O objetivo desta prática é implementar uma versão simplificada das calculadoras HP, baseada na notação polonesa inversa. Nossa calculadora fará operações sobre números a ponto flutuante: seu estado será dado a cada instante por uma pilha de números e pela operação a ser aplicada sobre os elementos desta pilha. Desta forma, a estutura de dados de base da calculadora será uma pilha dinâmica que deve ser implementada utilizando-se uma lista. Nesta prática nossa classe `Pilha` será genérica e utilizará portanto uma lista encadeada genérica.

Como funciona a notação polonesa inversa?

Em notação polonesa inversa, a operação (por exemplo +) é explicitada depois dos operandos (por exemplo 2 e 3). Desta forma, o que escrevemos em notação tradicional "2+3" se escreve "2 3 +" em notação polonesa inversa. A notação polonesa inversa permite economizar os parênteses, e portanto, permite um tratamento mais fácil de expressões aritméticas. Para melhor compreender como isto funciona, abaixo temos uma tabela comparativa de uma expressão aritmética na notação tradicional e na notação polonesa inversa:

Notação tradicional	Notação Polonesa Inversa
2 + 3	2 3 +
(3 + 5) / (7 + 6)	3 5 + 7 6 + /
8 * (4 - 3 +1)	8 4 3 - 1 + *

O tratamento de uma expressão em notação polonesa inversa se faz da seguinte maneira: nós lemos a expressão da esquerda para a direita; sempre que encontramos um número, nós o colocamos na pilha; sempre que encontramos um operador, nós retiramos os dois primeiros elementos da pilha e depois empilhamos o resultado da operação sobre estes dois números. Desta forma, para a expressão "2 3 +", empilhamos 2, depois 3, e depois os desempilhamos para poder efetuar a soma. Em seguida, empilhamos o resultado.

Nota cultural

A notação polonesa inversa foi desenvolvida em 1920 por Jan Lukasiewicz a fim de escrever fórmulas matemáticas sem o uso de parêntesis. Hewlett-Packard Co. adotou a notação

polonesa inversa em sua primeira calculadora científica de bolso. Era o modelo HP35, criado em 1972, que foi um sucesso planetário.

1. Implementação de uma pilha genérica

A classe da biblioteca Java que implementa as pilhas genéricas é a classe `Stack`. Se mais tarde você tiver a necessidade de uma pilha, é recomendado que vocês a utilizem pois esta é a maneira mais simples e segura de proceder, já que seu código foi testado anteriormente. Entretanto, nesta prática, por razões pedagógicas, nós iremos implementar a nossa própria classe genérica `Pilha` que sera parametrizada pelo tipo `X` dos objetos armazenados.

Para o armazenamento propriamente dito utilizaremos a classe genérica `LinkedList` fornecida pela biblioteca Java. Como seu nome indica, esta classe fornece uma implementação de listas encadeadas. Antes de mais nada, clique no link `http://docs.oracle.com/javase/1.5.0/docs/api/java/util/LinkedList.html` para acessar a documentação da classe `LinkedList`. Entre os diversos métodos fornecidos, vocês utilizarão:

- O construtor sem argumentos `LinkedList()` que instancia uma lista vazia,
- `String toString()` que retorna uma cadeia de caracteres descrevendo o conteúdo da lista,
- `bool isEmpty()` que retorna `true` si a lista está vazia e `false` caso contrário,
- `void addFirst(X x)` que adiciona o elemento `x` ao começo da lista,
- `X getFirst()` que retorna o elemento do começo da lista, se ele existir,
- `X removeFIrst()` que retorna o elemento do começo da lista e o retorna da lista, se ele existir,
- `int size()` que retorna o tamanho da lista, i.e., o número de elementos que ela contem,
- `void clear()` que apaga o conteúdo da lista, i.e., remove todos os seus elementos.

Repare que a classe `LinkedList` se encontra no pacote `util` de Java. Para utilizá-lo é necessário adicionar a linha `import java.util.LinkedList;` no começo do seu arquivo `.java`.

