

Лекция 12b

Приложения на
параметризацията по
тип(generics)

*Пресмятане на
аналитичен израз*

Основни теми

- Приложение на стек за анализ и пресмятане на аритметичен израз.
- **`infix`** и **`postfix`** представяне на аритметичен израз
- Представяне в **обратен полски запис (RPN-*Reverse Polish Notation*)**
- Пресмятане на **`postfix`** представяне на аритметичен израз

- 12b.1** class Stack – преговор
- 12b.2** Пресмятане на аритметични изрази
- 12b.3** Видове представяне на аритметични изрази
- 12b.4** Преобразуване от *infix* в *postfix* означения
- 12b.5** Пресмятане на аритметичен израз в *postfix* означение

Задачи

Литература:

Н. М. Deitel, Р. J. Deitel *Java How to Program, 7 Edition, глава 17*

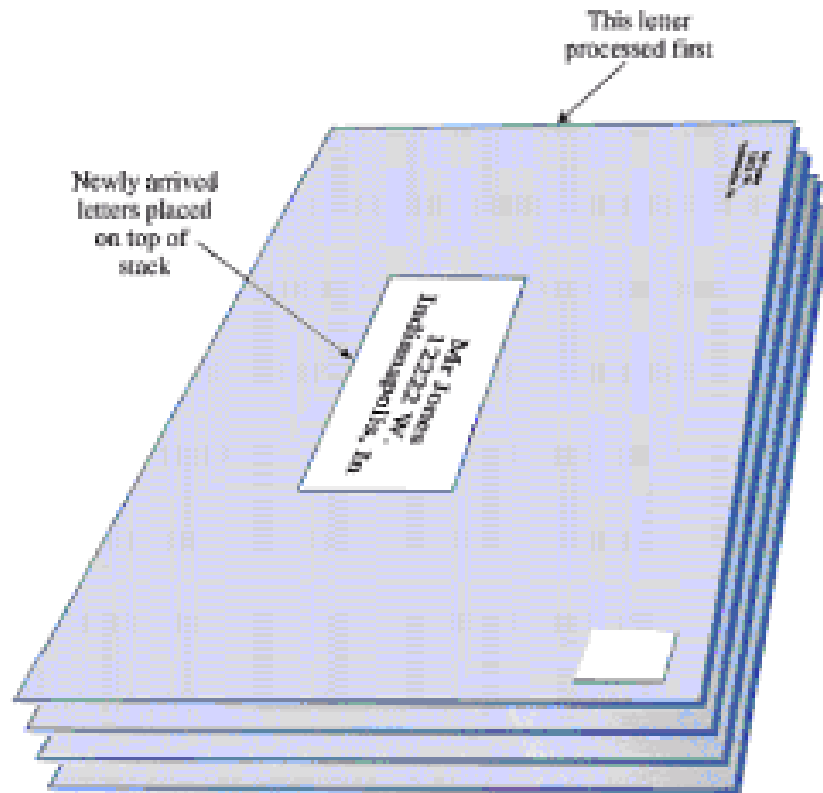
12b.1 class Stack - Въведение

Stack- **частен случай** на **свързан списък**, но не е задължително да се реализира като свързан списък

Нови елементи се **добавят и изтриват само в началото** на стека (*последният `ListNode` има данна `next = null`*)

- Last-in, first-out (LIFO) структура от данни с основни методи
 - Метод **push** добавя нов елемент в началото на стека
 - Метод **pop** премахва елемент от началото на стека и връща данната, съдържаща в отстранения елемент.

