מחשבון

רקע

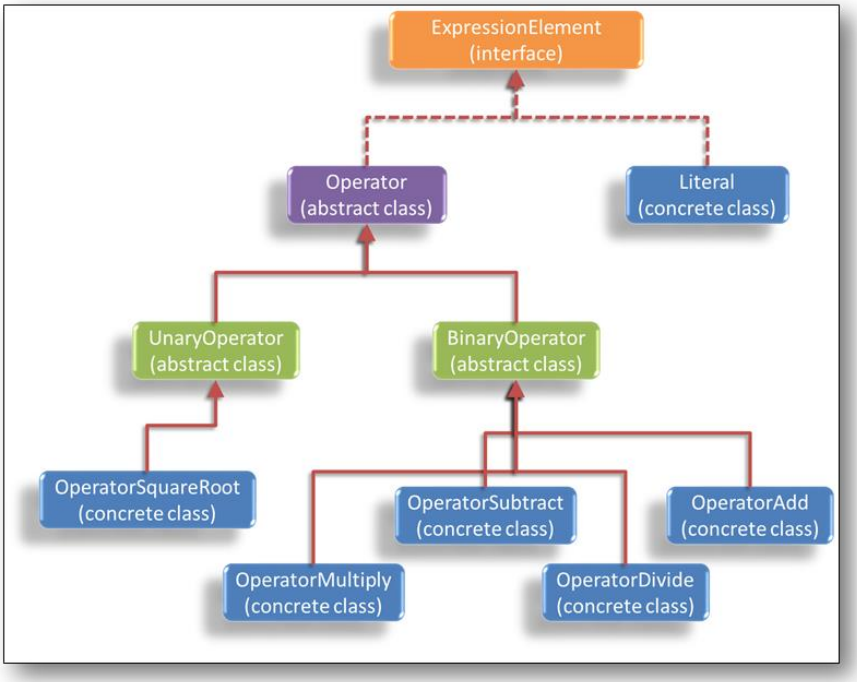
המחשבון הוא יישום המאפשר למשתמש להכניס ביטוי מתמטי. לאחר מכן הוא מחשב את הביטוי ומציג את התוצאות.

ביישום המחשבון שלנו, אין לנו אינטראקציה עם המשתמש (המכונה "ממשק משתמש"). אנחנו לא כותבים תוכנית, אלא את המחלקות המגדירות את הנתונים בבעיה. כל מי שכותב יישום ממשק המשתמש עבור מחשבון יכול להשתמש במחלקות שלנו כדי לחשב את הביטוי שהזין המשתמש, ולהראות את התוצאות. במשימה זו אנו רק מספקים להם מחלקות אשר מייצגות את כל האלמנטים של ביטוי אפשרי, וממשק המאפשר לדעת אילו שיטות נתמכות במחלקות אלו.

כן, זה קל מאוד לבקש מהמשתמש ביטוי חשבוני, לפענח מחרוזת, להמיר מספרים ולהפעיל את הפעולות תוך שימוש בבלוקים של if…else פשוטים, והכל בפעולת main פשוטה ובסיסית. עם זאת, זה ייחשב עיצוב רע. הסיבה שזה עיצוב גרוע, היא ששיטה אחת ארוכה (או שיטות רבות) היא קשה מאוד לשמירה. זו הסיבה שאנחנו בדרך כלל נעצב מחשבון הבנוי ממספר מחלקות. **אנחנו ננצל הירושה של Java וכלים פולימורפיזם כדי לעשות זאת בצורה אלגנטית וקלה.**

המחשבון שנבצע מבוסס על העיצוב המפורט להלן. מה שאתה רואה נקרא "Class Diagram", אשר מציג את הטיפוסים השונים של המחשבון ואת מערכות היחסים ביניהם. חץ המקווקו פירושו "יישום של ממשק" החץ מוצק פירושו "ירושה". לדוגמה, המחלקה המופשטת Operator מיישמת את ממשק ExpressionElement.

למד את תרשים המחלקות להלן, וודא שאתה מבין את העצמים השונים ואת מערכות היחסים ביניהם.



להלן קוד המיועד לחישוב הביטוי החשבוני **((15.4-3)*2)+(4*(1+5)) :**

```
// define literals
Literal l1  = new Literal(1);
Literal l2  = new Literal(2);
Literal l3  = new Literal(3);
Literal l4  = new Literal(4);
Literal l5  = new Literal(5);
Literal l15 = new Literal(15.4);

// build the expression
OperatorAdd a1 = new OperatorAdd(l1, l5);  // 1+5
OperatorMultiply m1 = new OperatorMultiply(l4, a1);   // 4*(1+5)
OperatorSubtract s1 = new OperatorSubtract(l15, l3);  // 15.4-3
OperatorMultiply m2 = new OperatorMultiply(s1, l2);   // (15.4-3)*2
OperatorAdd a2 = new OperatorAdd(m2, m1);  // ((15.4-3)*2)+(4*(1+5))

// calculate and print
System.out.println(a2 + " = " + a2.evaluate());
```

תיאור המחלקות

interface `ExpressionElement`

`ExpressionElement` הוא רכיב שניתן למצוא בביטוי מתמטי. רכיב כזה יכול להיות, למשל, סימן פלוס, סימן מינוס, מספר, וכן הלאה. ביטוי מתמטי הוא הרכבה של מספר אלמנטים שונים כאלה.

