Fernuniversität Hagen, Informatik

Kurs 1618 – Einführung in die objektorientierte Programmierung

Klausurvorbereitung

Lernheft

Till Menke
Kosselstr. 4 – 69115 Heidelberg – Tel. 06221/3538228

Fertigstellung: 15.07.2018
http://link.tillmenke.de/20180715-1

Inhaltsverzeichnis

1 theoretische Grundlagen 5
2 Java-Kontrollstrukturen 6
2.1 Parallelität 11
2.2 konzeptuelle Entscheidungen in Java 11
3 Streams am Beispiel des Dateizugriffs 12
4 Netzwerk 13
4.1 Kommunikation über Sockets 13
4.2 Kommunikation über RMI 13
4.2.1 auf beiden Seiten notwendig verfügbare Elemente 13
4.2.2 nur auf Serverseite notwendig verfügbare Elemente 13
4.2.3 Verwendung auf Clientseite 13
5 Fenster mit AWT 14
5.1 Grundstruktur 14
5.2 mögliche Ereignisse der Komponenten 15
5.3 Layouts 15

1 theoretische Grundlagen

- Unterscheidung Kovarianz/Kontravarianz
    - **Kovarianz**: nach unten in der Klassenhierarchie ist alles erlaubt
    - **Kontravarianz**: nach oben in der Klassenhierarchie ist alles erlaubt
    - Zugrundeliegendes Prinzip heißt „Liskovsches Substitutionsprinzip“
- Arten der Polymorphie (nach der Terminologie des Kurses)
    - **Subtyppolymorphie**: Subtyp kann anstelle des Supertyps verwendet werden
        - bis Java 5 einzige Art der Polymorphie
        - Problem: Basistypen nicht in obersten Supertyp „Object“ enthalten → Notwendigkeit von Wrapperklassen (sind als Java-Standardklassen mit demselben Namen wie die Basisdatentypen vorhanden, bloß großgeschrieben (Ausnahme: int wird zu Integer); Java packt wenn notwendig automatisch Basisdatentypen in solch eine Klasse ein oder aus)
        - Problem: Compiler weiß bei Abrufen eines Werts nicht, welche konkrete Klasse das Objekt hat → Cast notwendig → keine Typsicherheit
    - Lösung: **parametrische Polymorphie**: Verwender einer Klasse soll Typ (z. B. für Behälter) selbst festlegen dürfen
        - seit Java 7 darf Typparameter hinten weggelassen werden, wenn durch Deklarationstyp klar (z. B. Musterklasse<> statt Musterklasse<Mustertyp>)
        - Problem: durch Klasse bestimmte Methoden (z. B. compareTo bei Sortable-Klassen im Maximum-Container) können nicht aufgerufen werden
        - im Bytecode meist wie Rawtypen umgesetzt (z. B. mit typeof-Prüfungen), um Abwärtskompatibilität weitestmöglich zu erhalten
    - Lösung: **beschränkte parametrische Polymorphie**: <T(ypparameter) extends Superklasse>
        - nicht: implements bei Interface, da extends nach Java-Logik Oberbegriff, implements als Spezialfall für Klasse
        - zur Erweiterung von Subtypklassen in der Standardbibliothek wurden im Sinne der Abwärtskompatibilität Rawtypen ermöglicht: ein Aufruf ohne Typparameter führt zur Initialisierung mit dem Typ Object (aber in manchen Compiler-Versionen zu einer Warnung) → Loch im Typsystem (neben null, Casts als explizite Aufhebung der Typkontrolle und der Situation der ArrayStoreException)
        - Fall der ArrayStoreExcpetion wird hier anders als bei Arrays schon durch Aufgabe der Subtypbeziehung verhindert → Problem: gemeinsamer Supertyp aller Subtyplisten → Lösung: Wildcards (z. B. List<? extends Supertyp>) für diesen Fall
            - Elementtyp nicht bekannt → keine Zuweisung möglich, da nicht klar, welche der Subklassen im konkreten Fall verlangt
            - Abrufen: Compiler kennt Typ nicht → Zuweisung nur an Variablen mit Supertyp möglich
            - Alternative: Begrenzung nach oben (z. B. List<? super Subtyp>) → mehr Zuweisungen möglich, aber stärkere Begrenzungen beim Abrufen (z. B. Zuweisung des Abgerufenen nur an Variablen mit Deklarationstyp Object als generellen Supertyp)
    - **Überladen/ad-hoc-Polymorphie** (eigentlich keine echte Polymorphie, da es sich um verschiedene Methoden mit demselben Namen handelt → nur Methodenname wird doppelt verwendet, bei Verständnis dieser Dopplung als Polymorphie müsste man diesen als Programmstruktur sehen, da kein sonstiger Inhalt mehrfachverwendet wird)
- **Delegation** als Alternative zum Subtyping: Objekt hält sich ein Objekt der Klasse mit dem gewünschten Code und schreibt Methoden, die die gewünschten Methoden dieses Objekts ausführen → Vorteil: nicht gewünschte Methoden gibt es nicht, Code einfacher austauschbar