12b.1 class **Stack** – пример за стек



12b.1 class Stack

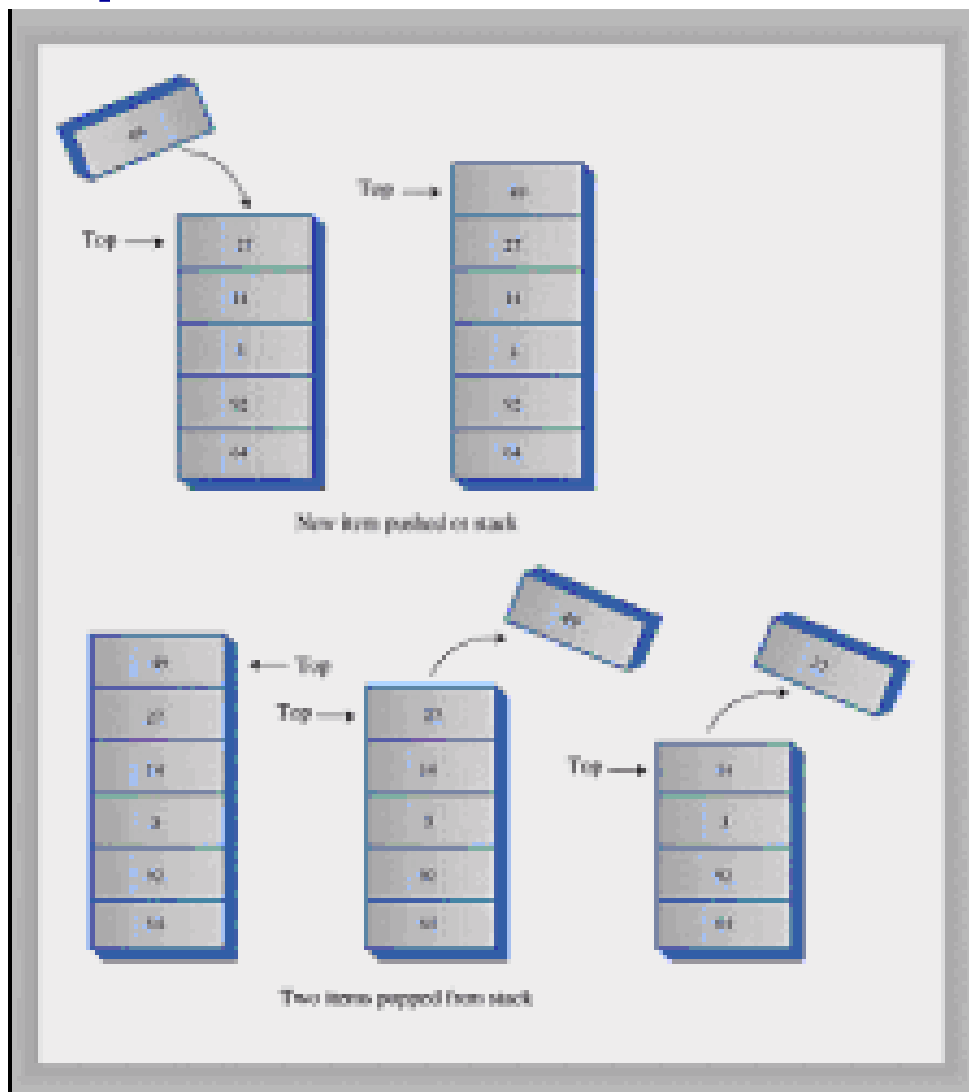
`class StackComposition` дефинира `Stack` **КОМПОЗИЦИЯ** на `class ArrayList` т.е. `class StackComposition` има референция към обект `ArrayList<E>`

Основни методи

`ArrayList<E> stackList` е данна на `StackComposition`

- `push()` се реализира като `stackList.add(0, object)`
- `pop()` се реализира като `stackList.remove(0)`
- `isEmpty()` се реализира като `stackList.isEmpty()`
- `peek()` – връща копие от данните в първия елемент (във върха на стека) и се реализира като `stackList.get(0)`
- `size()` се реализира като `stackList.size()`
- `toString` се реализира като `stackList.toString()`

