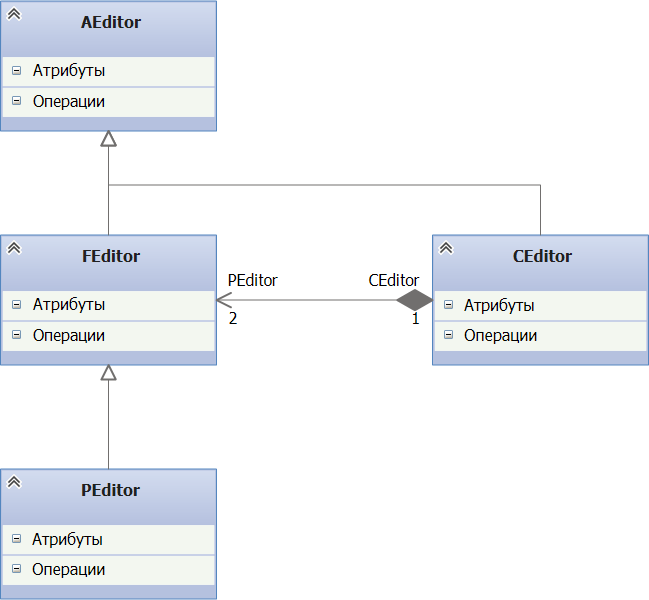
На унифицированном языке моделирования UML (Unified Modeling Language) диаграмма класса АEditor выглядит следующим образом:

|  |
| --- |
| **AEditor (Редактор)** |
| строка: String |
| числоЕстьНоль: Boolean |
| добавитьЗнак: String |
| добавитьР-ичную цифру(a: Integer): String |
| добавитьНоль: String |
| забойСимвола: String |
| очистить: String |
| читатьСтрокаВформатеСтроки: String (метод свойства) |
| писатьСтрокаВформатеСтроки(a: String) (метод свойства) |
| редактировать(a: Integer): String |
| Обязанность:  ввод, хранение и редактирование строкового представления чисел |

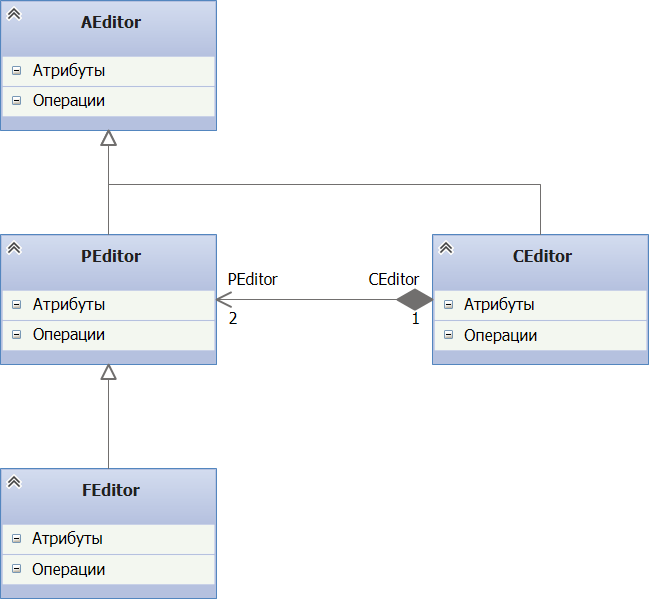
1. Класс должен отвечать за ввод и редактирование строкового представления чисел. Класс должен обеспечивать:
   * добавление символов;
   * добавление и изменение знака;
   * добавление разделителя;
   * забой символа, стоящего справа (BackSpace);
   * установку нулевого значения числа (Clear);
   * чтение строкового представления;
   * запись строкового представления;
2. Иерархия классов «Редактор» представлена на (Рис. 4, Рис. 5, Рис. 6, Рис. 7). Они отличаются тем, что первая иерархия не обеспечивает повторного использования кода, в отличие от второй. Третья и четвёртая иерархии обеспечивает работу со всеми типами чисел в системе счисления с выбранным основанием. Здесь AEditor – абстрактный редактор, PEditor – редактор р-ичных чисел, FEditor – редактор дробей, CEditor – редактор комплексных чисел.
3. Протестировать каждый класс иерархии.



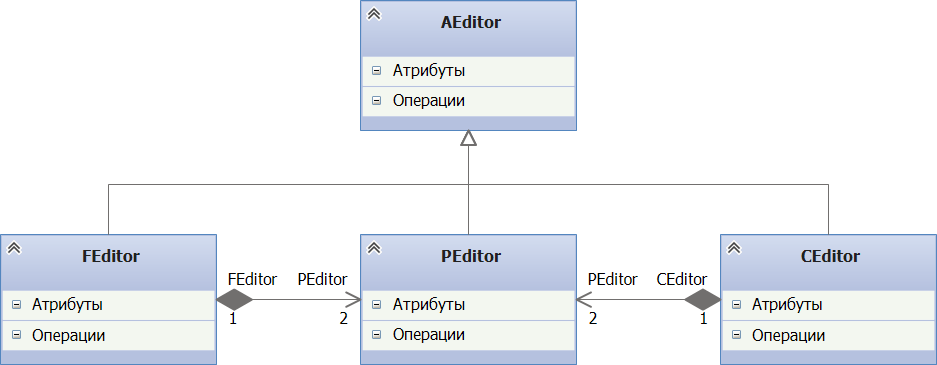
**Рис. 4. Иерархия классов редактор без повторного использования кода**



**Рис. 5. Иерархия классов редактор с повторным использованием кода**



**Рис. 6. Иерархия классов редактор с повторным использованием кода для работы с числами в заданной системе счисления.**



**Рис. 7. Иерархия классов редактор с повторным использованием кода для работы с числами в заданной системе счисления.**

**Практическая работа**

**Абстрактный тип данных (ADT) Память**

**Задание**

1. В соответствии с приведенной ниже спецификацией реализовать абстрактный тип данных «память для Число», используя класс.
2. Оттестировать каждую операцию, определенную на типе данных одним из методов тестирования.
3. Оттестировать тип данных в целом.

**Спецификация типа данных «память для Число». ADT** TMemory

**Данные**

Память для Число(тип TMemory, в дальнейшем - память) - это память для хранения указателя (ссылки) на объект «Число» (тип ANumber) и значения

«состояние памяти». Объект «память для Число» - изменяемый. Он имеет два состояния, обозначаемых значениями: «Включена» (\_On), «Выключена» (\_Off). Её изменяют операции: Записать (Store), Добавить (Add), Очистить (Clear).

**Операции**

|  |  |
| --- | --- |
| ***Конструктор*** |  |
| Начальные  значения: | Указатель (ссылка) на Число (тип  ANumber со значением 0) |
| Процесс: | Создаёт объект «память» типа TMemory. Инициирует поле Mem (тип ANumber) объекта «память» (тип TMemory) указателем на копию объекта «Число» (тип TPNumber), полученным через параметр. Память устанавливается в состояние «Выключена», в поле  «состояние памяти» заносится значение (\_Off). |
| ***Записать*** |  |
| Вход: | (E) – указатель (ссылка) на объект  «Число» (тип ANumber). |
| Предусловия: | Нет. |
| Процесс: | В объект «память» (тип TMemory) в поле указатель (ссылка) на «Число»  записывается копия объекта (Е). Память |
|  | устанавливается в состояние «Включена», в поле «состояние памяти» заносится  значение (\_On). |
| Выход: | Нет. |
| Постусловия: | Состояние памяти – «Включена» (\_On). |
|  |  |
| ***Взять*** |  |
| Вход: | Нет. |
| Предусловия: | Нет. |
| Процесс: | Создаёт копию объекта типа ANumber, хранящуюся в объекте «память» (тип TMemory) и возвращает ссылку  (указатель) на неё. |
| Выход: | Ссылка (указатель) на объект типа  ANumber. |
| Постусловия: | Состояние памяти – «Включена» (\_On). |
|  |  |
| ***Добавить*** |  |
| Вход: | (Е) – ссылка (указатель) на Число (объект  типа ANumber). |
| Предусловия: | Нет. |
| Процесс: | В поле ссылка (указатель) на «Число» объекта «память» (тип TMemory) записывается объект «Число» (тип ANumber), полученный в результате сложения «Числа» по ссылка (указателю)  (Е) и «Числа», ссылка (указатель) на которое хранится в памяти. |
| Выход: | Нет. |
| Постусловия: | Состояние памяти – «Включена» (\_On). |
|  |  |
| ***Очистить*** |  |
| Вход: | Нет. |
| Предусловия: | Состояние памяти – «Включена» (\_On). |
| Процесс: | В поле ссылка (указатель) на «Число» объекта «память» (тип TMemory) записывается объект «Число» (тип ANumber), инициализированный  значением 0. Память устанавливается в состояние «Выключена» (\_Off). |
| Выход: | Нет. |
| Постусловия: | Состояние памяти – «Выключена» (\_Off). |
|  |  |

|  |  |
| --- | --- |
| ***ЧитатьСостояниеПам яти*** |  |
| Вход: | Нет. |
| Предусловия: | Нет. |
| Процесс: | Копирует и возвращает значение поля  «состояние памяти» объекта «память» (тип TMemory) в формате строки. |
| Выход: | Значение типа AnsiString. |
| Постусловия: | Нет. |
|  |  |
| ***ЧитатьЧисло*** |  |
| Вход: | Нет. |
| Предусловия: | Нет. |
| Процесс: | Возвращает значение Числа, хранящегося  в памяти, в формате строки. |
| Выход: | Значение типа AnsiString. |
| Постусловия: | Нет. |

***end*** TMemory

**Практическая работа**

**Абстрактный тип данных Процессор**

**Задание**

1. В соответствии с приведенной ниже спецификацией реализовать абстрактный тип данных «Процессор», используя класс.
2. Оттестировать каждую операцию, определенную на типе данных одним из методов тестирования.
3. Оттестировать тип данных в целом.

**Спецификация типа данных «Процессор». ADT** TProc

**Данные**

Процессор (тип TProc) выполняет двухоперандные операции TOprtn = (None, Add, Sub, Mul, Dvd) и однооперандные операции - функции TFunc = (Rev, Sqr). Если операция или функция не может быть выполнена, в поле Error типа String заносится сообщение об ошибке. Левый операнд и результат операции хранится в поле Lop\_Res, правый - в поле Rop. Оба поля имеют тип TANumber. Процессор может находиться в состоянии «операция установлена»: поле Operation не равно None (значение типа TOprtn) или в состоянии операция не установлена: поле Operation = None. Значения типа TProc - изменяемые. Они изменяются операциями: Сброс операции (OprtnClear), Выполнить операцию (OprtnRun), Вычислить функцию (FuncRun), Установить операцию (OprtnSet), Установить левый операнд (Lop\_Res\_Set), Установить правый операнд (Rop\_Set), Сброс калькулятора (ReSet).

