Real World Interactions - Homework 3

In this assignment you'll practice the following concepts and skills:

- Runtime polymorphism,
- Dynamic containers (ICollection),
- Interfaces & inheritance,
- UML class diagrams

Short description

You already have a working RPN calculator. Now you'll change its design to take advantage of *runtime polymorphism*. You'll focus on making the calculator more extensible by turning operators into separate classes that can be added to, or removed from, an existing RPN calculator.

Requirements

1. Your `RPNCalculator` must not have any built-in operations that is supports.
2. Your `RPNCalculator` class must be able to add/ remove new operations at runtime.
3. Your `RPNCalculator` class must implement the `ICollection<IOperator>` interface or the `ICollection<KeyValuePair<string, IOperator>>` interface.
4. Your `RPNCalculator` class must be able to flexibly use unary and binary operators implemented as external classes.
5. Your `RPNCalculator` class must support mathematical constants (e.g. π, `e`). Those must be implemented as *nullary operators*.
6. When completed, your program **should create new operations/ constant in its `Main` method** and add them to the calculator. Like in the example below:

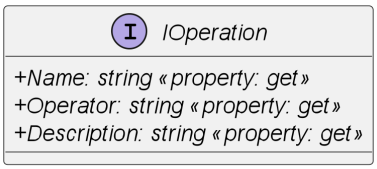
```
public class Program {
    static void Main(string[] args) {

    var calculator = new RPNCalculator.RPNCalculator();
    calculator.Add(new Addition());
    calculator.Add(new Subtraction());
    calculator.Add(new Multiplication());
    calculator.Add(new Division());
    calculator.Add(new Power());
    calculator.Add(new Sqrt());
    calculator.Add(new Exponentiation());
    calculator.Add(new Logarithm());
    calculator.Add(new Constant("pi", "pi", "constant pi", Math.PI));
    calculator.Add(new Constant("e", "e", "constant e", Math.E));

    var parser = new Parser(calculator.SupportedOperators);
    var menu = new TextMenu(calculator.OperationsHelpText);

    var controller = new Controller(calculator, parser, menu);

    controller.Run();
    ...
    }
}
```

Example design

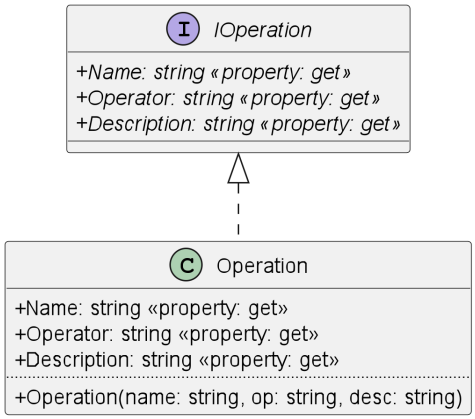
It's not easy to wrap your head around the design of the calculator as described in the requirements section. So here's a quick example of such a design.

Operations

1. Start with the `IOperation` interface. Since operations can have different arities, you won't add the `Calculate` method to it. Just the basic properties that every operation must have:



2. Then, create a class `Operation` that implements `IOperation`. This will be the base class for all other operations.



   Notice that `Operation` has a constructor that initializes the `Name`, `Operator`, and `Description` properties. Besides those `Operation` doesn't have any other functionality (this will come later in concrete operations)

3. Now it's time to add three more interfaces.

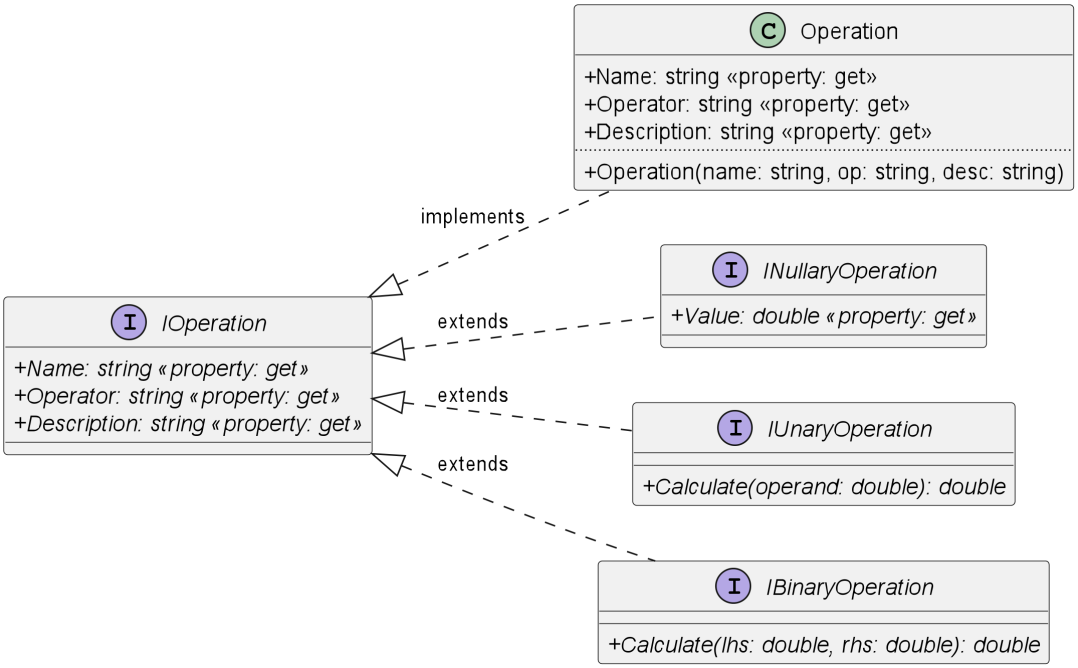
   The first one is `INullaryOperation` and it will be used to implement mathematical constants. This one is a bit special - it doesn't calculate anything, but it has a `Value` property, that's used to obtain the numerical value of a mathematical constant.

