IFM 2.1 Programmiermethoden, S20

B04 'Lexer': RegExp, Annotationen, Reflection

Carsten Gips, FH Bielefeld

Praktikum: 03.07.2020

ILIAS: 02.07.2020

1 Aufgabe Praktikum

In dieser Aufgabe soll schrittweise ein Programm zum Syntax-Highlighting für (einfachen) Java-Quelltext entstehen. Sie kennen dies von Entwicklungsumgebungen wie beispielsweise Eclipse oder auch von Editoren.

Lesen Sie dieses Blatt bitte bis zum Ende durch, bevor Sie mit der Bearbeitung beginnen.

1.1 Reguläre Ausdrücke

Im ersten Schritt muss der Java-Quelltext in passende Token aufgeteilt werden.[1] Beispiel: Der Java-Quellcode

```
// total irre: public void foo()
public void foo(String s) {}
```

enthält die Token `// total irre: public void foo()`, `public`, `void` und `foo(String s) {}`.

Ihr Lexer erhält den zu untersuchenden Quellcode als String über den Aufruf von `Lexer#tokenize`. Er muss dann mit der Methode `Lexer#testTokens` schrittweise prüfen, ob ein bestimmtes Token am Anfang des Strings vorliegt. Falls dies so ist, wird die gefundene Zeichenkette aus dem String entfernt und ein entsprechendes Token-Objekt in der Ergebnisliste gespeichert. Dies wird so lange wiederholt, bis der String komplett abgearbeitet ist. Anschließend wird die Ergebnisliste der erkannten Token zurückgeliefert.

Ihr Lexer soll folgende Token unterscheiden und erkennen:

1. Klasse `Comment`: Einzeilige Kommentare (beginnend mit `//` bis zum Zeilenende)
2. Klasse `MultilineComment`: Mehrzeilige Kommentare (alles zwischen `/*` und `*/`)
3. Klasse `StringContent`: Strings (alles zwischen `"` und `"` bzw. zwischen `'` und `'`)
4. Klasse `KeyWord`: Ausgewählte Java-Schlüsselwörter (Implementieren Sie mindestens 5 Java-Schlüsselwörter.)
5. Klasse `NewLine`: Zur Darstellung von Zeilenumbrüchen

   Diese Tokenklasse hat eine höhere Priorität als das "Catch-All"-Token, muss also vor diesem geprüft werden. Zeilenumbrüche innerhalb von mehrzeiligen Kommentaren gehören aber nach wie vor zum Kommentartext!
6. Klasse `CatchAll`: Sonstigen Text, d.h. alle restlichen Zeichen zwischen zwei beliebigen Token

   Dabei wird **zeichenweise** gearbeitet, d.h. pro Zeichen wird ein Objekt der Klasse `CatchAll` angelegt.

Für jedes Token gibt es eine entsprechende Token-Klasse mit dem passenden regulären Ausdruck, der die jeweilige Zeichenkette erkennt.[2] Die Tokenklassen bilden eine Klassenhierarchie mit gemeinsamer abstrakter Basisklasse.

Halten Sie dabei die Abbildung 1 dargestellten Schnittstellen bzw. Klassenhierarchien ein. Sie können eigene Attribute und Methoden ergänzen.

Die Methode `Token#match` wird in der abstrakten Basisklasse implementiert und kann nicht überschrieben werden (`final`). Sie erzeugt aus dem übergebenen String und dem im Feld `pattern` in den Unterklassen hinterlegten regulären Ausdruck ein Matcher-Objekt, welches sie im Feld `matcher` für die spätere Verwendung speichert. Bei erfolgreichem Match liefert sie unter Benutzung der Methode `Token#getToken` eine neue Instanz der konkreten Tokenklasse zurück, sonst `null`.[3]

Die Methode `Token#getToken` ist abstrakt und wird erst in den konkreten Token-Klassen implementiert. Die konkreten Token-Klassen belegen dann auch das Feld `pattern` mit dem jeweils nötigen Pattern. Die Methode `Token#getToken`

[1] Ein solches Programm wird auch **Lexer** genannt und stellt die erste Phase eines Parsers dar. Der nächste Schritt wäre dann der Aufbau eines Syntaxbaumes und das Prüfen der korrekten Syntax. Dies ist zwar noch viel spannender, geht aber leider deutlich über Ihre aktuellen Kenntnisse in Theoretischer Informatik hinaus. Darüber sprechen wir im Master in "Compilerbau" :-)

[2] Dies ist im Compilerbau eher unüblich, dient hier aber zum Üben mit unterschiedlichen regulären Ausdrücken.

