

MEMORY & FILES

Heap, Stack und der Garbage Collector

Struktur und Inhalt des Kurses wurden 2012 von Markus Heckner entwickelt. Im Anschluss haben Alexander Bazo und Christian Wolff Änderungen am Material vorgenommen. Die aktuellen Folien wurden von Alexander Bazo erstellt und können unter der [MIT-Lizenz](#) verwendet werden.

AKTUELLER SEMESTERFORTSCHRITT (WOCHE 12)

Kursabschnitt	Themen		
Grundlagen	Einführung	Einfache Programme erstellen	Variablen, Klassen & Objekte
	Kontrollstrukturen & Methoden	Arrays & komplexe Schleife	
Klassenmodellierung	Grundlagen der Klassenmodellierung	Vererbung & Sichtbarkeit	
Interaktive Anwendungen	Event-basierten Programmierung	String- & Textverarbeitung	
Datenstrukturen	Listen, Maps & die Collections	Speicherverwaltung	Umgang mit Dateien
Software Engineering	Debugging	Planhaftes Vorgehen bei der Softwareentwicklung	
	Qualitätsaspekte von Quellcode		

RÜCKBLICK

- Vorhersehbare Ausnahmefehler werden in Java über Exception-Objekte kommuniziert und müssen in der Regel im Code abgefangen werden.
- Neben *Arrays* und *ArrayList* existieren weitere Datenstrukturen, die z.T. im *Collections*-Framework organisiert sind.
- *HashMaps* bieten die Möglichkeit, *Schlüssel/Wert*-Paare zu speichern und schnell abzurufen.
- Viele - in sich nicht sortierte - Datenstrukturen können über Iterator vollständig iteriert werden.

AUSBLICK AUF DIE NÄCHSTEN WOCHEN

- In der nächsten Woche finden die letzte Vorlesung statt: *Wir werfen einen Blick auf die Klausur und die kommenden Semester*
- Die Klausur findet am 18. Februar von 12:30 bis 14:00 Uhr statt (Weitere Informationen folgen)
- Vielen Dank für die Teilnahme an unserer *Logging*-Studie: Wir würden uns freuen, wenn Sie uns auch in dieser Woche Daten aus den Übungen bereitstellen

Hinweis: Vergessen Sie nicht die **Klausuranmeldung** in Flexnow vom 1. bis zum 9. Februar!

DAS PROGRAMM FÜR HEUTE

- Speicherverwaltung in Java
- Der *Garbage Collector*
- Tipps und Tricks zum Lesen und **Schreiben** von Dateien

SPEICHER AUF IHREM COMPUTER: FLÜCHTIGER SPEICHER

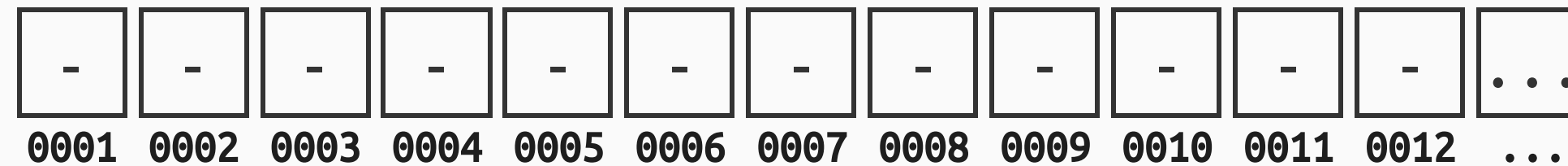
VON WELCHEM SPEICHER SPRECHEN WIR?



Quelle: [https://de.wikipedia.org/wiki/Datei:RAM_Module_\(SDRAM-DDR4\).jpg](https://de.wikipedia.org/wiki/Datei:RAM_Module_(SDRAM-DDR4).jpg)

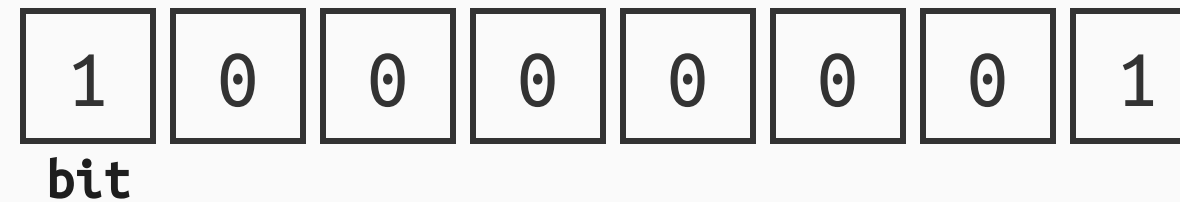
SPEICHERVERWALTUNG IM BETRIEBSSYSTEM UND IN UNSEREN ANWENDUNGEN

Der Speicher ist in Felder aufgeteilt, die jeweils eine bestimmte Menge an Informationen speichern können. Sie sind adressierbar, d.h. hier können (vom OS) Daten gespeichert bzw. gelesen werden.