**Операции**

|  |  |
| --- | --- |
| *Конструктор* |  |
| Начальные  значения: | два указателя TANumber\* L, TANumber\* R |
| Процесс: | Создаёт объект Процессор типа TProc. Поля Lop\_Res, Rop инициализируются копиями объектов (L, R). В поле Error заносится пустая строка. Процессор устанавливается в состояние: «операция не установлена»:  (Operation = None). |
|  |  |
| *Сброс процессора* |  |
| Вход: | Нет. |
| Предусловия: | Нет. |
| Процесс: | Поля объекта процессор: Lop\_Res, Rop инициализируются объектами текущего типа со значением 0. В поле Error заносится пустая строка. Процессор устанавливается в состояние: «операция не установлена»:  (Operation = None). |
| Выход: | Нет. |
| Постусловия: | Состояние процессора – «операция  сброшена» (Operation = None). |
|  |  |
| *Сброс операции* |  |
| Вход: | Нет. |
| Предусловия: | Нет. |
| Процесс: | Процессор устанавливается в состояние:  «операция не установлена»: (Operation = None). |
| Выход: | Нет. |
| Постусловия: | Состояние процессора – «операция  сброшена» (Operation = None). |
|  |  |
| *Выполнить операцию* |  |
| Вход: | Нет. |
| Предусловия: | Нет. |
| Процесс: | Вызывает выполнение текущей операции (записанной в поле Operation). Операция (Operation) выполняется над значениями, хранящимися в полях Rop и Lop\_Res. Результат сохраняется в поле Lop\_Res. Если Operation = None, никакие действия не выполняются. Если операция не может быть выполнена, в поле Error заносится сообщение об ошибке. Состояние объекта  не изменяется. |
| Выход: | Нет. |
| Постусловия: | Изменяется состояние поля Lop\_Res (если текущая операция выполнена) или Error – в  противном случае. |
|  |  |
| *Вычислить функцию* |  |
| Вход: | Вид функции (Func: TFunc). |
| Предусловия: | Нет. |
| Процесс: | Вызывает выполнение текущей функции  (Func). Функция (Func) выполняется над |

|  |  |
| --- | --- |
|  | значением, хранящимся в поле Rop. Результат сохраняется в нём же. Если операция не может быть выполнена, в поле Error заносится сообщение об ошибке.  Состояние объекта не изменяется. |
| Выход: | Нет |
| Постусловия: | Изменяется состояние поля Rop (если текущая операция выполнена) или Error – в  противном случае. |
|  |  |
| *Читать левый операнд* |  |
| Вход: | Нет. |
| Предусловия: | Нет. |
| Процесс: | Создаёт копию Lop\_Res и возвращает  указатель на неё. |
| Выход: | Указатель на Lop\_Res. |
| Постусловия: | Состояние процессора не изменяется. |
|  |  |
| *Записать левый*  *операнд* |  |
| Вход: | Переменная (Operand) типа TANumber. |
| Предусловия: | Нет. |
| Процесс: | Создаёт копию Operand и заносит указатель  на неё в поле Lop\_Res. |
| Выход: | Нет. |
| Постусловия: | Значение поля Lop\_Res меняется на  значение (Operand). |
|  |  |
| *Читать правый*  *операнд* |  |
| Вход: | Нет. |
| Предусловия: | Нет. |
| Процесс: | Создаёт копию Rop и возвращает указатель  на неёё. |
| Выход: | Указатель на Rop. |
| Постусловия: | Состояние процессора не меняется. |
|  |  |
| *Записать правый*  *операнд* |  |
| Вход: | Переменная (Operand) типа TANumber. |
| Предусловия: | Нет. |
| Процесс: | Создаёт копию Operand и заносит указатель  на неё в поле Rop. |

|  |  |
| --- | --- |
| Выход: | Нет. |
| Постусловия: | Значение поля Rop меняется на значение  (Operand). |
|  |  |
| *Читать состояние* |  |
| Вход: | Нет. |
| Предусловия: | Нет. |
| Процесс: | Копирует и возвращает значение поля  Operation. |
| Выход: | Значение поля Operation. |
| Постусловия: | Состояние процессора не изменяется. |
|  |  |
| *Записать состояние* |  |
| Вход: | Переменная (Oprtn) типа TOprtn. |
| Предусловия: | Нет. |
| Процесс: | Заносит значение Oprtn в поле Operation. |
| Выход: | Нет. |
| Постусловия: | Значение поля Operation изменяется на  Oprtn. |
|  |  |
| *Читать ошибку* |  |
| Вход: | Нет. |
| Предусловия: | Нет. |
| Процесс: | Копирует и возвращает значение поля Error. |
| Выход: | Значение поля Error. |
| Постусловия: | Состояние процессора не изменяется. |
|  |  |
| *Сброс ошибки* |  |
| Вход: | Нет. |
| Предусловия: | Нет. |
| Процесс: | Очищает поле Error. |
| Выход: | Нет. |
| Постусловия: | Изменяется значение поля Error. |
|  |  |

**Практическая работа**

**Управление калькулятором**

**Задание**

1. Разработать и реализовать класс «Управление универсальным калькулятором» тип TCtrl, используя класс С++.

На Унифицированном языке моделирования UML (Unified Modeling Language) наш класс можно обозначить следующим образом:

|  |
| --- |
| **УправлениеКалькуляторомПростыхДробей (тип TCtrl)** |
| состояниеКалькулятора: TCtrlState |
| редактор: TEditor |
| процессор: TProc |
| память: TMemory |
| число: TANumber |
| выполнитьКомандуКалькулятора(a: Integer; var b, MState: String):  String |
| выполнитьКомандуРедактора(a: Integer): String |
| выполнитьОперацию(a: Integer): String |
| выполнитьФункцию(a: Integer): String |
| вычислитьВыражение(a: Integer): String |
| установитьНачальноеСостояниеКалькулятора(a: Integer): String |
| выполнитьКомандуюПамяти(a: Integer; var MState: String): String |
| читатьПисатьСостояниеКалькулятора: TCtrlState |
| выполнитьКомандуБуфераОбмена(a: Integer; var b: String): String |

|  |
| --- |
| конструктор |
| деструктор |
| Обязанность:  управление выполнением команд калькулятора |

1. Класс должен отвечать за управление выполнением команд калькулятора. Он распределяет команды калькулятора между объектами («редактор»,

«процессор», «память», «буфер обмена»), которые должны эти команды выполнять.

Протестировать каждый метод класса

**Практическая работа**

**Интерфейс**

**Задание**

1. Разработать и реализовать класс «Интерфейс универсального калькулятора» тип TClcPnl наследник TForm, используя С++.

На Унифицированном языке моделирования UML (Unified Modeling Language) наш класс можно обозначить следующим образом:

|  |
| --- |
| **ИнтерфейсКалькулятораПростыхДробей** |
| строкаЧисло: TStaticText |
| состояниеПамяти: TStaticText |
| кнопки ввода: TBitButton |
| FormCreate(Sender: TObject) |
| ButtonClick(Sender: TObject) |
| FormKeyPress(Sender: TObject; var Key: Char) |
| Методы для обработки команд меню |
| Обязанность:  Обеспечить пользователю возможность управления калькулятором через клавиатуру и командные кнопки для выполнения вычислений |

1. Класс должен отвечать: 2.1.за ввод:
   * команд редактирования,
   * команд памяти,
   * команд процессора; 2.2.отображение:
   * вводимого числа,
   * результата вычисления,
   * состояния памяти;
   1. класс должен обеспечить возможность:
      * ввода перечисленных команд с помощью командных кнопок и клавиатуры;
      * выполнение команд для работы с буфером обмена:
        + копировать,
        + вставить;
      * настройки на в зависимости от варианта- типа чисел, обрабатываемых калькулятором.
2. Протестировать каждый метод класса.

**Текст программы**

**TNumber.cs**

using System;

using System.Globalization;

namespace Calculator

{

public abstract class TANumber

{

public abstract TANumber Add(TANumber a);

public abstract TANumber Mul(TANumber a);

public abstract TANumber Div(TANumber a);

public abstract TANumber Sub(TANumber a);

public abstract bool IsZero();

public abstract object Square();

public abstract object Reverse();

public abstract void SetString(string str);

}

public sealed class TNumber : TANumber

{

public double Number

{

get;

}

public TNumber()

{

Number = 0;

}

public TNumber(double n)

{

Number = n;

}

public TNumber(string str)

{

if (double.TryParse(str, NumberStyles.Float, CultureInfo.InvariantCulture, out double NewNumber))

{

Number = NewNumber;

}

else

{

Number = 0;

}

}

public TNumber(TNumber num)

{

Number = num.Number;

}

public override TANumber Add(TANumber a)

{

return new TNumber(Number + (a as TNumber).Number);

}

public override TANumber Div(TANumber a)

{

if ((a as TNumber).IsZero())

{

throw new DivideByZeroException();

}

return new TNumber(Number / (a as TNumber).Number);

}

public override TANumber Sub(TANumber a)

{

return new TNumber(Number - (a as TNumber).Number);

}

public override TANumber Mul(TANumber a)

{

return new TNumber(Number \* (a as TNumber).Number);

}

public override bool IsZero()

{

return Number == 0;

}

public override object Reverse()

{

return new TNumber(1 / Number);

}

public override object Square()

{

return new TNumber(Number \* Number);

}

public override string ToString()

{

return Number.ToString("0.###", CultureInfo.InvariantCulture);

}

public override bool Equals(object obj)

{

var number = obj as TNumber;

return number != null && Number == number.Number;

}

public override void SetString(string str) { }

public static TNumber operator +(TNumber a, TNumber b)

{

return new TNumber(a.Number + b.Number);

}

public static TNumber operator \*(TNumber a, TNumber b)

{

return new TNumber(a.Number \* b.Number);

}

public static TNumber operator -(TNumber a, TNumber b)

{

return new TNumber(a.Number - b.Number);

}

public static TNumber operator /(TNumber a, TNumber b)

{

if (b.IsZero())

{

throw new DivideByZeroException();

}

return new TNumber(a.Number / b.Number);

}

public static bool operator ==(TNumber a, TNumber b)

{

return a.Number == b.Number;

}

public static bool operator ==(TNumber a, int b)

{

return a.Number == b;

}

public static bool operator !=(TNumber a, TNumber b)

{

return a.Number != b.Number;

}

public static bool operator !=(TNumber a, int b)

{

return a.Number != b;

}

public static bool operator >(TNumber a, TNumber b)

{

return (a.Number) > (b.Number);

}

public static bool operator <(TNumber a, TNumber b)

{

return (a.Number < b.Number);

}

public static bool operator >(TNumber a, int b)

{

return (a.Number) > (b);

}

public static bool operator <(TNumber a, int b)

{

return (a.Number < b);

}

public static TNumber operator -(TNumber a)

{

return new TNumber(-a.Number);

}

public static TNumber operator /(TNumber a, int b)

{

if (b == 0)

{

throw new DivideByZeroException();

}

return new TNumber(a.Number / b);

}

public static TNumber operator /(TNumber a, long b)

{

if (b == 0)

{

throw new DivideByZeroException();

}

return new TNumber(a.Number / b);

