Quelltextanalyse

- Sowohl der Konstruktor als auch die Methode **abheben()** zeigen durch die Klausel **throws Exception** an, dass sie eine allgemeine Ausnahme werfen könnten.
- Für den Anwender einer Klasse heißt das, dass er das Instantiieren und das Aufrufen der Methode in eine **try-catch**-Umgebung einbetten **muss**.
- Mit dem Schlüsselwort **throw** können wir eine Ausnahme werfen. In diesem Beispiel wird eine neue **Exception** erzeugt, die wir mit einem bestimmten String versehen.
- In der **catch**-Umgebung wird dieser String dann ausgedruckt.

6.5 Eine spezielle Ausnahme erzeugen

Man kann in der Fehlerbehandlung noch viel stärker differenzieren, indem man eigene Klassen von Ausnahmen erzeugt.

Exceptions/Geld/4/SparbuchException.java

```
6  public class SparbuchException extends Exception
7  {
8    enum SBFehlerTyp
9    {
10     UNBEKANNT, KONSTRUKTOR, ABHEBEN;
11   }
12
13   private SBFehlerTyp fehlerTyp;
14
15   public SparbuchException()
16   {
17     fehlerTyp = SBFehlerTyp.UNBEKANNT;
18   }
19
20   public SparbuchException(SBFehlerTyp ft)
21   {
22     fehlerTyp = switch (ft)
23       {
24         case KONSTRUKTOR -> ft;
25         case ABHEBEN -> ft;
26         default -> SBFehlerTyp.UNBEKANNT;
27       };
28   }
29
30   public SBFehlerTyp leseFehlerTyp() { return fehlerTyp; }
31
32   @Override public String toString()
33   {
34     return switch (fehlerTyp)
35       {
36         case KONSTRUKTOR -> "Sparbuch: Fehler beim Erzeugen des Sparbuchs";
37         case ABHEBEN     -> "Sparbuch: Fehler beim Abheben";
38         default          -> "Sparbuch: unbekannter Fehler";
39       };
40   }
41 }
```


Quelltextanalyse

- Mit der Klausel **extends Exception** zeigt die Klasse **SparbuchException** an, dass sie als spezielle Ausnahme zu verwenden ist.
- Wir erweitern hier die allgemeine Ausnahme um einen Aufzählungtyp für spezielle Fehlercodes, das Attribut **fehlerTyp** und um die Methode **leseFehlerTyp()**.

Exceptions/Geld/4/Sparbuch.java

```
28   public float abheben(float s) throws SparbuchException
29   {
30     if (k < s)
31       throw new SparbuchException(SparbuchException.SBFehlerTyp.ABHEBEN);
```

Quelltextanalyse

- Der Konstruktor von **Sparbuch** und die Methode **abheben()** werfen nun die spezialisierte Ausnahme **SparbuchException**.
- Beim Erzeugen der Ausnahme mit **new** wird nun ein entsprechender Fehlertyp übergeben.
- In der **catch**-Anweisung kann dieser dann durch Aufruf der Methode **leseFehlerTyp()** ausgewertet werden.

6.6 Eine alternative clone-Implementierung

- In Abschnitt 5.9 hatten wir **clone()** von Hand so implementiert, dass ein neues Objekt mit gleichen Attributen zurückgeliefert wird.
- Stattdessen können wir das Kopieren der Attribute auch von der von **Object** geerbten Methode **clone()** erledigen lassen.

  aber Vorsicht: Bei Referenzattributen werden nur die Adressen kopiert!
- Da wir **Cloneable** implementieren, kann die zu fangende **CloneNotSupportedException** niemals auftreten.
- Seit Java 22 können wir die unbenannte Variable _ (Unterstrich) verwenden und müssen dem unbenutzten Exception-Parameter keinen Namen geben.

Objektorientierung/Autos/13-super.clone/Auto.java

```
6  public class Auto implements Cloneable
7  {
56   @Override public Auto clone()
57   {
58     try { return (Auto)super.clone(); }
59     catch (CloneNotSupportedException _) { throw new InternalError(); }
60   }
```

6.7 Zusammenfassende Fragen

- Was sind Ausnahmen und was geht uns das an?
- Was ist der Unterschied zwischen allgemeinen Ausnahmen und Laufzeitausnahmen?
- Was bedeutet Werfen und Fangen einer Ausnahme?
- Wie fängt man Ausnahmen?
- Welche Möglichkeiten hat man, mit einer gefangenen Ausnahme umzugehen?
- Wie realisiert man das Einbetten in die **`try-catch-finally`**-Kontrollstruktur?
- Was ist dabei besonders zu beachten?
- Wie kann man herausfinden, wann ein Einbetten notwendig ist?
- Wie wirft man Ausnahmen?
- Wie erzeugt man eigendefinierte Ausnahmen?
- Schlüsselwörter **`try`**, **`catch`**, **`finally`**, **`throw`** und **`throws`**

7 Generics

Inhalt

Inhalt/Ziele: Klassen und statische Methoden können generische (variable) Datentypen verwenden.