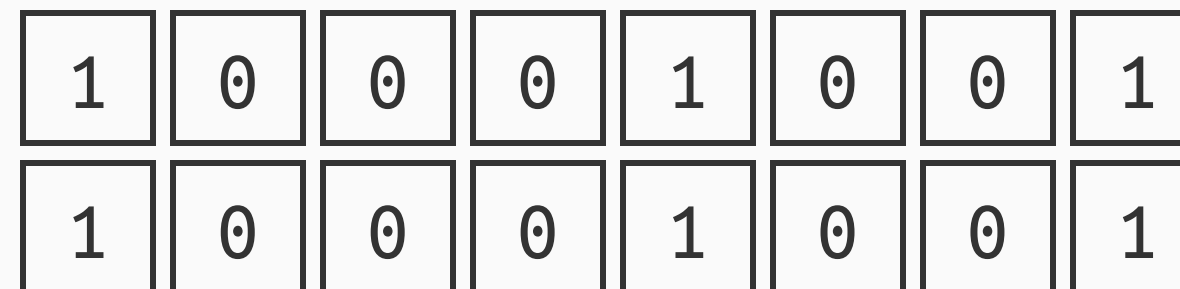


WICHTIGE EINHEITEN IM SPEICHER

BYTE (8 BIT)



WORD (KLEINSTE ADRESSIERBARSTE EINHEIT)



word ist eine Abstrakte Bezeichnung für die kleinste adressierbarste Einheit, d.h. die praktische Größe eines Elements des Speichers, zb. 16bit in einem x86-System. In Java werden 4bytes als Wortgröße verwendet.

GRÖSSENEINHEITEN

Bezeichner	Speichergröße
<i>Kilobyte</i> (KB)	1024 byte (2^{10} byte)
<i>Megabyte</i> (MB)	1024 KB (2^{10} KB)
<i>Gigabyte</i> (GB)	1024 MB (2^{10} MB)
<i>Terabyte</i> (TB)	1024 GB (2^{10} GB)

SPEICHERADRESSEN

Jedes praktisch verwendbare Element des Speicher kann über eine eindeutige Adresse vom Betriebssystem (und Anwendungen) angesprochen werden. Intern werden sowohl für die Adressen als auch für die gespeicherten Werte *binär Darstellungen verwendet*. Für die Repräsentation nach Außen (z. B. für ProgrammiererInnen) wird oft eine alternative Schreibweise gewählt, die *Hexadezimal*-Darstellung:

Binär	0	1																		
Oktal	0	1	2	3	4	5	6	7												
Dezimal	0	1	2	3	4	5	6	7	8	9										
Hexadezimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F				

EXEMPLARISCHER AUFBAU DES SPEICHERS

Adresse	Inhalt (Speicherzelle)
0000	0010
...	
A000	0000
A001	0100
A002	0001
...	
FFFF	0000

Hinweis: Die Menge des verfügbaren Speichers (*Anzahl der Adressen*) und die tatsächliche Größe einer Speicherzelle hängt vom verwendeten System ab.