12b.1 class **Stacks** – основни операции



StackComposition
.java

(1 от 2)

```
1 // Fig. 17.12: StackComposition.java
2 // Class StackComposition definition with composed List object.
3 import java.util.ArrayList;
4
5 public class StackComposition<E>
6 {
7     private ArrayList<E> stackList;
8
9     // no-argument constructor
10    public StackComposition()
11    {
12        stackList = new ArrayList<>();
13    } // end StackComposition no-argument constructor
14
15    // add object to stack
16    public void push( E object )
17    {
18        stackList.add( 0, object );
19    } // end method push
20
```

private List клас
данна

Метод **push** се изпълнява
посредством метод
insertAtFront на клас
List


```
21 // remove object from stack
22 public E pop() throws IndexOutOfBoundsException
23 {
24     return stackList.remove(0);
25 } // end method pop
26
27 // determine if stack is empty
28 public boolean isEmpty()
29 {
30     return stackList.isEmpty();
31 } // end method isEmpty
32
33 // output stack size
34 public int size()
35 {
36     return stackList.size();
37 } // end method size
38
39 // output contents of the top stack Object
40 public E peek ()
41 {
42     return stackList.get(0);
43 } // end method peek
44
45 // output stack contents
46 public String toString()
47 {
48     return stackList.toSstring();
49 } // end method print
50 } // end class StackComposition
```

Метод **pop** изпълнява **List**
метода **removeFromFront**

Метод **isEmpty** изпълнява **List**
метода **isEmpty**

Метод **size** изпълнява
List метода **size**

Метод **peek** изпълнява **List**
метода **peekFirst**

Метод **toString** изпълнява
List метода **toSstring**

12b.2 Пресмятане на аритметични изрази

Класически алгоритми за пресмятане на аритметични изрази.

Базиран се на разработките от 1962 на Доналд Кнут



<http://www-cs-faculty.stanford.edu:80/~knuth/> .

12b.2 Пресмятане на аритметични изрази

Кнут извършва пресмятането на изрази на три стъпки:

- Прочитане на аритметичния израз в *infix* вид
- Преобразуване на *infix* израза в *postfix* вид
- Пресмятане на *postfix* израз

12b.3 Видове представяне на аритметични изрази

Съществуват три формата за представяне на аритметичен израз :

- *infix*,
- *prefix*,
- *postfix*

12b.3 Видове представяне на аритметични изрази- *infix*

infix е видът , в който **най- често се пишат аритметични изрази**

Нарича се *infix* , защото **всеки аритметичен оператор се поставя между операндите си.**

Това е възможно само за бинарни оператори като операторите за *събиране, изваждане, делене, умножение и делене по модул*).

Аритметични изрази в този вид **имат нужда от скоби и правила за приоритет** на операциите с цел избягване на двусмислие в израза.

12b.3 Видове представяне на аритметични изрази- *infix*

infix примери

Syntax: **operand1** **operator** **operand2**

Примери:

$(A+B) * C - D / (E+F)$

$3 + 4$

$7 / 9$

$(A+B) * (C-D)$

$((A+B) * C - (D-E)) \% (F+G)$

12b.3 Видове представяне на аритметични изрази- *infix*

prefix е видът , при който операторът се изписва преди операндите си в аритметичния израз

Използва се за създаване на компилатори.

Нарича се *Polish notation* в чест на Полския математик, който е въвел (1920) това представяне на аритметични изрази

Jan Lukasiewicz (1878-1956)

12b.3 Видове представяне на аритметични изрази- *prefix*

prefix примери

Syntax: **operator operand1 operand2**

Пример:

+AB

съответства на израза **A+B**

- * + ABC / D + EF

съответства на израза

(A+B) * C - (D / (E+F))

12b.3 Видове представяне на аритметични изрази- *postfix*

postfix е видът , при който операторът се изписва след операндите си в аритметичния израз

Нарича се *Reverse Polish notation* и се използва за лесно пресмятане на аритметични изрази

Charles Hamblin (средата на 1950-те)

Използват се на калкулатори НР още от началото на 1960-те

12b.3 Видове представяне на аритметични изрази- *postfix*

postfix примери

Syntax: **operand1 operand2 operator**

Пример:

AB+

съответства на израза A+B

AB+C*DEF+/-

съответства на израза

(A+B) *C - (D/ (E+F))

12b.3 Видове представяне на аритметични изрази

prefix и *postfix* сравнение

prefix и *postfix* означенията имат три общи черти:

1. **Операндите са в същия ред,** както в зададения аритметичен израз в *infix* означения.
2. Не са нужни **скоби**.
3. **Приоритетът** на операторите е без значение.