[3] Dabei wird das Entwurfsmuster "Template Method Pattern" bzw. "Schablonenmethode" genutzt/realisiert ...

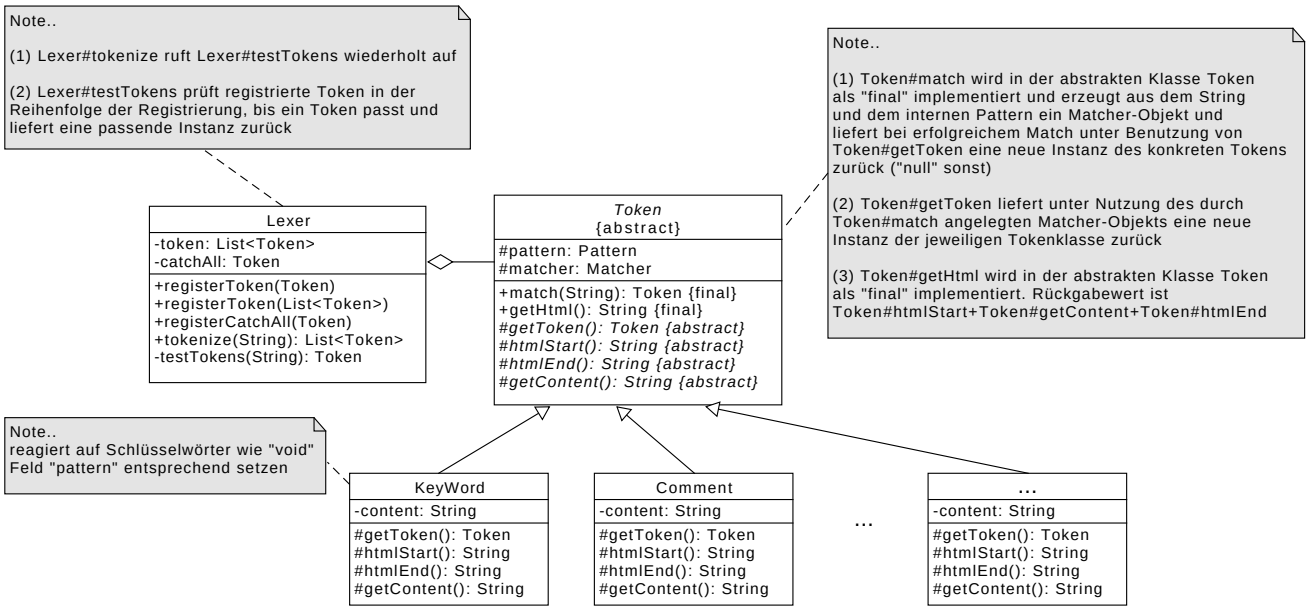

Abbildung 1: UML-Diagramm für Lexer

nutzt das von `Token#match` erzeugte Matcher-Objekt, um den jeweils gematchten Inhalt zu holen und diesen als neue Instanz der konkreten Token-Klasse zurück zu liefern.

Die Token sind untereinander nicht gleichberechtigt: Kommentare und Strings können zwar andere Token (beispielsweise andere Kommentare oder Java-Schlüsselwörter) prinzipiell enthalten, diese sollen aber dann nicht als eigene Token erkannt werden. Der Lexer muss beim Abarbeiten des Strings darauf achten:

- Die Tokenklassen werden in der entsprechenden Reihenfolge im Lexer "registriert".[4]
- In dieser Reihenfolge sollen sie entsprechend in der Methode `Lexer#testTokens` abgearbeitet werden.

