The goals of this assignment are:

- to implement a data structure whose capacity increases dynamically
- to describe the interpretation of computer programs
- to extend a stack-based interpreter (virtual machine) and
- to define, throw and catch Java exceptions

In this homework you are to write a mini-interpreter for PostScript, the stack-based language that controls printing in almost every modern printer. When you print using a Postscript printer, the print dialogue creates a program written in the Postscript language and sends it to the printer. The printer executes the program, the result being a set of commands that causes the print engine to create the document.

The interpreter of this stack-based programming language scans a program from left to right, pushes and pops operands from and to a stack during the evaluation of the program. The programming language also comprises several operations for drawing onto a two dimensional surface. The result of the execution of a program can be a drawing.

Make sure you download the provided code.

Map and Dictionary

First you need to implement your own Map and Dictionary according to the specifications below.

To complete this part you will need to work with two of the provided files found in part1code folder:

- Token.java: a class that captures the element of a mini-postscript input program. A token is either a number (double) or a symbol (String). See the description below.
- DictionaryTest.java: a JUnit test file.

```java
class Token

Methods | Description
---------|---------------------------------------------------
Token(double number) | Constructs a Token to represent a number.
Token(String symbol) | Constructs a Token to represent a symbol.
boolean isNumber() | Returns true if the object was created using the constructor Token(int number).
boolean isSymbol() | Returns true if the object was created using the constructor Token(String symbol).
int getNumber() | Returns the number stored in this object. The method can only be called if the object was created using the constructor Token(double number), throws IllegalArgumentException otherwise.
String getSymbol() | Returns the symbol stored in this object. The method can only be called if the object was created using the constructor Token(String symbol), throws IllegalArgumentException otherwise.
```
Map.java and Dictionary.java are not provided. However you should use the JUnit test file DictionaryTest to incrementally develop your code. Remember the setup you used in the linked-list lab.

interface Map

Declare the interface Map. A map stores key-value associations. For this project, a map can contain duplicate keys, in which cases the methods get, put, replace, and remove refer to the leftmost (last) association for the given key.

Map is a generic type having two type parameters, $K$ and $V$, where $K$ is the type of the keys, and $V$ is the type of the values. A map has the following methods:

- $V$ get($K$ key): returns the leftmost value associated with this specified key. The methods throws java.util.NoSuchElementException if the element being requested does not exist. It throws NullPointerException if the specified key is null.

- boolean contains($K$ key): returns true if an association exists for the specified key. It throws NullPointerException if the specified key is null.

- void put($K$ key, $V$ value): creates a new association key-value. It throws NullPointerException if the specified key or value is null.

- void replace($K$ key, $V$ value): replaces the value of the leftmost occurrence of the association for the specified key. It throws NoSuchElementException if no such association exists. It throws NullPointerException if the specified key or value is null.

- $V$ remove($K$ key): removes the leftmost association for the specified key, and returns the value that was associated with the key. It throws java.util.NoSuchElementException if the element being requested does not exist. It throws NullPointerException if the specified key is null.

class Dictionary

Dictionary keeps track of String-Token (identifier-value) associations. Such data structure is sometimes called a symbol table.

- The class Dictionary implements the interface Map<String, Token>.

- It uses an array to store the associations. The elements of this array are of type Pair, a private static nested class. A Pair object stores an association, a key and a value, of type String and Token, respectively.

- Finally, a Dictionary stores an arbitrarily large number of associations. Accordingly, implement the dynamic array technique, presented in class (remember Vector implementation). The initial size of the array is 10. It will double, whenever the array is full.

- DictionaryTest comprises a series of tests for your implementation. Use them as you are developing your code.

You can complete this part independently of the rest of the program.

You will need these two files, Dictionary and Map in the morecode folder: copy them. morecode folder contains the files needed for the rest of this assignment.
**Interpreter**

The class `Interpreter` implements a Virtual Machine for our stack-based language. The method `execute` implements the “read-eval” loop of the interpreter. At each iteration of the `while` loop, the interpreter fetches the next element i.e., `Token` from the input program.

- Numbers are simply pushed onto the operands stack.
- Symbols cause the interpreter to execute specific actions.

**Overview**

For example the process to interpret the command

```
15 10 sub
```

the following “read-eval” iterations are executed:

1. the interpreter reads and pushes the token 15 onto the operands stack.
2. the interpreter reads and pushes 10 onto the operands stack.
3. the interpreter reads the symbol `sub` and calls the method `execute_sub()`, which pops two operands from the stack, subtract one from the other, and stores the result onto the operands stack.

The program terminates with the operands stack containing only one element, the value 5. Run the provided code, the `main` is in `Viewer` and type `15 10 sub` in the gray text area as the bottom on the viewer window and push `Run` to execute the line: you don’t see much. So try the following two programs

```
15 10 sub pstack
15 10 pstack sub pstack
```

`pstack` is really useful! You should see the content of the operands stack in the bluish panel. Use it as you to understand and complete the operations below. Try more programs

```
1 3 4 mul add pstack
1 3 add 4 mul pstack
```

**More Operators**

Now you are ready to add more operators. Read the last section of this document to understand the implemented one (description of the provided operators for the virtual machine).