כל רכיב כזה יכול להיות מוצג כמחרוזת, או שזה יכול להיות בעל-ערך. הערך של (מספר בלבד) Literal הוא רק מספר. ההערכה של הפעולה פלוס, הוא תוצאה של הוספת האופרנד השמאלי עם האופרנד הימני, שהם גם אלמנטים של הביטוי.

כפי שניתן לראות בתרשים המחלקות, כל המחלקות בפרוייקט הקטן שלנו מיישמות את ממשק זה.

כל רכיב כזה "יודע" לומר מה ערכו (double evaluate()) ולהציג את עצמו (String toString()).

Class `Literal`

Literal הינה מחלקה המייצגת קבועים, ביישום שלנו Literal מייצג מספר המופיע בביטוי חשבוני ולכן כל קבוע מיישם את הממשק ExpressionElement.

למען הפשטות נניח כי כל המספרים הם ממשיים, ולכן נשתמש ב- double.

בשלב זה, אתה כבר יכול לבדוק את הקוד שלך !

כתוב את מחלקת Calculator ובה את הפעולה הסטטית main ונסה את הקוד הבא:

```
ArrayList<ExpressionElement> elements = new ArrayList < >();
elements.add(new Literal(14));
elements.add(new Literal(0));
elements.add(new Literal(-4));
elements.add(new Literal(13.1415));
elements.add(new Literal(-0.1));
for (ExpressionElement element : elements)  // foreach
{
    System.out.println(element + " = " + element.evaluate());
}
```

קוד זה מייצג פולימורפיזם בצורתו הטהורה ביותר: למרות ש-`elements` הוא אוסף של הפניות מסוג ExpressionElements, אנו יכולים להוסיף הפניות של סוגים אחרים, כל עוד הם יורשים מ-ExpressionElement.

יתר על כן, זימון של הפעולות `toString` ו- `evaluate` באוסף מפעילים את הפעולות המתאימות במחלקת Literal כי בזמן ריצה, הפניות אלו מפנות לעצמים מסוג Literal .

התוצאות צריכות להיראות ככה:

```
14 = 14
0 = 0
-4 = -4
13.1415 = 13.1415
-0.1 = -0.1
```

Abstract class `Operator`

מחלקת Operator מיישמת את ממשק ExpressionElement, אבל לא ברמה קונקרטית, מה שאומר שאתה לא יכול ליצור מופע של אובייקט של מסוג Operator.

מחלקה מופשטת זו מייצגת אופרטורים אריתמטיים. אופרטורים אריתמטיים היא כל פעולה בחשבון שיכולה לפעול על אחד או יותר אופרנדים. כמה דוגמאות:

- add הוא אופרטור אשר מחבר שני אופרנדים. סימנו הוא "+".
- devision הוא אופרטור אשר מחלק שני אופרנדים. סימנו הוא "/".
- שורש ריבועי הוא אופרטור אשר ימצא את השורש הריבועי של האופרנד היחיד. סימנו הוא "√" (מכיוון שלא ניתן להשתמש בתו הזה בקוד ב-Java נשתמש ב- "sqrt" במקום).

כפי שניתן לראות, האופרטור עובד עם אופרנד אחד או שניים (או יותר). למחלקת Operator זה לא אכפת. היא פשוט מייצגת אופרטור מופשט.

- למרות שלא ניתן ליצור עצם ממחלקה זו יש לכתוב בנאי. הסיבה היא כי במחלקה יש תכונה המשותפת לכל סוגי האופרטורים, סמל האופרטור (`stringRepresentation`).
- יישום הפעולה evaluate עלול להיות מבלבל. הנה רמז: ניתן לקרוא לשיטה אחרת (כמו operate) גם אם היא לא יושמה עדיין. מספיק שיש הכרזה עליה. בזמן ריצה, מנגנון הפולימורפיזם יוודא שהפעולה הנכונה תזומן.

class `BinaryOperator`

כפי שראית בתיאור של מחלקת Operator, יכולים להיות מספר סוג אופרטורים. האופרטור יכול לדרוש אופרנד יחיד (אופרטור אונרי), שני אופרנדים (אופרטור בינארי) או יותר. אנחנו נשקף הכללה זו על ידי הוספת רמה נוספת של מחלקה מופשטת, מחלקה לכל סוג אופרטור היורשת ממחלקת Operator.