12b.4 Преобразуване от *infix* в *postfix* означения

Компютрите “*предпочитат*” *postfix* означенията за аритметични изрази.

За пресмятане на сложен *infix* израз, е необходим компилатор, който да прочете, валидира и преобразува *infix* аритметичен израз в *postfix* означения и накрая да пресметне *postfix* версията.

Всеки от използваните алгоритми:

- извършва **единствено обхождане отляво надясно** на израза
- използва *stack* **обект** за тези цели, но при различните алгоритми *stack* обекта се използва за **различни цели**

12b.4 Преобразуване от *infix* в *postfix* алгоритъм

Ще напишем `class InfixToPostfixConverter` за преобразуване на зададен аритметичен израз в `infix` означения до `postfix` означения.

Например

$(6 + 2) * 5 - 8 / 4$

ще преобразуваме до

$6\ 2\ +\ 5\ *\ 8\ 4\ /\ -$

Програмата ще чете аритметичния израз в `StringBuffer infix` и ще използва `stack class` за генериране на съответното *postfix* означение в `StringBuffer postfix`.

12b.4 Преобразуване от *infix* в *postfix* алгоритъм

Ще ограничим програмирането до **операнди**, които са **едноцифрени** числа.

Предполагаме, че зададения аритметичен израз е написан правилно в *infix* означения.

12b.4 Преобразуване от *infix* в *postfix* алгоритъм

1. **Push** a left parenthesis ' (' on the `stack`.
2. **Append** a right parenthesis ')' to the end of `infix`.
3. **While** the **stack is not empty**, read `infix` from left to right and do the following:
 - If the **current character** in `infix` is a **digit**, **append** it to `postfix`.
 - If the current character in `infix` is a **left parenthesis**, **push** it onto the `stack`.
 - If the current character in `infix` is an **operator**:
 - **Pop** operators (if there are any) at the top of the `stack` while they have **equal** or **higher precedence** than the current operator, and **append** the popped operators to `postfix`.
 - **Push** the current character in `infix` onto the `stack`.
 - If the current character in `infix` is a right parenthesis:
 - **Pop** operators from the **top** of the `stack` and append them to `postfix` until a **left parenthesis** is at the top of the `stack`.
 - **Pop** (and discard) the **left parenthesis** from the `stack`.

12b.4 Преобразуване от *infix* в *postfix* алгоритъм

Пример

(вижте анимацията на алгоритъма в [Postfix Notation Mini Lecture](#))

$A + B * C$

Read	Stack	Postfix	Comment
A	(A	read first operand, set $A + B * C$)
+	(+	A	read operator push on stack
B	(+	AB	read operand
*	(+ *	AB	read operator push on stack (no- висок приоритет)
C	(+ *	ABC	read operand
)	empty	ABC*+	(pop up stack elements)

12b.4 Преобразуване от *infix* в *postfix* алгоритъм

Пример

(вижте анимацията на алгоритъма в [Postfix Notation Mini Lecture](#))

$A * B + C$

Read	Stack	Postfix	Comment
A	(A	read first operand, set $A * B + C$)
*	(*	A	read operator push on stack
B	(*	AB	read operand
+	(+	AB*	read operator push on stack (<i>по- нисък приоритет</i>)
C	(+	AB*C	read operand
)	empty	AB*C+	(pop up stack elements)

12b.4 Преобразуване от *infix* в *postfix* алгоритъм

Ще реализираме следните оператори

+ събиране

– изваждане

*** умножение**

/ делене

^ повдигане на степен

% делене по модул

Ще използваме стек, който има едносвързан списък

12b.4 Преобразуване от *infix* в *postfix* алгоритъм Stack методи

Метод `convertToPostfix`, преобразува `infix` израз в `postfix` означения.