   The second one is `IUnaryOperation`, that's one for unary operations (e.g. `sqrt`)

   The third one is `IBinaryOperation`. It's used to indicate that an operation is a binary operation (e.g. `+`).

   Perhaps a bit surprising, but all three of them *extend* the `IOperation` interface. That's because all three are operations at heart, so they should offer the basic functionality an operation offers. Reminder: extending an interface by another interface is expressed in code as:

```
public interface IBinaryOperation : IOperation {
   /* methods of IBinaryOperation be here */
}
```



4. The fun part begins - it's time to create concrete operation classes. Implement at least the four basic arithmetic operations (addition, subtraction, multiplication, division) and at least two unary operations of choice (e.g. `Sqrt`, `Logarithm`). Each of those classes must inherit from `Operation` and implement an interface that corresponds with the number of operands the operation needs (`IUUnaryOperation` or `IBinaryOperation`). Those classes will be very short, besides the obvious `Calculate` method, they only need a constructor that initializes the `Name`, `Operator`, and `Description` properties by calling the base class (`Operation`) constructor. This is done in the following way:

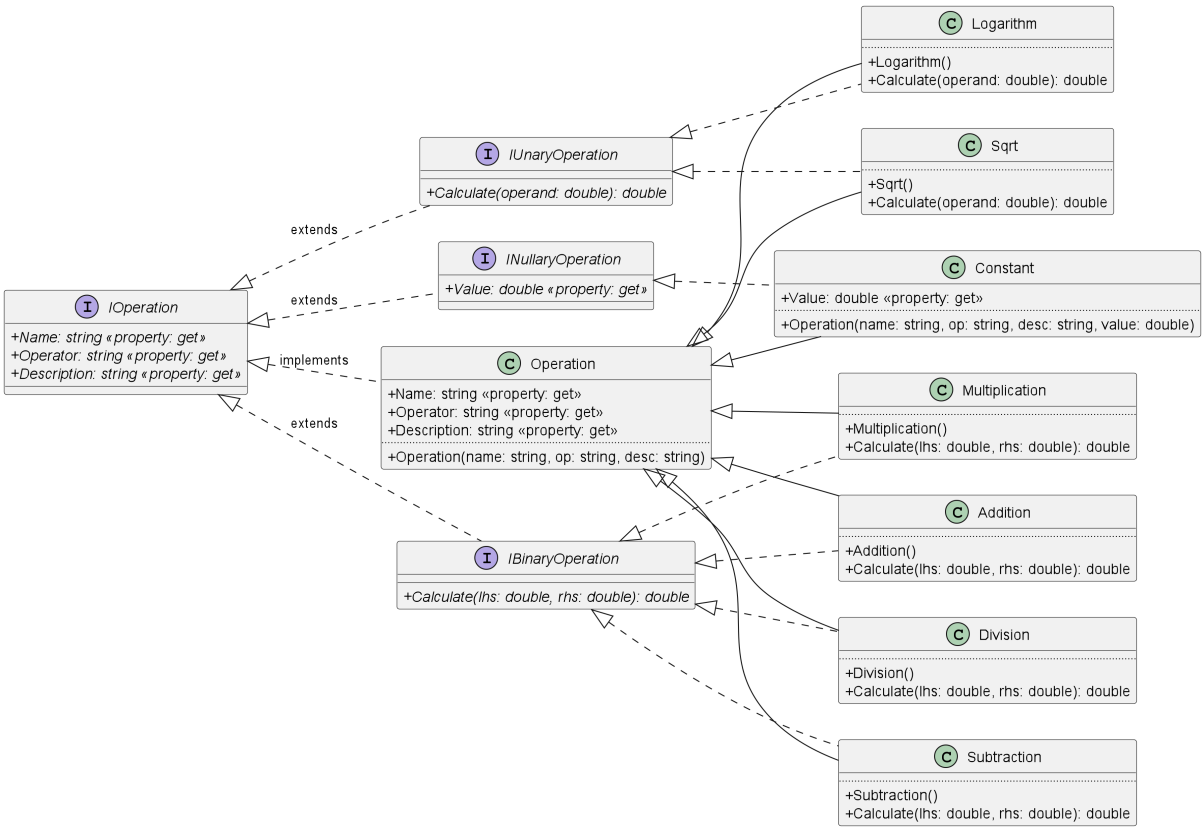
```
public class Logarithm: Operation, IUnaryOperation
{
    // Logarithm's constructor calls the base class's (Operation) constructor,
    // initializing the `Name`, `Operator`, and `Description` properties.
    public Logarithm() :
        base("Logarithm", "ln", "calculates the natural logarithm of a number")
    {
    }

    // + Calculate method implementation
}
```