You will notice that the operators `add`, `div`, `clear`, `dup` and `exch` are not implemented in `Interpreter.java`. On the model of `sub` complete the code to permit the `add` operation

```
0. add – pops off the top two elements from the operands stack, adds them together and pushes back the result onto the stack.
```

Inspired by the provided code implement the following operators:

```
1. div – pops off the top two elements from the operands stack, divides them and pushes back the result onto the stack. (5/2) would be represented as 5 2 div. A later call to `pstack` should display on the bluish panel `pstack (1)` `2.5`
```

3
2. **dup** – adds a duplicate copy of the top object of the stack to the stack.

3. **clear** – removes all the elements from the operands stack. The execution of the following program
   \[
   10 \ 20 \ 30 \ 40 \ 50 \ 60 \ 60.2 \ \text{pstack} \ \text{clear} \ \text{pstack}
   \]
   produces the following result in the Viewer’s console:

   \[
   \text{pstack (1)} \\
   60.2 \\
   50.0 \\
   40.0 \\
   30.0 \\
   20.0 \\
   10.0 \\
   \text{pstack (2)}
   \]

4. **exch** – exchange the top two stack values

For these four operators I encourage you to experiment with the real PostScript program (shell version) if needed: see instructions on Moodle. If you can’t run it you may look on the web for more details for a particular operator.

By the end many programs should work. For example the following one line programs should work

\[
10.1 \ \text{dup mul pstack pop} \\
3 \ 1 \ \text{exch sub pstack clear pstack} \\
9 \ 2 \ 4 \ \text{mul 5 sub div pstack} \\
9 \ \text{pstack 2 pstack 4 pstack mul pstack 5 pstack sub pstack div pstack}
\]

Try your own programs and document them in your **readme**, i.e. copy and paste the input and the output. Use **pstack**! to be confident your program is correct.

Don’t forget to try the graphical programs provided: see **ps_graphics_samples** folder.

### Quoted Symbols

Now that you have basic operations implemented, you extend your program to use the **Dictionary** class you wrote first. It permits to define variables.

You will add three (3) elements to the syntax of our mini-postscript:

1. the syntax for quoted symbols
2. **def** operation and
3. the evaluation of symbols.

The **def** operation defines symbols and creates **String-Token** associations that should be stored in a **symbol table**, i.e. the **Dictionary** instance that will be part of the interpreter state

To define a symbolic value we specify a “quoted” symbol (preceded by a slash) and the value, all followed by the operator **def**:

\[
/\pi \ 3.141592653 \ \text{def}
\]

The syntax **/pi** is a **quoted** symbol, the forward-slash prevents the evaluation of the symbol **pi**. Specifically this program is executed as follow
1. the interpreter recognized a quoted symbol i.e. \( \pi \)
2. the interpreter pushes the symbol itself (!) \( \pi \) on the stack,
3. the interpreter reads the \( 3.141592653 \), which is a number and therefore is pushed onto the stack.
4. the symbol \texttt{def} is read: the interpreter then retrieves two operands from the stack, two elements are popped: the number \( 3.141592653 \) and the symbol \( \pi \).
5. the interpreter creates an association for \( \pi \) and \( 3.141592653 \) and stores it into the dictionary = the new instance variable that you added to your Interpreter class.

Once we define a symbol, it can be used in computations. For example
\[
/\pi \ 3.141592653 \ \texttt{def} \\
/radius \ 1.6 \ \texttt{def} \\
\pi \ radius \ \texttt{dup} \ \texttt{mul} \ \texttt{mul} \ \texttt{pstack} \\
\text{print the area of a circle with radius 1.6. The output console is} \\
pstack \ (1) \\
8.042477191680002
\]
To observe the stack in details you have to understand that the program
\[
/pi \ \texttt{pstack} \ 3.141592653 \ \texttt{def} \\
5.0 \ \texttt{pstack} \\
\pi \ \texttt{pstack}
\]
produces on the output console
\[
pstack \ (1) \\
\pi \\
pstack \ (2) \\
5.0 \\
pstack \ (3) \\
3.141592653 \\
5.0
\]
Make all the necessary changes to the class \texttt{Interpreter} to implement the following instructions

1. quoted symbol – the syntax forward slash followed by a symbol is called a quoted symbol. The forward slash prevents the evaluation of the symbol. When it encounters a quoted symbol, the interpreter simply pushes the symbol onto the stack (without the forward slash).
2. \texttt{def} – is used to create an association between a value and a symbol. Here is the syntax for defining an association:
\[
/\times \ 100 \ \texttt{def}
\]
Here, \( /\times \) is a quoted symbol, 100 is the value that will be associated with the symbol \( \times \), and \texttt{def} is the symbol definition operation. The interpreter will add an association between the identifier \( \times \) and the value 100 in the dictionary.