Метод `isOperator`, определя дали прочетен символ `char` е оператор.

Метод `precedence`, определя дали приоритетът на `operator1` (от `infix` израза) е по-малък, равен или по-голям приоритет от `operator2` (в `stack`-а). Методът връща `true` ако `operator1` е с по-нисък приоритет от `operator2`. В противен случай, връща `false`.

Метод `stackTop` (е методът `peek()` в `stack` класа), връща стойността на върха на стека без да я изхвърля от `stack`-а

12b.4 Преобразуване от *infix* в *postfix*

```
package postfixconvert;
```

```
import stackcomposition.*;
```

```
public class InfixToPostfixConverter
```

```
{
```

```
// take out the infix and change it into postfix
```

```
public static StringBuffer convertToPostfix( StringBuffer infix )
{
```

```
    StackComposition<Character> charStack = new StackComposition<>();
```

```
    StringBuffer temporary = new StringBuffer( "" );
```

```
    // push a left paren onto the stack and
```

```
    // add a right paren to infix
```

```
    charStack.push( '(' );
```

```
    infix.append( ')' );
```

Клас с метод за преобразуване
от **infix** в **postfix**
означения

Създава **CharacterStack**
и **StringBuffer** за
записване на **postfix**
означението

Добавя лява отваряща скоба в
стека и затваряща скоба
infix израза

12b.4 Преобразуване от *infix* в *postfix*

```
// convert the infix expression to postfix
for ( int infixCount = 0; !charStack.isEmpty(); ++infixCount )
{
    if ( Character.isDigit( infix.charAt( infixCount ) ) )
        temporary.append( infix.charAt( infixCount ) + " " );
    else if ( infix.charAt( infixCount ) == '(' )
        charStack.pushChar( '(' );
    else if ( isOperator( infix.charAt( infixCount ) ) )
    {
        while ( isOperator( charStack.peek() ) &&
            precedence( charStack.peek(),
                infix.charAt( infixCount ) ) )
            temporary.append( charStack.pop() + " " );

        charStack.push( infix.charAt( infixCount ) );
    } // end else if
    else if ( infix.charAt( infixCount ) == ')' )
    {
        while ( charStack.peek() != '(' )
            temporary.append( charStack.popChar() + " " );

        charStack.pop();
    } // end else if
} // end for

return temporary;
} // end method convertToPostfix
```

Четем в цикъл **infix** изрази
докато **charStack** не стане
празен т.е. когато прочетем
последната затваряща скоба

12b.4 Преобразуване от *infix* в *postfix*

```
// check if c is an operator
private static boolean isOperator( char c )
{
    if ( c == '+' || c == '-' || c == '*' || c == '/' || c == '^' )
        return true;
    else
        return false;
} // end method isOperator
```

Определя дали прочетения символ
е оператор

12b.4 Преобразуване от *infix* в *postfix*

```
private static boolean precedence( char operator1, char operator2 )
{
    if ( operator1 == '^' )
        return true;
    else if ( operator2 == '^' )
        return false;
    else if ( operator1 == '*' || operator1 == '/' )
        return true;
    else if ( operator1 == '+' || operator1 == '-' )
    {
        if ( operator2 == '*' || operator2 == '/' )
            return false;
        else
            return true;
    } // end else if

    return false;
} // end method precedence
} // end class InfixToPostfixConverter
```

Определя дали приоритета на
върха на стека спрямо този на
прочетен оператор е по- голям
или равен