Nach einem erfolgreichen Durchlauf ergibt sich eine Folge von Token-Objekten. Wenn man die String-Repräsentation der Token in dieser Reihenfolge verkettet, sollte sich wieder der ursprüngliche Text ergeben.

Hinweis: Das "Catch-All"-Token dient dazu, Text zwischen zwei anderen Token aus dem String zu entfernen. Da man nicht weiss, wieviel Text zwischen zwei Token kommen wird und da das "Catch-All"-Token keine Kenntnis über die anderen Token-Klassen haben soll, passt das "Catch-All"-Token auf genau **ein beliebiges** Zeichen. Es wird in der Methode `Lexer#testTokens` als letzte Tokenklasse geprüft, wenn keine der anderen Tokenklassen gepasst hat und entfernt genau ein Zeichen vom Anfang des Strings. D.h. bei den in diesem Semester zu implementierenden Tokenklassen wäre `foo(String s) {}` kein eigenständiges Token, sondern eine Folge von `CatchAll`-Tokens für jeden einzelnen Buchstaben (pro Buchstabe eine Instanz von `CatchAll`).

Hinweis: Der zu verarbeitende String kann natürlich Zeilenumbrüche enthalten. Suchen Sie einmal nach "*multi line regular expression*" ...

Hinweis: Die in Abbildung 1 dargestellten Methoden `htmlStart`, `htmlEnd`, `getHtml` der Token-Klassen sind für diese Teilaufgabe nicht relevant und können ignoriert werden. Diese Methoden werden erst für die Tutoriumsaufgabe "Syntax-Highlighting" benötigt.

Hinweis: Alle von Ihnen implementierten Klassen sollen sich im Package `lexer` befinden!

Beachten Sie: Hier geht es vor allem um den Umgang mit regulären Ausdrücken. Nutzen Sie also nicht irgendwelche Lexer-/Parser-/Scanner-**Generatoren** o.ä., sondern implementieren diesen Lexer "zu Fuß" mit Hilfe der in den Klassen `java.lang.String`, `java.util.regex.Pattern` und `java.util.regex.Matcher` zur Verfügung stehenden Methoden!

Thema: Sicherer Umgang mit regulären Ausdrücken, Anwendung dynamischer Polymorphie, Lesen komplexerer UML-Klassendiagramme

[4] In dieser Teilaufgabe kann dies zunächst fest codiert geschehen, in der nächsten Teilaufgabe wird dies dynamisch zur Laufzeit realisiert.

1.2 Laden der Token per Reflection

Die Tokenklassen sollen nun per Reflection zur Laufzeit geladen werden. Dazu sollen aus `.class`-Dateien, die sich in einem bestimmten Ordner *außerhalb* des Java-Classpath des Projektes befinden, alle Klassen mit dem Obertyp `lexer.Token` geladen und registriert werden.

Für die Reihenfolge bei der Registrierung sollen selbst erstellte Annotationen genutzt werden, mit denen die Tokenklassen jeweils ausgezeichnet werden:

Nutzen Sie für die Priorisierung die beiden Annotationen `@Prio` und `@CatchAll`:

- Die Annotation `@Prio` soll ein Attribut `value` vom Typ `int` haben, so dass man die Priorität über den Wert angeben kann (Werte: 1, 2 oder 3; 1 ist die höchste Priorität, 3 die niedrigste Priorität). Es können jeweils mehrere Tokenklassen mit der selben Priorität ausgezeichnet werden, die Reihenfolge der Registrierung ist dann innerhalb der selben Priorität nicht spezifiziert.
- Es muss genau eine Tokenklasse mit `@CatchAll` markiert werden (niedrigste Priorität).

Markieren Sie alle Tokenklassen passend mit den Annotations-Varianten.

Die selbst erstellten Annotationen sollen nur an Klassendeklarationen genutzt werden dürfen und müssen zur Laufzeit zur Verfügung stehen.

Hinweis: Beachten Sie, dass sich die Token-Klassen im Package `lexer` befinden!

Hinweis: Denken Sie daran, dass die Token-Klassen nicht im Classpath für den Lexer sein sollen. Dies erreichen Sie am einfachsten, indem Sie ein zweites Projekt für die Token-Klassen anlegen. Zur Kompilierung benötigen beide Projekte dann die Klasse `lexer.Token`. Die Kompilierung der beiden Projekte lässt sich sehr gut durch geeignete Buildskripte synchronisieren.