DER SPEICHER IN UNSEREN ANWENDUNGEN

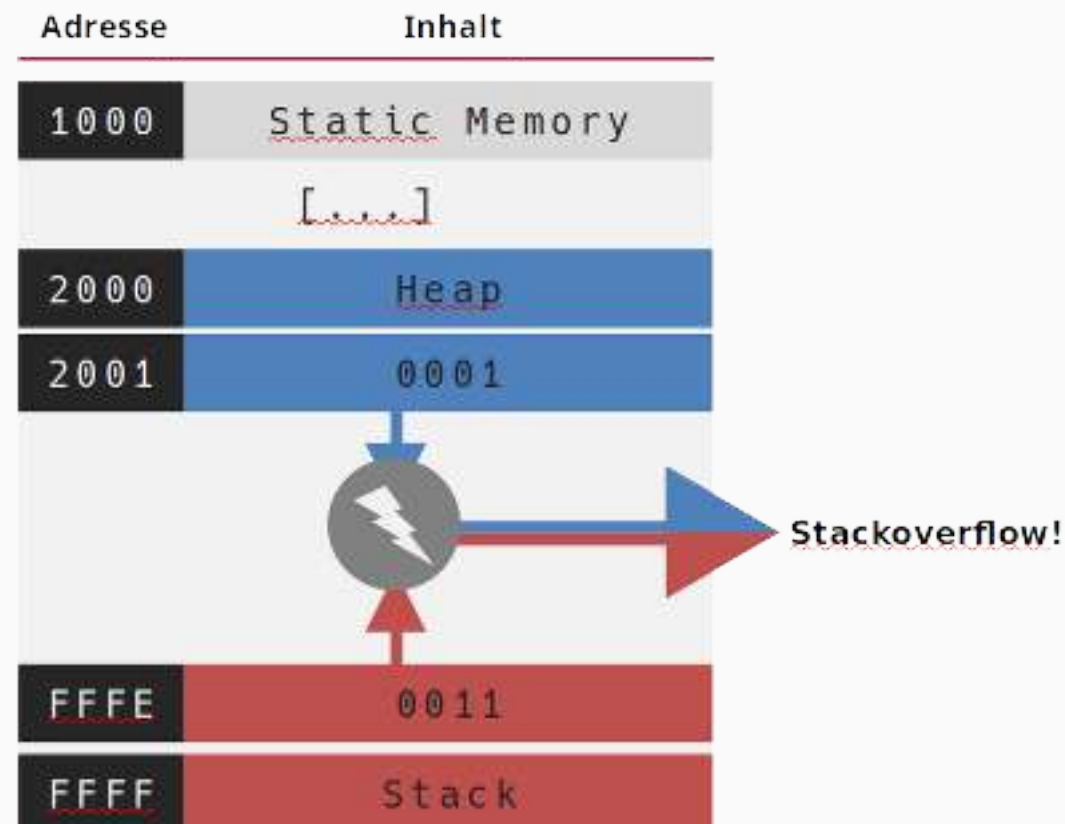
Beim Starten einer Java-Anwendung wird ein bestimmter Bereich im Arbeitsspeicher für diese Anwendung reserviert. **Dies ist der verfügbare Speicher.** Dieser Speicherbereich unterteilt sich in:

Static Memory: Hier werden statische Variablen und Konstanten abgelegt.

Heap: Hier werden dynamische Variablen, d.h. alle Objekte und deren Instanzvariablen abgelegt. Der *Heap* wird von der *Runtime* verwaltet. Ungenutzte Objekte, die keinen Bezug mehr zur Anwendung haben, werden *automatisch* gelöscht.

Stack: Hier werden lokale Variablen von Methoden und Parameter gespeichert.

STACKOVERFLOW!



Der *Heap* wächst zur Laufzeit nach "unten". D.h. die verwendeten Adressen werden "größer". Der *Stack* wächst nach "oben". D.h. die verwendeten Adressen werden kleiner.

Heap und *Stack* bewegen sich im vorher reservierten Speichereich. Sobald der *Stack* versucht, in den vom *Heap* verwendeten Bereich zu schreiben, wird ein Stackoverflow-Fehler ausgelöst.

Die genauen Positionen von *Heap* und *Stack* im Speicher sind nicht festgelegt. Die meisten *Runtime*-Implementierungen verfolgen aber dieses Schema.

SPEICHER UND DATENTYPEN

Unterschiedliche Datentypen benötigen unterschiedlich viel Speicher:

- int: 4 byte
- double: 8 byte
- char: 2 byte (Ein char ist kleiner als die Java-*Word*-Größe!)

Bei dynamischen Datentypen (Objekten) wird neben den eigentlichen Daten (z.B. Instanzvariablen) auch ein Overhead-Speicher benötigt.

DIE KLASSE POINT

```
1 public class Point {  
2     /* instance variables */  
3     private int px;  
4     private int py;  
5  
6     public Point(int x, int y) {  
7         px = x;  
8         py = y;  
9     }  
10  
11     public void move(int dx, int dy) {  
12         px += dx;  
13         py += dy;  
14     }  
15 }
```


UNSERE DEMO-ANWENDUNG

```
1 public class DemoApp extends GraphicsApp {  
2  
3     // ...  
4  
5     public void initialize() {  
6         Point p1 = new Point(5, 4);  
7         Point p2 = new Point(2, 3);  
8         p1.move(10, 15);  
9     }  
10  
11 }
```

[zurück](#)

Speicherzuweisung

[weiter](#)

Heap

Stack

ERZEUGEN UND ZUWEISEN VON OBJEKTEN

```
1 Point p1 = new Point(1, 2);  
2 Point p2 = p1;  
3 p2.move(10, 10);
```

Eine Zuweisung eines existierenden Objekts zu einer neuen Variable ändert nur den *Zeiger*. Die Variablen auf dem *Stack* zeigen auf das selbe Objekt, d.h. in diesem Fall, dass auch p1 durch die move-Methode verändert wird.

