題目內容

題目描述

標題：構建一個支持四則運算的抽象語法樹（AST）

背景：你將設計一個表達式計算系統，該系統支持基本的四則運算（加法、減法、乘法和除法），並用面向對象的方式來設計程式碼結構。通過使用抽象語法樹（AST），你將能夠對表達式進行評估並轉換為中序和後序表示法。

要求：

1. **類別設計**：使用 OOP 來組織程式碼，確保程式包含四大特性：
   - **封裝**：隱藏類的內部細節，只提供必要的介面給外界使用。
   - **繼承**：使用繼承來實現不同類型表達式的共同特性。
   - **多態**：使用多態使程式能根據對象的實際類型執行對應的方法。
   - **抽象**：使用抽象類別或介面來定義共同行為。
2. **動態調度**：設計的程式必須支持動態調度，以在運行時根據對象的實際類型執行對應的方法。
3. **抽象語法樹（AST）**：構建一個 AST，並實作節點來表示不同類型的運算（加、減、乘、除）。這些節點應能夠被計算並轉換為中序（infix）和後序（postfix）表示法。
4. **應用情境**：模擬一個簡單計算器，能夠對表達式進行評估並轉換為中序和後序表示法。

題目要求

1. **建立 `Exp` 抽象類別**：
   - `Exp.java` 為所有表達式的基礎抽象類別，定義以下方法：

```
public abstract class Exp {
    public abstract int eval();  // 計算表達式的值
    public abstract String toInfix();  // 轉換為中序表示法
    public abstract String toPostfix();  // 轉換為後序表示
```

```
法
}
```

2. **實作不同的表達式類別：**

- `IntLiteral.java`：代表整數字面值。

```
public class IntLiteral extends Exp {
    private int value;

    public IntLiteral(int value) {
        this.value = value;
    }

    @Override
    public int eval() {
        return value;
    }

    @Override
    public String toInfix() {
        return String.valueOf(value);
    }

    @Override
    public String toPostfix() {
        return String.valueOf(value);
    }
}
```

- `AddExp.java`、`SubExp.java`、`MulExp.java`、`DivExp.java`：分別表示加法、減法、乘法和除法節點。

```
public class AddExp extends Exp {
    private Exp left, right;
```

```
    public AddExp(Exp left, Exp right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public int eval() {
        return left.eval() + right.eval();
    }

    @Override
    public String toInfix() {
        return "(" + left.toInfix() + " + " + right.toI
nfix() + ")";
    }

    @Override
    public String toPostfix() {
        return left.toPostfix() + " " + right.toPostfix
() + " +";
    }
}

public class SubExp extends Exp {
    private Exp left, right;

    public SubExp(Exp left, Exp right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public int eval() {
        return left.eval() - right.eval();
    }
```

```
    @Override
    public String toInfix() {
        return "(" + left.toInfix() + " - " + right.toI
nfix() + ")";
    }

    @Override
    public String toPostfix() {
        return left.toPostfix() + " " + right.toPostfix
() + " -";
    }
}

   public class MulExp extends Exp {
       private Exp left, right;

       public MulExp(Exp left, Exp right) {
           this.left = left;
           this.right = right;
       }

       @Override
       public int eval() {
           return left.eval() * right.eval();
       }

       @Override
       public String toInfix() {
           return "(" + left.toInfix() + " * " + right.
toInfix() + ")";
       }

       @Override
       public String toPostfix() {
           return left.toPostfix() + " " + right.toPost
fix() + " *";
```

```
        }
    }

    public class DivExp extends Exp {
        private Exp left, right;

        public DivExp(Exp left, Exp right) {
            this.left = left;
            this.right = right;
        }

        @Override
        public int eval() {
            return left.eval() / right.eval(); // 注意：需考慮除數為 0 的情況，在實際應用中可加入錯誤處理
        }

        @Override
        public String toInfix() {
            return "(" + left.toInfix() + " / " + right.toInfix() + ")";
        }

        @Override
        public String toPostfix() {
            return left.toPostfix() + " " + right.toPostfix() + " /";
        }
    }
```

