# **CalcTest**

Calc c = new Calc();

Создаётся новый калькулятор **c** – объект класса **Calc**.

## Calc

public Calc() {  
 s = new StackInt();  
}

В конструкторе калькулятора определяется объект **s** класса **StackInt** через соответствующий конструктор.

### **StackInt**

public StackInt() {  
 array = new int[*DEFSIZE*];  
 head = 0;  
}

В данном классе определяется целочисленный массив **array**, а выставляется указатель **head** на нулевой элемент массива.

# **CalcTest**

Далее запускается бесконечный цикл, принимающий за каждую итерацию одно вычисляемое выражение.

c.compile(in.next().toCharArray());

После того, как выражение введено, к калькулятору применяется метод компиляции **compile(char[] str)**, в который передается строка с формулой, разбитая посимвольно на массив.

## Calc

В описанном выше методе переменная **super** ссылается на соответствующий метод в родительском классе:

super.compile(str);

Родительский класс – **Compf**.

### **Compf**

Компиляция выражения происходит следующим образом:

processSymbol('(');  
  
for (int i = 0; i < str.length; i++)  
 processSymbol(str[i]);  
  
processSymbol(')');

Обрабатывается мнимая левая открывающая скобка, затем обрабатывается каждый символ выражения, а после обрабатывается мнимая правая закрывающая скобка.

Для обработки символов используется метод **processSymbol(char c)**, описанный в этом же классе. Его задача заключается в получении текущего символа, передачи этого символа на анализ в метод **symType(char c)**, а затем (в зависимости от возвращаемого значения метода) переключение на один из предложенных кейсов:

case *SYM\_LEFT*:  
 *flag* = false;  
 push(c);  
 break;  
case *SYM\_RIGHT*:  
 *flag* = false;  
 processSuspendedSymbols(c);  
 pop();  
 break;  
case *SYM\_OPER*:  
 *flag* = false;  
 processSuspendedSymbols(c);  
 push(c);  
 break;  
case *SYM\_OTHER*:  
 nextOther(c);  
 *flag* = true;  
 break;

В случае с левой скобкой **SYM\_LEFT** символ просто добавляется в массив символов, описанный в классе **Stack**.

Если же символ оказывается правой скобкой **SYM\_RIGHT**, то он попадает в метод **processSuspendedSymbols(char c)** на обработку, а после удаляется.

Аналогично происходит в случае с оператором **SYM\_OPER**. Отличие от правой скобки заключается в том, что символ не удаляется, а добавляется в массив символов.

Последний кейс работает с символами **SYM\_OTHER** английского алфавита в первоначальной конфигурации программы. Однако затем некоторые методы класса **Compf** были переписаны в классе **Calc**, чтобы трансформировать компилятор в калькулятор целочисленных неотрицательных чисел.

private void processSuspendedSymbols(char c) {  
 while (precedes(top(), c))  
 nextOper(pop());  
}

Рассмотрим целиком метод **processSuspendedSymbols**. Он ничего не возвращает, зато внутри него вызывается еще один метод данного класса **nextOper(char c)**.

Метод вызывается в цикле, условием выполнения которого является метод **precedes(char a, char b)**. В рамках рассматриваемого данный метод берет на вход верхний элемент стека класса **Stack** и передаваемое значение символьной переменной.

Каким образом работает метод **precedes**?

if (symType(a) == *SYM\_LEFT*) return false;  
if (symType(b) == *SYM\_RIGHT*) return true;  
  
return priority(a) >= priority(b);

Рассматриваются 4 возможных случая:

1. Если скобка открывающая, то возвращается false, поскольку в таком случае еще нет выражения, над которым необходимо провести операцию.
2. Если приоритет второго полученного на вход значения больше, то также возвращается false (например, если : в стеке чисел уже лежат значения 6 и 2, а в стеке символов – «+», однако поступившая на вход операция умножения должна выполнятся раньше, чем лежащий сверху стека плюс, но второго ее множителя еще нет в стеке чисел, поэтому никаких операций на данном этапе выполнятся не будет).
3. Если скобка закрывающая, то возвращается true, поскольку выражение или его часть готовы к вычислению.
4. Если приоритет второго полученного на вход значения больше, то возвращается true (опять же, если возвращаться к примеру выше, то ; если выражение записано так, то, как только в стек чисел попадет «4», произведется умножение, ведь его приоритет выше, чем компилируемый следом «+»).

Эти случаи могут показаться непонятными на первый взгляд.

Однако давайте представим, что ввели выражение: . Сразу же преобразуем к виду, читаемому программой. Получается:

Пронумеруем элементы символьного массива, полученные после преобразования строки в **CalcTest**:

Тогда по ***обратной польской записи*** получаем: . Такая расстановка достигается посредством описанного метода, который, учитывая приоритет операций и скобок, располагает их в порядке произведения расчетов.

В случае выполнения условий цикла символы будут выводиться в консоли именно согласно данной записи с помощью метода **nextOper**.

Более наглядное представление работы программы через заполнение стеков можно увидеть **на gif-изображении**.

Вернемся в метод **symType(char c)**. После вызова и отработки метода **processSuspendedSymbols** в кейсе по закрывающей скобке происходит ее удаление из массива, а в кейсе по операциям – добавление. Также есть кейс по остальным символам, который, если речь идет только о компиляторе, ссылается через метод **nextOther** на метод **nextOper**.

## Calc

Когда разобраны все взаимосвязи методов в классе **Compf**, посмотрим внимательнее на содержимое класса **Calc**. По сути из нового данный класс содержит только конструктор и метод перевода символов в целочисленные значения. Нет нужды разбирать это.

Однако также тут есть переписанные методы **nextOper** и **nextOther**.