STRINGS VERGLEICHEN

```
1 String s1 = new String("hello");  
2 String s2 = new String("hello");  
3 System.out.println(s1 == s2);
```

Wir vergleichen hier die Adressen der beiden Objekte auf dem Heap, nicht den Inhalt der beiden String-Objekte. Daher gibt der Ausdruck auch `false` zurück (Beide Objekte wurden durch den expliziten Konstruktoraufbau als separate Instanzen erstellt).

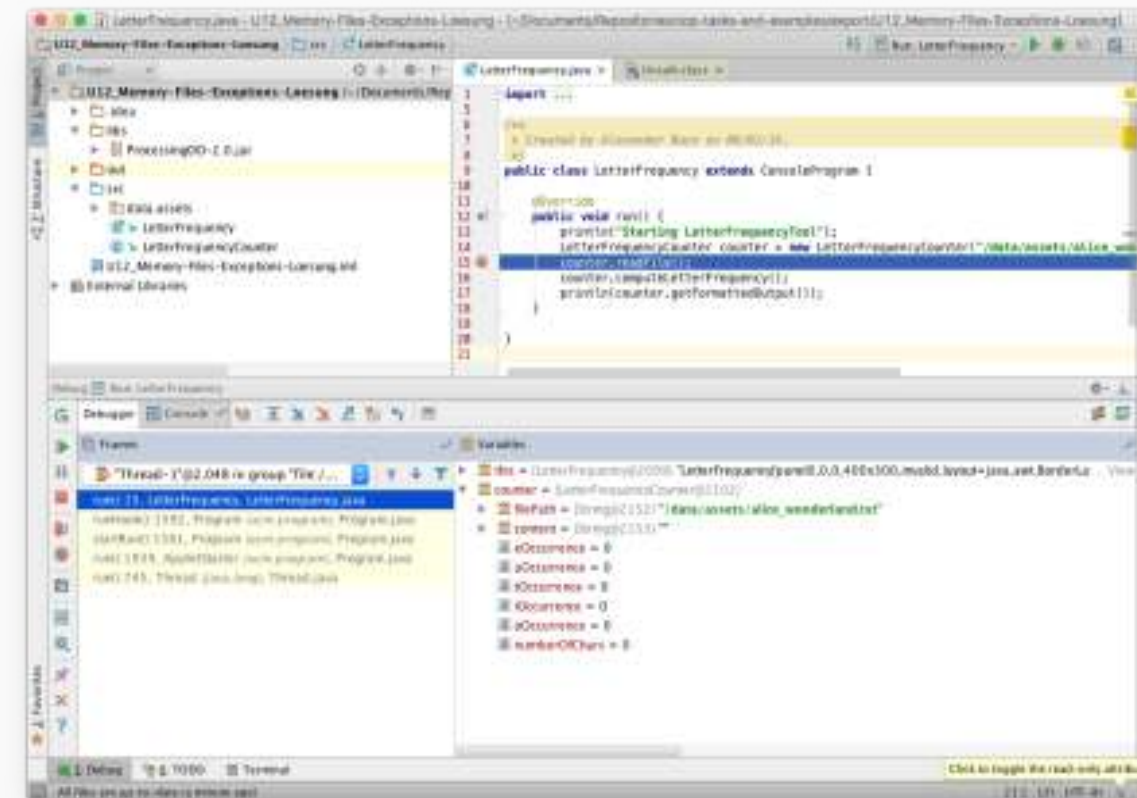
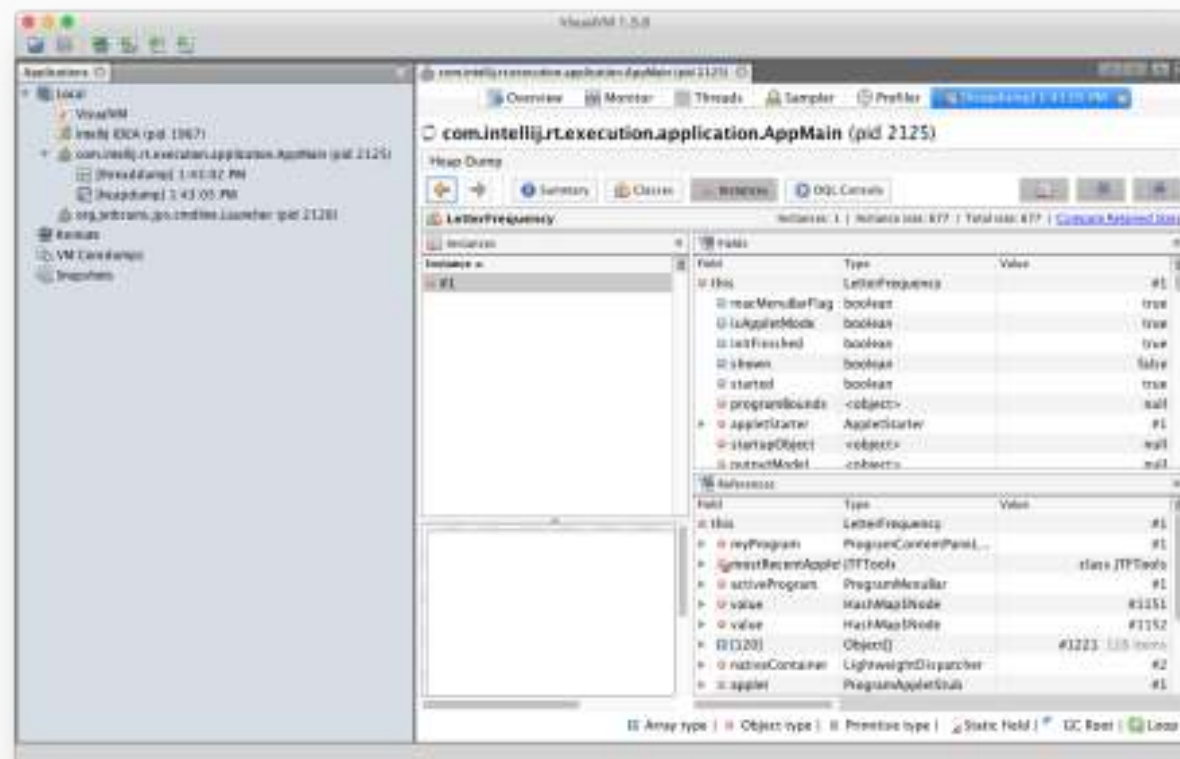
NULLPOINTER

```
1 Point p3;  
2 p3.move(9,7);
```