5. Instead of creating separate classes for each mathematical constant, you can create a single class that implements `INullaryOperation` and inherits from `Operation`. This class will be called `Constant` and it will have a constructor that takes the `name`, `operator`, and `description` arguments and initializes the corresponding properties of the base class. It will also have a `Value` property that will be used to obtain the numerical value of the constant, this one should also be initialized in the constructor from an argument. Then you can create new constants with:

   ```
   var pi = new Constant("pi", "pi", "constant pi", Math.PI);
   ```

   Of course, you can also make a new class for every constant if you wish.

All in all, your design should now look like this:



You can check if everything works as expected by running a piece of code similar to:

```
var ops = new List<IOperation>();
ops.Add(new Addition());
ops.Add(new Subtraction());
ops.Add(new Multiplication());
ops.Add(new Division());
ops.Add(new Sqrt());
ops.Add(new Logarithm());
ops.Add(new Constant("pi", "pi", "constant pi", Math.PI));
ops.Add(new Constant("e", "e", "constant e", Math.E));

foreach (var op in ops)
{
    double lhs = 2.0;
    double rhs = 3.0;

    switch (op)
    {
        case INullaryOperation nullaryOp:
            Console.WriteLine($"I am a constant {nullaryOp.Name}
                with a value of {nullaryOp.Value}");
            break;

        case IUnaryOperation unaryOp:
            Console.WriteLine($"I am an unary operation {unaryOp.Name}
                and can calculate: {unaryOp.Operator} {rhs} -> {unaryOp.Calculate(rhs)}");
            break;
        case IBinaryOperation binaryOp:
            Console.WriteLine($"I am a binary operation {binaryOp.Name} and can calculate:
                {lhs} {binaryOp.Operator} {rhs} -> {binaryOp.Calculate(lhs, rhs)}");
            break;
```

```
    }
}
```

