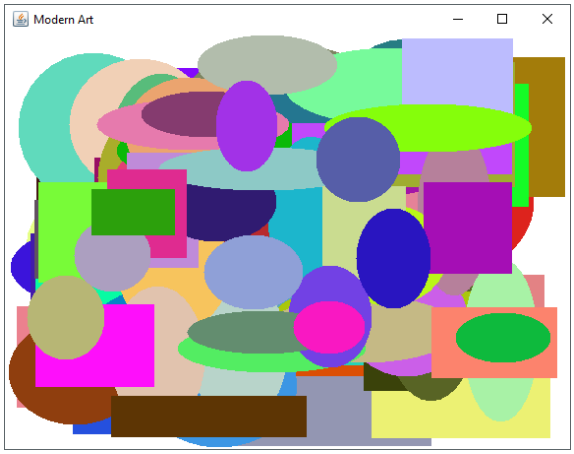
252-0027-00: Einführung in die Programmierung I

Übungsblatt 11

Abgabe: 13. Dezember 2016, 13:00

Checken Sie die neue Übungs-Vorlage aus. Vergessen Sie nicht, Tests zu schreiben!

Aufgabe 1: Comparable Modern Art



In der Vorlesung haben Sie gesehen, wie man in Java einfache Graphiken anzeigen kann. In dieser Übung sollen Sie in einem Fenster "moderne Kunst" anzeigen und dabei ein generisches Interface `Comparable<T>` verwenden.

Das Programm `ModernArtApp` in der Vorlage enthält bereits eine `main()`-Methode. Diese erstellt ein Fenster (`JFrame`), fügt ein `ModernArtPanel` ein und öffnet das Fenster. Danach ruft sie im Abstand von 100 ms die `repaint()`-Methode des Fensters auf, damit das Fenster neu gezeichnet wird. Ihre Aufgabe ist es, die `ModernArtPanel`-Klasse so zu erweitern, dass zufällig generierte geometrische Formen angezeigt werden. Bisher ist die Klasse leer, d.h., sie verhält sich genau so wie ihre Superklasse `JPanel`, welche einfach eine leere Fläche anzeigt.

a) Erstellen Sie einen Konstruktor ohne Parameter, welche den Hintergrund des Panels auf Weiss setzt. Verwenden Sie die `setBackground()`-Methode und die `Color`-Klasse[1].

   Nun überschreiben Sie die `paintComponent()`-Methode von `JPanel`. Rufen Sie zu Beginn die `super`-Methode auf, welche den Hintergrund malt. Fügen Sie einige Zeichenbefehle hinzu, indem Sie auf das `Graphics`-Argument verschiedene `draw*()`- oder `fill*()`-Methoden aufrufen. Mit `setColor()` ändern Sie die Zeichenfarbe. Führen Sie das Programm aus.

b) Damit das Ganze als moderne Kunst durchgeht, sollen Sie jetzt zufällig generierte Formen zeichnen. Erstellen Sie ein Interface `Shape` mit einer `draw(Graphics)`-Methodendeklaration und zwei Klassen `Rectangle` und `Oval`, welche `Shape` implementieren. Die Klassen sollen die

[1] `Color` und andere Graphik-Klassen befinden sich in den Packages `java.awt` und `javax.swing`. Für beide ist bereits eine `import`-Anweisung vorhanden

Eigenschaften Grösse (als `java.awt.Dimension`), Position (als `java.awt.Point`) und Farbe (als `java.awt.Color`) haben. Die Position bezeichnet beim Rechteck die linke obere Ecke und beim Oval (der Einfachheit halber) die linke obere Ecke des umschliessenden Rechtecks.

Das `ModernArtPanel` soll sich jetzt eine Liste (`ArrayList`) von `Shapes` merken und dieser bei jedem Aufruf von `paintComponent()` eine neue, zufällig generierte Form hinzufügen. Falls die Grösse der Liste eine festgelegte Grenze überschreitet, soll die Method per Zufall eine Form wieder entfernen. Schliesslich sollen alle Formen in der Liste (mithilfe der `Shape.draw()`-Methode) gezeichnet werden. Wenn Sie das Programm jetzt ausführen, sollten Sie Formen erscheinen und wieder verschwinden sehen. Moderne Kunst in stetem Wandel!

c) Erweitern Sie das Programm so, dass die Formen der Grösse nach gezeichnet werden (die kleinsten zuletzt). Dazu soll das Interface `Shape` das Interface `Comparable<T>` erweitern.