Für die Variable p3 wird auf dem Stack eine Speicherzelle reserviert. Diese ist solange "leer" (`null`) bis ein konkretes Objekt erstellt und dessen *Heap*-Adresse eingetragen wird. Greifen wir vorher auf die leere Speicherzelle zu, wird eine `NullPointerException` ausgelöst: p3 verweist auf **nichts**

EINBLICKE IN HEAP UND STACK

Mit dem eingebauten Debugger können Sie in IntelliJ den aktuellen Zustand des Heaps einsehen. Alternativ können Sie laufende Java-Anwendungen auch mit externen Tools wie **VisualVM** analysieren.



DER GARBAGE COLLECTOR

Was passiert mit *Heap* und *Stack*, wenn die dort gespeicherten Daten nicht mehr benötigt werden?



VERWAISTE EINTRÄGE IM HEAP

Zur Laufzeit kann der Fall eintreten, dass *Reste* im *Heap* zurückbleiben:

1. In einer Methode (*stack frame*) wird ein Objekt referenziert.
2. Wird die Methode beendet, wird die Referenz aus dem Stack entfernt.
3. Die Inhalte im *Heap* bleiben vorerst (unreferenziert) bestehen.
4. Der *Garbage Collector* der *Runtime* sucht regelmäßig nach *Heap*-Inhalten, die von keiner Stelle des Programms referenziert werden.

Findet der *Garbage Collector* entsprechende Daten-Reste, werden diese automatisch entfernt.

DATEIEN ALS MÖGLICHKEIT ZUR DAUERHAFTEN SPEICHERUNG VON INFORMATIONEN

VON WELCHEM SPEICHER SPRECHEN WIR?