}

public static TNumber operator +(TNumber a, int b)

{

return new TNumber(a.Number + b);

}

public static TNumber operator -(TNumber a, int b)

{

return new TNumber(a.Number - b);

}

public static TNumber operator \*(TNumber a, int b)

{

return new TNumber(a.Number \* b);

}

}

}

**TFrac.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text.RegularExpressions;

namespace Calculator

{

public sealed class TFrac : TANumber

{

public TNumber Numerator;

public TNumber Denominator;

static void Swap<T>(ref T lhs, ref T rhs)

{

T temp;

temp = lhs;

lhs = rhs;

rhs = temp;

}

public static long GCD(long a, long b)

{

a = Math.Abs(a);

b = Math.Abs(b);

while (b > 0)

{

a %= b;

Swap(ref a, ref b);

}

return a;

}

public TFrac()

{

Numerator = new TNumber(0);

Denominator = new TNumber(1);

}

public TFrac(TNumber a, TNumber b)

{

try

{

if (a < 0 && b < 0)

{

a \*= -1;

b \*= -1;

}

else if (b < 0 && a > 0)

{

b \*= -1;

a \*= -1;

}

else if (a == 0 && b == 0 || b == 0 || a == 0 && b == 1)

{

Numerator = new TNumber(0);

Denominator = new TNumber(1);

return;

}

Numerator = new TNumber(a);

Denominator = new TNumber(b);

long gcdResult = GCD((long)a.Number, (long)b.Number);

if (gcdResult > 1)

{

Numerator /= gcdResult;

Denominator /= gcdResult;

}

}

catch

{

throw new OverflowException();

}

}

public TFrac(int a, int b)

{

if (a < 0 && b < 0)

{

a \*= -1;

b \*= -1;

}

else if (b < 0 && a > 0)

{

b \*= -1;

a \*= -1;

}

else if (a == 0 && b == 0 || b == 0 || a == 0 && b == 1)

{

Numerator = new TNumber(0);

Denominator = new TNumber(1);

return;

}

Numerator = new TNumber(a);

Denominator = new TNumber(b);

long gcdResult = GCD(a, b);

if (gcdResult > 1)

{

Numerator /= gcdResult;

Denominator /= gcdResult;

}

}

public TFrac(string fraction)

{

Regex FracRegex = new Regex(@"^-?(\d+)/(\d+)$");

Regex NumberRegex = new Regex(@"^-?\d+/?$");

if (FracRegex.IsMatch(fraction))

{

List<string> FracParts = fraction.Split('/').ToList();

Numerator = new TNumber(FracParts[0]);

Denominator = new TNumber(FracParts[1]);

if (Denominator.IsZero())

{

Numerator = new TNumber(0);

Denominator = new TNumber(1);

return;

}

long gcdResult = GCD((long)Numerator.Number, (long)Denominator.Number);

if (gcdResult > 1)

{

Numerator /= gcdResult;

Denominator /= gcdResult;

}

return;

}

else if (NumberRegex.IsMatch(fraction))

{

Numerator = new TNumber(fraction);

Denominator = new TNumber(1);

return;

}

else

{

Numerator = new TNumber(0);

Denominator = new TNumber(1);

return;

}

}

public TFrac(TFrac anotherFrac)

{

Numerator = anotherFrac.Numerator;

Denominator = anotherFrac.Denominator;

}

public static TFrac operator +(TFrac a, TFrac b)

{

return new TFrac(a.Numerator \* b.Denominator + a.Denominator \* b.Numerator, a.Denominator \* b.Denominator);

}

public static TFrac operator \*(TFrac a, TFrac b)

{

return new TFrac(a.Numerator \* b.Numerator, a.Denominator \* b.Denominator);

}

public static TFrac operator -(TFrac a, TFrac b)

{

return new TFrac(a.Numerator \* b.Denominator - a.Denominator \* b.Numerator, a.Denominator \* b.Denominator);

}

public static TFrac operator /(TFrac a, TFrac b)

{

if (b.IsZero())

{

throw new DivideByZeroException();

}

return new TFrac(a.Numerator \* b.Denominator, a.Denominator \* b.Numerator);

}

public static TFrac operator -(TFrac a)

{

return new TFrac(-a.Numerator, a.Denominator);

}

public static bool operator ==(TFrac a, TFrac b)

{

return a.Numerator == b.Numerator && a.Denominator == b.Denominator;

}

public static bool operator !=(TFrac a, TFrac b)

{

return a.Numerator != b.Numerator && a.Denominator != b.Denominator;

}

public static bool operator >(TFrac a, TFrac b)

{

return (a.Numerator / a.Denominator) > (b.Numerator / b.Denominator);

}

public static bool operator <(TFrac a, TFrac b)

{

return (a.Numerator / a.Denominator) < (b.Numerator / b.Denominator);

}

public override TANumber Add(TANumber a)

{

return new TFrac(Numerator \* (a as TFrac).Denominator + Denominator \* (a as TFrac).Numerator, Denominator \* (a as TFrac).Denominator);

}

public override TANumber Mul(TANumber a)

{

return new TFrac((a as TFrac).Numerator \* Numerator, (a as TFrac).Denominator \* Denominator);

}

public override TANumber Div(TANumber a)

{

return new TFrac((a as TFrac).Numerator \* Denominator, (a as TFrac).Denominator \* Numerator);

}

public override TANumber Sub(TANumber a)

{

return new TFrac((a as TFrac).Numerator \* Denominator - (a as TFrac).Denominator \* Numerator, (a as TFrac).Denominator \* Denominator);

}

public override object Square()

{

return new TFrac((TNumber)Numerator.Square(), (TNumber)Denominator.Square());

}

public override object Reverse()

{

return new TFrac(Denominator, Numerator);

}

public override bool IsZero()

{

return Numerator.IsZero();

}

public override void SetString(string str)

{

TFrac TempFrac = new TFrac(str);

Numerator = TempFrac.Numerator;

Denominator = TempFrac.Denominator;

}

public override string ToString()

{

return Numerator.ToString() + "/" + Denominator.ToString();

}

public override bool Equals(object obj)

{

var frac = obj as TFrac;

return frac != null &&Numerator == frac.Numerator && Denominator == frac.Denominator;

}

}

}

**TComplex.cs**

**using System;**

**using System.Collections.Generic;**

**using System.Linq;**

**using System.Text.RegularExpressions;**

**namespace Calculator**

**{**

**public sealed class TComplex : TANumber**

**{**

**public TNumber Real;**

**public TNumber Imaginary;**

**const string Separator = " + i \* ";**

**const int OverflowStringLimit = 15;**

**public double Abs()**

**{**

**return Math.Sqrt(Real.Number \* Real.Number + Imaginary.Number \* Imaginary.Number);**

**}**

**public double GetRad()**

**{**

**if (Real > 0)**

**{**

**return Math.Atan((Imaginary / Real).Number);**

**}**

**else if (Real == 0 && Imaginary > 0)**

**{**

**return Math.PI / 2;**

**}**

**else if (Real < 0 && Imaginary.Number >= 0)**

**{**

**return Math.Atan((Imaginary / Real).Number + Math.PI);**

**}**

**else if (Real < 0 && Imaginary.Number < 0)**

**{**

**return Math.Atan((Imaginary / Real).Number - Math.PI);**

**}**

**else if (Real == 0 && Imaginary < 0)**

**{**

**return -Math.PI / 2;**

**}**

**return 0;**

**}**

**public double GetDegree()**

**{**

**return GetRad() \* 180 / Math.PI;**

**}**

**public TComplex Pwr(int n)**

**{**

**return new TComplex(Math.Pow(Abs(), n) \* Math.Cos(n \* GetRad()), Math.Pow(Abs(), n) \* Math.Sin(n \* GetRad()));**

**}**

**public TComplex Root(int n, int i) {**

**if (i >= n || i < 0 || n < 0)**

**{**

**return new TComplex();**

**}**

**return new TComplex(Math.Pow(Abs(), 1.0 / n) \* Math.Cos((GetDegree() + 2 \* Math.PI \* i) / n), Math.Pow(Abs(), 1.0 / n) \* Math.Sin((GetDegree() + 2 \* Math.PI \* i) / n));**

**}**

**public TComplex()**

**{**

**Real = new TNumber(0);**

**Imaginary = new TNumber(0);**

**}**

**public TComplex(double anReal, double anImaginary)**

**{**

**Real = new TNumber(anReal);**

**Imaginary = new TNumber(anImaginary);**

**}**

**public TComplex(int anReal, int anImaginary)**

**{**

**Real = new TNumber(anReal);**

**Imaginary = new TNumber(anImaginary);**

**}**

**public TComplex(TNumber anReal, TNumber anImaginary)**

**{**

**Real = anReal;**

**Imaginary = anImaginary;**

**}**

**public TComplex(TComplex anotherComplex)**

**{**

**Real = anotherComplex.Real;**

**Imaginary = anotherComplex.Imaginary;**

**}**

**public TComplex(string str) {**

**Regex FullNumber = new Regex(@"^-?(\d+.?\d\*)\s+\+\s+i\s+\\*\s+-?(\d+.?\d\*)$");**

**Regex LeftPart = new Regex(@"^-?(\d+.?\d\*)(\s+\+\s+i\s+\\*\s+)?$");**

**if (FullNumber.IsMatch(str))**

**{**

**List<string> Parts = str.Split(new string[] { Separator }, StringSplitOptions.None).ToList();**

**Real = new TNumber(Parts[0]);**

**Imaginary = new TNumber(Parts[1]);**

**}**

**else if (LeftPart.IsMatch(str))**

**{**

**if (str.Contains(Separator))**

**str = str.Replace(Separator, string.Empty);**

**Real = new TNumber(str);**

**Imaginary = new TNumber();**

**}**

**else**

**{**

**Real = new TNumber(0);**

**Imaginary = new TNumber(0);**

**}**

**}**

**public static TComplex operator +(TComplex a, TComplex b)**

**{**

**TComplex toReturn = new TComplex(a.Real + b.Real, a.Imaginary + b.Imaginary);**

**if (toReturn.Real.ToString().Length > OverflowStringLimit)**

**{**

**throw new OverflowException();**

**}**

**else if (toReturn.Imaginary.ToString().Length > OverflowStringLimit)**

**{**

**throw new OverflowException();**

**}**

**return toReturn;**

**}**

**public static TComplex operator \*(TComplex a, TComplex b)**

**{**

**TComplex toReturn = new TComplex(a.Real \* b.Real - a.Imaginary - b.Imaginary, a.Real \* b.Imaginary + b.Imaginary \* a.Real);**