2 Java-Kontrollstrukturen

```
1.  package de.tillmenke.studium.informatik.kurs1618.klausurvorbereitung;
2.
3.  import java.util.Iterator;
4.
5.  public class Struktur {
6.
7.      public static void main(String[] args) {/*Main-
    Methode als Programmeinstiegspunkt*/
8.          int variable1 = 1;
9.          if (variable1 > 0) {/* für Wahrheitswerte möglich: logische Operatoren & (un
    d), | (oder), ^ (ausschließendes oder); als && oder || wird ggf. 2. Teil nicht ausge
    führt (sinnvoll für if-Bedingungen) */
10.             System.out.println("Variable 1 ist positiv");
11.             System.out.print("wirklich!");
12.         } else if (variable1 == 0) {
13.
14.         } else {
15.             System.out.println("Variable 1 ist negativ");
16.             System.out.println();
17.         }
18.
19.         /* verkürzte if-Schleife als Ausdruck */
20.         boolean boolvar = true;
21.         String boolstring = boolvar ? "wahr" : "falsch";
22.         System.out.println(boolstring);
23.
24.         /* Laufanweisung (for-Schleife), zugleich: Arraydeklaration */
25.         int[] array = new int[5]; /* leeres Array */
26.         array = new int[] { 1, 2, 3 };
27.         for (int i = 0; i < array.length; i++) {
28.             System.out.println(array[i]);
29.         }
30.         /* Kurzschreibweise für von Interface Iterable ableitende Klassen */
31.         for (int arrayvar : array) {
32.             System.out.println(arrayvar);
33.         }
34.
35.         /* Kopfschleife (Ausführung nur, wenn Schleifenbedingung erfüllt) */
36.         int j = 2;
37.         while (j != 0) {
38.             j--;
39.             System.out.println(j);
40.         }
41.         /* Fußschleife (Ausführung einmal, danach nur, wenn Schleifenbedingung erfül
    lt) */
42.         do {
43.             j++;
44.             System.out.println(j);
45.         } while (j != 2);
46.
47.         /* Switch */
48.         int month = 2;
49.         switch (month) {
50.         case 1:
51.         case 3:
52.         case 5:
53.         case 7:
54.         case 8:
55.         case 10:
56.         case 12:
57.             System.out.println("31 Tage");
58.             break;
59.         case 2:
60.             System.out.println("28 oder 29 Tage");
61.             break;
```

```
62.         case 4:
63.         case 6:
64.         case 9:
65.         case 11:
66.             System.out.println("30 Tage");
67.             break;
68.         default:
69.             System.out.print("Ungültiger Monat!");
70.             break;
71.         }
72.
73.         Behälter<? extends Superinterface1> bA = new Behälter<Subinterface>();
74.         Behälter<? super Subinterface> bB = new Behälter<>(); /* Kurzschreibweise hi
    nten seit Java 7 */
75.         bA = bB;
76.         bA.toString();
77.
78.
79.         /* Threading */
80.         Thread myThread = new MyThread();
81.         myThread.start();
82.
83.         Runnable target = new SubklassenThread();
84.         Thread myThread2 = new Thread(target);
85.         myThread2.start();
86.         /* Probleme der Synchronisation von Threads werden außerhalb des Codebeispie
    ls behandelt */
87.
88.         /* Rückruffunktionen (am Beispiel eines Filters) */
89.         Filter<String> filter = new Filter<>(new java.util.ArrayList<String>());
90.         /*klassische Implementierung, egal ob globale oder lokale (Sub-)Klasse*/
91.         java.util.List<String> result = filter.filter(new FilterPredicateImpl<String
    >()); /*Implementierung auch als lokale Klasse möglich; lokale Klassen werden am Bei
    spiel von AWT gezeigt (s. u.)*/
92.         /*Implementierung mit anonymer Klasse, Übersetzung (Verzeichnis „bin“) als e
    igene Datei, die durchnummeriert wird, z. B. BeinhaltendeKlasse$1.class */
93.         result = filter.filter(new FilterPredicate<String>() {
94.             @Override
95.             public boolean isMatching(String b) {
96.                     return b.toString().length() > 5;
97.             }
98.         });
99.         /*Implementierung mit Methodenreferenz (ab Java 8)*/
100.        result = filter.filter(b -> b.toString().length() > 5); /*Filter-
    Interface muss ein Interface mit genau einer Methode sein, damit Kurzschreibweise er
    laubt*/
101.        System.out.println("Es verbleiben "+result.size()+" Textstränge.");
102.    }
103.
104.    int teilen(int m, int n) throws Exception {
105.        try {
106.            int ergebnis = m / n;
107.            return ergebnis;
108.        } catch (ArithmeticException e) {
109.            System.err.println("Division durch null.");
110.            throw new Exception(
111.                    "Division durch null."); /* muss deklariert werden, da nicht Sub
    klasse von RuntimeException */
112.        } finally {