A classe `Pilha`

Crie a classe `Pilha<X>` genérica contendo um campo `LinkedList<X> conteudo` assim como os métodos a seguir:

- Um construtor `Pilha` que inicializa o campo `conteudo`,
- `boolean estaVazia()` que retorna `true` se a pilha está vazia e `false` caso contrário,
- `void empilha(X x)` que adiciona o elemento `x` ao topo da pilha, i.e., no começo da lista `conteudo`,

- `X desempilha()` que retira o elemento do topo da pilha e retorna o seu valor,
- `X topo()` que retorna o valor do elemento no topo da pilha sem removê-lo.

Se a pilha está vazia, os métodos `desempilha` e `topo` devem lançar erros com a ajuda da instrução `throw new Error(‘‘...’’)`, onde "..." será uma cadeia de caracteres de sua escolha. Para testar o seu código, adicione à classe `Pilha` o código a seguir:

```
static void test1() {
    Pilha<Double> aPilha = new Pilha<Double>();
    aPilha.empilha(1.1);
    aPilha.empilha(2.1);
    aPilha.empilha(3.1);
    aPilha.empilha(4.1);
    aPilha.empilha(5.1);
    double valor = 0.0;
    valor = aPilha.topo();
    System.out.println("topo pilha = " + valor);
    valor = aPilha.desempilha();
    System.out.println("topo pilha = " + valor);
    valor = aPilha.desempilha();
    System.out.println("topo pilha = " + valor);
    valor = aPilha.desempilha();
    System.out.println("topo pilha = " + valor);
    valor = aPilha.topo();
    System.out.println("topo pilha = " + valor);
    valor = aPilha.desempilha();
    System.out.println("topo pilha = " + valor);
}
public static void main(String[] args) {
    test1();
}
```

Nota: Preste atenção na utilização da classe `Double` ao invés do tipo primitivo `double` logo da declaração da variável `aPilha`. As classes genéricas como `Pilha` necessitam de fato que seus parâmetros sejam eles mesmos uma classe, e não um tipo primitivo. Java fornece uma classe para cada tipo primitivo, cuja utilização é transparente pois as conversões de tipo se fazem automaticamente. Por exemplo, a instrução `aPilha.empilha(1.1)` acima converte automaticamente o valor 1.1 em `Double` antes de inseri-lo na pilha. De modo inverso, a instrução `valor = aPilha.topo()` converte automaticamente o objeto do tipo `Double` situado no topo da pilha em um `double` antes de atualizar a variável `valor`.

Com o código acima você deverá obter o resultado a seguir:

```
topo pilha = 5.1
topo pilha = 5.1
```

```
topo pilha = 4.1
topo pilha = 3.1
topo pilha = 2.1
topo pilha = 2.1
```

Impressão por meio do método `toString`

Afim de poder imprimir o estado da pilha enquanto a calculadora funcionar, adicione agora o método `public String toString()` à classe `Pilha`. Este método imprimirá o conteúdo da lista `conteudo` pela chamada ao método `toString` da classe `LinkedList`.

Para testar o seu código, modifique o código da função `main` da classe `Pilha` para fazê-la chamar o método a seguir:

```
static void test2() {
Pilha<Double> aPilha = new Pilha<Double>();
System.out.println(aPilha);
aPilha.empilha(1.1);
System.out.println(aPilha);
aPilha.empilha(2.1);
System.out.println(aPilha);
aPilha.empilha(3.1);
System.out.println(aPilha);
double valor = 0.0;
valor = aPilha.desempilha();
System.out.println("topo pilha = " + valor );
System.out.println(aPilha);
valor = aPilha.desempilha();
System.out.println("topo pilha = " + valor );
System.out.println(aPilha);
valor = aPilha.desempilha();
System.out.println("topo pilha = " + valor );
System.out.println(aPilha);
}
```