3. symbol evaluation – when the interpreter encounters a symbol for which there exists an association in the dictionary, the value associated with the symbol is pushed onto the stack. Once a symbol has been defined, the evaluation of the symbol returns the associated value:
\[
/x \ 100 \ \texttt{def} \\
/width \ 200 \ \texttt{def} \\
x \ width \ \texttt{pstack} \\
mul \\
pstack
\]
The above program displays in the console the following:

```
pstack (1)
200.0
100.0
pstack (2)
20000.0
```

**InterpreterSyntaxException**

Finally you have to define a new run-time exception named `InterpreterSyntaxException`. See the class `EmptyStackException` in for inspiration.

Make the change to the interpreter so that when the following situation is detected, an exception of type `InterpreterSyntaxException` is thrown.

This exception is thrown when an illegal symbol has been found. An illegal symbol is a symbol that is nor a reserved keyword (operation), nor an existing association in the dictionary.

When an illegal symbol is found the interpreter displays on the Java console the appropriate messages, including:

- the symbol table and
- the operands stack.

For example the following program

```
/x 100 def
/y pstack 200 def
3 pstack y z
```

should produce on the *Java console* something similar to the following

```
InterpreterSyntaxException:
ILLEGAL SYMBOL: z
Operands stack:
200.0
3.0
Symbol table
x --> 100.0
y --> 200.0
```

**Extensions**

You are encourage to go beyond in this assignment. Especially if you are aiming for an A+!! Extensions are however optional. Please document your achievement(s) in your *readme*

- Implement some more operations. `procedure`, `if` and `ifelse` would be nice.
- Implement some more graphical operations. I would suggest operations such as transformations in addition to pixel images and movies...

The following site introduces interesting operators. Please come and see me if you have ideas you want to talk about.
Submit

Turn in a .zip file containing Dictionary.java, Map.java, Interpreter.java, InterpreterSyntaxException.java, any other .java file you modified, and a readme, which includes

- your student information at the top,
- sequences of input you used to test all the functionality of the interpreter you implemented, and with output of the program that help to determine its correctness, and
- if you have not fully implemented the interpreter and dictionary, list the parts that work and the ones you attempted so as to receive partial credits.

Clearly acknowledge any help you received.

Credit  This assignment was created by Marcel Turcotte.

Appendix: Description of our Virtual Machine

Classes Overview

Run

The class Viewer starts the application by opening a window attached to Interpreter instance

```java
java Viewer
run Viewer (in DrJava)
```

Interpreter

Here are some details of the class Interpreter:

- Instance variables.
  - A Reader object is used to for the lexical analysis of the input program.
  - A stack is used to store the operands during the execution of a program.
  - A graphics state is needed to store a pair of coordinates, x and y, as well as a default color (of type java.awt.Color). The pair (x, y) represents the current position of the pen.
A Dictionary will be needed!

- public void execute(String program, Graphics g)

  The execution of a mini-postscript program always starts with an empty stack. The pen is moved to the location (0, 0), and the default color is set to blue (Color.BLUE).

A Reader object is used to read the program one Token at a time.

The essential of the method consists of a loop to read and execute the input program; this is sometimes called the “read-eval” loop of the interpreter.

This loop terminates when the whole program has been read (hasMoreTokens() returns false). At each iteration, a Token is read. Inside the loop, the method must implement each of the operations of the programming language listed below. Numbers are simply pushed onto the operands stack.

For each of the following operations there should be a corresponding method in the interpreter named execute_operation, for instance execute_sub in the case of the operation sub.

- sub: pops off the top two elements from the operands stack, subtracts them together and pushes back the result onto the stack. For example, the expression (3 − 1) is represented in mini-postscript as 3 1 sub (remember postfix we don’t need parenthesis!).

- mul: pops off the top two elements from the operands stack, multiplies them together and pushes back the result onto the stack.

- pop: removes the top element of the stack.

- pstack: displays the content of the stack in the Viewer window’s output panel.

- quit: exits the application (calls System.exit(0)).

- moveto: sets the position of the pen to \((x', y')\), where \((x', y')\) are read from the stack. For example, if the current location of the pen is \((10, 30)\) and the content of the stack is \([80, 20, 400, 100, 50, 30]\) the operation moveto sets the position of the pen to \((20, 80)\). After the execution of the operation, the stack contains the following elements: \([400, 100, 50, 30]\).

- lineto: draws a line from \((x, y)\) to \((x', y')\), where \((x, y)\) is the current location of the pen (which is part of the graphics state of the interpreter), and \((x', y')\) are read from the stack. Once the line has been drawn, the position of the pen, \((x, y)\), is set to \((x', y')\). For example, if the current location of the pen is \((10, 30)\) and the content of the stack is \([80, 20, 400, 100, 50, 30]\). The operation lineto draws a line from \((10, 30)\) to \((20, 80)\). After the execution of the operation, the stack contains the following elements: \([400, 100, 50, 30]\). and the new position of the pen is \((20, 80)\). For drawing a line onto a Graphics2D object, the method uses ((Graphics2D) g).draw(new Line2D.Double(x1, y1, x2, y2)).

- arc: draws an arc with a Arc2D.Double(Rectangle2D ellipseBounds, double start, double extent, int type). Please see me if you are interested in the details!