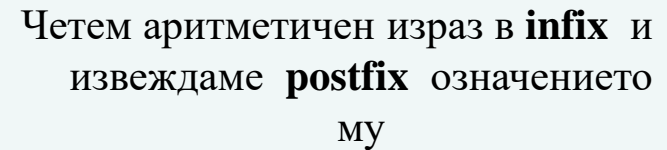
12b.4 Преобразуване от *infix* в *postfix*

```
package postfixconvert;
// Infix to postfix conversion
import java.util.Scanner;
public class InfixToPostfixConverterTest
{
    public static void main( String args[] )
    {
        // get infix expression
        Scanner scanner = new Scanner( System.in );
        System.out.println( "Please enter an infix expression:" );
        StringBuffer infix = new StringBuffer( scanner.nextLine() );
        System.out.printf(
            "\nThe original infix expression is:\n%s\n", infix );

        // change from infix notation into postfix notation
        StringBuffer postfix =
            InfixToPostfixConverter.convertToPostfix( infix );

        System.out.printf(
            "The expression in postfix notation is:\n%s\n", postfix );
    } // end main
} // end class InfixToPostfixConverterTest
```

Четем аритметичен израз в **infix** и
извеждаме **postfix** означението
му



12b.5 Пресмятане на *postfix* израз

алгоритъм

Пресмятането на *postfix* израз е по-просто от директно пресмятане на *infix* израз. В *postfix* означения, се елиминира нуждата от скоби и приоритета на операторите е без значение.

Алгоритъм за пресмятане на *postfix* изрази:

1. Инициализираме празен *stack*.
2. Прочитаме *postfix* изразът отляво надясно.
3. Ако символът е операнд, поставяме го (*push*) го върху стека.
4. Ако символът е оператор, **взимаме два операнда** като ги изхвърляме (*pop*) от стека, **изпълняваме съответния оператор** и **поставяме** (*push*) резултата върху *stack*-а . Ако не може да вземем два операнда от *stack*-а , то синтаксиса на *postfix* изразът е грешен.
5. **При прочитане на края** на *postfix* израза, изхвърляме (*pop*) резултата от *stack*-а . Ако стека се окаже не празен в този момент, синтаксиса на *postfix* изразът е грешен.

12b.5 Пресмятане на *postfix* израз

алгоритъм

5 1 2 + 4 * + 3 -

Input	Operation	Stack	Comment
5	Push operand	5	
1	Push operand	5, 1	
2	Push operand	5, 1, 2	
+	Add	5, 3	Pop two values (1, 2) and push result (3)
4	Push operand	5, 3, 4	
*	Multiply	5, 12	Pop two values (3, 4) and push result (12)
+	Add	17	Pop two values (5, 12) and push result (17)
3	Push operand	17, 3	
-	Subtract	14	Pop two values (17, 3) and push result (14)

$$5 + ((1 + 2) * 4) - 3 = 14$$

12b.5 Пресмятане на *postfix* израз

алгоритъм

Програмна реализация в `class PostfixEvaluator`, с която може за пресмятане `postfix` изрази от вида

6 2 + 5 * 8 4 / -

Програмата прочита `postfix` израз състоящ се от цифри и оператори в един `StringBuffer`. Допускаме, че `postfix` израза е с вярна синтакс и разглеждаме оператори

+ събиране

- изваждане

* умножение

/ делене

^ повдигане на степен

% делене по модул

Ще използваме стек, който има едносвързан списък

12b.5 Пресмятане на *postfix* израз алгоритъм – програмна реализация

- а) Добавяме дясна скоба ')' в края на *postfix* израза. При прочитане на тази скоба извеждаме резултата от пресмятането.
- б) Ако не е прочетена дясна скоба, четем *postfix* израза отляво надясно.
- Ако **прочетеният символ е цифра**, поставяме целочислената ѝ стойност на **stack** (*числената стойност на цифра е разликата на Unicode стойностите на символа на прочетената цифра и символа за цифрата '0'*).
 - В противен случай, ако е **прочетен символ на оператор** :
 - Изхвърляме (**pop**) два елемента от **stack**-а и ги записваме в променливите **х** и **у**.
 - Пресмятаме резултата от израза **у оператор х**.
 - Поставяме получения резултат върху **stack**-а.
- в) При прочитане на дясна скоба в *postfix* израза, изхвърляме (**pop**) елемент от **stack**-а. Това е резултатът от пресмятането на *postfix* израза.