Im Gegensatz zum `Comparable` aus dem letzten Übungsblatt ist das `Comparable<T>` im `java.lang`-Package generisch. Wenn eine Klasse dieses Interface implementiert (oder ein Interface es erweitert), kann sie genau festlegen, mit welchem Typ sie vergleichbar ist (normalerweise nur mit sich selbst). In dieser Übung sollen `Shapes` mit anderen `Shapes` vergleichbar sein. Fügen Sie dazu zusätzlich eine `area()`-Methodendeklaration zu `Shape` hinzu und implementieren Sie in `Rectangle` und `Oval` beide Methoden `area()` und `compareTo()`[2].

Um die Formen der Grösse nach zu zeichnen, können Sie die `Shape`-Liste sortieren. Verwenden Sie dazu Methoden aus der `Collections`-Klasse. Wenn Sie alles richtig gemacht haben, sollten grosse Formen im Hintergrund und kleine im Vordergrund erscheinen.

Aufgabe 2: Expression Evaluator

In dieser und in folgenden Übungen werden Sie eine Reihe von Programmen schreiben, welche andere Programme interpretieren, kompilieren oder (in kompilierter Form) ausführen. Die Programmiersprachen definieren wir selber.

Als Einstieg schreiben Sie ein Programm, welches mathematische Ausdrücke (*expressions*) auswertet. Die Ausdrücke bestehen aus Zahlen, Variablen, Operatoren wie $+$ oder $-$ und einfachen Funktionen wie $\sin()$ oder $\cos()$. Die genaue Syntax für diese Ausdrücke finden Sie als EBNF-Beschreibung in Abbildung 1. Beachten Sie, dass die Beschreibung rekursiv ist: Auf der RHS (rechten Seite) von *expr* kommt *term* vor und auf der RHS von *term* kommt wiederum *expr* vor.

Ein Programm, das Ausdrücke auswertet, muss natürlich entscheiden, ob eine gegebene Zeichenkette überhaupt ein gültiger Ausdruck ist[3]. Das nennt man *parsen* und ein solches Programm heisst *Parser*. Aus einer EBNF-Beschreibung wie dieser kann man einfach einen Parser erstellen[4]:

- Regeln werden zu Methoden.
- Alternativen werden zu `if`-Anweisungen.
- Regeln auf der RHS werden zu Methodenaufrufen.

Man unterscheidet dabei zwischen zwei Arten von Regeln: *Parser-Regeln* und *Tokenizer-Regeln*. Zuerst teilt ein *Tokenizer* die Zeichenkette aufgrund der Tokenizer-Regeln in eine Reihe von Tokens

[2] Sie werden merken, dass `compareTo()` in beiden Klassen identisch implementiert werden kann. In solchen Fällen könnte man stattdessen eine `default`-Methode verwenden. Diese sind aber nicht Teil des Unterrichtsstoffs.

[3] Ähnlich wie Sie, wenn Sie mit einer Tabelle überprüfen, ob ein Symbol einer EBNF-Beschreibung entspricht.

[4] Im Allgemeinen, d.h. für gewisse andere EBNF-Beschreibungen, ist das leider nicht möglich.

$$
\begin{array}{lcl}
\textit{digit} & \Leftarrow & \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9} \\
\textit{char} & \Leftarrow & \texttt{A} \mid \texttt{B} \mid \ldots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z} \\
\textit{num} & \Leftarrow & \textit{digit}\ \{\ \textit{digit}\ \}\ [\ \texttt{.}\ \textit{digit}\ \{\ \textit{digit}\ \}\ ] \\
\textit{var} & \Leftarrow & \textit{char}\ \{\ \textit{char}\ \} \\
\textit{func} & \Leftarrow & \textit{char}\ \{\ \textit{char}\ \}\ \texttt{(} \\
\textit{op} & \Leftarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\^{}} \\
\textit{open} & \Leftarrow & \texttt{(} \\
\textit{close} & \Leftarrow & \texttt{)} \\
\\
\textit{atom} & \Leftarrow & \textit{num} \mid \textit{var} \\
\textit{term} & \Leftarrow & \textit{open}\ \textit{expr}\ \textit{close} \mid \textit{func}\ \textit{expr}\ \textit{close} \mid \textit{atom} \\
\textit{expr} & \Leftarrow & \textit{term}\ [\ \textit{op}\ \textit{term}\ ]
\end{array}
$$