Você deverá obter o resultado a seguir (a impressão com colchetes e vírgulas separando os elementos da lista é feito automaticamente pelo método `toString` da classe `LinkedList`)

```
[1.1]
[2.1, 1.1]
[3.1, 2.1, 1.1]
topo pilha = 3.1
[2.1, 1.1]
topo pilha = 2.1
[1.1]
topo pilha = 1.1
[]
```

2. Calculadora básica

Crie um arquivo `CalcRPN.java` com o esqueleto de classe a seguir:

```
import java.io.*;

class CalcRPN {

// variáveis da instancia :
// uma pilha para os cálculos
Pilha<Double> aPilha;

// construtor
CalcRPN () {
throw new Error("a ser completado");
}

// Adição de dois elementos do topo da pilha
void mais() {
throw new Error("a ser completado");
}


// Subtração de dois elementos do topo da pilha
void menos() {
throw new Error("a ser completado");
}


// Multiplicação de dois elementos do topo da pilha
void vezes() {
throw new Error("a ser completado");
}

// Divisão de dois elementos no topo da pilha
void dividido() {
throw new Error("a ser completado");
}

// retorna o conteudo do topo da pilha
Double resultado() {
throw new Error("a ser completado");
}

// interpretador de comandos
```

```
void exec(String cmd) {
throw new Error("a ser completado");
}
}
```

Operadores básicos

Complete os métodos a seguir:

- O construtor `CalcRPN`, que inicializa a pilha `aPilha`,
- `mais,menos,vezes,divido` que retiram os dois elementos do topo da pilha, efetuando a operação correspondente, e depois empilham o resultado. Para os operadores `menos` e `dividido`, que são não-simétricos, nós adotaremos a convenção a seguir: em uma pilha da forma $[x; y; \ldots]$ o operador `menos` deve fazer a pilha igual a $[y - x; \ldots]$; da mesma forma, em uma pilha da forma $[x; y; \ldots]$ o operador `dividido` deve fazer a pilha igual a $[y/x; \ldots]$;
- `resultado` que retorna o valor no topo da pilha sem retirá-lo.

Para testar o seu código, adicione à classe `CalcRPN` o método de teste a seguir, a ser chamado a partir do método `main`:

```
 static void test() {
     CalcRPN calc = new CalcRPN() ;

     System.out.print("3 2 + = ");
     calc.aPilha.empilha(3.0);
     calc.aPilha.empilha(2.0);
     calc.mais();
     System.out.println(calc.resultat());

     calc = new CalcRPN();
     System.out.print("3 2 - = ");
     calc.aPilha.empilha(3.0);
     calc.aPilha.empilha(2.0);
     calc.menos();
     System.out.println(calc.resultat());

     calc = new CalcRPN();
     System.out.print("3 2 * = ");
     calc.aPilha.empilha(3.0);
     calc.aPilha.empilha(2.0);
     calc.vezes();
     System.out.println(calc.resultado());
```

```
        calc = new CalcRPN();
        System.out.print("3 2 / = ");
        calc.aPilha.empilha(3.0);
        calc.aPilha.empilha(2.0);
        calc.dividido();
        System.out.println(calc.resultado());

        calc = new CalcRPN();
        System.out.print("1 2 + 3 4 - / 10 3 - * = ");
        calc.aPilha.empilha(1.0);
        calc.aPilha.empilha(2.0);
        calc.mais();
        calc.aPilha.empilha(3.0);
        calc.aPilha.empilha(4.0);
        calc.menos();
        calc.dividido();
        calc.aPilha.empilha(10.0);
        calc.aPilha.empilha(3.0);
        calc.menos();
        calc.vezes();
        System.out.println(calc.resultado());
}
```

Você deve obter o resultado a seguir:

```
3 2 + = 5.0
3 2 - = 1.0
3 2 * = 6.0
3 2 / = 1.5
1 2 + 3 4 - / 10 3 - * = -21.0
```

Interface com o usuário

Complete o método `exec` da classe `CalcRPN` que toma como entrada uma cadeia de caracteres `cmd` e executa o comando correspondente na calculadora. Por enquanto, vamos supor que o parâmetro `cmd` vale ou "+", ou "-", ou "*", ou "/", ou um número representado sob a forma de uma cadeia de caracteres. Por exemplo se `cmd` vale "+", entaão chamaremos o método `mais`; se `cmd` vale "-264.37", então nós inserimos o número -264.37 em `aPilha`. Para transformar `cmd` em `double`, você pode utilizar o método `static double parseDouble(String s)` da classe `Double` de Java, que retorna o `double` correspondente a string `s`.

