Code Editor (auf Basis eines Formel-Editors)

Im gegebenen Projekt *CodeEditor_Base* wurde ein Formeleditor implementiert, welcher in der Lage ist, in mathematische Formeln farbliche Hervorhebungen vorzunehmen. Betrachten Sie folgende Anwendung des Editors und nebenstehende Ausgabe:

Testprogramm	Konsolenausgabe

`public class Main {`
`  public static void main(String[] args) {`
`  FormulaEditor editor = new FormulaEditor();`
`  editor.appendLine("1: a^2 + b^2 = c^2");`
`  editor.appendLine("2: sin(x) + cos(x) = 1");`
`  editor.appendLine("3: y = 2*a*b*cos(gamma)");`
`  editor.appendLine("IV: A = r^2 * pi");`
`  editor.appendLine("V: f(x) = 1.245 * x^2 + 3.45");`
`  editor.print();`
`  }`
`}`
	`1: a^2 + b^2 = c^2`
	`2: sin(x) + cos(x) = 1`
	`3: y = 2*a*b*cos(gamma)`
	`IV: A = r^2 * pi`
	`V: f(x) = 1.245 * x^2 + 3.45`

Beachten Sie:

- Der Formeleditor verfügt über eine Methode *appendLine()* zum Hinzufügen von Formeln.
- Die *print()*-Methode druckt die Formeln in der Konsole aus, wobei unterschiedliche Formelelemente farblich unterschiedlich hervorgehoben werden:
  - Die Zeilennummer (jeglicher Text bis zum ersten ":") wird *rot* gefärbt.
  - Texte (Buchstabenfolgen) werden unterschieden in Funktionen (*grün* gefärbt) andere Text (z. B. Variable, Konstante, … - *lila* gefärbt).
  - Zahlen beginnen mit einer Ziffer, gefolgt von einer Ziffer oder einem Dezimalpunkt. Diese werden *blau* gefärbt.
  - Andere Formelteile werden ungefärbt übernommen.

Analysieren Sie den gegebenen Programmcode genau und beachten Sie dabei folgende Punkte:

- Die Konsolenausgabe wird gefärbt, indem spezielle ANSI-Strings eingefügt werden. In der Klasse **ConsoleColor** sind eine Reihe von String-Konstanten für die gängigen Farben definiert, die Sie verwenden können. Die Konstante *ANSI_RESET* setzt die Farbe wieder auf die Standardfarbe zurück. Anwendungsbeispiel, um ein Wort rot zu färben:

```
String s = "Eine " + ConsoleColor.ANSI_RED + "rote" + ConsoleColor.ANSI_RESET + " Ampel";
System.out.println(s);
```

```
Eine rote Ampel

Process finished with exit code 0
```

- Der Formel-Editor verfügt über einfache Methoden zum Editieren und Drucken der Formeln (*appendLine(), deleteAll(), print()*).
- Das eigentliche Formatieren erfolgt durch die Methode *format()* und eine Reihe von Hilfsmethoden, welche einzelne Formelteile einlesen und teilweise auch formatieren. Dabei wurde folgender gängiger Ansatz beim sequenziellen Lesen und Verarbeiten von Texten gewählt:
  - Es gibt eine Variable *text*, welche den gesamten Editortext unformatiert speichert. Dieser Text wird durch das Formatieren auch nicht verändert.
  - Die Verarbeitung beim Formatieren des Formeltextes erfolgt zeichenweise. Dazu wird eine globale Variable *pos* verwendet, um die Position des nächsten zu lesenden Zeichens zu markieren. Weiters enthält die Variable *ch* jenes Zeichen, das als nächstes zu verarbeiten ist.
  - Darüber hinaus gibt es eine Konstante *EOF*, die zum Erkennen des Textendes verwendet wird, sowie eine Konstante *FUNCTION*, die eine Liste aller bekannten Funktionen enthält.
  - Die Hilfsmethode *nextChar()* holt das nächste Zeichen an der Position *pos* und speichert dieses in *ch* ab, bzw. liefert *EOF*, wenn das Ende des Formeltexts erreicht wurde. Pos wird dann um 1 erhöht.
  - Die *format()*-Methode beginnt am Anfang des Formeltextes und liest Zeichen für Zeichen, bis das Ende erreicht wurde. Nach jedem Zeichen wird entschieden:

- Am Beginn einer Zeile wird immer eine Zeilennummer erwartet. Also werden mit der Hilfsmethode *readAndFormatFormulaNumber()* alle Zeichen bis zum nächsten ":" gelesen und formatiert.
- Ist man nicht am Beginn einer Zeile, so wird differenziert:
  - Ziffern leiten eine Zahl ein, also wird mit der Methode *readNumber()* eine Ziffernsequenz gelesen und formatiert.
  - Buchstaben leiten einen Text ein. Mit der Methode *readAndFormatWord()* werden Buchstabenfolgen eingelesen. Dabei erfolgt in der Methode auch die unterschiedliche Formatierung von Funktionen bzw. anderen Texten.
  - Ein Zeilenumbruch wird entsprechend verarbeitet.
  - Alle sonstigen Zeichen werden unformatiert in das Ergebnis übernommen.

Aufgabenstellung:

Aus dem Formel-Editor soll ein allgemeiner Code-Editor entwickelt, sodass dieser für verschiedenste Arten von Texten (Formeln, HTML-Quellcode, Java-Quellcode, etc.) ein jeweils geeignetes Syntax-Highlighting vornehmen kann. Dadurch soll beispielhaft folgende Anwendung möglich sein:

Testprogramm

```
public class Main {
  public static void main(String[] args) {
    CodeEditor editor = new CodeEditor(CodeType.HTML);
    editor.appendLine("<html lang=\"de\">");
    editor.appendLine("   <head>");
    editor.appendLine("      <meta charset=\"UTF-8\">");
    editor.appendLine("      <meta name=\"viewport\">");
    editor.appendLine("      <meta http-equiv=\"X-UA-Compatible\">");
    editor.appendLine("      <title>Meine kleine HTML-Seite</title>");
    editor.appendLine("   </head>");
    editor.appendLine("   <body>");
    editor.appendLine("      <h1>Willkommen auf meiner Seite</h1>");
    editor.appendLine("      <p>Hier steht ein bisschen Text.</p>");
    editor.appendLine("   </body>");
    editor.appendLine("</html>");
    editor.print();

    editor.deleteAll();
    editor.setCodeType(CodeType.JAVA);
    editor.appendLine("public class TestProgramm {");
    editor.appendLine("  public static void main(String[] args) {");
    editor.appendLine("    // A loop that iterates from 1 to 10");
    editor.appendLine("    for (int i = 1; i <= 10; i++) {");
    editor.appendLine("      // Check if the number is even or odd");
    editor.appendLine("      if (i % 2 == 0) {");
    editor.appendLine("        System.out.println(i + \" is an even number.\");");
    editor.appendLine("      } else {");
    editor.appendLine("        System.out.println(i + \" ist an odd number.\");");
    editor.appendLine("      }");
    editor.appendLine("    }");
    editor.appendLine("  }");
    editor.appendLine("}");
    editor.print();
  }
}
```

Konsolenausgabe

```
<html lang="de">
   <head>
      <meta charset="UTF-8">
      <meta name="viewport">
      <meta http-equiv="X-UA-Compatible">
      <title>Meine kleine HTML-Seite</title>
   </head>
   <body>
      <h1>Willkommen auf meiner Seite</h1>
      <p>Hier steht ein bisschen Text.</p>
   </body>
</html>

public class TestProgramm {
  public static void main(String[] args) {
    // A loop that iterates from 1 to 10
    for (int i = 1; i <= 10; i++) {
      // Check if the number is even or odd
      if (i % 2 == 0) {
        System.out.println(i + " is an even number.");
      } else {
        System.out.println(i + " ist an odd number.");
      }
    }
  }
}
```

Verwenden Sie das **Strategie-Pattern** damit der Editor unterschiedliche Formatierungsstrategien nutzen kann. Orientieren Sie sich dabei an folgenden Vorgaben:

- Benennen Sie die **FormulaEditor**-Klasse in **CodeEditor** um.
- Die Enumeration **CodeType** zählt die unterstützten Codearten auf: FORMULA, HTML, JAVA, DEFAULT.
- Die **CodeEditor** wie folgt geändert werden:
  - Beim Konstruktor *CodeEditor(CodeType)* wird die Art des Codes gesetzt.
  - Ein Ändern des CodeTyps ist zur Laufzeit möglich mittels *setCodeType(CodeType)*.
- Hinweise zum Strategie-Pattern für die unterschiedlichen Formatierungsstrategien:
  - Erstellen Sie ein Interface ***CodeFormatter***, das die Methode *String format(String)* bereitstellt.
  - Erstellen Sie eine Klasse ***FormulaFormatter***, welche das Formatieren von Formeln übernimmt. Verschieben Sie dazu jegliche Funktionalität (Methoden, Felder) betreffend das Formatieren in diese Klasse und testen Sie dann, ob das Formatieren wie bisher funktioniert.
  - Erstellen Sie weitere Formatter für HTML, Java und eine Default-Strategie (ohne jegliche Formatierung). Jeder Formatter soll unterschiedliche spezifische Sprachelemente, z. B. HTML-Attribute, String-Literale, Java-Keywords, Java-Kommentare, etc. mit unterschiedlichen Farben darstellen (siehe Beispielausgaben oben).
- Sie können die *StringBuilder*-Klasse verwenden, um Manipulationen an Strings effizient abzubilden.

Detaillierte Beschreibung, wie z. B. die HTML-Formatierung umgesetzt werden kann:

- Der grundsätzliche Verarbeitungsprozess (sequentielles Lesen, etc.) kann vom *FormulaFormatter* übernommen werden.
- Erstellen Sie eine Hilfsmethode ***boolean isHtmlLetter(char)***, welches prüft, ob ein gegebenes Zeichen gültiger Teil eines HTML-Tags bzw. Attributs ist (Buchstabe a-z, A-Z, '-').
- Erstellen Sie eine Hilfsmethode ***String readWord()***, welches ein gesamtes Wort (HTML-Letter-Folge) einliest und zurückgibt.
    - Dabei ist davon auszugehen, dass *ch* am Beginn eines HTML-Worts steht. Der Rückgabewert ist das gesamte HTML-Wort.
- Erstellen Sie eine Hilfsmethode ***String readStringLiteral()***, welches einen String-Literal (z. B. "Hallo Welt") inkl. der einschließenden Hochkomma einliest und zurückgibt.
    - Dabei ist davon auszugehen, dass *ch* am öffnenden " des String-Literals steht. Es müssen alle folgenden Zeichen bis zum schließenden " eingelesen und zurückgegeben werden.
- Erstellen Sie eine Hilfsmethode ***String readAndFormatTag()***, welches eine Tag einliest, formatiert und das formatierte Tag zurückgibt.
    - Dabei ist davon auszugehen, dass *ch* am öffnenden "<" steht. Es müssen alle folgenden Zeichen bis zum schließenden ">" gelesen und verarbeitet werden.
    - Verwenden Sie zunächst *readWord()*, um das erste Wort (Tag) einzulesen, und übernehmen Sie dieses formatiert in den Ergebnis-String.
    - Solange Sie nicht beim schließenden ">" angekommen sind:
        - Wenn ein Buchstabe folgt, so ist ein Html-Attribut gegeben. Lesen Sie dieses mit *readWord()* und übernehmen Sie das formatierte Attribut in den Ergebnisstring.
        - Wenn ein " gelesen wird, so folgt ein String-Literal. Lesen Sie diesen mit *readStringLiteral()* und übernehmen Sie diesen formatiert in den Ergebnisstring.
        - Alle anderen Zeichen (Leerzeichen, =) können Sie unformatiert in den Ergebnisstring übernehmen.
    - Der Rückgabewert ist das gesamte formatierte Tag.
    - Überlegen Sie, wie schließende Tags berücksichtigen werden können!