113.            System.out.println("Programmende.");
114.        }
115.    }
116.
117.    enum Musteraufzählung {
118.        A, B, C
119.    }
120.
```

```
121.     enum MusteraufzählungMitFunktionen {
122.         A(1) {
123.             @Override
124.             public void Musterfunktion() {
125.             }
126.         },
127.         B {
128.             @Override
129.             public void Musterfunktion() {
130.             }
131.         },
132.         C {
133.             @Override
134.             public void Musterfunktion() {
135.             }
136.         };
137.
138.         MusteraufzählungMitFunktionen() {
139.
140.         }
141.
142.         MusteraufzählungMitFunktionen(int i) {
143.             // i kann verarbeitet werden
144.         }
145.
146.         public abstract void Musterfunktion();
147.
148.         // Rest des Klassenrumpfs
149.     }
150.}
151.
152.abstract class Superklasse {
153.     abstract void zuImplementieren();
154.}
155.
156.interface Superinterface1 {
157.     int publicfinalstaticint = 5; /*bei Interfaces andere Standardmodifikatoren*/
158.
159.     String getName();
160.}
161.
162.interface Superinterface2 {
163.     int getNumber();
164.}
165.
166.interface Subinterface extends Superinterface1, Superinterface2 {
167.}
168.
169.class Subklasse extends Superklasse implements Superinterface1, Subinterface {
170.     /*Klassen als Muster für Objekt statt Prototyp wie in JavaScript*/
171.     private int lokal;
172.
173.     @SuppressWarnings("hiding")
174.     Subklasse(int lokal) {
175.         super(); /* ermöglicht Aufruf eines bestimmten Superkonstruktors */
176.         this.lokal = lokal; /* ermöglicht Zugriff auf verdeckte Variable */
177.     }
178.
179.     @Override
180.     public String getName() {
181.         return "Sub";
182.     }
183.
184.     @Override
185.     public int getNumber() {
186.         return lokal;
187.     }
```

```
188.
189.     @Override
190.     void zuImplementieren() {
191.
192.     }
193.
194.     class DynamischeInnereKlasse {
195.         int dynamischerZugriff() {
196.             return Subklasse.this.lokal; /* Zugriff auf nicht statische Elemente der
     äußeren Klasse möglich, da keine statische innere Klasse; Kapselung nicht innerhalb
     einer Klasse */
197.         }
198.     }
199.}
200.
201.class Behälter<T extends Superinterface1>
202.         implements Iterable<T> { /* nicht: implements bei Interface als Typschranke,
     da extends nach Java-Logik Oberbegriff, implements als Spezialfall für Klasse */
203.     private T content;
204.     private int size = 1;
205.
206.     T getContent() throws ContentNotFoundException {
207.         return content;
208.     }
209.
210.     class ListIterator implements Iterator<T> {/*Aufruf von außen: Behälter.new List
    Iterator();*/
211.         private int nextIndex = 0;
212.
213.         @Override
214.         public boolean hasNext() {
215.             return nextIndex != size;
216.         }
217.
218.         @Override
219.         public T next() {
220.             if (nextIndex == size) { throw new java.util.NoSuchElementException(); }

221.             T elem = content;
222.             nextIndex++;
223.             return elem;
224.         }
225.     }
226.
227.     @Override
228.     public Iterator<T> iterator() {
229.         return new ListIterator();
230.     }
231.}
232.
233.class ContentNotFoundException extends Exception {
234.}
235.
236.class MyThread extends Thread {
237.     @Override
238.     public void run() {
239.         // Hier steht der Code, der in einem eigenen
240.         // VM-Thread ausgefuehrt werden soll.
241.     }
242.}
243.
244.class SubklassenThread extends Superklasse implements Runnable {
245.     @Override
246.     public void run() {
247.         zuImplementieren();
248.     }
249.
```

```
250.    @Override
251.    void zuImplementieren() {
252.
253.    }
254.}
255.
256.class Filter<T> {
257.    private java.util.List<T> sourceList;
258.
259.    @SuppressWarnings("hiding")
260.    public Filter(java.util.List<T> sourceList) {
261.        this.sourceList = sourceList;
262.    }
263.
264.    java.util.List<T> filter(FilterPredicate<T> filterPredicate) {
265.        java.util.ArrayList<T> resultList = new java.util.ArrayList<T>();
266.        for (T elem : sourceList) {
267.            if (filterPredicate.isMatching(elem)) {
268.                resultList.add(elem);
269.            }
270.        }
271.        return resultList;
272.    }
273.}
274.
275.interface FilterPredicate<T> {
276.    public boolean isMatching(T b);
277.}
278.
279.class FilterPredicateImpl<T> implements FilterPredicate<T> {
280.    @Override
281.    public boolean isMatching(T b) {
282.        return b.toString().length() > 5;
283.    }
284.}
```

