Programmieren II (Java)

1. Praktikum: Grundlagen

Sommersemester 2025
Christopher Auer

Lernziele

- Erstes Beschnuppern von Java
- Arbeiten mit Kontrollstrukturen und primitiven Datentypen
- Arithmetik
- Implementieren einer Konsolenanwendung
- Implementierung eines Algorithmus nach einer Spezifikation
- Arbeiten mit einem Debugger

- Schauen Sie sich die Tutorial-Videos auf der *Moodle-Seite* mit *wichtigen Hinweisen* zum Praktikum und VS-Code an!
- Sie dürfen die Aufgaben *alleine* oder zu *zweit* bearbeiten und abgeben
- Sie müssen *4 der 5* Praktika bestehen
- *Kommentieren* Sie Ihren Code
  - Jede *Methode* (wenn nicht vorgegeben)
  - *Wichtige* Anweisungen/Code-Blöcke
  - *Nicht kommentierter* Code führt zu *Nichtbestehen*
- Bestehen Sie eine Abgabe *nicht* haben Sie einen *zweiten Versuch*, in dem Sie Ihre Abgabe *verbessern müssen*.
- *Wichtig*: Sie sind einer *Praktikumsgruppe* zugewiesen, *nur* in dieser werden Ihre Abgaben *akzeptiert*!

Aufgabe 1: Java zu Fuß

Erstellen Sie eine Java-Datei mit folgendem Inhalt und dem Namen `HelloJava.java`.

```java
public class HelloJava{
  public static void main(String[] args){
    System.out.println("Hello Java!");
  }
}
```

HelloJava.java

- ☑ Installieren Sie das JDK in der Version *21*[1] („Java Development Kit“) von Oracle *oder* OpenJDK *oder* Adoptium JDK (wählen Sie als Version *21* und *JDK* als Package Type).
- ☑ Starten Sie eine Kommandozeile und navigieren Sie (mit `cd`) in das Verzeichnis, in dem die Datei `HelloJava.java` liegt.
- ☑ Übersetzen Sie die Datei folgenden Kommando in eine `.class`-Datei:

```
javac HelloJava.java
```

- ☑ Führen Sie das Programm aus mit

```
java HelloJava
```

  Das Programm sollte `Hello Java!` ausgeben.

Hinweise

- Sie benötigen für diese Aufgabe keine Entwicklungsumgebung.
- Diese Aufgabe müssen Sie *nicht abgeben*

[1] Versionsnummer der Form 21.x.y verwenden

Aufgabe 2: Debuggen eines Java-Programms ★★

In dieser Aufgabe werden Sie einen *Debugger* verwenden, um einem Fehler (*Bug*) in einem Programm zu beheben. Bei dem Programm handelt es sich um die (noch etwas *fehlerhafte*) Lösung einer Praktikumsaufgaben aus dem *Sommersemester 2024*.

Wikipedia sagt zu Debugger: „A debugger is a computer program used to test and debug other programs (the “target” programs). Common features of debuggers include the ability to run or halt the target program using breakpoints, step through code line by line, and display or modify the contents of memory, CPU registers, and stack frames.“

In *Visual Studio Code* müssen Sie zum Debuggen von Java-Programmen die Erweiterung „Debugger for Java“ installiert haben. Diese wird mit der Erweiterung Extension Pack for Java automatisch installiert. Hinweise zur Verwendung des Debuggers in *Visual Studio Code* finden Sie hier.

Lesen Sie sich zunächst die *Aufgabenstellung aus dem Sommersemester 2024* durch, die Sie *nicht bearbeiten müssen*!

Aufgabenstellung Sommersemester 2024: Monte-Carlo-Integration

In dieser Aufgabe nähern wir das Integral einer reellen Funktion an. Als Vereinfachung betrachten wir die Funktion im Intervall $[0, 1]$ und verlangen, dass die Funktionswerte in dem Intervall ebenfalls im Intervall $[0, 1]$ liegen.