DATEIEN

- Dateien speichern Inhalte (Informationen) in codierter, maschinenlesbarer Form
- Die Struktur der Informationen können vom Computer unterschiedlich interpretiert werden: Bilder, Musik, Text, ...
- Dateien sind die Basis für die (praktische) persistente Speicherung von Daten über die Laufzeit eines Programms hinaus (wenn man keine Datenbank oder ein anderes Datenmanagement-System benutzt – letztlich speichert aber auch eine Datenbank ihre Inhalte in Dateien)

VERWENDUNG VON DATEIEN

Dateien (und deren Inhalte) werden von vielen Programmen genutzt:

- Textverarbeitung (Textdokumente)
- IDEs wie IntelliJ (Quellcode: *.java-Dateien)
- Browser (Cookies)
- Spiele (Speicherstände)
- Mediaplayer (Musik- und Videodateien) ...

WO HABEN SIE BEREITS (IMPLIZIT) MIT DATEIEN GEARBEITET?

DATEIEN IN JAVA

Dateien werden in Java über die Klasse `File` repräsentiert, in deren Konstruktor Sie den (relativen oder absoluten) Pfad zur bestehenden oder neu anzulegenden Datei angeben können:

```
1 File fileA = new File("\data\assets\file.txt");  
2 File fileB = new File("C:\data\file.txt");
```

Diese Objekte sind die Grundlage für Funktionen wie z.B. das Einlesen oder Schreiben von Dateien.

DATEIEN EINLESEN: SCANNER

Wir kennen bereits eine einfache Möglichkeit, Inhalte aus Dateien einzulesen: die Scanner-Klasse:

```
1 Scanner in = new Scanner(new File("input.txt"));
2 while (in.hasNext()) {
3     System.out.println.add(in.nextLine());
4 }
```

Intern arbeitet die Klasse mit *Streams* (Datenströmen) und Reader-Klassen, die die Grundlagen für die Datenverarbeitung in Java bilden. Es lohnt sich, im Kontext der Dateiverarbeitung, einen Blick auf diesen Bereich zu werfen, da auch andere Probleme in Java (z.B. die *Netzwerkkommunikation* über die gleichen Mechanismen gelöst werden.

DAS PAKET JAVA.IO

Das Paket `java.io` enthält die Datenstrukturen, die eine Behandlung von Datenströmen, insbesondere Dateiein- und -ausgabe ermöglichen. Dazu gehören u.a. die folgenden Basisklassen:

- `Reader`
- `Writer`
- `InputStream`
- `OutputStream`

Grundsätzlich wird unterschieden in gepufferte und nichtgepufferte Ein- und Ausgabeströme für *byte*- oder *character*-Daten.

DIE ARBEIT MIT DEM BUFFEREDREADER

```
1 // Datei öffnen
2 File file = new File("data.txt")
3 BufferedReader br = new BufferedReader(new FileReader(file));
4
5 // Zeilen auslesen
6 // Der interne Zeiger in br zeigt immer auf eine Zeile
7 String line1 = br.readLine();
8 // readLine() gibt die aktuelle Zeile zurück und setzt den Zeiger auf die nächste
9 String line2 = br.readLine();
10 String line3 = br.readLine();
11 String line4 = br.readLine();
12 // Kann keine Zeile (mehr) gelesen werden, gibt readLine() null zurück
13 String line5 = br.readLine(); // Gibt null zurück
14
15 // Datei schließen
16 br.close();
```

DATEIEN VOLLSTÄNDIG AUSLESEN

```
1 File file = new File("data.txt")
2 BufferedReader br = new BufferedReader(new FileReader(file));
3
4 while (true) {
5     String line = br.readLine();
6     if (line == null) { break; }
7     // Hier kann die Zeile verarbeiten werden
8 }
9
10 br.close();
```