Adicione agora à classe `CalcRPN` o método a seguir, a ser chamada a partir do método `main` (por razões técnicas, a assinatura de `main` deverá ser modificada para `public static void main (String[] args) throws IOException`):

```
static void interfaceUsuario() throws IOException {
CalcRPN calc = new CalcRPN() ;
String line;
BufferedReader reader = new BufferedReader
(new InputStreamReader (System.in));
while((line = reader.readLine()) != null) {
if (line.isEmpty())
continue;
for (String s : line.split(" "))
calc.exec(s);
System.out.println("Pilha = " + calc.aPilha);
}
System.out.println("Até logo");
}
```

Este código simplesmente lê os comandos digitados no teclado pelo usuário (separados por espações ou ¡enter¿) e os comunica sob a forma de cadeias de caracteres à calculadora via método `exec` da classe `calcRPN`. Para testar o seu código, vocês podem executar o programa e depois digitar sucessivamente as linhas (seguidas da tecla ¡enter¿) no console:

```
3 2 +
3 2 -
3 2 *
3 2 /
1 2 + 3 4 - / 10 3 - *
```

Você obterá após cada linha o estado da pilha ao fim do cálculo, com o topo da pilha representando o resultado do cálculo da linha. Para os exemplos acima os resultados devem ser os mesmos obtidos na seção **Operadores básicos**.

Você pode também utilizar o seu código com outras fórmulas, a fim de melhor compreender o funcionamento da calculadora.

A instrução `clear` ´ Vamos agora adicionar uma instrução "clear" a nossa calculadora, que apagará o conteúdo da pilha.

Adicione à classe `Pilha` um método `reinicialize` que retira todos os elemento da lista `conteudo`.

Complete em seguida o método `exec` da classe `CalcRPN` para que a calculadora reaja ao comando "clear" removendo o conteúdo de `aPilha`.

Para testar o seu código, você pode executar o programa e depois digitar as linhas a seguir no console:

```
3 2 +
3 2 -
clear
3 2 +
clear 3 2 -
```

o que produzirá o resultado:

```
Pilha = [5.0]
Pilha = [1.0, 5.0]
Pilha = []
Pilha = [5.0]
Pilha = [1.0]
```

3. Gestão do histórico

Nesta parta da prática, nós iremos melhorar a nossa calculadora adicionando a ela um histórica das operações efetuadas. Isto permitirá entre outras coisas cancelas as últimas operações. Cada operação será representada por um objeto da classe `Operacao` definida como abaixo. O histórico terá a forma de uma pilha de operações e será portanto do tipo `Pilha<Operacao>`.

A classe `Operacao`

Nós vamos representar dois tipos de operação: os empilhamentos e as operações básicas aritméticas (+, -, *, /).

Crie a classe `Operacao` com 3 campos:

- `char code` que codifica a natureza da operação. Por convenção, + terá o código '+', - o código '-', * o código '*', / o código '/', e os empilhamentos o código 'e'
- dois campos `a,b` do tipo `double` que armazena os operandos. Por convenção, no caso de um empilhamento (código 'e') o valor empilhado será armazenado no campo `a`.

Adicione dois construtores:

- `public Operation(double a)` que cria uma operação de empilhamento (código 'e') de valor `a`,
- `public Operation(char code, double a, double b)` que cria uma operação aritmética de código `code` com os operandos `a` e `b`.