Die rot schattierte Fläche entspricht dem Integral der Funktion im Intervall $[0, 1]$.

Es ist im Allgemeinen nicht so einfach das Integral einer Funktion exakt zu berechnen, aber glücklicherweise gibt es verschiedene Näherungsverfahren. Wir betrachten hier die *Monte-Carlo-Integration*: Bezeichnen wir mit A_f den Flächeninhalt unterhalb der Funktion begrenzt durch die x-Achse und mit A_Q den Flächeninhalt des Quadrats $[0, 1] \times [0, 1]$. Dann ist A_f der Wert des Integrals und $A_Q = 1 \cdot 1 = 1$. Erzeugt man zufällig[2] N Punkte innerhalb des Quadrats und zählt wie oft Punkte *unterhalb* der Funktion landen (bezeichnet durch N_f, schattierter Bereich), dann gilt

[2] Für Stochastik-Spezialisten: *uniform verteilt* und *unabhängig*

für *große N*:

$$\frac{N_f}{N} \approx \frac{A_f}{A_Q}$$

Wenn wir für $A_q = 1$ einsetzen, bekommen wir:

$$\frac{N_K}{N} \approx A_f$$

Den Wert des Integrals kann man also annähern indem man zählt wieviele Punkte unterhalb der Kurve liegen und dies durch die Anzahl aller Punkte teilt.

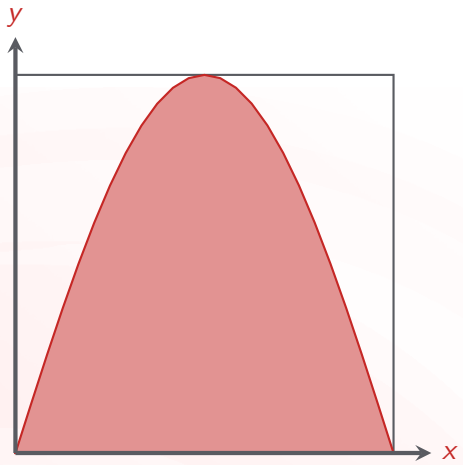
- ☑ Erstellen Sie eine Java-Klasse `MonteCarloIntegration` mit einer `main`-Methode.
- ☑ Implementieren Sie eine *statische Methode* `public static double function(double x)`, die den Funktionswert berechnet. Als sehr einfaches Beispiel, verwenden Sie die Funktion:

  $$f(x) = x$$

  Dies entspricht einer Gerade mit *Steigung* 1, die im Ursprung beginnt. Das Integral hat hier den Wert 0.5.
  Als Vorschlag für eine interessantere Funktion, verwenden Sie:

  $$f(x) = \sin \pi \cdot x$$

  Die Kurve sieht wie folgt aus:



  Und der (exakte) Wert des Integrals hat den Wert $\frac{2}{\pi} \approx 0.6366$.[3]
- ☑ Um die Anzahl der Iterationen zu begrenzen, deklarieren Sie eine Konstante `MAX_ITERATIONS` in der Klasse `MonteCarloIntegration` mit dem Wert `100_000`. Deklarieren Sie zusätzliche eine Konstante `MIN_CHANGE` mit dem Wert 10^{-5}. Die Funktion der Variable wird weiter unten klar.
- ☑ Implementieren Sie folgenden Algorithmus zur Annäherung des Integrals in der `main`-Methode
  - Deklarieren Sie drei lokale Variablen mit geeigneten Datentypen und Initialwerten:
    - `allPoints` — Anzahl der Punkte (entspricht der Anzahl der bisherigen Iterationen)
    - `pointsUnderCurve` — Anzahl der Punkte *unter dem Funktionsgraphen*
    - `double approxInt` — *Annäherung des Integrals*
  - Solange die Anzahl der Iterationen *kleiner* als `MAX_ITERATIONS` ist *und* der Absolutbetrag der Änderung des Werts von `approxInt` im Vergleich zur vorherigen Iteration *mindestens* `MIN_CHANGE`, wiederhole:
    - Erzeuge eine zufällige x-Koordinate zwischen 0 und 1 (verwenden Sie dazu `Math.random()`)

[3] D.h. wenn Sie den Kehrwert des Integrals mit 2 multiplizieren, sollte ungefähr π herauskommen.