3. **測試程式：**
   - 建立一組測試資料，驗證 `eval()`、`toInfix()` 和 `toPostfix()` 方法。

```
public class Main {
    public static void main(String[] args) {
        Exp[] expressions = {
            new IntLiteral(5),
            new AddExp(new IntLiteral(2), new IntLitera
l(3)),
            new SubExp(new IntLiteral(10), new IntLiter
al(4)),
            new MulExp(new IntLiteral(2), new IntLitera
l(6)),
            new DivExp(new IntLiteral(8), new IntLitera
l(2)),
            new AddExp(new MulExp(new IntLiteral(2), ne
w IntLiteral(3)), new IntLiteral(4))
        };

        for (Exp exp : expressions) {
            System.out.println("Evaluation: " + exp.eva
l());
            System.out.println("Infix: " + exp.toInfix
());
            System.out.println("Postfix: " + exp.toPost
fix());
            System.out.println();
        }
    }
}
```

測試資料與結果

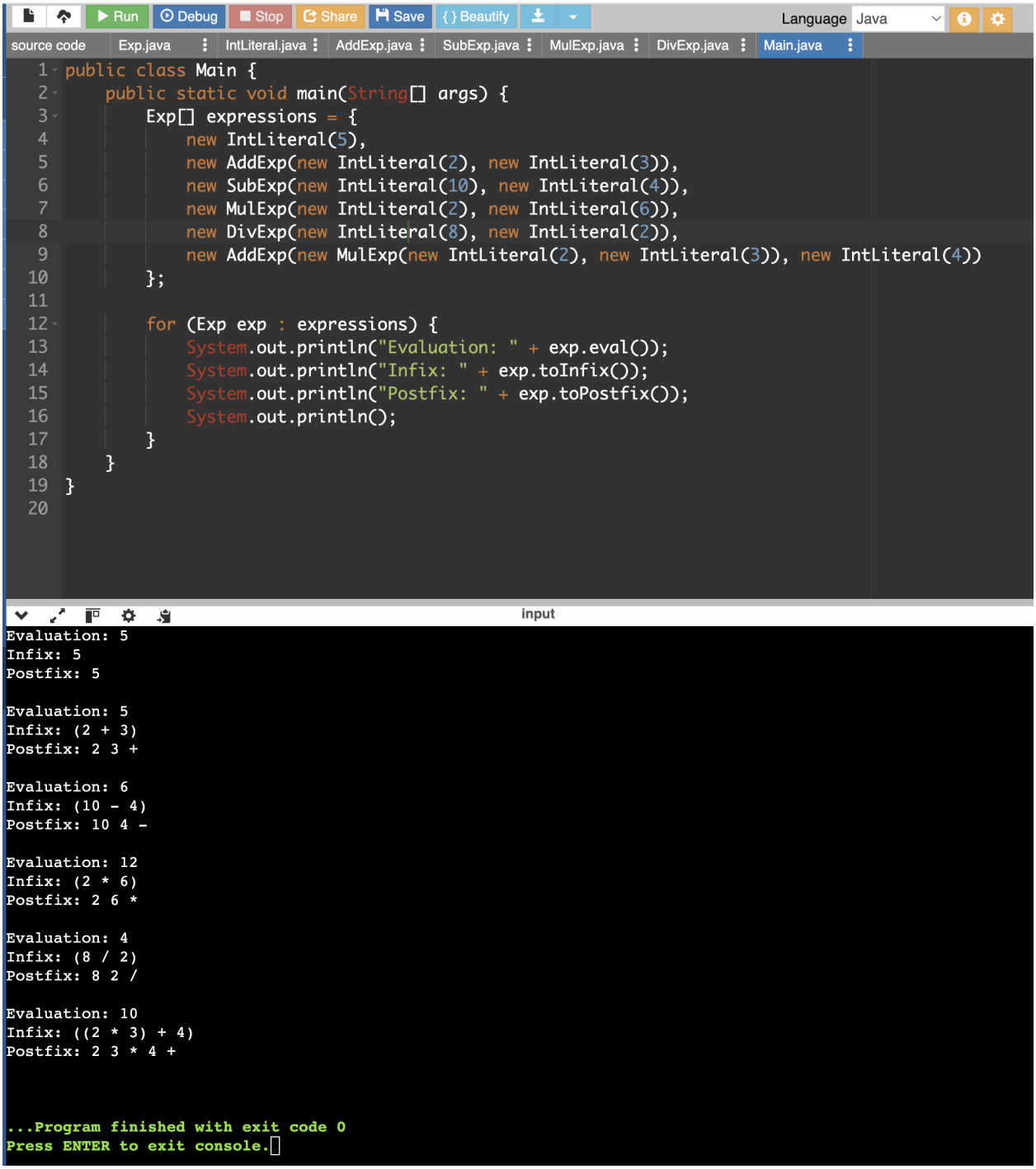
測試資料：

- 表達式：`5`、`2 + 3`、`10 - 4`、`2 * 6`、`8 / 2`、`(2 * 3) + 4`

預期輸出：

```
Evaluation: 5
Infix: 5
Postfix: 5

Evaluation: 5
Infix: (2 + 3)
Postfix: 2 3 +

Evaluation: 6
Infix: (10 - 4)
Postfix: 10 4 -

Evaluation: 12
Infix: (2 * 6)
Postfix: 2 6 *

Evaluation: 4
Infix: (8 / 2)
Postfix: 8 2 /

Evaluation: 10
Infix: ((2 * 3) + 4)
Postfix: 2 3 * 4 +
```

實際輸出



與 LLM 的對話紀錄擷圖

請幫我生成一道程式題目，該題目需要滿足以下程式語言規範：
