Lab 1: JavaRPNCalculator

Part 1 due: Feb 4th at 11:59pm

Overview

In this lab, you will implement an RPN (Reverse Polish Notation) calculator in Java using object inheritance and generic collections. Two files will be provided to you via the SVN repository, `javaRPNCalculator.java` and `Operator.java`. You will checkout the project from the SVN repository to get started; instructions are below.

Deliverables

Your submissions must include:

- `README`
- `javaRPNCalculator.java`
- `Operator.java`

Your `README` file must include, (1) your name, (2) the name of all files in your submission, and (3) a short description of the program and your development process. Failure to include a `README` or not including sufficient information in the `README` will adversely affect your grade on this lab.

Project Retrieval and Submission Instructions

To retrieve the project for this lab, in Eclipse new project menue, choose a new project from SVN. Use the following URI and check out the project into the workspace:

```
https://cs71.cs.swarthmore.edu/svn/cs71/labs/01/JavaRPNCalculator
```

Once the project is checked out, you must then *disconnect* the project by right clicking on the project in the package explorer (Team→Disconnect). Also remove the SVN meta-data.

To submit the project, you will again right click on the project in the package explorer and choose Team→Share Project. Use the following URI as the svn repository location:

```
https://cs71.cs.swarthmore.edu/svn/cs71/submission/<username>/01
```

Where `<username>` is replaced with your swarthmore username. Click finish, and then follow the on-screen instructions to com mitt. You may update and commit your project until the deadline. All commits after the deadline will be rolled back to the submission just prior to the deadline.

1 RPN Calculators and Post-fix Notation

Reverse Polish notation (RPN) calculators refers to a style of calculator that uses post-fix notation and a stack to maintain computation state. To implement your calculator, you must first understand post-fix notation. As a comparison, consider in-fix notation which we are already familiar with.

```
 (9 + 3) * 10  / 2
```

The "fix" refers to the location of the operation with relation to the values it operates over. For example, the plus operator "+" is in-fixed between the numbers being computed, 9 and 3. This style of notation is very natural for humans because we read the statement "9+3" as "nine plus three," but in-fix notation is complicated for computers to parse, and it also requires strict adherence to order of operations, lest the improper result is calculated.

Post-fix notation, in contrast, is harder for humans to read but much easier for computers to parse. Again, the "fix" refers to the location of the operands, and post-fix notation places the operator following the values being operated over. So: 1 + 2 is written 1 2 +, and we can rewrite the example above as:

```
 9 3 + 10 * 2 /
```

As you might have observed, the advantage to post-fix notation is that the order of operations is syntax directed; that is, there is no need for parenthesis to indicate which operations should be applied first. Instead, the order of the input and the operations indicate directly which computation should occur first by reading from left to right. In the example above, first 9 3 + is performed, whose results is the first input to 10 *, whose result is the first input to 2 /, and so on.

An RPN calculator takes this process one step further by maintaining the states of the calculation in a stack, the last in first out data structure. The idea is that each value that occurs (i.e., a number) is *pushed* onto the stack and operators *pop* from the stack to retrieve their inputs, *pushing* the result of the operation back onto the stack. As refernce, consider the computation above using an RPN calculator output below. The "-*-" indicates an empty stack, and you should consider this output as an example of how your calculator should function.

```
0: -*-

RPN>10 3 9

2:10.0
1:3.0
0:9.0

RPN>+

1:10.0
0:12.0

RPN>*

0:120.0

RPN>2 /

0:60.0

RPN>
```

The head of the stack, index 0, always stores the result of the last operation or the last value pushed onto the stack. Traversing this execution trace, you can see that first the values 10, 3 and 9 are pushed onto the stack. The plus operation consumes 9 and 3 and pushes 12 onto the stack. Next the multiplication operation consumes 10 and 12 and pushes the result 120 onto the stack, and, finally, 2 is pushed onto the stack followed by the division operation which results with 60 on the stack.

2 Development

There are two java files provided to aid your development. In these, you will find comments and `TODO`s that will help guide you through the development. Part of the goal for this assignment is for you to learn to read and understand java source without too much direction. However, there are some key design aspect that you must meet for full credit.

Operator Object

The operator object is defined in the `Operator.java` source file. This source file represents defines an operator object, both a unary operator and binary operator. A unary operator is a operation that requires a single input. For example, the square root function is a unary operator because it only requires the number to take the square root of. A binary operator is a operation that requires two inputs, such as +.