**if (toReturn.Real.ToString().Length > OverflowStringLimit)**

**{**

**throw new OverflowException();**

**}**

**else if (toReturn.Imaginary.ToString().Length > OverflowStringLimit)**

**{**

**throw new OverflowException();**

**}**

**return toReturn;**

**}**

**public static TComplex operator -(TComplex a, TComplex b) {**

**TComplex toReturn = new TComplex(a.Real - b.Real, a.Imaginary - b.Imaginary);**

**if (toReturn.Real.ToString().Length > OverflowStringLimit)**

**{**

**throw new OverflowException();**

**}**

**else if (toReturn.Imaginary.ToString().Length > OverflowStringLimit)**

**{**

**throw new OverflowException();**

**}**

**return toReturn;**

**}**

**public static TComplex operator /(TComplex a, TComplex b) {**

**TComplex toReturn = new TComplex((a.Real \* b.Real + a.Imaginary \* b.Imaginary) / (b.Real \* b.Real + b.Imaginary + b.Imaginary), (b.Real \* a.Imaginary - a.Real \* b.Imaginary) / (b.Real \* b.Real + b.Imaginary \* b.Imaginary));**

**if (toReturn.Real.ToString().Length > OverflowStringLimit)**

**{**

**throw new OverflowException();**

**}**

**else if (toReturn.Imaginary.ToString().Length > OverflowStringLimit)**

**{**

**throw new OverflowException();**

**}**

**return toReturn;**

**}**

**public static TComplex operator -(TComplex a)**

**{**

**return new TComplex(-a.Real, a.Imaginary);**

**}**

**public static bool operator ==(TComplex a, TComplex b)**

**{**

**return (a.Real == b.Real && a.Imaginary == b.Imaginary);**

**}**

**public static bool operator !=(TComplex a, TComplex b)**

**{**

**return (a.Real != b.Real || a.Imaginary != b.Imaginary);**

**}**

**public override TANumber Add(TANumber a)**

**{**

**TComplex toReturn = new TComplex(Real + (a as TComplex).Real, Imaginary + (a as TComplex).Imaginary);**

**if (toReturn.Real.ToString().Length > OverflowStringLimit)**

**{**

**throw new OverflowException();**

**}**

**else if (toReturn.Imaginary.ToString().Length > OverflowStringLimit)**

**{**

**throw new OverflowException();**

**}**

**return toReturn;**

**}**

**public override TANumber Mul(TANumber a)**

**{**

**TComplex toReturn = new TComplex(Real \* (a as TComplex).Real - Imaginary - (a as TComplex).Imaginary, Real \* (a as TComplex).Imaginary + (a as TComplex).Imaginary \* Real);**

**if (toReturn.Real.ToString().Length > OverflowStringLimit)**

**{**

**throw new OverflowException();**

**}**

**else if (toReturn.Imaginary.ToString().Length > OverflowStringLimit)**

**{**

**throw new OverflowException();**

**}**

**return toReturn;**

**}**

**public override TANumber Div(TANumber a)**

**{**

**TComplex toReturn = new TComplex((Real \* (a as TComplex).Real + Imaginary \* (a as TComplex).Imaginary) / ((a as TComplex).Real \* (a as TComplex).Real + (a as TComplex).Imaginary + (a as TComplex).Imaginary), ((a as TComplex).Real \* Imaginary - Real \* (a as TComplex).Imaginary) / ((a as TComplex).Real \* (a as TComplex).Real + (a as TComplex).Imaginary \* (a as TComplex).Imaginary));**

**if (toReturn.Real.ToString().Length > OverflowStringLimit)**

**{**

**throw new OverflowException();**

**}**

**else if (toReturn.Imaginary.ToString().Length > OverflowStringLimit)**

**{**

**throw new OverflowException();**

**}**

**return toReturn;**

**}**

**public override TANumber Sub(TANumber a)**

**{**

**TComplex toReturn = new TComplex(Real - (a as TComplex).Real, Imaginary - (a as TComplex).Imaginary);**

**if (toReturn.Real.ToString().Length > OverflowStringLimit)**

**{**

**throw new OverflowException();**

**}**

**else if (toReturn.Imaginary.ToString().Length > OverflowStringLimit)**

**{**

**throw new OverflowException();**

**}**

**return toReturn;**

**}**

**public override object Square()**

**{**

**TComplex toReturn = new TComplex(Real \* Real - Imaginary \* Imaginary, Real \* Imaginary + Real \* Imaginary);**

**if (toReturn.Real.ToString().Length > OverflowStringLimit)**

**{**

**throw new OverflowException();**

**}**

**else if (toReturn.Imaginary.ToString().Length > OverflowStringLimit)**

**{**

**throw new OverflowException();**

**}**

**return toReturn;**

**}**

**public override object Reverse()**

**{**

**TComplex toReturn = new TComplex(Real / (Real \* Real + Imaginary \* Imaginary), -(Imaginary / (Real \* Real + Imaginary \* Imaginary)));**

**if (toReturn.Real.ToString().Length > OverflowStringLimit)**

**{**

**throw new OverflowException();**

**}**

**else if (toReturn.Imaginary.ToString().Length > OverflowStringLimit)**

**{**

**throw new OverflowException();**

**}**

**return toReturn;**

**}**

**public override bool IsZero()**

**{**

**return Real.IsZero() && Imaginary.IsZero();**

**}**

**public override void SetString(string str)**

**{**

**TComplex temp = new TComplex(str);**

**Real = temp.Real;**

**Imaginary = temp.Imaginary;**

**}**

**public override string ToString()**

**{**

**return Real.ToString() + Separator + Imaginary.ToString();**

**}**

**public override bool Equals(object obj)**

**{**

**var complex = obj as TComplex;**

**return complex != null &&**

**EqualityComparer<TNumber>.Default.Equals(Real, complex.Real) &&**

**EqualityComparer<TNumber>.Default.Equals(Imaginary, complex.Imaginary);**

**}**

**}**

**}**

**TPnumber.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text.RegularExpressions;

namespace Calculator

{

public sealed class TPNumber : TANumber

{

public static class Conver\_10\_p

{

public static string Do(double n, int p, int c)

{

if (p < 2 || p > 16)

{

throw new IndexOutOfRangeException();

}

if (c < 0 || c > 10)

{

throw new IndexOutOfRangeException();

}

string LeftSideString;

string RightSideString;

long LeftSide = 0;

double RightSide = 0f;

try

{

LeftSide = (long)n;

RightSide = n - LeftSide;

if (RightSide < 0)

{

RightSide \*= -1;

}

LeftSideString = int\_to\_P(LeftSide, p);

RightSideString = flt\_to\_P(RightSide, p, c);

}

catch

{

throw new OverflowException();

}

return LeftSideString + (RightSideString == String.Empty ? "" : ".") + RightSideString;

}

public static char int\_to\_Char(long d)

{

if (d < 0 || d > 15)

{

throw new IndexOutOfRangeException();

}

string SymbolArray = "0123456789ABCDEF";

return SymbolArray.ElementAt((int)d);

}

public static string int\_to\_P(long n, long p)

{

if (p < 2 || p > 16)

{

throw new IndexOutOfRangeException();

}

if (n == 0)

{

return "0";

}

bool HaveMinus = false;

if (n < 0)

{

HaveMinus = true;

n \*= -1;

}

string PNumber = string.Empty;

while (n > 0)

{

PNumber += int\_to\_Char(n % p);

n /= p;

}

if (HaveMinus)

{

PNumber += "-";

}

char[] TempArray = PNumber.ToCharArray();

Array.Reverse(TempArray);

return new string(TempArray);

}

public static string flt\_to\_P(double n, int p, int c)

{

if (p < 2 || p > 16)

{

throw new IndexOutOfRangeException();

}

if (c < 0 || c > 10)

{

throw new IndexOutOfRangeException();

}

string PNumber = string.Empty;

for (int i = 0; i < c; ++i)

{

PNumber += int\_to\_Char((int)(n \* p));

n = n \* p - (int)(n \* p);

}

PNumber = PNumber.TrimEnd('0');

return PNumber;

}

}

public static class Conver\_p\_10

{

private static int char\_To\_num(char ch)

{

string AllVariants = "0123456789ABCDEF";

if (!AllVariants.Contains(ch))

{

throw new IndexOutOfRangeException();

}

return AllVariants.IndexOf(ch);

}

private static double convert(string P\_num, int P, double weight)

{

if (weight % P != 0)

{

throw new Exception();

}

long Degree = (long)Math.Ceiling(Math.Log(weight, P)) - 1;

double Result = 0.0f;

for (int i = 0; i < P\_num.Length; ++i, --Degree)

{

Result += char\_To\_num(P\_num.ElementAt(i)) \* Math.Pow(P, Degree);

}

return Result;

}

public static double dval(string P\_num, int P)

{

if (P < 2 || P > 16)

{

throw new IndexOutOfRangeException();

}

bool HaveMinus = false;

if (P\_num.First() == '-')

{

HaveMinus = true;

P\_num = P\_num.Remove(0, 1);

}

foreach (char ch in P\_num)

{

if (ch == '.')

{

continue;

}

if (char\_To\_num(ch) > P)

{

throw new Exception();

}

}

double Number = 0.0f;

Regex LeftRight = new Regex("^[0-9A-F]+\\.[0-9A-F]+$");

Regex Right = new Regex("^0\\.[0-9A-F]+$");

Regex Left = new Regex("^[0-9A-F]+.?$");

if (LeftRight.IsMatch(P\_num))

{

Number = convert(P\_num.Remove(P\_num.IndexOf('.'), 1), P, Math.Pow(P, P\_num.IndexOf('.')));

}

else if (Left.IsMatch(P\_num))

{

if (P\_num.Last() == '.')

{

P\_num = P\_num.Remove(P\_num.Length - 1);

}

Number = convert(P\_num, P, Math.Pow(P, P\_num.Length));

}

else if (Right.IsMatch(P\_num))

{

Number = convert(P\_num.Remove(P\_num.IndexOf('.'), 1), P, 0);

}

else

{

throw new Exception();

}

return HaveMinus ? -Number : Number;

}

}

public TNumber Number;

public TNumber Notation;

public TNumber Precision;

public TPNumber()

{

Number = new TNumber();

Notation = new TNumber(10);

Precision = new TNumber(5);

}

public TPNumber(TNumber num, TNumber not, TNumber pre)

{

if (not < 2 || not > 16 || pre < 0 || pre > 10)

{

Number = new TNumber();

Notation = new TNumber(10);

Precision = new TNumber(5);

}

else

{

Number = new TNumber(num);

Notation = new TNumber(not);

Precision = new TNumber(pre);

}

}

public TPNumber(TNumber num, int not, int pre)

{

if (not < 2 || not > 16 || pre < 0 || pre > 10)

{

Number = new TNumber();

Notation = new TNumber(10);

Precision = new TNumber(5);

}

else

{

Number = new TNumber(num);

Notation = new TNumber(not);

Precision = new TNumber(pre);

}

}

public TPNumber(double num, int not, int pre)

{

if (not < 2 || not > 16 || pre < 0 || pre > 10)

{

Number = new TNumber();

Notation = new TNumber(10);

Precision = new TNumber(5);

}

else {

Number = new TNumber(num);

Notation = new TNumber(not);

Precision = new TNumber(pre);

}

}

public TPNumber(TPNumber anotherTPNumber)

{

Number = anotherTPNumber.Number;

Notation = anotherTPNumber.Notation;

Precision = anotherTPNumber.Precision;