Abbildung 1: Übersicht über Java-Kontrollstrukturen in Beispielform

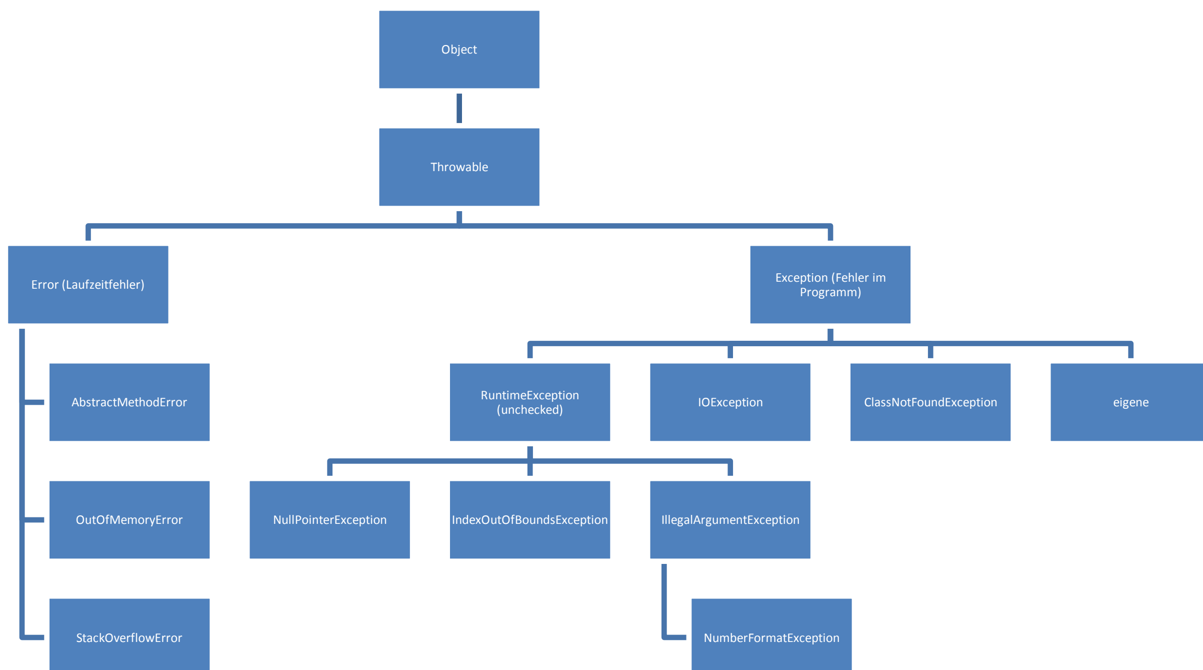


Abbildung 2: Klassenhierarchie der Standardausnahmen

2.1 Parallelität

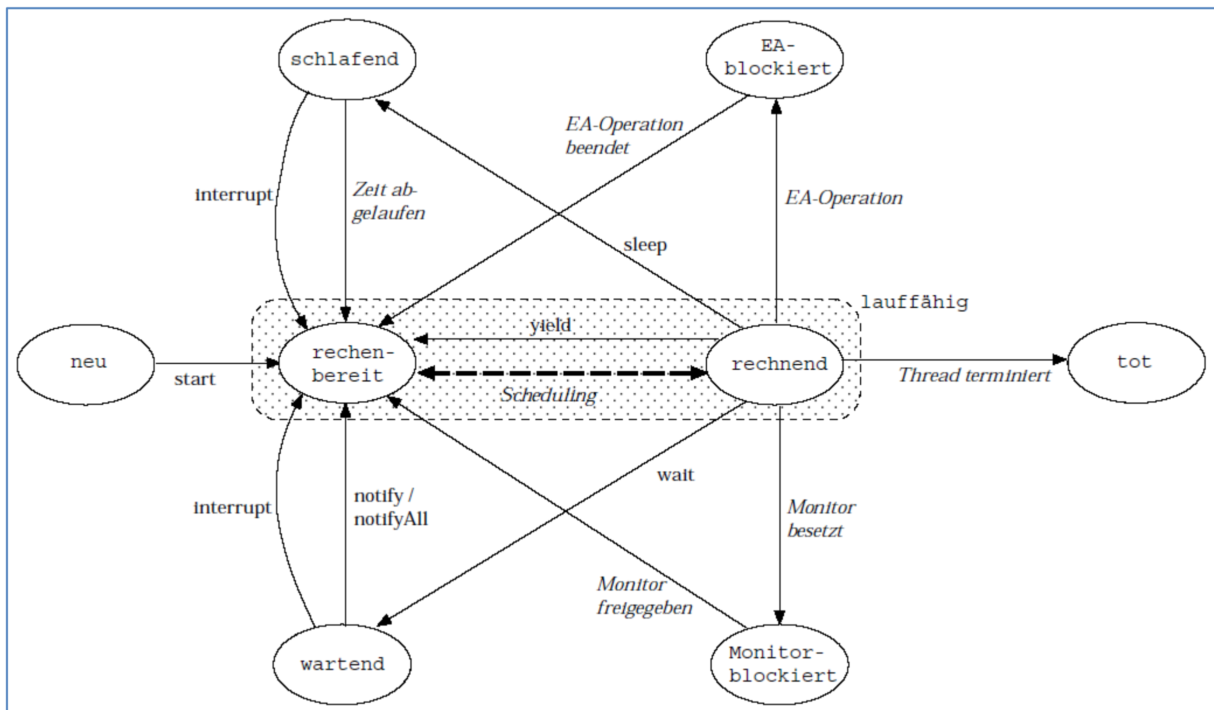


Abbildung 3: Zustandsverhalten von Threads; normalgedruckte Begriffe zeigen Java-Befehle für Zustandswechsel

- Problem, wenn temporärer Zwischenspeicher von Werten, die anderer Thread ändert → Blöcke lassen sich mit Schlüsselwort `synchronized` so konfigurieren, dass nur ein Block eines Monitors (grundsätzlich this, Ausnahmen: Klassenmethoden (auf Klassenobjekt synchronisiert) und explizite Angabe eines Monitorobjekts in Klammern bei Blöcken innerhalb von Methoden (z. B. statisches Lock-Objekt)) gleichzeitig ausgeführt wird
- Problem, wenn mehrere Threads auf dasselbe Objekt zugreifen, da Java Zwischenspeicherung erlaubt → Attribute lassen sich mit dem Schlüsselwort `volatile` so konfigurieren, dass keine Zwischenspeicherung erfolgt
- Überprüfen, ob interrupt-Flag gesetzt mit `Thread.currentThread().isInterrupted()`, zusätzlich zurücksetzen mit `interrupted()`
- Problem der Verklemmung wenn nicht darauf geachtet wird, dass wenn dieselben Monitore gesperrt werden müssen, dieselbe Reihenfolge eingehalten wird