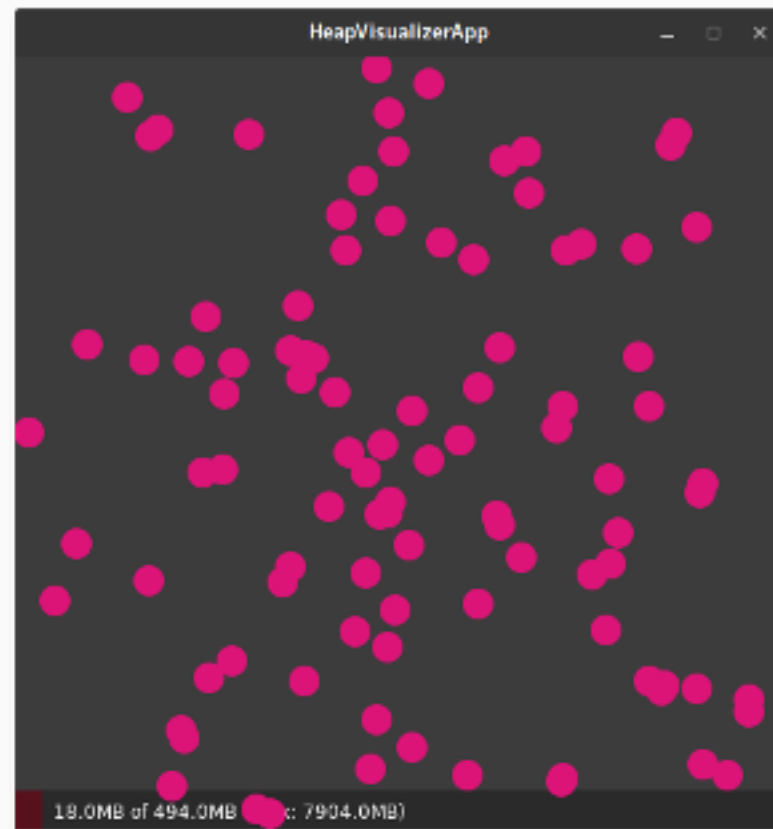
DATEIEN SCHREIBEN

Das Schreiben in Dateien funktioniert nach dem gleichen Prinzip, wird aber durch den Einsatz anderer Klassen, z.B. `FileWriter` realisiert.

```
1 File file = new File("data.txt")
2
3 try{
4     FileWriter fw = new FileWriter(file);
5     fw.write("Welcome to javaTpoint.");
6     fw.close();
7 } catch(Exception e){
8     e.printStackTrace();
9 }
```

Hinweis: Öffnen Sie eine existierende Datei zum Schreiben, wird deren Inhalt permanent überschrieben. Seien Sie vorsichtig, wenn Sie das nicht wollen! Schauen Sie sich die `FileWriter`-Klasse an, wenn Sie wissen möchten, wie Sie Inhalte an Dateien anhängen können.

WALKTHROUGH: HEAP VISUALIZER



Eine beliebige Anzahl an Bällen wird in einer ArrayList gespeichert und auf dem Bildschirm dargestellt. Die Bälle bewegen sich mit zufälliger Geschwindigkeit in jedem Tick auf der x- und y-Achse. Kollidiert ein Ball mit einem anderen, wird dessen Bewegungsrichtung umgekehrt. Das Kollisionsziel bewegt sich mit der ursprünglichen Richtung des Balls fort. Verlässt ein Ball die Zeichenfläche, wird er entfernt. In jedem Tick werden alle "freien" Stellen der Liste mit einer Wahrscheinlichkeit von 0.5 aufgefüllt.

KONFIGURATION ÜBER EXTERNE DATEI (1/2)

Zentrale Parameter der Anwendung werden über eine Textdatei verwaltet, die bei Start der Anwendung eingelesen wird. Die Simulation kann dadurch angepasst werden, ohne dass Änderungen am Code notwendig wären.

```
1 WIDTH:500
2 HEIGHT:500
3 BACKGROUND_COLOR:60,60,60
4 MAX_BALLS:100
5 BALL_RADIUS:10
6 BALL_MIN_VELOCITY:-3
7 BALL_MAX_VELOCITY:3
8 BALL_COLOR:220,20,120
```

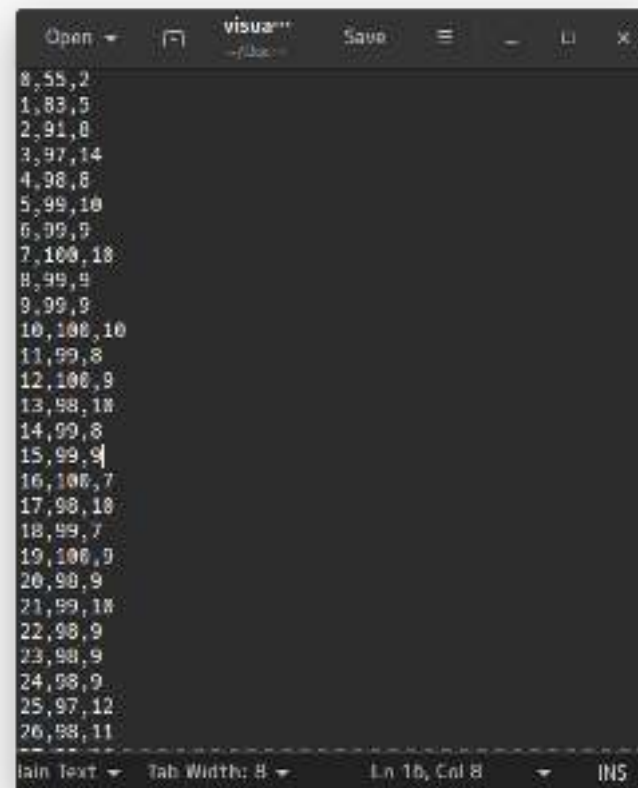
KONFIGURATION ÜBER EXTERNE DATEI (2/2)