}

public TPNumber(string str, TNumber not, TNumber pre)

{

Notation = not;

Precision = pre;

try

{

Number = new TNumber(Conver\_p\_10.dval(str, Convert.ToInt32(not.Number)));

}

catch

{

throw new System.OverflowException();

}

}

public TPNumber(string str, int not, int pre)

{

try

{

Number = new TNumber(Conver\_p\_10.dval(str, not));

Notation = new TNumber(not);

Precision = new TNumber(pre);

}

catch

{

throw new System.OverflowException();

}

}

public static TPNumber operator +(TPNumber a, TPNumber b)

{

return new TPNumber(a.Number + b.Number, a.Notation, a.Precision);

}

public static TPNumber operator \*(TPNumber a, TPNumber b)

{

return new TPNumber(a.Number \* b.Number, a.Notation, b.Notation);

}

public static TPNumber operator -(TPNumber a, TPNumber b)

{

return new TPNumber(a.Number - b.Number, a.Notation, a.Precision);

}

public static TPNumber operator /(TPNumber a, TPNumber b)

{

return new TPNumber(a.Number / b.Number, a.Notation, a.Precision);

}

public static TPNumber operator -(TPNumber a)

{

return new TPNumber(-a.Number, a.Notation, a.Precision);

}

public static bool operator ==(TPNumber a, TPNumber b)

{

return a.Number == b.Number;

}

public static bool operator !=(TPNumber a, TPNumber b)

{

return a.Number != b.Number;

}

public static bool operator >(TPNumber a, TPNumber b)

{

return a.Number > b.Number;

}

public static bool operator <(TPNumber a, TPNumber b)

{

return a.Number < b.Number;

}

public override TANumber Add(TANumber a)

{

return new TPNumber((a as TPNumber).Number + Number, Notation, Precision);

}

public override TANumber Mul(TANumber a)

{

return new TPNumber((a as TPNumber).Number \* Number, Notation, Precision);

}

public override TANumber Div(TANumber a)

{

return new TPNumber((a as TPNumber).Number / Number, Notation, Precision);

}

public override TANumber Sub(TANumber a)

{

return new TPNumber((a as TPNumber).Number - Number, Notation, Precision);

}

public override object Square()

{

return new TPNumber((TNumber)Number.Square(), Notation, Precision);

}

public override object Reverse()

{

return new TPNumber((TNumber)Number.Reverse(), Notation, Precision);

}

public override bool IsZero()

{

return Number.IsZero();

}

public override void SetString(string str)

{

Number = new TNumber(Conver\_p\_10.dval(str, Convert.ToInt32(Notation.Number)));

}

public override string ToString()

{

string str;

try

{

str = Conver\_10\_p.Do(Number.Number, Convert.ToInt32(Notation.Number), Convert.ToInt32(Precision.Number));

}

catch

{

throw new OverflowException();

}

return str;

}

public override bool Equals(object obj)

{

var number = obj as TPNumber;

return number != null &&

EqualityComparer<TNumber>.Default.Equals(Number, number.Number) &&

EqualityComparer<TNumber>.Default.Equals(Notation, number.Notation) &&

EqualityComparer<TNumber>.Default.Equals(Precision, number.Precision);

}

}

}

**TAEditor.cs**

using System;

namespace Calculator

{

public abstract class TAEditor

{

public abstract string Number

{

get;

set;

}

public enum Command

{

cZero, cOne, cTwo, cThree, cFour, cFive, cSix,

cSeven, cEight, cNine, cA, cB, cC, cD, cE, cF,

cSign, cSeparator, cBS, CE, cNumbSeparator, cNone

}

public abstract bool IsZero();

public abstract string ToogleSign();

public abstract string AddNumber(int num);

public abstract string AddZero();

public abstract string RemoveSymbol();

public abstract string Clear();

public abstract string Edit(Enum com);

public abstract string AddSeparator();

public abstract bool HaveSeparator();

}

}

**TFracEditor.cs**

using System;

using System.Linq;

namespace Calculator

{

public sealed class TFracEditor : TAEditor

{

const int BothSideLimit = 22;

const int LeftSideOnlyLimit = 14;

const string Separator = "/";

const string ZeroFraction = "0/";

private string number;

public override string Number

{

get => number;

set

{

number = new TFrac(value).ToString();

}

}

public TFracEditor()

{

number = "0";

}

public TFracEditor(int a, int b)

{

number = new TFrac(a, b).ToString();

}

public TFracEditor(string str)

{

number = new TFrac(str).ToString();

}

public override bool IsZero()

{

return number.StartsWith(ZeroFraction) || number == "0";

}

public override string ToogleSign()

{

if (number.ElementAt(0) == '-')

{

number = number.Remove(0, 1);

}

number = "-" + number;

return number;

}

public override string AddNumber(int a)

{

if ((!HaveSeparator() && number.Length > LeftSideOnlyLimit)

|| (number.Length > BothSideLimit)

|| (a < 0 || a > 9))

{

return number;

}

if (a == 0)

{

AddZero();

}

else if (number == "0" || number == "-0")

{

number = number.First() == '-' ? "-" + a.ToString() : a.ToString();

}

else

{

number += a.ToString();

}

return number;

}

public override bool Equals(object obj)

{

return obj is TFracEditor editor && number == editor.number;

}

public override string AddZero()

{

if ((HaveSeparator() && number.Last().ToString() == Separator)

|| (number == "0" || number == "0/"))

{

return number;

}

number += "0";

return number;

}

public override string RemoveSymbol()

{

if (number.Length == 1)

{

number = "0";

}

else if (number.Length == 2 && number.First() == '-')

{

number = "-0";

}

else

{

number = number.Remove(number.Length - 1);

}

return number;

}

public override string Clear()

{

number = "0";

return number;

}

public override string Edit(Enum com)

{

switch (com)

{

case Command.cZero:

AddZero();

break;

case Command.cOne:

AddNumber(1);

break;

case Command.cTwo:

AddNumber(2);

break;

case Command.cThree:

AddNumber(3);

break;

case Command.cFour:

AddNumber(4);

break;

case Command.cFive:

AddNumber(5);

break;

case Command.cSix:

AddNumber(6);

break;

case Command.cSeven:

AddNumber(7);

break;

case Command.cEight:

AddNumber(8);

break;

case Command.cNine:

AddNumber(9);

break;

case Command.cSign:

ToogleSign();

break;

case Command.cSeparator:

AddSeparator();

break;

case Command.cBS:

RemoveSymbol();

break;

case Command.CE:

Clear();

break;

default:

break;

}

return Number;

}

public override string AddSeparator()

{

if (!number.Contains(Separator))

{

number += Separator;

}

return number;

}

public override bool HaveSeparator()

{

return number.Contains(Separator);

}

public override string ToString()

{

return Number;

}

}

}

**TComplexEditor.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text.RegularExpressions;

namespace Calculator

{

public sealed class TComplexEditor : TAEditor

{

private string number;

const string Separator = " + i \* ";

readonly Regex ZeroComplex = new Regex(@"^-?(0+.?0\*)(\s\*\+\s\*i\s\*\\*\s\*-?(0+.?0\*)|(\s\*\+\s\*i\s\*\\*\s\*-?))?$");

public override string Number

{

get => number;

set

{

number = new TComplex(value).ToString();

}

}

public TComplexEditor()

{

number = "0";

}

public TComplexEditor(int a, int b)

{

number = new TComplex(a, b).ToString();

}

public TComplexEditor(string str)

{

number = new TComplex(str).ToString();

}

public override bool IsZero()

{

return ZeroComplex.IsMatch(number);

}

public override string ToogleSign()

{

if (HaveSeparator())

{

List<string> Parts = new List<string>();

Parts = Number.Split(new string[] { Separator }, StringSplitOptions.None).ToList();

if (Parts[0].First() == '-')

{

Parts[0] = Parts[0].Remove(0, 1);

}

else

{

Parts[0] = '-' + Parts[0];

}

if (Parts[1].First() == '-')

{

Parts[1] = Parts[1].Remove(0, 1);

}

else

{

Parts[1] = '-' + Parts[1];

}

number = Parts[0] + Separator + Parts[1];

return number;

}

if (number.First() == '-')

{

number = number.Remove(0, 1);

}

else

{

number = '-' + number;

}

return number;

}

public override string AddNumber(int a)

{

if (a < 0 || a > 9)

{

return number;

}

if (a == 0)

{

AddZero();

}

else if (number == "0" || number == "-0")

{

number = number.First() == '-' ? "-" + a.ToString() : a.ToString();

}

else if (number.EndsWith(" 0") || number.EndsWith(" -0"))

{

number = number.Remove(number.Length - 1) + a.ToString();

}

else

{

number += a.ToString();

}

return number;

}

public override string AddZero()

{

if (number == "0" || number == "-0"

|| number.EndsWith(" 0") || number.EndsWith(" -0")

|| number.EndsWith(Separator))

{

return number;

}

number += "0";

return number;

}

public override string RemoveSymbol()

{

if (number.Length == 1)

{

number = "0";

}

else if (number.Length == 2 && Number.First() == '-')

{

number = "-0";

}

else if (HaveSeparator() && Number.ElementAt(Number.Length - 2) == ' ')

{

number = number.Remove(number.IndexOf(Separator));

}

else

{

number = number.Remove(number.Length - 1);

}

return number;

}

public override string Clear()

{

number = "0";

return Number;

}

public override string Edit(Enum com)

{

switch (com)

{

case Command.cZero:

AddZero();

break;

case Command.cOne:

AddNumber(1);

break;

case Command.cTwo:

AddNumber(2);

break;

case Command.cThree:

AddNumber(3);

break;

case Command.cFour:

AddNumber(4);

break;

case Command.cFive:

AddNumber(5);

break;

case Command.cSix:

AddNumber(6);

break;

case Command.cSeven:

AddNumber(7);

break;

case Command.cEight:

AddNumber(8);

break;

case Command.cNine:

AddNumber(9);

break;

case Command.cSign:

ToogleSign();

break;

case Command.cSeparator:

AddNumberSeparator();

break;

case Command.cBS:

RemoveSymbol();

break;

case Command.CE:

Clear();

break;

case Command.cNumbSeparator:

AddSeparator();

break;

default:

break;

}

return Number;

}

public override string AddSeparator()

{

if (!HaveSeparator())

{

Number = string.Concat(Number, Separator, "0");

}

return Number;

}

public override bool HaveSeparator()

{

return Number.Contains(Separator);

}

public override string ToString()

{

return Number;

}

public string AddNumberSeparator()

{

if (!HaveSeparator() && !number.Contains("."))

{

number += ".";

}

else if (HaveSeparator())

{

List<string> Parts = new List<string>();

Parts = Number.Split(new string[] { Separator }, StringSplitOptions.None).ToList();

if (!Parts[1].Contains("."))