2.2 konzeptuelle Entscheidungen in Java

- nur Einfachvererbung möglich (keine Mehrfachvererbung)
- klassenbasiertes, nicht prototypbasiertes Erstellen von Objekten
- Bindungsart nicht einheitlich: Objekte dynamisch gebunden, Attribute statisch gebunden
- Auflösung einer Überladung
    - erfolgt durch Compiler, der eine Methodensignatur (Name, geordnete Liste der Parametertypen) ermittelt und in den Bytecode schreibt → für Ermittlung der Methodensignatur Typenprüfung nur nach Deklarationstyp, nicht nach dynamischen Typ → es kommen nur Methoden der Superklassen[1] der deklarierten Klasse in Betracht, auch wenn zur Laufzeit tatsächlich ein Objekt einer Subklasse vorliegt, die weitere Methoden hat; da die Auflösung der Methodensignatur zur Laufzeit durch die VM erfolgt, werden Überschreibungen aber berücksichtigt[2]

[1] Eine Klasse ist zur Vermeidung unsinniger Fallunterscheidung stets Super- und Subklassen von sich selbst.
[2] Dies gilt generell bei Verwendung von Superklassen als Deklarationstyp: Es kann nur auf Methoden zugegriffen werden, die auch im deklarierten Typ vorhanden sind.

- Ablauf der Ermittlung
  - Erstellen einer Liste aller in Frage kommenden Methodensignaturen
  - wenn mehrere vorhanden: most-specific-Auswahl: paarweiser Vergleich der Signaturen aus der Liste ohne Berücksichtigung des Aufrufs: Methode A ist spezieller als Methode b, wenn jeder Parameter der Methode A Subtyp[1] des entsprechenden Parameters in Methode B ist.
  - wenn immer noch mehrere vorhanden: Compilerfehler

- ohne expliziten Superkonstrukterverweis (`super()`, ggf. mittelbar durch mit `this()` referenzierten anderen Konstruktur der aktuellen Klasse) wird implizit parameterloser Superklassenkonstruktur aufgerufen; Zulässigkeit des expliziten Verweises nur an erster Stelle, da erst Aktionen im Super-, dann im Subtyp ausgeführt werden
- Unterschied zwischen `protected` und default-Zugriffsebene (paketlokal): `protected` erlaubt zusätzlich Zugriff in abgeleiteter Klasse (aber nur als Element der abgeleiteten, nicht der originalen Klasse); Zweck: Sichtbarmachen des Elements in Subklassen zum Überschreiben (nötig insbesondere bei Frameworks[3])
- Prinzip des „Catch or declare“ (mit „`throws`“) für Exceptions
  - Grundsatz: Prinzip gilt („checked exceptions”)
  - Ausnahme: Von „`RuntimeException`“ (Subklasse von „`Exception`“) abgeleitete Exceptions („unchecked exceptions“) → sollten in guten Programmen nicht abgefangen, sondern vermieden werden (z. B. Division durch Null, `NullPointerException`, `ArrayIndexOutOfBoundsException`)

3 Streams am Beispiel des Dateizugriffs

```
1.  package de.tillmenke.studium.informatik.kurs1618.klausurvorbereitung;
2.  
3.  import java.io.*;
4.  
5.  class FileAccess {
6.      public static void writeTextToFile(String fileName, String text) throws IOExcept
    ion {
7.          FileWriter fw = new FileWriter(fileName);
8.          BufferedWriter bw = new BufferedWriter(fw);
9.          bw.write(text);
10.         bw.close();
11.     }
12. 
13.     public static void readTextFromFile(String fileName) throws IOException {
14.         FileInputStream fis = new FileInputStream(fileName);
15.         InputStreamReader isr = new InputStreamReader(fis);
16.         BufferedReader br = new BufferedReader(isr);
17.         String line = null;
18.         while ((line = br.readLine()) != null) {
19.             System.out.println(line);
20.         }
21.         br.close();
22.     }
23. 
24.     /*ObjectOutput/InputStream erlaubt exportieren von Objekten, deren Klassen das I
    nterface Serializable implementieren*/
25. }
```

[3] „Rahmenwerke“, die erlauben, an wichtigen Stellen eigenen Code einzusetzen (z. B. Motor/Fahrer im Auto)