- Erzeuge eine zufällige y-Koordinate zwischen 0 und 1
- Erhöhen Sie den Wert von `allPoints` um 1
- Wenn der y-Wert *kleiner oder gleich* dem Funktionswert an der Stelle x ist, so erhöhen Sie `pointsUnderCurve` um 1.
- Berechnen Sie anhand der obigen Formel `approxInt`.
- Geben Sie das aktuelle Ergebnis aus, z.B.:

```
Iteration 36257: 0.63745 (0.000010)
```

Die bisherigen Annäherung des Integrals soll mit *fünf Nachkommastellen* dargestellt werden.

☑ Testen Sie Ihr Programm *mehrmals*! Manchmal kommt es zu einem seltsamen Verhalten:

```
Iteration 1: 1.00000 (1.000000)
Iteration 2: 1.00000 (0.000000)
```

Warum tritt das Verhalten auf und wie könnten Sie es *beheben* bzw. zumindest *unwahrscheinlicher* machen?

Debuggen der Lösung

Einen *Lösungsvorschlag* für die vorherige Aufgabe finden Sie in monte-carlo-integration .

- Importieren Sie das *Gradle-Projekt* in Ihre *IDE* und betrachten Sie sich die Lösung.
- Führen Sie das Programm *mehrmals* aus — es wird zu dem oben beschriebenen *seltsamen Verhalten kommen*.
- Verwenden Sie den *Debugger Ihrer IDE* um dem Fehler auf die Schliche zu kommen. Bei der Abnahme des Praktikums werden Sie aufgefordert folgende Aktionen durchzuführen:
  - Setzen Sie einen *Breakpoint* in *Zeile 21* und starten Sie die *Ausführung mit dem Debugger*.
  - Führen Sie das Programm *schrittweise* aus und *springen Sie in die Methode* `samplePoint`, wenn diese aufgerufen wird.
  - Setzen Sie einen *conditional Breakpoint* in Zeile 39, mit der Bedingung `change == 0.0` und führen Sie das Programm fort. Wenn der Breakpoint „triggered“, was sind die Werte der *lokalen Variablen*?
  - *Beheben* Sie das Problem, dass das Integral manchmal *nicht korrekt berechnet wird*.

Aufgabe 3: Zahlenraten

Wir implementieren ein einfaches Zahlenraten-Spiel auf der Konsole. Der Nutzer muss sich eine Zahl denken und wird gefragt, ob die Zahl in verschiedenen Zahlenauflistungen vorkommt. Hier hat sich der Nutzer die Zahl 17 ausgedacht:

```
Wieviele Bits soll die Zahl haben?
5
Denken Sie sich eine Zahl zwischen 0 und 31
Kommt Ihre Zahl in folgender Aufzählung vor? (j/n)
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31
j
Kommt Ihre Zahl in folgender Aufzählung vor? (j/n)
2 3 6 7 10 11 14 15 18 19 22 23 26 27 30 31
n
Kommt Ihre Zahl in folgender Aufzählung vor? (j/n)
4 5 6 7 12 13 14 15 20 21 22 23 28 29 30 31
n
Kommt Ihre Zahl in folgender Aufzählung vor? (j/n)
8 9 10 11 12 13 14 15 24 25 26 27 28 29 30 31
n
Kommt Ihre Zahl in folgender Aufzählung vor? (j/n)
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
j
Die Zahl ist: 17
```

Der Trick ist einfach zu verstehen, wenn man weiß, dass 17 die *Binärdarstellung* 10011 hat und die erste Aufzählung alle Zahlen von 0 bis 31 enthält, die an ersten (*rechtesten*) Stelle eine 1 haben, die zweite Aufzählung an der zweiten Stelle eine 1 usw. Die Antworten des Nutzers, in umgekehrter Reihenfolge, entsprechen der Binärdarstellung: *jnnjj* für 10011.