{

number += ".";

}

}

return number;

}

}

}

**TPnumberEditor.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text.RegularExpressions;

namespace Calculator

{

public sealed class TPNumberEditor : TAEditor

{

private string number;

public TNumber Notation;

public TNumber Precision;

const int BothSideLimit = 22;

const int LeftSideOnlyLimit = 12;

const string Separator = ".";

readonly Regex ZeroPNumber = new Regex("^-?(0+|.?0+|0+.(0+)?)$");

public override string Number

{

get

{

return number;

}

set

{

number = new TPNumber(value, Notation, Precision).ToString();

}

}

public TPNumberEditor()

{

number = "0";

Notation = new TNumber(10);

Precision = new TNumber(2);

}

public TPNumberEditor(string str, TNumber not, TNumber pre)

{

if (not < 2 || not > 16 || pre < 0 || pre > 10)

{

number = "0";

Notation = new TNumber(10);

Precision = new TNumber(2);

}

else

{

Notation = not;

Precision = pre;

number = new TPNumber(str, Notation, Precision).ToString();

}

}

public TPNumberEditor(double num, TNumber not, TNumber pre)

{

if (not < 2 || not > 16 || pre < 0 || pre > 10)

{

number = "0";

Notation = new TNumber(10);

Precision = new TNumber(2);

}

else

{

Notation = not;

Precision = pre;

number = new TPNumber(num, Convert.ToInt32(Notation.Number), Convert.ToInt32(Precision.Number)).ToString(); ;

}

}

public TPNumberEditor(double num, int not, int pre)

{

if (not < 2 || not > 16 || pre < 0 || pre > 10)

{

number = "0";

Notation = new TNumber(10);

Precision = new TNumber(2);

}

else {

Notation = new TNumber(not);

Precision = new TNumber(pre);

number = TPNumber.Conver\_10\_p.Do(num, not, pre);

}

}

public TPNumberEditor(string str)

{

Notation = new TNumber(10);

Precision = new TNumber(2);

number = new TPNumber(str, Notation, Precision).ToString();

}

public override bool IsZero()

{

return ZeroPNumber.IsMatch(number);

}

public override string ToogleSign()

{

if (number.ElementAt(0) == '-')

{

number = number.Remove(0, 1);

}

else

{

number = "-" + number;

}

return number;

}

public override string AddNumber(int num)

{

if (!HaveSeparator() && number.Length > LeftSideOnlyLimit)

return number;

else if (number.Length > BothSideLimit)

return number;

if (num < 0 || num >= Notation.Number)

return number;

if (num == 0)

AddZero();

else if (number == "0" || number == "-0")

number = number.First() == '-' ? "-" + TPNumber.Conver\_10\_p.int\_to\_Char(num).ToString() : TPNumber.Conver\_10\_p.int\_to\_Char(num).ToString();

else

number += TPNumber.Conver\_10\_p.int\_to\_Char(num).ToString();

return number;

}

public override bool Equals(object obj)

{

var editor = obj as TPNumberEditor;

return editor != null &&

number == editor.number &&

EqualityComparer<TNumber>.Default.Equals(Notation, editor.Notation) &&

EqualityComparer<TNumber>.Default.Equals(Precision, editor.Precision) &&

Number == editor.Number &&

EqualityComparer<Regex>.Default.Equals(ZeroPNumber, editor.ZeroPNumber);

}

public override string RemoveSymbol()

{

if (number.Length == 1)

{

number = "0";

}

else if (number.Length == 2 && number.First() == '-')

{

number = "-0";

}

else

{

number = number.Remove(number.Length - 1);

}

return number;

}

public override string Clear()

{

number = "0";

return number;

}

public override string Edit(Enum com)

{

switch (com) {

case Command.cZero:

AddZero();

break;

case Command.cOne:

AddNumber(1);

break;

case Command.cTwo:

AddNumber(2);

break;

case Command.cThree:

AddNumber(3);

break;

case Command.cFour:

AddNumber(4);

break;

case Command.cFive:

AddNumber(5);

break;

case Command.cSix:

AddNumber(6);

break;

case Command.cSeven:

AddNumber(7);

break;

case Command.cEight:

AddNumber(8);

break;

case Command.cNine:

AddNumber(9);

break;

case Command.cA:

AddNumber(10);

break;

case Command.cB:

AddNumber(11);

break;

case Command.cC:

AddNumber(12);

break;

case Command.cD:

AddNumber(13);

break;

case Command.cE:

AddNumber(14);

break;

case Command.cF:

AddNumber(15);

break;

case Command.cSign:

ToogleSign();

break;

case Command.cSeparator:

AddSeparator();

break;

case Command.cBS:

RemoveSymbol();

break;

case Command.CE:

Clear();

break;

default:

break;

}

return Number;

}

public override string AddSeparator()

{

if (!number.Contains(Separator))

{

number += Separator;

}

return number;

}

public override bool HaveSeparator()

{

return number.Contains(Separator);

}

public override string AddZero()

{

if (HaveSeparator() && number.Last().ToString() == Separator

|| (number == "0" || number == "0."))

{

return number;

}

number += "0";

return number;

}

public override string ToString()

{

return number;

}

}

}

**TProc.cs**

namespace Calculator

{

public sealed class TProc<T> where T : TANumber, new()

{

public enum Oper

{

None, Add, Sub, Mul, Div

}

public enum Func

{

Rev, Sqr

}

Oper operation;

T lop\_Res;

T rop;

public T Lop\_Res

{

get => lop\_Res;

set => lop\_Res = value;

}

public T Rop

{

get => rop;

set => rop = value;

}

public Oper Operation

{

get => operation;

set => operation = value;

}

public TProc()

{

operation = Oper.None;

Lop\_Res = new T();

Rop = new T();

}

public TProc(T leftObj, T rightObj)

{

operation = Oper.None;

Lop\_Res = leftObj;

Rop = rightObj;

}

public void Reset()

{

operation = Oper.None;

T newObj = new T();

Lop\_Res = Rop = newObj;

}

public void DoOper()

{

try

{

dynamic a = Lop\_Res;

dynamic b = Rop;

switch (operation)

{

case Oper.Add:

Lop\_Res = (T)(a + b);

break;

case Oper.Sub:

Lop\_Res = (T)(a - b);

break;

case Oper.Mul:

Lop\_Res = (T)(a \* b);

break;

case Oper.Div:

Lop\_Res = (T)(a / b);

break;

default:

break;

}

}

catch

{

throw new System.OverflowException();

}

}

public void DoFunc(Func func)

{

dynamic a = Rop;

switch (func)

{

case Func.Rev:

a = a.Reverse();

Rop = (T)a;

break;

case Func.Sqr:

a = a.Square();

Rop = (T)a;

break;

default:

break;

}

}

}

}

**TCtrl.cs**

namespace Calculator

{

public sealed class TCtrl<T, Editor>

where T : TANumber, new()

where Editor : TAEditor, new()

{

public enum TCtrlState

{

cStart, cEditing, FunDone, cOperDone, cExpDone, cOpChange, cError

}

Editor edit;

TProc<T> proc;

TMemory<T> memory;

TCtrlState curState;

public TCtrlState CurState

{

get => curState;

set => curState = value;

}

public TProc<T> Proc

{

get => proc;

set => proc = value;

}

public TMemory<T> Memory

{

get => memory;

set => memory = value;

}

public Editor Edit

{

get => edit;

set => edit = value;

}

public TCtrl()

{

Edit = new Editor();

Proc = new TProc<T>();

Memory = new TMemory<T>();

curState = TCtrlState.cStart;

}

public string ExecCommandEditor(TAEditor.Command command)

{

string ToReturn;

if (CurState == TCtrlState.cExpDone)

{

Proc.Reset();

CurState = TCtrlState.cStart;

}

if (CurState != TCtrlState.cStart)

{

CurState = TCtrlState.cEditing;

}

ToReturn = Edit.Edit(command);

T TempObj = new T();

if (TempObj is TPNumber)

{

dynamic a = TempObj;

dynamic b = Edit;

a.Notation = new TNumber(b.Notation);

TempObj = a;

}

TempObj.SetString(ToReturn);

Proc.Rop = TempObj;

return ToReturn;

}

public string ExecOperation(TProc<T>.Oper oper)

{

if (oper == TProc<T>.Oper.None)

{

return Edit.Number;

}

string ToReturn;

try

{

switch (CurState)

{

case TCtrlState.cStart:

Proc.Lop\_Res = Proc.Rop;

Proc.Operation = oper;

CurState = TCtrlState.cOperDone;

Edit.Clear();

break;

case TCtrlState.cEditing:

Proc.DoOper();

Proc.Operation = oper;

Edit.Clear();

CurState = TCtrlState.cOperDone;

break;

case TCtrlState.FunDone:

if (Proc.Operation == TProc<T>.Oper.None)

{

Proc.Lop\_Res = Proc.Rop;

}

else

{

Proc.DoOper();

}

Proc.Operation = oper;

Edit.Clear();

CurState = TCtrlState.cOpChange;

break;

case TCtrlState.cOperDone:

CurState = TCtrlState.cOpChange;

Edit.Clear();

break;

case TCtrlState.cExpDone:

Proc.Operation = oper;

Proc.Rop = Proc.Lop\_Res;

CurState = TCtrlState.cOpChange;

Edit.Clear();

break;

case TCtrlState.cError:

Proc.Reset();

return "ERR";

case TCtrlState.cOpChange:

Proc.Operation = oper;

Edit.Clear();

break;

default:

break;

}

ToReturn = Proc.Lop\_Res.ToString();

}

catch

{

Reset();

return "ERROR";

}

return ToReturn;

}

public string ExecFunction(TProc<T>.Func func)

{

string ToReturn;

try

{

if (CurState == TCtrlState.cExpDone)

{

Proc.Rop = Proc.Lop\_Res;

Proc.Operation = TProc<T>.Oper.None;

}

Proc.DoFunc(func);

CurState = TCtrlState.FunDone;

ToReturn = Proc.Rop.ToString();

}

catch

{

Reset();

return "ERROR";

}

return ToReturn;

}

public string Calculate()

{

string ToReturn;

try

{

if (CurState == TCtrlState.cStart)

{

Proc.Lop\_Res = Proc.Rop;

}

Proc.DoOper();

CurState = TCtrlState.cExpDone;

ToReturn = Proc.Lop\_Res.ToString();

}

catch

{

Reset();

return "ERROR";

}

return ToReturn;

}

public string Reset()

{

Edit.Clear();

Proc.Reset();

Memory.Clear();

curState = TCtrlState.cStart;

return Edit.ToString();

}

public (T, TMemory<T>.NumStates) ExecCommandMemory(TMemory<T>.Commands command, string str)

{

T TempObj = new T();

TempObj.SetString(str);

(T, TMemory<T>.NumStates) obj = (null, TMemory<T>.NumStates.OFF);

try

{

obj = Memory.Edit(command, TempObj);

}

catch

{

Reset();

return obj;

}

if (command == TMemory<T>.Commands.Copy)

{

Edit.Number = obj.Item1.ToString();

Proc.Rop = obj.Item1;

}

return obj;

}

}

}

**TMemory.cs**

namespace Calculator

{

public sealed class TMemory<T> where T : TANumber, new()