4 Netzwerk

4.1 Kommunikation über Sockets

```
1. package de.tillmenke.studium.informatik.kurs1618.klausurvorbereitung;
2.
3. import java.io.*;
4. import java.net.*;
5.
6. class Server {
7.     public static void main(String[] args) throws IOException {
8.         @SuppressWarnings("resource") /* kann nicht geschlossen werden, da Server er
   st mit Beenden des Programms beendet
9.                                        * werden muss */
10.         ServerSocket serversocket = new ServerSocket(1234);
11.         while (true) {
12.             Socket socket = serversocket.accept(); /*Socket kann als Parameter an be
   arbeitenden Thread übergeben werden*/
13.             BufferedReader fromClient = new BufferedReader(new InputStreamReader(soc
   ket.getInputStream(), "UTF-8"));
14.             String request = fromClient.readLine();
15.             System.out.println(request);
16.             BufferedOutputStream toClient = new BufferedOutputStream(
17.                     socket.getOutputStream());
18.             toClient.write('b');
19.             toClient.flush();
20.             socket.close();
21.         }
22.     }
23. }
24.
25. class Client {
26.     public static void main(String[] args) throws IOException {
27.         Socket socket = new Socket("server1.tillmenke.de", 1234);
28.         BufferedOutputStream toServer = new BufferedOutputStream(socket.getOutputStr
   eam());
29.         toServer.write('b');
30.         toServer.flush();
31.         BufferedReader fromServer = new BufferedReader(new InputStreamReader(socket.
   getInputStream(), "UTF-8"));
32.         String response = fromServer.readLine();
33.         System.out.println(response);
34.         socket.close();
35.     }
36. }
```

4.2 Kommunikation über RMI

4.2.1 auf beiden Seiten notwendig verfügbare Elemente

- Übertragungsklasse für Parameter/Antwort `implements Serializable`
- `Schnittstelleninterface extends java.rmi.Remote`, alle Funktionen `throws java.rmi.RemoteException`

4.2.2 nur auf Serverseite notwendig verfügbare Elemente

- Implementierung des Schnittstelleninterfaces `extends java.rmi.server.UnicastRemoteObject` mit Konstruktur `throws java.rmi.RemoteException`
- Registry
  - `java.rmi.registry.LocateRegistry.createRegistry(1099);`
  - `Naming.rebind("Übermittlungsname", Objekt);`

4.2.3 Verwendung auf Clientseite

Abruf der Objektreferenz mit `Klasse Objekt = (Klasse) Naming.lookup("rmi://localhost/Übermittlungsname")`