Thema: Erstellung von Annotationsklassen, Einsatz von Reflection

1.3 Logger I

Bauen Sie in Ihren Lexer ein Logging auf der Basis von `java.util.logger` ein.

Jedes Laden und Benutzen von Token soll geeignet geloggt werden, damit man den Ablauf des Programms nachvollziehen kann. Dies betrifft den Aufruf der Methoden `match` und `getToken` in den Token-Klassen sowie die Methoden `registerToken`, `registerCatchAll` und `tokenize` im `Lexer`: Nutzen sie hier das Level `INFO`.

Überlegen Sie sich den sinnvollen Einsatz eines weiteren Log-Levels und implementieren Sie dies entsprechend.

Thema: Umgang mit der Java-Logging-API `java.util.logger`

2 Tutorium

Lösen Sie die folgenden Aufgaben selbstständig **vor** dem Tutorium und diskutieren Sie Ihre Lösung und ggf. aufgetauchte Probleme und Fragen im Tutorium.

Hinweis: Beachten Sie dabei auch den Zeitplan der Themen in der Vorlesung: Sie werden entsprechend nicht alle Fragen gleich zum ersten Tutorium lösen können!

2.1 Template Method Pattern

Führen Sie eine Recherche durch und erklären Sie die Funktionsweise des *Template Method Pattern*.

Thema: Selbstständige Literaturrecherche, Entwurfsmuster "Template Method"

2.2 Build-Skripte: Ant oder Gradle

Erstellen Sie ein oder mehrere Build-Skripte (etwa für Ant oder für Gradle), mit denen Sie Ihr Projekt unabhängig von der IDE übersetzen und starten können.

2.3 Logger II

Erweitern Sie das Logging in Ihrem Lexer:

- Nutzen Sie zunächst einen `ConsoleHandler`. Schreiben Sie sich einen Formatter, den Sie im `ConsoleHandler` nutzen und der die Logmeldungen in etwa so formatiert:

  ```
  ------------------------------------------------------------
  Logger: <name>
      Level: <level>
      Class: <class>
      Method: <method>
      Message: <message>
  ------------------------------------------------------------
  ```

- Um eine zusätzliche Ausgabe in eine HTML-Datei zu ermöglichen, ergänzen Sie die Logger nun noch um einen `FileHandler`. Schreiben Sie sich einen weiteren Formatter, der die Logmeldung als HTML-Schnipsel formatiert (analog zur Formatierung für die Ausgabe in der GUI) und nutzen Sie diesen im `FileHandler`.

  Je nach Level soll die Farbe und Hervorhebung der Meldung variieren, beispielsweise *rot* und *fett* für das Level `SEVERE`. Schauen Sie sich das generierte Logfile mit einem Browser an.

- Unterdrücken Sie zusätzlich erscheinende Konsolenausgaben, die nicht von Ihren Formattern stammen.

- Wie können Sie die Empfindlichkeit des Loggings in der `main()`-Methode steuern? Wie kann man dort einzelne Logger komplett abschalten? Nutzen Sie dazu die in der nächsten Teilaufgabe implementierte CLI-Option.

Überlegen Sie sich geeignete Testfälle und führen Sie diese durch.[5]

Thema: Sicherer Umgang mit der Java-Logging-API `java.util.logger`

[5] Passende Aufrufe in `main()` und manuelle Kontrolle der Konsolenausgabe bzw. der Zieldatei reichen. Sie können aber auch gern mit JUnit arbeiten.

2.4 Konfiguration

Bauen Sie unter Nutzung von Apache Commons CLI eine Auswertung von Kommandozeilen-Parametern in Ihren Lexer ein.

Es soll mindestens folgende Optionen geben:

1. Ausgabe einer Hilfe für den Start des Programms: Welche Optionen gibt es, wie werden diese geschrieben, welche Parameter werden erwartet, ...
2. Aktivierung (bzw. Deaktivierung) des Loggers.
3. Eingabe eines Ordners, aus dem die Token per Reflection geladen werden sollen.

