Imperial College London – Department of Computing

MSc in Computing Science

536: Introduction to Java
Tutorial 1 — Expressions

In this exercise you will implement a group of Java classes representing simple arithmetic expressions. These classes will provide methods to print and evaluate expressions.

Aims

This tutorial will help you understand how to:

- Write simple Java classes
- Compile, debug and execute Java programs
- Implement inheritance in Java
- Use abstract classes in Java
- Inherit from the class `Object` and override its methods
- Work with strings in Java

Expressions

For the purposes of the exercise an *expression* is defined inductively as one of

- an integer, Z;
- an addition of expressions, $E_1 + E_2$;
- a multiplication of two expressions, $E_1 \times E_2$.

OO Implementation

Expressions can be implemented using the following class hierarchy.

- An `Expression` is an object with a value and a precedence. The class `Expression` will not possess enough information to be instantiated directly. It will be an abstract super class.

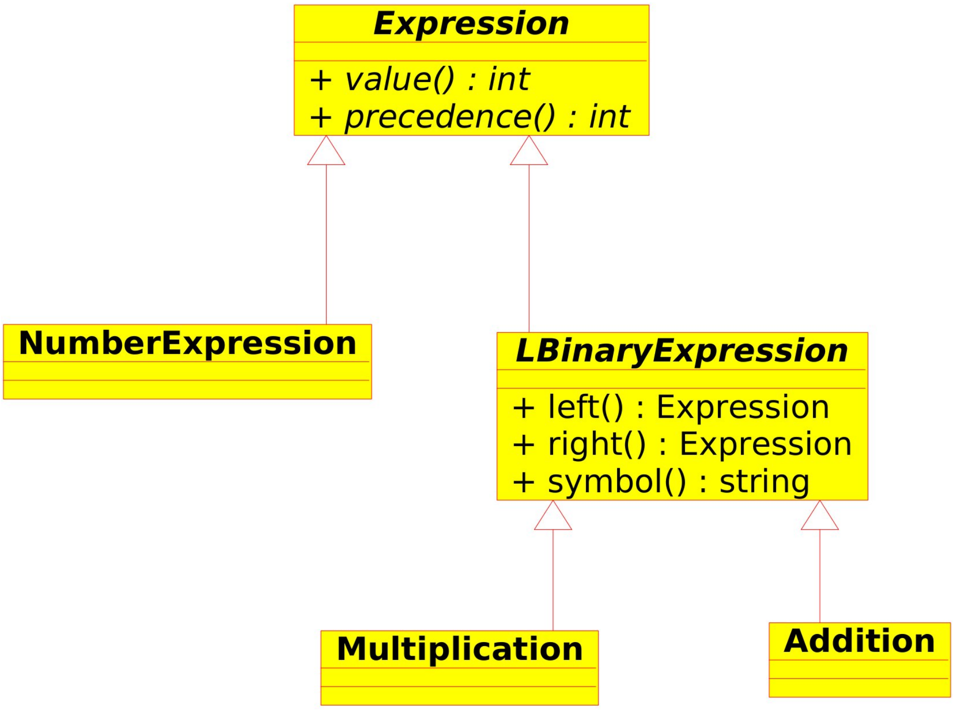

Figure 1: The `Expression` classes.

- A `NumberExpression` is a kind of `Expression` that encapsulates an integer value.
- An `LBinaryExpression` is a kind of `Expression` that represents a left-associative binary operation between two sub-expressions. It has a left operand, an operator symbol, and a right operand. An operation # is left-associative if $E_1 \# E_2 \# E_3$ means $(E_1 \# E_2) \# E_3$.
- `Addition` and `Multiplication` are two kinds of `LBinaryExpression`. The symbol of an `Addition` is " + ", and the symbol of a `Multiplication` is " * ".
- All `Expressions` can be evaluated. The evaluation result is the same as that of the arithmetic expression it represents.
- All `Expressions` can be turned to a string. This string is the text representation of the expression, using minimum brackets. Brackets are required only when the precedence of some operand expression is *lower* than the precedence of the binary expression of which it is a part. See the implementation of this feature in the C++ code below.

What To Do

You have been given the code for `Expression.java`. Write Java implementations of the other four classes described above. A C++ implementation is shown below, for reference.

- Save `Expression.java` in a suitable directory and add more source files in the same place. Remember each Java class `NameOfClass` should have its own file. This file must be named `NameOfClass.java` or your code will not compile.

- Start by implementing `NumberExpression`. This is a non-abstract class that should subclass `Expression`.
- Note that, in the Java implementation, all printed output is generated by the (non-abstract, static) `Expression.printExpression` method. This (implicitly) calls the expression's `toString()` method to obtain its string representation. The `toString()` method is inherited from `Object` and should be overridden in the different expression classes.
- `Expression` has been given a `main` method that creates three `NumberExpressions` and prints them out. Look at `main` to see how `NumberExpressions` are constructed. Once your `NumberExpression` is complete, you should be able to compile both files with

  ```
  javac Expression.java
  ```

  and then run `Expression.main` with

  ```
  java Expression
  ```

- Now implement the binary expression classes. Uncomment the rest of `Expression.main` to test as you go.

C++ Code

```
class Expression
{
  public:
    virtual int value() const = 0;
    virtual int prec() const = 0;
    virtual void print(ostream &) const = 0;
};
```

```
class NumberExpression : public Expression
{
  int _value;
  public:
    NumberExpression(int value) : _value(value) {
    }
    virtual int value() const {
      return _value;
    }
    virtual int prec() const {
      return _value >= 0 ? 10000 : 10;
    }
    virtual void print(ostream &o) const {
      o << _value;
    }
};
```

```
class LBinaryExpression : public Expression
{
  Expression &_left, &_right;
  protected:
    virtual int calculate(int v1, int v2) const = 0;
  public:
    LBinaryExpression(Expression &l, Expression &r) : _left(l), _right(r) {
    }
    Expression &left() const {
      return _left;
    }
    Expression &right() const {
      return _right;
    }
    virtual const char *symbol() const = 0;
    virtual void print(ostream &o) const {
      bool left_lower = _left.prec() < prec();
      if (left_lower) {
        o << "(";
      }
      _left.print(o);
      if (left_lower) {
        o << ")";
      }
      o << symbol();
      bool right_not_greater = _right.prec() <= prec();
      if (right_not_greater) {
        o << "(";
      }
      _right.print(o);
      if (right_not_greater) {
         o << ")";
      }
    }
    virtual int value() const {
      return calculate(_left.value(), _right.value());
    }
};

```

```cpp
class Addition : public LBinaryExpression
{
  protected:
    int calculate(int v1, int v2) const {
      return v1 + v2;
    }
  public:
    Addition(Expression &l, Expression &r) : LBinaryExpression(l, r) {
    }
    virtual int prec() const {
      return 10;
    }
    const char *symbol() const {
      return "+";
    }
};
```

```cpp
class Multiplication : public LBinaryExpression
{
  protected:
    int calculate(int v1, int v2) const { return v1 * v2; }
  public:
    Multiplication(Expression &l, Expression &r) : LBinaryExpression(l, r) {
    }
    virtual int prec() const {
      return 20;
    }
    virtual const char *symbol() const {
      return "*";
    }
};
```