Los geht's

Erstellen Sie ein neues Java-Projekt und mit der `main`-Klasse `Zahlenraten`:

- *Deklarieren* Sie die `main`-Methode und zwei *lokalen* **`int`**-*Variablen* `bits` und `max`, wobei `bits` die Anzahl der Bits der zu ratenden Zahl ist.
- Lesen Sie `bits` mit Hilfe eines `Scanner`-Objekts ein:

  ```
  Wieviel Bits soll die Zahl haben?
  8
  ```

  Dabei akzeptieren Sie nur Wert $0 < bits <= 8$.
- Setzen Sie `max` auf den Wert 2^{bits}.
- Testen Sie Ihr Programm bis hierhin, indem Sie den Wert von `bits` und `max` ausgeben.

Welche Zahlen haben welche Bits?

Implementieren Sie eine Methode **`private static void`** `printNumbersWithBit(`**`int`** `max, `**`int`** `bit)`, die alle **`int`**-Zahlen von `0` bis `max-1` ausgibt, deren Bit an der Stelle mit der Wertigkeit `bit` („von rechts“) eine `1` stehen haben. Die Ausgabe von `printNumbersWithBit(32, 2)` ist bspw.:

```
4 5 6 7 12 13 14 15 20 21 22 23 28 29 30 31
```

wobei $4 = (100)_2$, $5 = (101)_2$, ..., $30 = (11110)_2$. Implementieren Sie `printNumbersWithBit`! Für den Fall, dass Sie *nicht weiterkommen*:

- In Java gibt es den *Bit-Shift-Operator* `a >> n`, der die *Bits* in `a` um `n` Stellen nach *rechts* verschiebt, bspw. ist `(22 >> 2)== 5` gleich da $22 = (10110)_2$ und damit `6>>2` in Binär $(101)_2 = 5$. Das *Bit an der Stelle* `n` steht nach der Operation *ganz rechts*.
- Um herauszufinden, ob das letzte Bit 0 oder 1 ist können Sie prüfen ob die Zahl *gerade* (0) oder *ungerade* (1) ist.

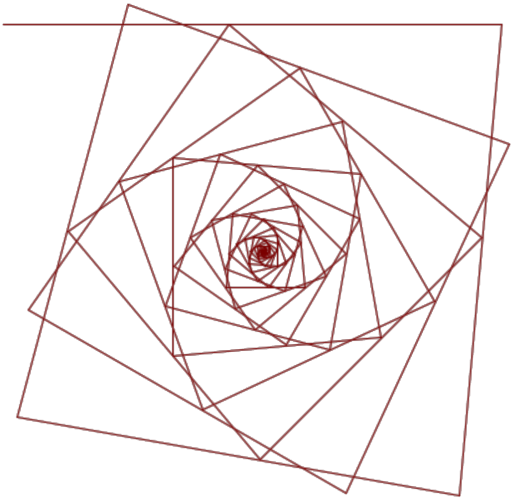
Zahl raten

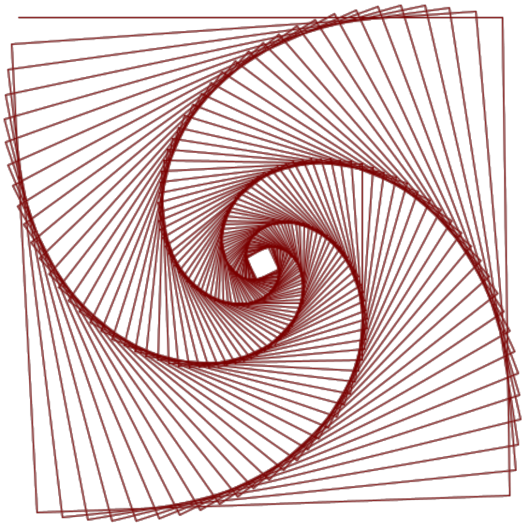
In der `main`-Methode:

- Deklarieren Sie eine *Variable* `int zahl = 0`.
- Implementieren Sie eine Schleife, die den Nutzer für jedes Bit `i` von `0` bis `bit` fragt, ob die gedachte Zahl in der Liste der Zahlen ist, die `printNumbersWithBit(max, i)` ausgibt.
- Wenn ja, dann zählen Sie zu `zahl` die Zahl 2^i dazu.
- Geben Sie `zahl` nach der letzten Frage aus — es handelt sich um die *gedachte Zahl*.

Aufgabe 4: Spiralen

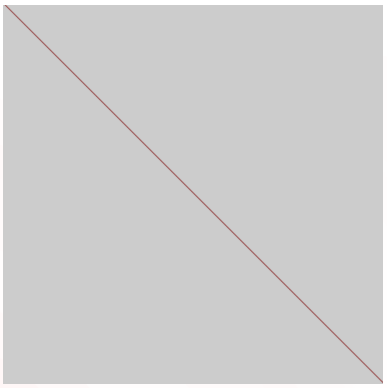
In dieser Übung zeichnen wir Spiralen. Hier zwei Beispiele:





Los geht's

Erstellen Sie ein *neues Java-Projekt* und kopieren Sie die *vorbereitete Java-Datei* spiralen/Spiralen.java in Ihr Projekt. Führen Sie das Projekt aus, sie sollten folgendes Fenster sehen:



Die Methode `drawSpiral`

Im Folgenden werden Sie die Methode `drawSpiral` befüllen — die anderen Methoden müssen/-dürfen Sie nicht verändern. `drawSpiral` besitzt zwei *Argumente*:

- `String[] args` — die *Kommandozeilenargumente* (wie an `main(String[] args)` übergeben)
- `Graphics graphics` — ein `Graphics`-Objekt: Mit diesem Objekt können Sie in das Fenster zeichnen. Im Folgenden benötigen wir nur die Methode `Graphics.drawLine(int x0, int y0, int x1, int y1)`, die eine Linie vom Punkt `(x0,y0)` zum Punkt `(x1,y1)` zeichnet. *Achtung:* Das Koordinatensystem in dem Fenster hat seinen Ursprung *links oben* und eine Größe von `600` mal `600` *Pixel*.

Zur Übung: Ändern Sie `drawSpiral` so, dass eine Linie vom Punkt `(10,10)` zum Punkt `(150,200)` gezeichnet wird.

Um Spiralen zu malen, verwenden wir folgendes Verfahren:

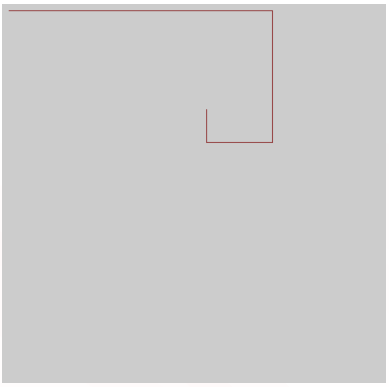
- Gegeben sind folgende *Eingabeparameter*
  - **int** `iterations` — Anzahl der *Wiederholungen*
  - **double** `angle` — Drehwinkel in *Grad* (mehr unten)
  - **double** `shrink` — Verkürzungsfaktor zwischen 0 und 1 (mehr unten)
- Wir starten mit einem Punkt mit Koordinaten `x=20` und `y=20` und einem Vektor mit den Komponenten `dx=400` und `dy=0`.
- Für die Anzahl `iterations` an Wiederholungen:
  - *Zeichne* eine Linie von `(x,y)` nach `(x+dx, y+dy)`
  - *Verschiebe* `(x,y)` nach `(x+dx,y+dy)`
  - *Drehe* den Vektor `(dx,dy)` um den Winkel `angle` mit der Formel:
    - `newDx = dx * Math.cos(Math.toRadians(angle))- dy * Math.sin(Math.toRadians(angle))`
    - `newDy = dx * Math.sin(Math.toRadians(angle))+ dy * Math.cos(Math.toRadians(angle))`