1. Object-Oriented Programming
 - 描述：利用 "物件" 來組織程式碼結構，並包含 OOP 的四大特性：封裝（Encapsulation）、繼承（Inheritance）、多態（Polymorphism）、抽象（Abstraction）。
2. Dynamic Dispatch
 - 描述：要求動態調度，以在運行時根據對象的實際類型執行對應的方法。
3. Abstract Syntax Tree (AST)
 - 描述：需要構建一個 AST，並實作該 AST 的節點（例如加法節點、乘法節點），以支援對表達式的計算和轉換。
4. Real-World Application Scenario
 - 描述：需模擬實際應用情境，例如計算器、編譯器、或語法解析器，並要求實作表達式的計算與轉換功能。

範例：
請設計一個使用類別來表示加法和乘法表達式的程式，並要求實作一個 AST 來計算並轉換這些表達式。此外，請提供測試案例來驗證程式是否正確地計算和轉換不同的表達式。

請生成完整的程式碼範例，包括介面和類別的實作，以及測試資料。並提供相應的解釋來說明程式碼中的各個部分是如何對應到上述規範的。

題目描述

標題：構建一個支持四則運算的抽象語法樹（AST）

背景：你將設計一個表達式計算系統，該系統支持基本的四則運算（加法、減法、乘法和除法），並用面向對象的方式來設計程式碼結構。通過使用抽象語法樹（AST），你將能夠對表達式進行評估並轉換為中序和後序表示法。

要求：

1. **類別設計**：使用 OOP 來組織程式碼，確保程式包含四大特性：
   - **封裝**：隱藏類的內部細節，只提供必要的介面給外界使用。
   - **繼承**：使用繼承來實現不同類型表達式的共同特性。
   - **多態**：使用多態使程式能根據對象的實際類型執行對應的方法。
   - **抽象**：使用抽象類別或介面來定義共同行為。
2. **動態調度**：設計的程式必須支持動態調度，以在運行時根據對象的實際類型執行對應的方法。
3. **抽象語法樹（AST）**：構建一個 AST，並實作節點來表示不同類型的運算（加、減、乘、除）。這些節點應能夠被計算並轉換為中序（infix）和後序（postfix）表示法。
4. **應用情境**：模擬一個簡單計算器，能夠對表達式進行評估並轉換為中序和後序表示法。

題目要求

1. **建立 `Exp` 抽象類別**：
   - `Exp` 為所有表達式的基礎抽象類別，定義以下方法：

```java
public abstract class Exp {
    public abstract int eval();  // 計算表達式的值
    public abstract String toInfix();  // 轉換為中序表示法
    public abstract String toPostfix();  // 轉換為後序表示法
}
```