{

public enum NumStates

{

OFF, ON

}

public enum Commands

{

Store, Add, Clear, Copy

}

T fNumber;

NumStates fState;

public T FNumber

{

get

{

fState = NumStates.ON; return fNumber;

}

set

{

fNumber = value;

fState = NumStates.ON;

}

}

public NumStates FState

{

get => fState;

set => fState = value;

}

public TMemory()

{

FNumber = new T();

FState = NumStates.OFF;

}

public TMemory(T number)

{

FNumber = number;

FState = NumStates.OFF;

}

public T Add(T number)

{

FState = NumStates.ON;

dynamic a = fNumber;

dynamic b = number;

fNumber = (T)(a + b);

return fNumber;

}

public void Clear()

{

fNumber = new T();

FState = NumStates.OFF;

}

public (T, NumStates) Edit(Commands command, T newNumber)

{

switch (command)

{

case Commands.Store:

FState = NumStates.ON;

fNumber = newNumber;

break;

case Commands.Add:

FState = NumStates.ON;

dynamic a = fNumber;

dynamic b = newNumber;

fNumber = (T)(a + b);

break;

case Commands.Clear:

Clear();

break;

}

return (fNumber, fState);

}

}

}

**Form1.cs (интерфейс)**

using System;

using System.Linq;

using System.Windows.Forms;

namespace Calculator

{

public partial class Form1 : Form

{

readonly TCtrl<TFrac, TFracEditor> frac\_ctrl;

readonly TCtrl<TPNumber, TPNumberEditor> pNumber\_ctrl;

readonly TCtrl<TComplex, TComplexEditor> complex\_ctrl;

const string OPERATIONS = "+-/\*";

const string TAG\_FRAC = "FRAC\_";

const string TAG\_COMPLEX = "COMPLEX\_";

const string TAG\_PNUMBER = "PNUMBER\_";

bool frac\_mode = true;

bool pNumber\_mode = true;

bool complex\_mode = true;

enum Complex\_func

{

Pwr, Root, Abs, Dgr, Rad

}

private string Number\_to\_state(string tag, string str)

{

if ("ERROR" == str)

{

return str;

}

string return\_str = str;

switch (tag)

{

case TAG\_PNUMBER:

break;

case TAG\_FRAC:

if (true == frac\_mode)

{

return\_str = str;

}

else if (new TFrac(str).Denominator == 1)

{

return\_str = new TFrac(str).Numerator.ToString();

}

break;

case TAG\_COMPLEX:

if (true == complex\_mode)

{

return\_str = str;

}

else if (new TComplex(str).Imaginary == 0)

{

return\_str = new TComplex(str).Real.ToString();

}

break;

}

return return\_str;

}

private static TAEditor.Command Char\_to\_edit\_command(char ch)

{

TAEditor.Command command = TAEditor.Command.cNone;

switch (ch)

{

case '0':

command = TAEditor.Command.cZero;

break;

case '1':

command = TAEditor.Command.cOne;

break;

case '2':

command = TAEditor.Command.cTwo;

break;

case '3':

command = TAEditor.Command.cThree;

break;

case '4':

command = TAEditor.Command.cFour;

break;

case '5':

command = TAEditor.Command.cFive;

break;

case '6':

command = TAEditor.Command.cSix;

break;

case '7':

command = TAEditor.Command.cSeven;

break;

case '8':

command = TAEditor.Command.cEight;

break;

case '9':

command = TAEditor.Command.cNine;

break;

case 'A':

command = TAEditor.Command.cA;

break;

case 'B':

command = TAEditor.Command.cB;

break;

case 'C':

command = TAEditor.Command.cC;

break;

case 'D':

command = TAEditor.Command.cD;

break;

case 'E':

command = TAEditor.Command.cE;

break;

case 'F':

command = TAEditor.Command.cF;

break;

case '.':

case '/':

command = TAEditor.Command.cSeparator;

break;

case '-':

command = TAEditor.Command.cSign;

break;

}

return command;

}

private static TProc<T>.Oper Char\_to\_operation\_command<T>(char ch) where T : TANumber, new()

{

TProc<T>.Oper command = TProc<T>.Oper.None;

switch (ch)

{

case '+':

command = TProc<T>.Oper.Add;

break;

case '-':

command = TProc<T>.Oper.Sub;

break;

case '\*':

command = TProc<T>.Oper.Mul;

break;

case '/':

command = TProc<T>.Oper.Div;

break;

}

return command;

}

private static TAEditor.Command Keycode\_to\_edit\_command(Keys ch)

{

TAEditor.Command command = TAEditor.Command.cNone;

switch (ch)

{

case Keys.Back:

command = TAEditor.Command.cBS;

break;

case Keys.Delete:

case Keys.Escape:

command = TAEditor.Command.CE;

break;

}

return command;

}

public Form1()

{

frac\_ctrl = new TCtrl<TFrac, TFracEditor>();

pNumber\_ctrl = new TCtrl<TPNumber, TPNumberEditor>();

complex\_ctrl = new TCtrl<TComplex, TComplexEditor>();

InitializeComponent();

Size = new System.Drawing.Size(355, 433);

}

private void Button\_Number\_Edit(object sender, EventArgs e)

{

TAEditor.Command parsed\_command;

Button button = (Button)sender;

string tag = button.Tag.ToString();

string part\_tag = tag.Substring(0, tag.IndexOf('\_')) + "\_";

switch (part\_tag)

{

case TAG\_PNUMBER:

pNumber\_ctrl.Edit.Notation = new TNumber(trackBar\_PNumber.Value);

Enum.TryParse(tag.Replace(TAG\_PNUMBER, string.Empty), out parsed\_command);

tB\_PNumber.Text = pNumber\_ctrl.ExecCommandEditor(parsed\_command);

break;

case TAG\_FRAC:

Enum.TryParse(tag.Replace(TAG\_FRAC, string.Empty), out parsed\_command);

tB\_Frac.Text = frac\_ctrl.ExecCommandEditor(parsed\_command);

break;

case TAG\_COMPLEX:

Enum.TryParse(tag.Replace(TAG\_COMPLEX, string.Empty), out parsed\_command);

tB\_Complex.Text = complex\_ctrl.ExecCommandEditor(parsed\_command);

break;

}

}

private void Button\_Number\_Operation(object sender, EventArgs e)

{

TProc<TFrac>.Oper parsed\_frac\_oper;

TProc<TPNumber>.Oper parsed\_pNumber\_oper;

TProc<TComplex>.Oper parsed\_complex\_oper;

Button button = (Button)sender;

string tag = button.Tag.ToString();

string part\_tag = tag.Substring(0, tag.IndexOf('\_')) + "\_";

switch (part\_tag)

{

case TAG\_PNUMBER:

Enum.TryParse(tag.Replace(TAG\_PNUMBER, string.Empty), out parsed\_pNumber\_oper);

tB\_PNumber.Text = pNumber\_ctrl.ExecOperation(parsed\_pNumber\_oper);

break;

case TAG\_FRAC:

Enum.TryParse(tag.Replace(TAG\_FRAC, string.Empty), out parsed\_frac\_oper);

tB\_Frac.Text = Number\_to\_state(TAG\_FRAC, frac\_ctrl.ExecOperation(parsed\_frac\_oper));

break;

case TAG\_COMPLEX:

Enum.TryParse(tag.Replace(TAG\_COMPLEX, string.Empty), out parsed\_complex\_oper);

tB\_Complex.Text = Number\_to\_state(TAG\_COMPLEX, complex\_ctrl.ExecOperation(parsed\_complex\_oper));

break;

}

}

private void Button\_Number\_Function(object sender, EventArgs e)

{

TProc<TFrac>.Func parsed\_frac\_func;

TProc<TPNumber>.Func parsed\_pNumber\_func;

TProc<TComplex>.Func parsed\_complex\_func;

Button button = (Button)sender;

string tag = button.Tag.ToString();

string part\_tag = tag.Substring(0, tag.IndexOf('\_')) + "\_";

switch (part\_tag)

{

case TAG\_PNUMBER:

Enum.TryParse(tag.Replace(TAG\_PNUMBER, string.Empty), out parsed\_pNumber\_func);

tB\_PNumber.Text = pNumber\_ctrl.ExecFunction(parsed\_pNumber\_func);

break;

case TAG\_FRAC:

Enum.TryParse(tag.Replace(TAG\_FRAC, string.Empty), out parsed\_frac\_func);

tB\_Frac.Text = Number\_to\_state(TAG\_FRAC, frac\_ctrl.ExecFunction(parsed\_frac\_func));

break;

case TAG\_COMPLEX:

Enum.TryParse(tag.Replace(TAG\_COMPLEX, string.Empty), out parsed\_complex\_func);

tB\_Complex.Text = Number\_to\_state(TAG\_COMPLEX, complex\_ctrl.ExecFunction(parsed\_complex\_func));

break;

}

}

private void Button\_Reset(object sender, EventArgs e)

{

Button button = (Button)sender;

string tag = button.Tag.ToString();

string part\_tag = tag.Substring(0, tag.IndexOf('\_')) + "\_";

switch (part\_tag)

{

case TAG\_PNUMBER:

tB\_PNumber.Text = pNumber\_ctrl.Reset();

label\_PNumber\_Memory.Text = string.Empty;

break;

case TAG\_FRAC:

tB\_Frac.Text = frac\_ctrl.Reset();

label\_Frac\_Memory.Text = string.Empty;

break;

case TAG\_COMPLEX:

tB\_Complex.Text = complex\_ctrl.Reset();

tB\_Complex\_SpecialOut.Text = label\_Complex\_Memory.Text = string.Empty;

break;

}

}

private void Button\_FinishEval(object sender, EventArgs e)

{

Button button = (Button)sender;

string tag = button.Tag.ToString();

string part\_tag = tag.Substring(0, tag.IndexOf('\_')) + "\_";

switch(part\_tag)

{

case TAG\_PNUMBER:

tB\_PNumber.Text = pNumber\_ctrl.Calculate();

break;

case TAG\_FRAC:

tB\_Frac.Text = Number\_to\_state(TAG\_FRAC, frac\_ctrl.Calculate()); ;

break;

case TAG\_COMPLEX:

tB\_Complex.Text = Number\_to\_state(TAG\_COMPLEX, complex\_ctrl.Calculate());

break;

}

}

private void Button\_Memory(object sender, EventArgs e)

{

TMemory<TPNumber>.Commands parsed\_pNumber\_command;

TMemory<TFrac>.Commands parsed\_frac\_command;

TMemory<TComplex>.Commands parsed\_complex\_command;

dynamic exec;

Button button = (Button)sender;

string tag = button.Tag.ToString();

string part\_tag = tag.Substring(0, tag.IndexOf('\_')) + "\_";

switch(part\_tag)

{

case TAG\_PNUMBER:

Enum.TryParse(tag.Replace(TAG\_PNUMBER, string.Empty), out parsed\_pNumber\_command);

exec = pNumber\_ctrl.ExecCommandMemory(parsed\_pNumber\_command, tB\_PNumber.Text);

if (TMemory<TPNumber>.Commands.Copy == parsed\_pNumber\_command)