    Warum benötigen wir die Methode `toRadians`?
  - *Multipliziere* `dx` und `dy` jeweils mit `shrink`

Für

- **int** `iterations = 4`
- **double** `angle = 90`
- **double** `shrink = 0.5`

bekommt man folgendes Ergebnis:



Die Spirale beginn *links oben*. Man erkennt die Drehungen um 90° und das halbieren der Länge um die Hälfte nach jedem Schritt.

Einlesen der Parameter

Lesen Sie zunächst die Eingabeparameter aus den *Kommandozeilenargumenten* in `args` aus:

- `args[0]` → **int** `iterations`
- `args[1]` → **double** `angle`
- `args[2]` → **double** `shrink`

Lesen Sie die Kommandozeilenargumente in die drei *lokalen Variablen* `iterations`, `angle` und `shrink` ein.

Sollten Sie *nicht weiterwissen*, wie Sie die Werte aus `args` in die Eingabeparameter einlesen, lesen Sie folgende *Hinweise*:

- Deklarieren Sie `iterations`, `angle` und `shrink` als *lokale Variablen* in `drawSpiral`.

- Bei `args[0]`, usw. handelt es sich um `String`s. Sie können einen **`int`**-Wert aus einem `String` mit der Methode `Integer.parseInt` und einen **`double`**-Wert mit `Double.parseDouble` auslesen.
- Zum Testen, verwenden Sie den *Debugger* und/oder geben Sie Ihre Werte mit `System.out.println` aus!

Spiralen zeichnen

Implementieren Sie das Verfahren wie oben angegeben! Überlegen Sie sich, welche Typen die Variablen `x`, `y`, `dx`, und `dy` haben sollten. Probieren Sie verschiedene Parameter aus!

Sollten Sie nicht weiterkommen, hier ein paar *Hinweise*:

- Implementieren Sie zunächst eine *Schleife*, die genau `iterations`-mal durchläuft, eine Linie von `(x,y)` nach `(x+dx, y+dy)` zeichnet und dann `x` auf `x+dx` und `y` auf `y+dy` setzt. Rufen Sie das Programm mit den *Kommandozeilenargumenten* `4 90 0.5` auf (die beiden letzten Parameter werden *noch* ignoriert).
- Implementieren Sie nun die *Drehung* anhand der *oberen Formel*. Das Ergebnis müsste nun ein Quadrat sein.
- Multiplizieren Sie `dx` und `dy` mit `shrink`. Das Ergebnis auf dem Bild oben, müsste erscheinen.
- Sollten Sie Probleme haben, verwenden Sie den *Debugger* und/oder geben Sie die Werte von `x`, `y`, `dx` und `dy` mit `System.out.println` aus.

Parameterprüfungen

Überlegen Sie sich, welche *ungültigen Werte* der Nutzer *übergeben* kann:

- Was passiert, wenn Sie *nicht alle drei Kommandozeilenargumente* beim Aufruf angeben? Wie können Sie darauf reagieren?
- Was passiert, wenn Sie statt einer Zahl ein *Wort* übergeben? Wie können Sie darauf reagieren?
- Was sind die *gültigen Werte* für die Eingabeparameter?

Implementieren Sie *zumindest eine Prüfung* auf Gültigkeit!