The operator object is not a standard class, rather it defines an *abstract class*. An abstract class, like a C++ interface, describes a mechanism for accessing the methods and data of an object that implements the interface; however, an abstract class differs in that some portion of the methods or data are fully defined while other parts are described as *abstract* and must be defined by the*implementing* class. Further, an abstract class need not be inherited to be instantiated, unlike an *interface*, and the abstract methods can be defined at instantiation, i.e., when `new` is used. This will be discussed in more detail below. Eclipse will help guide you through which methods need defining if you allow it to auto-fill.

Implementing the `operation()` Method

There are two abstract methods for an operator object, or a single overloaded method, named `operation()`. The reason this method is overloaded is that there are two types of operations in your calculator, unary and binary operations, and each takes a different number of arguments. For example, if you are implementing the plus form of the operator object, the `operation(int x, int y)` method implements plus while the `operation(int x)` will throw an exception. In code, this will look like this:

```
operators.insert(new Operator("+",2){

  @Override
  public double operation(double x, double y)
  throws OperatorException {
    return x+y;
  }

  @Override
  public double operation(double x) throws OperatorException {
    throw new Operator.OperatorException("binary operator");

  }

});
```

There are two things to notice here: First, there is no need to define a third class that *implements* the Operator object, but rather the abstract methods can be defined as part of the `new` operation; and second, the unary operation method throws an `OperatorException` which is also defined in `Operator.java` and, further, each of operation methods indicates that such an exception can occur in their function definition. Similarly, when defining a unary operation, the binary operation method should throw an exception. This is good software engineering practice so that the instances of the object is used as intended. Additionally, the operator object provides a data element, `argc`, that indicated which of the two operation methods is active, i.e., 1 for unary and 2 for binary. You can `switch` on `argc` in your calculator. You may also notice that the new operator is stored in a data-structure `operators`, which is a generic collection to store the operators. More details on generics and collections is found in the next section.

Generics and Collections Objects

There are two key generic collections defined in the calculator object:

```
  private Stack<Double> stack;
  private Hashtable<String, Operator> operators;
```

You should already be familiar with the use of these data objects, so, instead, direct your attention to the generic aspects. Recall that java defines two types of data, primitives and objects. A double precision floating point number (`double`) is a primitive, not an object, but generics are designed for objects only! Java provides a straightforward work araound for this using an *object wrapper*. The generic collection for the stack is `Double` objects not `double` primitives; however, the `Double` object acts like its primitive counter part in all ways. That means you can do this without error:

```
  stack.push(1.0);
```

The other generic data structure used in the calculator implementation is a hash table. Notice that the generic definition takes two objects, the type of the key and type of the value. In this case, the hash table maps strings to operator objects, i.e., it describes a mapping between the string form of the operation and object that performs that operation. So the `plus` operator object defined above can be inserted into the hash table like:

```
operator.insert("+",plus);
```

For more information about the methods and data associate with these data structures, as well as more data structures you can use in your development, refer to the java documentation.

Exception Driven Control Flow and Error Reporting

When reviewing the provided code, you should notice that I have defined two exceptions that you should use to indicate various error conditions. There are other exceptions you should be aware of, particularly, `NumberFormatException` which is thrown when trying to convert a string to a number. Your development should *catch* these exceptions as part of the control flow of the program, and, similarly, your development should throw exceptions to affect control flow. Where and when this should occur will be obvious, but I strongly encourage you to think ahead about the kinds of errors that can occur. Further, **your development must report and handle computation and numeric errors in a natural way.** Errors should be reported to the user and the calculator state should be minimally affected when an error occurs. For example:

```
0: -*-

RPN>1 2 a

1:1.0
0:2.0
Unknown Operator: op='a'
RPN>
```

Finally, for exceptions that are defined in a class file as a subclass, such as `RPNCalcualtorException`, you can refer to them directly using the dot (".") operator. Like in C++ and Python and Java member access, the dot-operator is used to refer to a sub-part of a larger structure.

Required Operations

Your calculator must implement the operations listed below. You should refer to the `java.util.math` class for the java implementation of many of these operations. Further, note that not all the operations below are math operations, such as `clr` and `help`, and there are also two constants.

- `+` : plus
- `-` : minus
- `*` : multiplication
- `/` : division
- `^` : exponentiation
- `sq` : square
- `sqr` : square root,
- `lg` : natural logarithm, e.g., e lg returns 1
- `logx` : logarithm base x, e.g., 100 10 logx returns 2
- `sin` : sin function in radians
- `cos` : co-sin function in radians
- `tan` : tangent function in radians
- `clr` : clear the stack
- `help` : help function that provides the user info about all operations and constants
- `e` : the natural logarithm root
- `pi` : the ratio of the circumference of a circle to its diameter