12b.5 Пресмятане на *postfix* израз алгоритъм – програмна реализация

`class PostfixEvaluator` реализира следните методи

- a) Метод `evaluatePostfixExpression`, пресмята *postfix* израза.
- b) Метод `calculate`, пресмята израза `op1 operator op2`.
- c) Метод `push`, който поставя стойност върху `stack`-а.
- d) Метод `pop`, който изхвърля стойност от `stack`.
- e) Метод `isEmpty`, който определя дали `stack`-а е празен
- f) Метод `printStack`, който отпечатва

12b.5 Пресмятане на *postfix* израз

```
package postfixevaluate;
// Using a stack to evaluate an expression in postfix notation
import stackcomposition.*;

public class PostfixEvaluator
{
    // evaluate the postfix notation
    public static int evaluatePostfixExpression( StringBuffer expr
    )
    {
        int i, popVal1, popVal2, pushVal;
        StackComposition<Integer> intStack =
            new StackComposition<>();

        char c;

        expr.append( " )" );
```

Създаваме празен стек и добавяме
затваряща скоба към postfix
израза

12b.5 Пресмятане на *postfix* израз

```
// until it reaches ")"
```

```
for ( i = 0; expr.charAt( i ) != ' )'; ++i )
```

```
{
```

```
    if ( Character.isDigit( expr.charAt( i ) ) )
```

```
    {
```

```
        pushVal = expr.charAt( i ) - '0';
```

```
        intStack.push( pushVal );
```

```
        System.out.println(intStack);
```

```
    } // end if
```

```
    else if ( !Character.isWhitespace( expr.charAt( i ) ) )
```

```
    {
```

```
        popVal2 = intStack.pop();
```

```
        System.out.println(intStack);
```

```
        popVal1 = intStack.pop();
```

```
        System.out.println(intStack);
```

```
        pushVal = calculate( popVal1, popVal2, expr.charAt( i ) );
```

```
        intStack.push( pushVal );
```

```
        System.out.println(intStack);
```

```
    } // end else if
```

```
} // end for
```

```
    return intStack.pop();
```

```
} // end method evaluatePostfixExpression
```

Прочитаме **postfix**
израза отляво
надясно

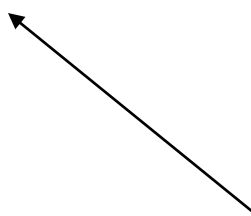
Ако е прочетена цифра
стойността ѝ се
записва в стека

Ако е прочетен оператор,
се изчислява междинен
резултат и се записва в
стека

12b.5 Пресмятане на *postfix* израз

```
// do the calculation
private static int calculate( int op1, int op2, char oper )
{
    switch( oper )
    {
        case '+':
            return op1 + op2;
        case '-':
            return op1 - op2;
        case '*':
            return op1 * op2;
        case '/':
            return op1 / op2;
        case '^':    // exponentiation
            return ( int )Math.pow( op1, op2 );
    } // end switch

    return 0;
} // end method calculate
} // end class PostfixEvaluator
```



Метод за извършване на пресмятания с операторите

Задачи

Задача 1.

Запишете в **postfix** следните **infix** изрази като приложите стъпка по стъпка алгоритъма за преобразуване в **postfix**

$A + B / (C - D)$

$(A+B) * (C-D)$

$((A+B) * C - (D-E)) \% (F+G)$

Задача 2.

Проследете стъпка по стъпка работата на алгоритъма за пресмятане на **postfix** израза

$6\ 2\ 3\ +\ -\ 3\ 8\ 2\ /\ +\ *$

Задачи

Задача 3

Използвайте дадения примерен код и реализирайте следните **допълнителна функционалност**:

- Извеждане на съобщение за грешка при прочитане грешен символ, неподдржан оператор или грешен формат на зададения `postfix` или `infix` израз
- Обработка операнди , които са цели числа по- голями от 9