אופרטור בינארי דורש שני אופרנדים מסוג ExpressionElement (למשל האופרנד "+").

לאופרטור בינארי טיפוסי יש שני אופרנדים: ימני ושמאלי. יש והסדר שלהם חסר חשיבות (כמו במקרה של הוספה) או חשוב (כמו במקרה של חלוקה, חיסור, או חזקה).

<u>שים ♥</u>:

הפעולה toString מחזירה מחרוזת המכילה את שני האופרטורים וביניהם הסימן. כל אופרטור שאיננו Literal יש להקיף בסוגריים.

למען הבהירות, ניתן להוסיף סוגריים עבור כל Literal שהוא שלילי.

מחלקת `OpeartorSubtract`

במחלקה זו ניישם את הביטוי המתימטי חיסור ("מינוס", "-").

יש ליישם 2 שיטות:

- בנאי המגדיר את סמל האופרטור תוך שימוש בבנאי האב. הסימן מינוס הוא "-".
- הפעולה operate מבצעת את החישוב בפועל של left_operand - right_operand.

זה הזמן לבדוק שהכל עובד:

שנה את הקוד שלך, בפעולה main, ובדוק את התוצאות:

```
// our test arrayList
  ArrayList<ExpressionElement> elements = new ArrayList < >();
// add literals
  elements.add(new Literal(14));
  elements.add(new Literal(0));
  elements.add(new Literal(-4));
  elements.add(new Literal(13.1415));
  elements.add(new Literal(-0.1));

// create subtraction operators
  ExpressionElement sub0 = new Subtract(elements[2], elements[2]);
  ExpressionElement sub1 = new Subtract(sub0, elements[2]);
  ExpressionElement sub2 = new Subtract(elements[0], sub1);
  ExpressionElement sub3 = new Subtract(elements[3], sub1);
  ExpressionElement sub4 = new Subtract(sub3, elements[4]);

// add subtraction operators
  elements.add(sub0);
  elements.add(sub1);
  elements.add(sub2);
  elements.add(sub3);
  elements.add(sub4);

// test
  for (ExpressionElement element : elements)  // foreach
  {
      System.out.println(element + " = " + element.Evaluate());
  }
}
```

הפלט הנדרש:

```
14 = 14
0 = 0
-4 = -4
13.1415 = 13.1415
-0.1 = -0.1
-4--4 = 0
(-4--4)--4 = 4
14-((-4--4)--4) = 10
13.1415-((-4--4)--4) = 9.1415
(13.1415-((-4--4)--4))--0.1 = 9.2415
```

עכשיו ניתן ליישם את שאר המחלקות, כלומר את כל שאר 3 האופרטורים הבינאריים (חיבור, כפל וחילוק), את המחלקה המופשטת UnaryOperator וה- subclass שלה OperatorSquareRoot.

שים ♥:

חלק מהאופרנדים לא יכולים להיות שליליים או אפס.

בסיום מימוש אופרטור ← יש לבדוק אותו. לא מומלץ להשאיר את כל הבדיקות לסוף.

בתחילת המשימה נמצא קוד לחישוב הביטוי החשבוני **((15.4-3)*2)+(4*(1+5))**, הריצו אותו ובידקו.

הרחבה:

- בחלק זה נרחיב את ה-Class Diagram ונוסיף עוד מחלקות.
- מטרת סעיף זה היא לאפשר לך לקבל בעצמך את ההחלטות לגבי תכנון ירושה וקידוד.
- תצטרך לחשוב ולהחליט אילו שיטות יש ליישם במחלקות, אלו אמורות להיות מופשטות, מה רמת הגישה של כל שיטה במה יש להשתמש וכיוצ"ב.
- על ידי כתיבת המחלקות שלך תוכל להבין עד כמה קל ואלגנטי זה להוסיף פונקציונליות לתוכנה המעוצבת נכון.

מה צריך לעשות:

1. יש לכתוב עוד אופרטור בינארי. (למשל: השורש ה-n, log, חזקה, מודולו, וכו ').
2. יש ליישם עוד אופרטור אונארי (למשל: שורש שלישי, סימן חלופי, absolute, וכו ').
3. יש לכתוב את המחלקה המופשטת TernaryOperator, עבור אופרטור הדורש 3 אופרנדים.
4. כתוב 2 מחלקות אשר יורשות מ-TernaryOperator .
5. יש ליישם מחלקה אחרת אשר מיישמת את הממשק ExpressionElement ישירות. ניתן לחשוב על קבועים מתמטיים, כגון E, PI, וכו '.

להלן Class Diagram הכולל את כל המחלקות (המחלקות החדשות הן בצהוב):