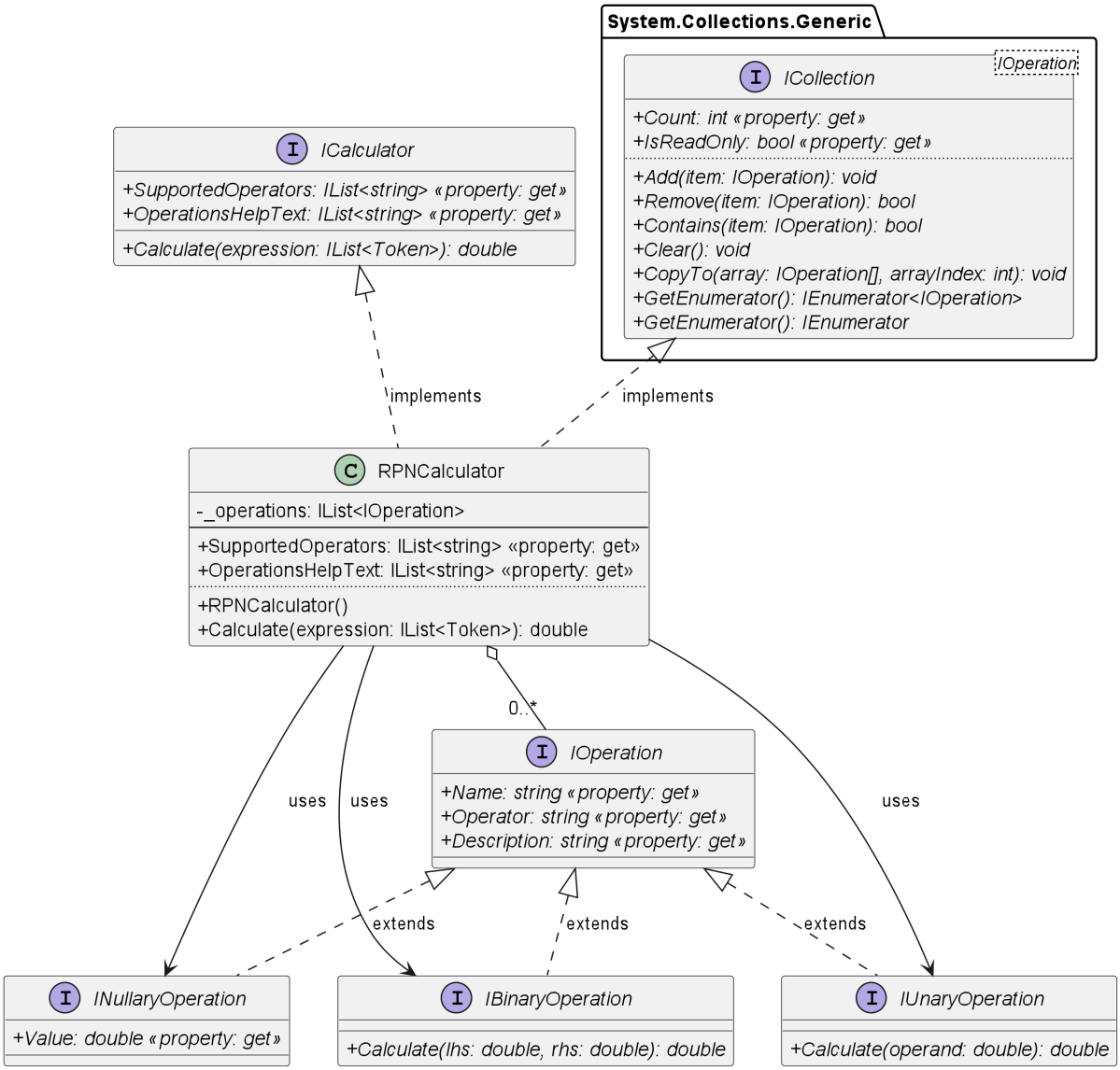
RPNCalculator

First get rid of any hard-coded operations that you might have in your RPNCalculator class.

Then:

1. Add a `private` field `List<IOperation> _operations` to your class. (Don't forget to initialize it in the constructor.)
2. Implement the `ICollection<IOperation>` interface in your RPNCalculator. This will come with methods such as `Add(IOperation operation)`, `Remove(IOperation)`, etc. Don't overdo the implementations of those methods - you can forward all those calls to `_operations`.
3. Alternatively you can implement the `ICollection<KeyValuePair<string, IOperation>>` interface in the `RPNCalculator`. You'll then need to use `Dictionary<string, IOperation>` to store the operations instead of `List<IOperation>`.
4. With the `ICollection` interface implemented you can now use the `Add` method to add new operations to the calculator in the `Main` function of the `Program` class.

If in doubt, the UML diagram of the `RPNCalculator` class and its dependencies might be of some help:



The key points are:

1. The `RPNCalculator` class implements the `ICollection<IOperation>` interface.
2. The `RPNCalculator` class implements the `ICalculator` interface.
3. The `RPNCalculator` class has no idea that the `Operation` class or any of the its subclasses exist.
4. The `RPNCalculator` uses the `INullaryOperation`, `IUnaryOperation` and `IBinaryOperation` interfaces.

About this last point, you'll need to change the `Calculate` method of the `RPNCalculator` class to use *dynamic polymorphism*. Since the calculator has no built-in knowledge of the operations it supports, when it encounters a token that's an operation, it must check if such an operation was added to its `_operations` array. For instance:

```
foreach (var token in expression)
{
    switch (token.Type)
    {
        case TokenType.OPERATOR:
            // find an operation with the matching operator
            // operations is a List<IOperation> here
            var op = _operations.Find(op => op.Operator.Equals(token.Value));
            if (op != null)
            {
                // the operation was found, so we can use it
            }
            else
            {
                // the operation was not found, throw an exception
                throw new InvalidOperationException($"Unknown operation: {token.Value}");
            }
            break;
        case TokenType.NUMBER:
            // do something with the number
            break;
    }
}
```

Your job finding a matching operation will be a bit easier if you use a `Dictionary` instead of a `List` to store the operations.

Once you've found the matching operation, you'll have to check if it's a nullary, unary or a binary operation and act accordingly. Checking can be done with simple type checking and casting (keywords `is` and `as`). Or you might consider pattern matching on types with `switch` statements (similarly as it was done in the code example just before the start of the **RPNCalcualtor** section above).

Finally, don't forget to change the `SupportedOperators` and `OperationsHelp` properties of the `RPNCalculator` class to return the information about operations you added to the calculator. This can be as simple as e.g.:

```
public IList<string> SupportedOperators => operations.Select(op => op.Operator).ToList();
```