Первый указанный метод здесь содержит в себе все операции, производимые над двумя верхними значениями стека, объекта класса **StackInt**:

int second = s.pop();  
int first = s.pop();  
  
switch (c) {  
 case '+':  
 s.push(first + second);  
 break;  
 case '-':  
 s.push(first - second);  
 break;  
 case '\*':  
 s.push(first \* second);  
 break;  
 case '/':  
 s.push(first / second);  
 break;  
 case '^':  
 s.push((int) Math.*pow*(first, second));  
 break;  
}

Если до этого программа выводила обратную польскую запись из символьного выражения типа , то теперь выполняет введенные операции над числами.

Второй метод **nextOther** аналогично был преобразован для работы с числами.

if (*flag*)  
 s.push(s.pop() \* 10 + *char2int*(c));  
else  
 s.push(*char2int*(c));

Здесь отдельно лежащие в стеке цифры начинают «склеиваться» в число, если между ними не стоит знака операции. Это условие заложено в переменную **flag**, которая задавалась еще в методе **processSymbol** класса **Compf**.

# **Double Upd.**

Попробуем теперь изменить тип чисел с целых на дробные.

Начнем с того, что представим себе примитивное возможное выражение в таком случае: .

Сначала в стек чисел упадет «3», затем «1» и они через переписанный метод **nextOther** преобразуются к «31». Все как обычно. Однако затем программа встретит на своем пути символ точки и тогда, отправив его на проверку в метод **symOther**, столкнется с ошибкой.

Исправим множество допустимых значений:

if ((c < '0' || c > '9') && c != '.') {  
 System.*out*.println("Invalid character: " + c);  
 System.*exit*(0);  
}

Теперь важно, чтобы значения после точки не «приклеивались» к предыдущим, а становились их дробной частью. Добавим и поставим новый флаг **isDouble** в методах **symType** и **processSymbol**. Также добавим целочисленный показатель степени **n**, который работает только при **isDouble** = true.

Обернем все на стадии определения символа в методе **symType**:

case '.':  
 *isDouble* = true;  
 *n* = 1;  
 return *SYM\_DOT*;

Где:

case *SYM\_DOT*:  
 break;

А так выглядит метод **nextOther** после модернизации:

if (*flag*)  
 if (*isDouble*) {  
 s.push(s.pop() + *char2int*(c) / Math.pow(10, n));  
 *n*++;  
 } else  
 s.push(s.pop() \* 10 + *char2int*(c));  
else  
 s.push(*char2int*(c));

Теперь самое главное: преобразовать класс **StackInt** в **StackDouble**.

# **Negative Upd.**

Чтобы решить, каким образом возможно модернизовать программу под обработку отрицательных чисел, рассмотрим 3 возможных случая:

Поскольку каждое выражение подразумевает еще мнимые скобки по краям, знак минуса никогда не идет первым в строке.

Значит, нужно работать с считыванием количества символов, не относящихся к операндам.

Добавим еще одну булеву переменную и назовем **isNegative**. По умолчанию она будет true. Также создадим счетчик количества операций **k**. При попадании в метод **compile** он будет принимать значение 1.

Счетчик при этом будет увеличиваться только при попадании в кейс **SYM\_OPER**.

switch (symType(c)) {  
 case *SYM\_LEFT*:  
 *flag* = false;  
 *isDouble* = false;  
 push(c);  
 break;  
 case *SYM\_RIGHT*:  
 if (*flag* && *k* > 1) *isNegative* = true;  
  
 *flag* = false;  
 *isDouble* = false;  
 processSuspendedSymbols(c);  
 pop();  
  
 *isNegative* = false;  
 *k* = 0;  
 break;  
 case *SYM\_OPER*:  
 if (*flag* && *k* > 1) *isNegative* = true;  
  
 *flag* = false;  
 *isDouble* = false;  
 processSuspendedSymbols(c);  
 push(c);

*isNegative* = true;  
 *k*++;  
 break;  
 case *SYM\_OTHER*:  
 *isNegative* = false;  
 if (*k* < 2) *k* = 0;  
 nextOther(c);  
 *flag* = true;  
 break;  
 case *SYM\_DOT*:  
 break;  
}

Вот так изменился метод **processSymbol** после его модернизации для обработки отрицательных значений.

Открывающие скобки никак не влияют на выявление и обработку отрицательных чисел.

Закрывающая скобка перед компиляцией проверяет, шло ли перед ней число (проверка на **flag** = true) и успел ли счетчик **k** дойти до 2+ символов операций (**k** > 1). Если оба условия соблюдены, то **isNegative** принимает значение true, что влияет на метод **nextOper:**

double first;  
if (*isNegative*) {  
 first = 0;  
 *k* = 0;  
} else   
 first = s.pop();

То бишь операции будут проводиться не над двумя числами из стека, а над последним его элементом (**second**) и нулем (**first**). Счетчик тем временем **k** обнуляется, поскольку выполнил свою задачу.

После компиляции закрывающей скобки переменные **isNegative** и **k** обнуляются, поскольку откомпилированная часть строки больше не будет влиять на алгоритм выявления отрицательных значений в выражении.

Кейс операций содержит аналогичное предыдущему описанному кейсу условие на **isNegative**, но в отличие от него увеличивает счетчик **k** и присваивает **isNegative** значение true. Это необходимо, чтобы учитывать откомпилированный символ в алгоритме выявления отрицательных значений.

И последний кейс остальных символов (чисел) еще перед компиляцией обнуляет счетчик **k**, если тот до достижения числового значения не выявил его отрицательности.

Трассировка программы проводилась на следующих примитивных выражениях:

Есть возможность оптимизации кода и выявления уязвимостей, однако ~~мне похуй~~ пока данный вопрос не рассматривается.