Complete a classe adicionando a ela um método `public String toString()`, que retorna uma cadeia de caracteres representando o valor empilhado no caso de um empilhamento ou o operador no outros caso. Você poderá testar o seu código adicionando o método `main` a seguir à classe `Operacao`.

```
public static void main (String[] args) {
Operacao[] op = new Operacao[9];
op[0] = new Operacao(16.0);
op[1] = new Operacao(8.0);
op[2] = new Operacao(4.0);
```

```
op[3] = new Operacao(2.0);
op[4] = new Operacao(1.0);
op[5] = new Operacao('+', 2.0, 1.0);
op[6] = new Operacao('-', 4.0, 3.0);
op[7] = new Operacao('*', 8.0, 1.0);
op[8] = new Operacao('/', 16.0, 8.0);
for (int i=0; i<op.length; i++)
    System.out.print(op[i] + " ");
System.out.println();
}
```

Você deve obter o resultado a seguir:

```
16.0 8.0 4.0 2.0 1.0 + - * /
```

Criação e impressão do histórico

Adicione à classe `CalcRPN` um campo `Pilha<Operacao> hist` e a inicialize dentro do construtor.

Agora que o histórico está inicializado, é preciso preenchê-lo. Para isto, modifique os métodos `mais`, `menos`, `vezes` e `dividido` a fim de que els empilhem as operações correspondentes em `hist`. Modifique igualmente o método `exec` a fim de que reporte em `hist` as operações de empilhamento de números na pilha.

Na classe `Pilha`, adicione um método `public String toStringInverse()` que retorna uma cadeia de caracteres correspondente ao conteúdo da lista `conteudo` na ordem inversa, entre dois colchetes "[" e "]", com "," como separador entre os elementos. Existem várias maneiras de fazer isto sem afetar a lista `conteudo`. Por exemplo, você pode criar uma pilha temporária na qual você empilha os elementos de `conteudo`, para em seguida desempilhar seus elementos à medida que você cria a cadeia de caracteres resultante.

Complete em seguida o método `exec` da classe `calcRPN` para que a calculadora reaja ao comando "hist" imprimindo o conteúdo do histórico, e ao comando "clear" que apaga o conteúdo do histórico além apagar também o conteúdo de `aPilha`.

Você pode testar o seu código executando o programa e depois digitando as duas linhas a seguir no console:

```
hist 1 2 3 + 7 - 16 8 / * hist
clear hist
```

o que deverá resultar em:

```
Historico = []
Historico = [1.0, 2.0, 3.0, +, 7.0, -, 16.0, 8.0, /, *]
Pilha = [-4.0, 1.0]
Historico = []
Pilha = []
```

É recomendado fazer testes amplos a fim de se assegurar que o seu histórico funciona bem.

Cancelamento dos últimos comandos

Graças ao histórico, nós podemos agora dar ao usuário a possibilidade de cancelar o último comando digitado. Para isto, é suficiente buscar no histórico qual é a última operação registrada, e depois retirá-la do histórico recolocando `aPilha` em seu estado precedente.

Para atualizar a pilha, nós procederemos como a seguir:

- Se a última operação foi um empilhamento, então nós simplesmente retiramos o elemento do topo da pilha
- senão, o último comando foi uma das quatro operações binárias (+, -, *, /). Neste caso retiramos o resultado do topo da pilha e recolocamos os operandos, na mesma ordem.

Na classe `CalcRPN`, adicione um método `void cancela()` que implementa o cancelamento do último comando como descrito acima. Modifique igualmente o método `exec` a fim de que ele chame o método `cancela` e imprima o novo histórico quando o usuário digite o comando "undo". Você pode testar o seu código executando o programa e depois digitando a linha a seguir no console:

```
3 2 4 8 - + * hist undo
```

o que deve fornecer o resultado a seguir:

```
Historico = [3.0, 2.0, 4.0, 8.0, -, +, *]
Historico = [3.0, 2.0, 4.0, 8.0, -, +]
Pilha = [-2.0, 3.0]
```

Aqui mais uma vez é recomendado que você faça vários testes a fim de assegurar a corretude do código.

*Este trabalho prático é de autoria de Steve Oudot (Poly, France)