Abbildung 1: **EBNF-Beschreibung** von *expr*

auf. In unserer EBNF-Beschreibung sind die Tokenizer-Regeln rot dargestellt. Die grauen Regeln werden zwar intern vom Tokenizer verwendet, aber erzeugen keine eigenen Tokens. Zum Beispiel erzeugt die Zeichenkette "`sin(1 + x) * 3.14`" die folgende Reihe von Tokens:

func : `sin(` *num* : `1` *op* : `+` *var* : `x` *close* : `)` *op* : `*` *num* : `3.14`

Danach entscheidet der Parser aufgrund der Parser-Regeln (oben in Schwarz dargestellt), ob eine solche Reihe von Tokens einen gültigen Ausdruck darstellt. Abbildung 2 zeigt, wie die Parser-Methode für *term* aussehen könnte.

a) In der Übungsvorlage finden Sie eine `Tokenizer`-Implementation, eine Vorlage für den `ExprParser` und eine `EvaluatorApp` mit einer `main()`-Methode. Diese parst die vom Benutzer eingegebenen Zeichenketten und gibt an, ob sie gültige Ausdrücke sind. Wenn der Benutzer "exit" eingibt, terminiert das Programm. Ihre Aufgabe ist es, den `ExprParser` zu schreiben.

   Erstellen Sie in der schon vorgegebenen `parse(String)`-Methode eine `Tokenizer`-Instanz. Das Interface des `Tokenizers` ist dem der `Scanner`-Klasse nachempfunden. Sie können also die `hasNext*()`-Methoden verwenden, um zu prüfen, welche Art von Token als nächstes kommt, und die `next*()`-Methoden, um Tokens zu "konsumieren". Schreiben Sie die nötigen `parse*(...)`-Methoden, eine für jede Parser-Regel. Die erste Ihrer `parse*(...)`-Methoden rufen Sie von `parse(String)` aus auf. Diese Methoden sollen eine `EvaluationException` mit einer sinnvollen Fehlermeldung werfen, falls die Zeichenkette kein gültiger Ausdruck ist. Falls z.B. nach "`(`" und einer *expr* das Token "`10`" statt "`)`" folgt, könnte die Fehlermeldung lauten:

   ```
   Syntax error: unexpected token '10', expected ')'
   ```

b) Um aus dem `ExprParser` einen `ExprEvaluator` zu machen, kann man die Methoden so ändern, dass sie im selben Zug das Resultat berechnen. Jede Methode überprüft dann nicht nur, ob die nächsten Tokens der Regel entsprechen, sondern gibt auch gleich den Wert des entsprechenden Ausdruck-Teils zurück. Dies sehen Sie in Abbildung 3.

```
/* checks if the next tokens form a valid term */
void parseTerm(...) {
    if(next token is a "open") {
        consume "open" token
        // check if the next tokens are a valid expr:
        parseExpr(...);
        check whether next token is a "close" & consume
    }
    else if(next token is a "func") {
        consume "func" token
        // check if the next tokens are a valid expr:
        parseExpr(...);
        check whether next token is a "close" & consume
    }
    else {
        // check if the tokens are a valid atom:
        parseAtom(...);
    }
}
```

Abbildung 2: Parser-Methode für *term*

```
/* evaluate the next tokens as a term */
double evalTerm(...) {
    if(next token is a "open") {
        consume "open" token
        double val = evalExpr(...);
        check whether next token is a "close" & consume
        return val;
    }
    else if(next token is a "func") {
        consume "func" token
        double arg = evalExpr(...);
        check whether next token is a "close" & consume
        double result = apply function to arg
        return result;
    }
    else {
        return evalAtom(...);
    }
}
```