Nutzen Sie benannte Parameter. Sehen Sie jeweils sowohl die Kurz- als auch die Langform vor. Beachten Sie, dass einige Parameter selbst Argumente haben.

Thema: Umgang mit der Apache CLI-Bibliothek

2.5 Syntax-Highlighting

Erweitern Sie den Logger um Funktionalität zum SyntaxHighlighting.

1. Implementieren Sie die Methoden zum Formatieren des Token-Inhalts.

   Die Methode `Token#getHtml` wird in der abstrakten Basisklasse `Token` implementiert und kann nicht überschrieben werden (`final`). Sie erzeugt den gewünschten HTML-formatierten String durch den Aufruf der abstrakten Methoden `Token#htmlStart`, `Token#getContent` und `Token#htmlEnd` und die entsprechende Konkatenation der jeweiligen Rückgabewerte. Die drei zuletzt genannten Methoden werden erst in den konkreten (abgeleiteten) Token-Klassen implementiert.[6]

   Nutzen Sie für die Formatierung **HTML-Tags**. Beispielsweise könnten Java-Schlüsselwörter fett und dunkelrot formatiert werden, Kommentare kursiv und in einem dunklen Grau. Die einzelnen Tokenarten sollen sich in ihrer Formatierung deutlich voneinander unterscheiden, die verschiedenen Kommentare können die gleiche Formatierung nutzen.

   Beispiel (Schlüsselwort `public` in rot+fett):

   ```
   <font color="red"><b>public</b></font>
   ```

   Verweis auf Textformatierung mit HTML: selfhtml.org.

2. Finden Sie heraus, welche von `JTextComponent` abgeleiteten Klassen HTML-Dokumente darstellen können und die Bearbeitung der dargestellten Inhalte ermöglichen.

   Bauen Sie eine Swing-GUI auf, deren zentrales Element eine Instanz dieser Klasse ist. Das Element soll auf Textänderungen reagieren, indem nach einer Änderung der aktuelle Textinhalt an den Lexer aus dem vorigen Aufgabenteil übergeben wird und aus der resultierenden Liste von Token jeweils die **formatierten Stringrepräsentationen** abgerufen werden.

   Diese Strings müssen in ein minimales HTML-Gerüst eingebettet werden und können dem Element als neuer Text gesetzt werden. Verweis auf Aufbau einer HTML-Datei: selfhtml.org: Grundgerüst. Ihre durch die Token erzeugten HTML-Schnipsel gehören zwischen die Tags `<body>` und `</body>`.

   Da das Element HTML-formatierten Text interpretieren und formatiert darstellen kann, haben Sie soeben ein einfaches Syntax-Highlighting realisiert.

   Hinweis: Das Highlighting muss nicht nach jeder Änderung am Eingabetext automatisch aktualisiert werden. Es ist ausreichend, eine entsprechende Aktualisierung durch Aktivierung eines Buttons o.ä. zu triggern. Weiterhin ist es ausreichend, wenn Sie ein Editorfenster (ohne Highlighting) und ein Vorschaufenster (Highlighting des Textes im Editor-Fenster) realisieren.

Thema: Formatierung von Strings mit HTML-Code; Einarbeitung in Swing

[6] Dabei wird das Entwurfsmuster "Template Method Pattern" bzw. "Schablonenmethode" genutzt/realisiert ...

2.6 Exception Handling

Arbeiten Sie die Kapitel zum Exception-Handling unter Java in der Semesterliteratur durch.

1) Erklären Sie die Bedeutung und die Funktionsweise des `finally`-Blockes bei der Exception-Behandlung mit `try` und `catch`.

   Was bedeutet “try-with-resources”, wie wird es angewendet?

2) Worin liegt der Unterschied zwischen “Checked Exceptions” und “Unchecked Exceptions”? Geben Sie passende Beispiele an.

3) Erklären Sie den Umgang mit Exceptions: Wann sollte man Exceptions fangen und bearbeiten und wann sollte man Exceptions propagieren?

Thema: Selbstständige Literaturrecherche, Umgang mit Exceptions in Java