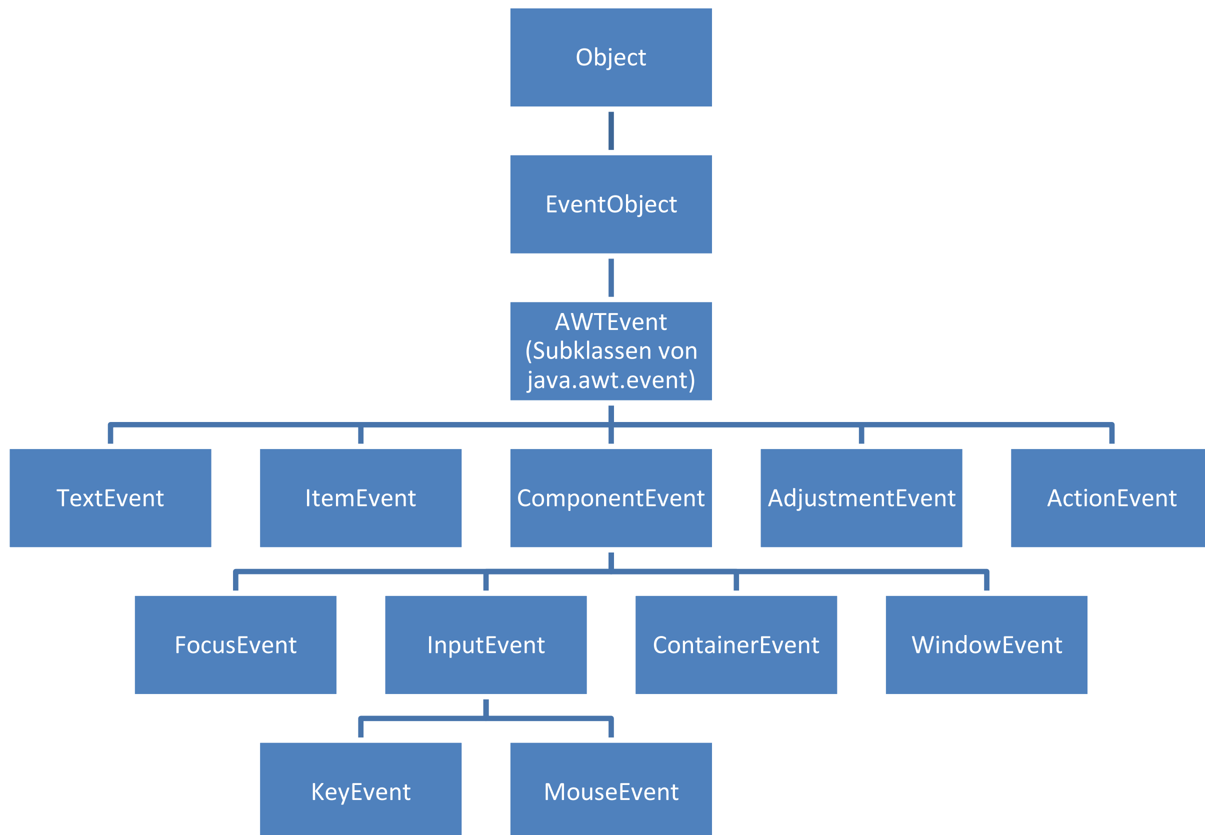
5 Fenster mit AWT

5.1 Grundstruktur

```
1.  package de.tillmenke.studium.informatik.kurs1618.klausurvorbereitung;
2.  
3.  import java.awt.*;
4.  import java.awt.event.*;
5.  
6.  public class AWT {
7.      public static void main(String[] args) {
8.          Frame f = new Frame();
9.          f.setSize(300,400);
10.         f.setLocation(100,100);
11.         f.add(new Label("Text"));
12.         Button b = new Button("Text"); /*auch möglich: TextField, Canvas*/
13.         b.setEnabled(true); /*Listener wie beim Fenster möglich*/
14.         f.setVisible(true);
15.         /*Implementierung Listener*/
16.         f.addWindowListener(new WindowListener() { /*anonyme Klasse*/
17.             @Override
18.             public void windowClosed(WindowEvent arg0) {
19.             
20.             }
21.             
22.             @Override
23.             public void windowActivated(WindowEvent arg0) {
24.             }
25.             
26.             @Override
27.             public void windowClosing(WindowEvent arg0) {
28.                 System.exit(0);
29.             }
30.             
31.             @Override
32.             public void windowDeactivated(WindowEvent arg0) {
33.             }
34.             
35.             @Override
36.             public void windowDeiconified(WindowEvent arg0) {
37.             
38.             }
39.             
40.             @Override
41.             public void windowIconified(WindowEvent arg0) {
42.             
43.             }
44.             
45.             @Override
46.             public void windowOpened(WindowEvent arg0) {
47.             
48.             }});
49.         /*technische Funktionsweise: Eintragen in Liste mit Listener-
    Objekten, wenn Aufrufbedingung eintritt dann Aufruf der entsprechenden Methoden (die