Abbildung 3: Evaluator-Methode für *term*

Benennen Sie als erstes die Klasse und die Methoden um[5], so dass sie die neue Funktionalität widerspiegeln. Nun können Sie entscheiden: Erstens, welche Funktionen möchten Sie erlauben? Für Aufgabe 3 sollten Sie mindestens sin(), cos() und tan() unterstützen, aber auch andere Funktionen wie abs() oder log() könnten später Spass machen. Schauen Sie sich die `Math`-Klasse für weitere Kandidaten an (oder implementieren Sie selber welche). Zweitens können Sie entscheiden, wie Sie mit Variablen umgehen. Sie sollten mindestens eine "x"-Variable unterstützen, und wir empfehlen, dass Sie den Wert dafür dem `ExprEvaluator`-Konstruktor übergeben. In jedem Fall sollten Sie eine Exception werfen, falls nicht-unterstützte Funktionen oder Variablen in einem Ausdruck vorkommen.

Am Schluss sollte die `EvaluatorApp` das Resultat der eingegebenen Ausdrücke ausgegeben, statt nur zu sagen, ob sie gültig sind. Wenn Sie wollen, können Sie dem Benutzer auch die Möglichkeit geben, Werte für Variablen zu definieren.

xkcd: (by Randall Munroe (CC BY-NC 2.5)

Aufgabe 3: Funktions-Plotter

Mit dem `ExprEvaluator` von Aufgabe 2 können wir ein praktisches Programm schreiben: einen Funktions-Plotter. Dieser interpretiert einen eingegebenen Ausdruck als Funktion $f(x)$, wertet $y = f(x)$ für verschiedene x aus und zeichnet die Funktion, indem er die resultierenden (x, y)-Punkte mit Linien verbindet.

[5] Verwenden Sie die äusserst praktische *Rename*-Funktion von Eclipse, welche überall im Code die nötigen Anpassungen vornimmt: Klicken Sie rechts auf den Klassen- oder Methoden-Namen und wählen Sie *Refactor* → *Rename...*

In der Übungsvorlage finden Sie die `PlotterApp`, welche ein kleines GUI (*graphical user interface*) erstellt, wo der Benutzer eine Funktion eingeben kann. Diese wird an eine Instanz von `PlotterPanel` übergeben. Ihre Aufgabe ist es, die `PlotterPanel.paintComponent()`-Methode so zu überschreiben, dass sie diese Funktion plottet.

a) Im ersten Schritt sollen Sie eine Koordinaten-Transformation von den Ur-Koordinaten (x, y) (in welchen $f(x)$ definiert ist) zu den GUI-Koordinaten (X, Y) implementieren. Dies ist u.a. nötig, weil die Y-Achse des GUI-Koordinatensystem nach unten verläuft, während wir es gewohnt sind, dass die y-Achse nach oben verläuft.

   Dem `PlotterPanel`-Konstruktor werden Werte für x_{min}, x_{max}, y_{min} und y_{max} übergeben. Diese Werte geben an, welcher Teil des Ur-Koordinatensystems im Panel sichtbar ist. Hier sind einige Beispiele für die Transformation (w und h stehen für die Breite und Höhe des `PlotterPanels`):

Ur-Koordinaten (x, y)	$\rightarrow$	(X, Y) GUI-Koordinaten
$(0, 0)$		$(\frac{w}{2}, \frac{h}{2})$
$(x_{\text{min}}, y_{\text{min}})$		$(0, h)$
$(x_{\text{max}}, y_{\text{max}})$		$(w, 0)$

   Implementieren Sie zwei Methoden `toGuiX()` und `toGuiY()`, welche die Transformation berechnen. Überschreiben Sie dann die `paintComponent()`-Methode so, dass sie die x- und y-Achse des Ur-Koordinatensystems zeichnet. Plotten Sie ein paar Punkte im Ur-Koordinatensystem um zu sehen, ob Ihre Berechnung korrekt ist (oder besser, schreiben Sie Tests).