{

tB\_PNumber.Text = exec.Item1.ToString();

}

label\_PNumber\_Memory.Text = exec.Item2 == TMemory<TPNumber>.NumStates.ON ? "M" : string.Empty;

break;

case TAG\_FRAC:

Enum.TryParse(tag.Replace(TAG\_FRAC, string.Empty), out parsed\_frac\_command);

exec = frac\_ctrl.ExecCommandMemory(parsed\_frac\_command, tB\_Frac.Text);

if (TMemory<TFrac>.Commands.Copy == parsed\_frac\_command)

{

tB\_Frac.Text = exec.Item1.ToString();

}

label\_Frac\_Memory.Text = exec.Item2 == TMemory<TFrac>.NumStates.ON ? "M" : string.Empty;

break;

case TAG\_COMPLEX:

Enum.TryParse(tag.Replace(TAG\_COMPLEX, string.Empty), out parsed\_complex\_command);

exec = complex\_ctrl.ExecCommandMemory(parsed\_complex\_command, tB\_Complex.Text);

if (TMemory<TComplex>.Commands.Copy == parsed\_complex\_command)

{

tB\_Complex.Text = exec.Item1.ToString();

}

label\_Complex\_Memory.Text = exec.Item2 == TMemory<TComplex>.NumStates.ON ? "M" : string.Empty;

break;

}

}

private void СправкаToolStripMenuItem\_Click(object sender, EventArgs e)

{

MessageBox.Show("Калькулятор чисел", "Калькулятор", MessageBoxButtons.OK, MessageBoxIcon.Information);

}

private void TrackBar\_PNumber\_ValueChanged(object sender, EventArgs e)

{

label\_PNumber\_P.Text = trackBar\_PNumber.Value.ToString();

pNumber\_ctrl.Edit.Notation = new TNumber(trackBar\_PNumber.Value);

tB\_PNumber.Text = pNumber\_ctrl.Reset();

label\_PNumber\_Memory.Text = string.Empty;

string AllowedEndings = "0123456789ABCDEF";

foreach (Control i in tabPage\_PNumber.Controls.OfType<Button>())

{

if (AllowedEndings.Contains(i.Name.ToString().Last()) && i.Name.ToString().Substring(i.Name.ToString().Length - 2, 1) == "\_")

{

int j = AllowedEndings.IndexOf(i.Name.ToString().Last());

if (j < trackBar\_PNumber.Value)

{

i.Enabled = true;

}

if ((j >= trackBar\_PNumber.Value) && (j <= 15))

{

i.Enabled = false;

}

}

}

pNumber\_ctrl.Proc.Lop\_Res.Notation = new TNumber(trackBar\_PNumber.Value);

pNumber\_ctrl.Proc.Rop.Notation = new TNumber(trackBar\_PNumber.Value);

}

private void Button\_Complex\_Special(object sender, EventArgs e)

{

try

{

Button button = (Button)sender;

Enum.TryParse(button.Tag.ToString().Replace(TAG\_COMPLEX, string.Empty), out Complex\_func ParsedEnum);

TComplex number = new TComplex(tB\_Complex.Text);

switch (ParsedEnum)

{

case Complex\_func.Pwr:

int PwrN = Convert.ToInt32(nUD\_Complex\_Pwr.Value);

tB\_Complex\_SpecialOut.Text = number.Pwr(PwrN).ToString();

break;

case Complex\_func.Root:

int RootN = Convert.ToInt32(nUD\_Complex\_Root\_N.Value);

int Rooti = Convert.ToInt32(nUD\_Complex\_Root\_i.Value);

tB\_Complex\_SpecialOut.Text = number.Root(RootN, Rooti).ToString();

break;

case Complex\_func.Abs:

tB\_Complex\_SpecialOut.Text = number.Abs().ToString();

break;

case Complex\_func.Dgr:

tB\_Complex\_SpecialOut.Text = number.GetDegree().ToString();

break;

case Complex\_func.Rad:

tB\_Complex\_SpecialOut.Text = number.GetRad().ToString();

break;

}

}

catch

{

tB\_Complex\_SpecialOut.Text = "ERROR";

}

}

private void Button\_Complex\_ShowAllRoots(object sender, EventArgs e)

{

ShowRoots RootsForm = new ShowRoots();

int RootN = Convert.ToInt32(nUD\_Complex\_Root\_N.Value);

TComplex number = new TComplex(tB\_Complex.Text);

for (int i = 0; i < RootN; ++i)

{

RootsForm.richTB\_Roots.Text += "Root " + i.ToString() + ": " + number.Root(RootN, i).ToString() + Environment.NewLine;

}

RootsForm.ShowDialog();

}

private void NUD\_Complex\_Root\_N\_ValueChanged(object sender, EventArgs e)

{

nUD\_Complex\_Root\_i.Maximum = nUD\_Complex\_Root\_N.Value;

}

private void TabControl\_SelectedIndexChanged(object sender, EventArgs e)

{

switch (tabControl.SelectedIndex)

{

case 0:

Size = new System.Drawing.Size(355, 433);

break;

case 1:

Size = new System.Drawing.Size(310, 382);

break;

case 2:

Size = new System.Drawing.Size(445, 418);

break;

default:

break;

}

}

private void Form1\_KeyPress(object sender, KeyPressEventArgs e)

{

switch (tabControl.SelectedIndex)

{

case 0:

{

if ((e.KeyChar >= '0' && e.KeyChar <= '9')

|| (e.KeyChar >= 'A' && e.KeyChar <= 'F')

|| (e.KeyChar == '.' && pNumber\_mode))

{

tB\_PNumber.Text = pNumber\_ctrl.ExecCommandEditor(Char\_to\_edit\_command(e.KeyChar));

}

else if (OPERATIONS.Contains(e.KeyChar))

{

tB\_PNumber.Text = Number\_to\_state(TAG\_PNUMBER, pNumber\_ctrl.ExecOperation(Char\_to\_operation\_command<TPNumber>(e.KeyChar)));

}

break;

}

case 1:

{

if ((e.KeyChar >= '0' && e.KeyChar <= '9') || e.KeyChar == '.')

{

tB\_Frac.Text = frac\_ctrl.ExecCommandEditor(Char\_to\_edit\_command(e.KeyChar));

}

else if (OPERATIONS.Contains(e.KeyChar))

{

tB\_Frac.Text = Number\_to\_state(TAG\_FRAC, frac\_ctrl.ExecOperation(Char\_to\_operation\_command<TFrac>(e.KeyChar)));

}

break;

}

case 2:

{

if ((e.KeyChar >= '0' && e.KeyChar <= '9') || e.KeyChar == '.')

{

tB\_Complex.Text = complex\_ctrl.ExecCommandEditor(Char\_to\_edit\_command(e.KeyChar));

}

else if (OPERATIONS.Contains(e.KeyChar))

{

tB\_Complex.Text = Number\_to\_state(TAG\_COMPLEX, complex\_ctrl.ExecOperation(Char\_to\_operation\_command<TComplex>(e.KeyChar)));

}

break;

}

default:

break;

}

}

private void Form1\_KeyDown(object sender, KeyEventArgs e)

{

switch (tabControl.SelectedIndex)

{

case 0:

{

if (e.KeyCode == Keys.Enter)

{

b\_PNumber\_Eval.PerformClick();

}

else

{

TAEditor.Command comm = Keycode\_to\_edit\_command(e.KeyCode);

if (comm != TAEditor.Command.cNone)

{

tB\_PNumber.Text = pNumber\_ctrl.ExecCommandEditor(comm);

}

}

break;

}

case 1:

{

if (e.KeyCode == Keys.Enter)

{

b\_Frac\_Eval.PerformClick();

}

else

{

TAEditor.Command comm = Keycode\_to\_edit\_command(e.KeyCode);

if (comm != TAEditor.Command.cNone)

{

tB\_Frac.Text = pNumber\_ctrl.ExecCommandEditor(comm);

}

}

break;

}

case 2:

{

if (e.KeyCode == Keys.Enter)

{

b\_Complex\_Eval.PerformClick();

}

else

{

TAEditor.Command comm = Keycode\_to\_edit\_command(e.KeyCode);

if (comm != TAEditor.Command.cNone)

{

tB\_Complex.Text = pNumber\_ctrl.ExecCommandEditor(comm);

}

}

break;

}

default:

break;

}

}

private void ДействительныеPNumberTSMI\_Click(object sender, EventArgs e)

{

целыеPNumberTSMI.Checked = false;

действительныеPNumberTSMI.Checked = true;

pNumber\_mode = true;

b\_PNumber\_Separator.Enabled = true;

b\_PNumber\_Clear.PerformClick();

}

private void ЦелыеPNumberTSMI\_Click(object sender, EventArgs e)

{

целыеPNumberTSMI.Checked = true;

действительныеPNumberTSMI.Checked = false;

pNumber\_mode = false;

b\_PNumber\_Separator.Enabled = false;

b\_PNumber\_Clear.PerformClick();

}

private void ДробьFracTSMI\_Click(object sender, EventArgs e)

{

дробьFracTSMI.Checked = true;

числоFracTSMI.Checked = false;

frac\_mode = true;

}

private void ЧислоFracTSMI\_Click(object sender, EventArgs e)

{

дробьFracTSMI.Checked = false;

числоFracTSMI.Checked = true;

frac\_mode = false;

}

private void КомплексноеComplexTSMI\_Click(object sender, EventArgs e)

{

комплексноеComplexTSMI.Checked = true;

действительноеComplexTSMI.Checked = false;

complex\_mode = true;

}

private void ДействительноеComplexTSMI\_Click(object sender, EventArgs e)

{

комплексноеComplexTSMI.Checked = false;

действительноеComplexTSMI.Checked = true;

complex\_mode = false;

}

private void КопироватьToolStripMenuItem\_Click(object sender, EventArgs e)

{

switch(tabControl.SelectedIndex)

{

case 0:

Clipboard.SetText(tB\_PNumber.Text);

break;

case 1:

Clipboard.SetText(tB\_Frac.Text);

break;

case 2:

Clipboard.SetText(tB\_Complex.Text);

break;

}

}

private void ВставитьToolStripMenuItem\_Click(object sender, EventArgs e)

{

string text = Clipboard.GetText();

switch (tabControl.SelectedIndex)

{

case 0:

tB\_PNumber.Text = "";

for (int i = 0; i < text.Length; i++)

{

tB\_PNumber.Text = pNumber\_ctrl.ExecCommandEditor(Char\_to\_edit\_command(text[i]));

}

break;

case 1:

tB\_Frac.Text = "";

for (int i = 0; i < text.Length; i++)

{

tB\_Frac.Text = frac\_ctrl.ExecCommandEditor(Char\_to\_edit\_command(text[i]));

}

break;

case 2:

tB\_Complex.Text = "";

for (int i = 0; i < text.Length; i++)

{

tB\_Complex.Text = complex\_ctrl.ExecCommandEditor(Char\_to\_edit\_command(text[i]));

}

break;

}

}

}

}