2. **實作不同的表達式類別**：
   - `IntLiteral`：代表整數字面值。

```java
public class IntLiteral extends Exp {
    private int value;

    public IntLiteral(int value) {
        this.value = value;
    }

    @Override
    public int eval() {
        return value;
    }

    @Override
    public String toInfix() {
        return String.valueOf(value);
    }

    @Override
    public String toPostfix() {
        return String.valueOf(value);
    }
}
```

- `AddExp`、`SubExp`、`MulExp`、`DivExp`：分別表示加法、減法、乘法和除法節點。

```java
public class AddExp extends Exp {
    private Exp left, right;

    public AddExp(Exp left, Exp right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public int eval() {
        return left.eval() + right.eval();
    }

    @Override
    public String toInfix() {
        return "(" + left.toInfix() + " + " + right.toInfix() + ")";
    }

    @Override
    public String toPostfix() {
        return left.toPostfix() + " " + right.toPostfix() + " +";
    }
}

public class SubExp extends Exp {
    private Exp left, right;

    public SubExp(Exp left, Exp right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public int eval() {
        return left.eval() - right.eval();
    }

    @Override
    public String toInfix() {
        return "(" + left.toInfix() + " - " + right.toInfix() + ")";
    }

    @Override
    public String toPostfix() {
        return left.toPostfix() + " " + right.toPostfix() + " -";
    }
}

// 同樣地，你可以為 `MulExp` 和 `DivExp` 類別編寫類似的程式碼。
```

3. **測試程式**：
   - **建立一組測試資料，驗證** `eval()`、`toInfix()` 和 `toPostfix()` 方法。

```java
public class Main {
    public static void main(String[] args) {
        Exp[] expressions = {
            new IntLiteral(5),
            new AddExp(new IntLiteral(2), new IntLiteral(3)),
            new SubExp(new IntLiteral(10), new IntLiteral(4)),
            new MulExp(new IntLiteral(2), new IntLiteral(6)),
            new DivExp(new IntLiteral(8), new IntLiteral(2)),
            new AddExp(new MulExp(new IntLiteral(2), new IntLiteral(3)), ne
        };

        for (Exp exp : expressions) {
            System.out.println("Evaluation: " + exp.eval());
            System.out.println("Infix: " + exp.toInfix());
            System.out.println("Postfix: " + exp.toPostfix());
            System.out.println();
        }
    }
}
```

測試資料與結果

測試資料：

- 表達式：`5`、`2 + 3`、`10 - 4`、`2 * 6`、`8 / 2`、`(2 * 3) + 4`

預期輸出：

```makefile
Evaluation: 5
Infix: 5
Postfix: 5

Evaluation: 5
Infix: (2 + 3)
Postfix: 2 3 +

Evaluation: 6
Infix: (10 - 4)
Postfix: 10 4 -

Evaluation: 12
Infix: (2 * 6)
Postfix: 2 6 *

Evaluation: 4
Infix: (8 / 2)
Postfix: 8 2 /

Evaluation: 10
Infix: ((2 * 3) + 4)
Postfix: 2 3 * 4 +
```

說明

1. **封裝（Encapsulation）**：所有類別的成員變數（如 `value`、`left`、`right`）都是私有的，並且通過公開的方法來訪問，實現了封裝。
2. **繼承（Inheritance）**：所有表達式類別都繼承自抽象類別 `Exp`，這使得類別之間可以共享相同的方法簽名並實作不同的行為。
3. **多態（Polymorphism）**：通過對 `Exp` 類型的變數調用 `eval()`、`toInfix()` 和 `toPostfix()`，可以根據不同的實際類型執行對應的方法實作。
4. **抽象（Abstraction）**：`Exp` 類別作為一個抽象層，對外界隱藏了具體的表達式運算實作細節，只提供了共用的介面方法。