     über Interface garantiert werden) aller vorhandenen Listener*/
50.         /*Kurzimplementierung Listener*/
51.         f.addWindowListener(new WindowAdapter() {
52.             @Override /* Adapterklasse enthält Implementierung des Interfaces Window
    Listener, sodass nur benötigte Methoden implementiert werden müssen */
53.             public void windowClosing(WindowEvent arg0) {
54.                 System.exit(0);
55.             }});
56.         
57.         f.setLocationRelativeTo(null); /*Fenster zentrieren */
58.     }
59.     
60.     class MyCanvas extends Canvas{
61.         @Override
```

```
62.         public void paint(Graphics g) {
63.             g.setColor(Color.BLUE);
64.             g.setFont(new Font("Name",10,10));
65.             g.drawLine(0,0,10,10);
66.             /*
67.             g.drawImage(...);
68.             g.drawOval(...);
69.             g.drawPolygon(...);
70.             g.drawPolyline(...);
71.             g.drawRect(...);
72.             g.drawRoundRect(...);
73.             g.drawString(...);
74.             g.fillRect(...);
75.             ...
76.             */
77.         }
78.         @Override
79.         public Dimension getPreferredSize() {
80.             return new Dimension(10,10);
81.         }
82.     }
83. }
```

5.2 mögliche Ereignisse der Komponenten



5.3 Layouts

- BorderLayout (Standardlayout) → fünf verschiedene mögliche Positionen (zweiter Parameter der add-Funktion)
    - BorderLayout.NORTH, BorderLayout.SOUTH (Behälterbreite, bevorzugte Höhe (getPreferredSize())
    - BorderLayout.WEST, BorderLayout.EAST (bevorzugte Breite, verbleibende Höhe)
    - BorderLayout.CENTER (verbleibender Platz in der Mitte)
- FlowLayout (Parameter des Konstruktors des Panelobjekts: `new FlowLayout (FlowLayout.LEFT)`)