Zur Laufzeit wird der Inhalt der Konfigurationsdatei durch eine Instanz der Klasse AppConfig repräsentiert.

```
1 // Einlesen der Datei
2 private void initConfig() {
3     File configFile = new File(CONFIG_FILE_PATH);
4     config = new AppConfig(configFile);
5     if(!config.isValid()) {
6         System.out.println("Invalid config file!");
7         System.exit(1);
8     }
9 }
10
11 // Zugriff auf Parameter
12 private void initHeapVisualizer() {
13     heapVisualizer = new HeapVisualizer(config.getWidth(), config.getHeight());
14 }
```

LOGGING IN DATEI (1/2)

Für eine vorgegebene Anzahl an *Frames* werden jeweils die Anzahl der Bälle sowie die registrierten Kollisionen gespeichert und in eine Log-Datei geschrieben. Die Log-Einträge werden im Speicher gesammelt und erst dann in die Datei geschrieben, wenn die vordefinierte Anzahl an Einträgen erreicht wurde



```
0,55,2
1,83,5
2,91,8
3,97,14
4,98,8
5,99,10
6,99,9
7,100,18
8,99,9
9,99,9
10,100,10
11,99,8
12,100,9
13,98,10
14,99,8
15,99,9
16,100,7
17,98,10
18,99,7
19,100,9
20,98,9
21,99,10
22,98,9
23,98,9
24,98,9
25,97,12
26,98,11
```




LOGGING IN DATEI (2/2)

```
1 private File logFile;
2 private ArrayList<String> log;
3
4 public void logFrame(int ballCount, int collisions) {
5     if(loggedFrames >= framesToLog) {
6         if(!fileWritten) { writeLog(); }
7         return;
8     }
9     log.add(loggedFrames + DELIMITER + ballCount + DELIMITER + collisions);
10    loggedFrames++;
11 }
12 private void writeLog() {
13     try {
14         PrintWriter fileWriter = new PrintWriter(new FileWriter(logFile));
15         for(String line: log) { fileWriter.println(line); }
16         fileWriter.close();
17         fileWritten = true;
18     } catch (IOException e) {
19     }
20 }
```


ZUSAMMENFASSUNG

- Speicher in Java unterteilt sich in *Heap* und *Stack*: Auf dem Stack werden lokale Variablen und Parameter abgelegt.
- Alle Objekte (komplexe Typen) liegen auf dem *Heap*.
- Der Stack verweist auf komplexe Datentypen auf dem *Heap* ("Pointer"), auch der *Heap* kann auf andere Objekte auf dem *Heap* verweisen.
- Der *Heap* wächst "von oben", der *Stack* "von unten" – Treffen sich beide in der Mitte, entsteht ein *Stack Overflow*.
- Der *Garbage Collector* räumt nicht mehr referenzierte Objekte auf und hilft somit *Stack Overflows* zu vermeiden und den Speicherbedarf eines Programms gering zu halten.

VIELEN DANK FÜR IHRE AUFMERKSAMKEIT. WENN SIE MÖCHTEN, SEHEN WIR UNS IM ANSCHLUSS IN DER ZENTRALÜBUNG!

Fragen oder Probleme? In allgemeinen Angelegenheiten und bei Fragen zur Vorlesung wenden Sie sich bitte an Alexander Bazo (alexander.bazo@ur.de ). Bei organisatorischen Fragen zu den Studienleistungen und Übungsgruppen schreiben Sie bitte Florin Schwappach (florin.schwappach@ur.de ). Bei inhaltlichen Fragen zu den Übungen, Beispielen und Studienleistungen schreiben Sie uns unter mi.oop@mailman.uni-regensburg.de .

QUELLEN

Eric S. Roberts, *The art and science of Java: an introduction to computer science, New international Edition*, 1. Ausgabe, Pearson, Harlow, UK, 2014.