這道題目完整體現了面向對象編程的特性、AST 的設計與構建以及真實世界應用情境，讓學生能夠透過程式實作加深對這些概念的理解。

題目解釋

1. Programming Paradigms

這道題目主要基於 **面向對象編程（OOP）** 的範式來設計。透過建立一個抽象語法樹（AST）的結構，這道題目展現了 OOP 中的四大核心特性：

- **封裝（Encapsulation）**：每個表達式類別如 `AddExp`、`SubExp`、`MulExp` 和 `DivExp` 都將自己的計算邏輯和表示法細節封裝在內部，外部只能通過公開的 `eval()` 和 `toInfix()`、`toPostfix()` 方法來訪問這些細節。
- **繼承（Inheritance）**：所有表達式類別都繼承自抽象類別 `Exp`，這樣可以共用父類別的方法定義，並根據需要進行具體實現。
- **多態（Polymorphism）**：由於多態性，當程式運行時，無論變數的靜態類型為 `Exp`，動態調用時會根據實際的對象類型執行對應的 `eval()` 或 `toInfix()` 等方法，這也體現了 **動態調度（Dynamic Dispatch）** 的特性。
- **抽象（Abstraction）**：抽象類別 `Exp` 定義了所有表達式必須實作的 `eval()`、`toInfix()` 和 `toPostfix()` 方法，並讓具體的子類負責各自的實現。

這種設計使得程式可以靈活擴展，未來若要添加新的表達式類型（如取餘運算 `%`），只需繼承 `Exp` 類並實作相關方法即可。

2. Design Patterns

題目應用了 **策略模式（Strategy Pattern）**，每個表達式類別（如 `AddExp`、`SubExp`）實際上是對不同運算邏輯的策略實現。這種設計模式允許我們根據不同情況，選擇不同的策略來計算或轉換表達式。具體來說，程式可以針對每一個運算符號節點，使用對應的 `eval()` 和 `toInfix()` 方法來處理計算與轉換工作。

3. Programming Language Support

Java 是一個天然支持 OOP 特性的語言，它提供了語法結構來實現繼承、多態、封裝等特性。此外，Java 支援 **動態調度**，即使我們在程式碼中宣告的變數類型是父類 `Exp`，在運行時會自動調用子類的具體實現。

例如，在運行 `Exp exp = new AddExp(new IntLiteral(1), new IntLiteral(2));` 時，當我們調用 `exp.eval()` 時，實際會調用 `AddExp` 類中的 `eval()` 方法，而不是 `Exp` 父類中的抽象方法。

4. Other Related Issues

- **擴展性**：這種 AST 設計模式具備高度的擴展性。如果將來需要添加新的運算類型，如 **模運算** 或 **指數運算**，只需定義一個新類繼承 `Exp` 並實作相應的 `eval()`、`toInfix()` 和 `toPostfix()` 方法即可。
- **可讀性與模組化**：每個運算符類別都封裝了自己的邏輯，程式碼具備良好的可讀性，並且模組化程度高，不同類別的運算邏輯不會互相干擾。
- **錯誤處理**：在實作除法運算 `DivExp` 時，應注意處理除數為零的情況。在實際應用中，可以在 `eval()` 方法中加入對除數為零的檢查，並拋出適當的異常。

這道題目涵蓋了 OOP 的核心概念，並運用抽象語法樹（AST）的設計來實現表達式的計算與轉換，使其具備擴展性和實際應用價值。通過這個題目，學生可以加深對 OOP 四大特性、動態調度以及 AST 概念的理解，並將其應用到程式設計中。

主要建構給 LLM 的 prompt

```
請幫我生成一道程式題目，該題目需要滿足以下程式語言規範：
1. Object-Oriented Programming
    - 描述：利用 "物件" 來組織程式碼結構，並包含 OOP 的四大特性：封裝（Encapsulation）、繼承（Inheritance）、多態（Polymorphism）、抽象（Abstraction）。
2. Dynamic Dispatch
    - 描述：要求動態調度，以在運行時根據對象的實際類型執行對應的方法。
3. Abstract Syntax Tree (AST)
    - 描述：需要構建一個 AST，並實作該 AST 的節點（例如加法節點、乘法節點），以支援對表達式的計算和轉換。
4. Real-World Application Scenario
    - 描述：需模擬實際應用情境，例如計算器、編譯器、或語法解析器，並要求實作表達式的計算與轉換功能。

範例：
請設計一個使用類別來表示加法和乘法表達式的程式，並要求實作一個 AST 來計算並轉換這些表達式。此外，請提供測試案例來驗證程式是否正確地計算和轉換不同的表達式。
```

請生成完整的程式碼範例，包括介面和類別的實作，以及測試資料。並提供相應的解釋來說明程式碼中的各個部分是如何對應到上述規範的。