Programming Language Concepts 2026

Assignment 1

15 Feb 2026, due 25 Feb 2026

---

1. Define an class to store information about (simple) arithmetic expressions, with the following fields and methods.

```
abstract public class Expr {
    Expr subexp1;
    Expr subexp2;

    public Expr(Expr e1, Expr e2) {
        subexp1 = e1;
        subexp2 = e2;
    }

    abstract String makeString();
    abstract public int evaluate();
}
```

   Create subclasses `NumExpr`, `AddExpr`, `SubExpr` and `MulExpr`, each of which implement the `makeString()` and `evaluate()` methods appropriately.

   - `NumExpr` stores an integer value
   - `makeString()` for `AddExpr` should create the string for `subexp1`, concatenate `"+"` to it, and concatenate `makeString` of `subexp2` to that. Similarly for `SubExpr` and `MulExpr`.
   - The `evaluate()` method has the obvious functionality.

---

2. Create a generic class `Stack` with the following functions.

```
public class Stack<T> {
    // fields
    // private class definitions, if necessary

    public Stack()           // constructor
    public void push(T elt)  // pushes element on top of the stack
    public T pop()           // remove the element on top of the stack
    public boolean isEmpty() // return true if stack is empty
    public T peek()          // return top element of the stack, without removing it
}
```

---

3. Create a class `Token` with the following fields and functions

```
public class Token {
    public enum TokType {Num, Add, Sub, Mul}
    private TokType toktype;
    private int tokval;

    public Token(TokType typ)          // constructor when token type is not Num

    public Token(TokType typ, int val) // constructor when type is Num

    public void hopOnStack(Stack<Expr> stack)
    // add the effect of a token to an expression stack.
    // The logic is the same as for a calculator of postfix expressions.
```

```
    // For token of type Num, just push the token onto the stack.
    // For other token types, pop the top two elements,
    //   create a new expression reflecting the outcome of this operation,
    //   and push this expression onto the stack.
}
```

---

4. Create a generic LinearList class with the following definitions.

```
public interface Iterator<T> {
    public boolean hasNext();
    public T next();
}

public class LinearList<T> {
    // fields
    // private class definitions as appropriate

    private class Iter implements Iterator<T> {
        // fields, constructor, code for hasNext() and next()
    }

    public LinearList()       // constructor
    public void add(T elt)    // add an element at end of list
    public Iter getIterator() // return a new iterator object
}
```

---

5. Create a class `Calculator` that uses all the classes defined above to implement a calculator for postfix expressions. The calculator takes as input a linear list of tokens, and uses a stack of expressions to compute the (infix) expression corresponding to the list of tokens.

```
public class Calculator {
    LinearList<Token> toklist;
    Stack<Expr> exprstack;

    public Calculator(LinearList<Token> toks) {
        toklist = toks;
        exprstack = new Stack<Expr>();
    }

    public Expr makeExpr()
    // for each token in toklist, make it hop on to the expression stack.
}
```

This whole module is tested using a file similar to `UseCalculator.java`, included as part of this assignment. On running the two commands:

```
javac UseCalculator.java
java UseCalculator
```

we should get the following output on screen:

```
((((22 + 5) * 3) + (4 * 3)) - 20)
The value of the expression is: 73
(((22 + (5 * 3)) + (4 * 3)) - 20)
The value of the expression is: 29
(((22 + 5) * 3) + (4 * (3 - 20)))
The value of the expression is: 13
```

General Instructions – Important

1. Do **not** add any package declarations
2. Do **not** submit a `main` method as part of your solution. Any solutions containing a `main` method will **not** be graded. We will test your solutions by supplying our own main methods in a separate file. Make sure that your implementation works when this happens.
3. Any submissions with compilation errors will **not** be graded.
4. Late submissions will **not** be graded.
5. All the files comprising your solution code should be in one folder (no subfolders) named `un`, where `un` is your CMI username. Compress the folder as a zip file named `un`.zip and submit it via Moodle. For example, username `madhavan` should put all files in a folder named `madhavan` and submit the compressed file `madhavan.zip`.