   Tipp: Die aktuelle Grösse des Panels bekommen Sie mit `getWidth()` und `getHeight()`.

b) Erweitern Sie `paintComponent()` so, dass die Funktion geplottet wird. Iterieren Sie dazu über alle Werte der X-Achse im GUI-Koordinatensystem ($0 \leq X < w$), finden Sie für jedes X das dazugehörige x im Ur-Koordinatensystem und berechnen Sie $y = f(x)$. Transformieren Sie diese y-Werte zurück ins GUI-Koordinatensystem und verbinden Sie die (X, Y)-Punkte mit Linien. Zusätzlich zur `toGuiY()`-Methode brauchen Sie dafür auch eine `fromGuiX()`-Methode.

   Um $y = f(x)$ zu berechnen, brauchen Sie natürlich Ihren `ExprEvaluator`. Übergeben Sie ihm in jeder Iteration den aktuellen x-Wert. Falls bei der Evaluation ein Fehler auftritt, sollen Sie mithilfe von `drawString()` die Fehlermeldung im `PlotterPanel` ausgeben.

c) **Optional:** Erweitern Sie Ihren Plotter um zusätzliche Features, zum Beispiel:

   - Achsen-Striche und -Beschriftung: Zeichnen Sie zusätzlich zu den Achsen Haupt- und Nebenstriche und fügen Sie die dazugehörigen Werte hinzu.
   - Automatische Skalierung der y-Achse: Berechnen Sie y_{min} und y_{max} basierend auf den erhaltenen y-Werten in jedem Aufruf von `paintComponent()` neu.
   - Schauen Sie sich an, wie `MouseListeners` funktionieren, und machen Sie Ihr `PlotterPanel` interaktiv. Erlauben Sie z.B. Rein- und Raus-zoomen oder Verschieben, oder zeigen Sie für den x-Wert, auf den die Maus zeigt, den y-Wert an. Falls Sie etwas Interaktives machen, sollten Sie an den entsprechenden Stellen `repaint()` auf das `PlotterPanel` aufrufen.

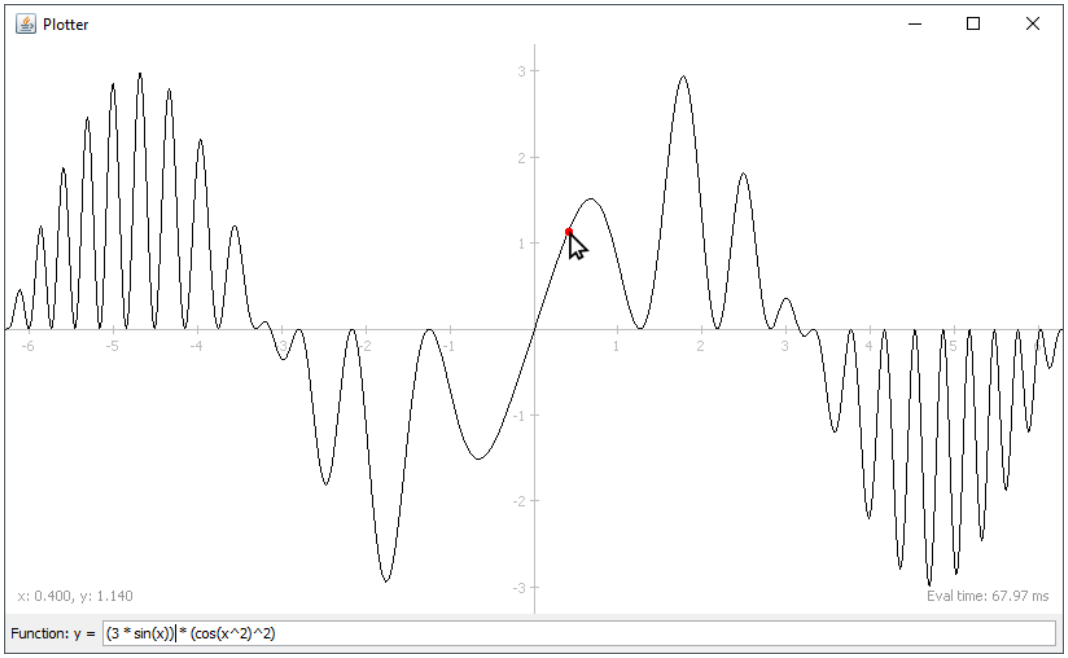


Abbildung 4: `PlotterPanel` mit Zusatz-Features