Diese sog. **Generics** ähneln den Templates in C++; sie sind verglichen mit diesen weniger flexibel aber dafür typsicher.

Eine **generische Klasse** ist eine Klasse, die **Typ-Variablen** besitzt.

Bei der Klassendeklaration wird der formale Typ-Parameter in „spitzen Klammern“ angegeben: **<T>**

Dieser formale Typ-Parameter **T** steht dabei für irgendeinen nicht näher spezifizierten Typ, der allerdings kompatibel mit **`Object`** sein muss (d. h. Referenztypen, aber keine einfachen Datentypen).

Es gibt grundsätzlich zwei Realisierungsmöglichkeiten von generischen Datentypen:

- heterogene Variante: Für jeden verwendeten Typ (etwa **`String`**, **`Integer`**) wird individueller Code erzeugt, also mehrere Klassen (Code-Spezialisierung).
- homogene Übersetzung: Für jede parametrisierte Klasse wird Code erzeugt, der statt des generischen Typs **`Object`** einsetzt (Code-Sharing). Für einen konkreten Typ werden Typanpassungen in die Anweisungen eingebaut.

Java benutzt die homogene Übersetzung (und C++ die heterogene).

Generics/1/Pair.java

```
7  public class Pair<T>
8  {
9    private T elem0;
10   private T elem1;
11
12   public Pair(T e0, T e1)
13   {
14     elem0 = e0;
15     elem1 = e1;
16   }
17
18   public T get0() { return elem0; }
19   public T get1() { return elem1; }
20
21   // public T getSum() { return elem0 + elem1; } // nicht möglich, denn
22   //   T + T ist abgesehen von String für keinen Referenztyp definiert
23
24   public static void main(String[] args)
25   {
26     Pair<Integer> p = new Pair<Integer>(12, 37); // Angabe des Typs bei Initialisierung
27
28     System.out.println("Summe: " + (p.get0() + p.get1())); // Ok wegen Unboxing Integer → int
29
30     // Pair<int> p2 = new Pair<int>(12, 14); // nicht möglich, int kein Referenztyp
31
32     int[] a = { 1 };
33     int[] b = { 3 };
34     Pair<int[]> p3 = new Pair<int[]>(a, b); // möglich, aber Integer besser für nur einen Wert
41   }
42 }
```

Mit Java 7 wurde **Typinferenz** eingeführt. Die spitzen Klammern können im Konstruktoraufruf leer bleiben, wenn der Typ aus dem Zusammenhang klar ist.

```
36     Pair<Integer> p4 = new Pair<>(3, 5);
```

seit Java 10 kombinierbar mit **var**:

```
37      var p5 = new Pair<>(4, 6);
```

Der Typ wird hier aus den Parameter abgeleitet, für Standardkonstruktoren (ohne Parameter) klappt **var** also nicht.

7.1 Unterklassen

Beispiel: Klassenhierarchie

```
class Base { … };
class Spec extends Base { … };
```

und die **Pair**-Variablen

```
Pair<Base> pbase;
Pair<Spec> pspec;
```

Dann ist **Pair<Spec>** keine Unterklasse von **Pair<Base>**, d. h. die Zuweisung **pbase = pspec** erzeugt einen Compilerfehler.

```
39      // nicht möglich, obwohl Integer extends Number
40      // Pair<Number> p5 = new Pair<Integer>(17, 4);
```

Über **p5** ließe sich sonst ein anderer **Number**-Wert (z. B. **1.5** als **Double**) eintragen.

7.2 Wildcard-Typen

Anstelle von generischen Superklassen können sogenannte **Wildcards** verwendet werden, z. B. ein Paar von Elementen unbekannten Datentyps **Pair<?>**.

Die Methode **get0()** kann für jeden Typ und damit auch für einen unbekannten Typ aufgerufen werden. Die Methode **set()** darf dagegen nicht mit Wildcards benutzt werden, da der Java-Compiler hierbei keine Typprüfung vornehmen kann.

Generics/2-wildcard/Pair.java

```
7   public class Pair<T>
8   {
21    public void set(T e0, T e1)
22    {
23      elem0 = e0;
24      elem1 = e1;
25    }
32      Pair<?> p2 = new Pair<Integer>(12, 37);
33      System.out.println("Elemente: " + p2.get0() + ", " + p2.get1());
34      // p2.set(1.41421, 1.73205);  // nicht möglich
35      // p2.set(13, 36);  // auch nicht
36      p2 = new Pair<String>("zwölf", "siebenunddreißig");
```


7.3 Typ-Variablen mit Einschränkungen

Formale Typvariablen können eingeschränkt, d. h. an einen Typ gebunden werden. Dies ist auch bei Wildcard-Typen möglich.

Mit **<T extends MeineKlasse>** ist die Typ-Variable **T** auf **MeineKlasse** selbst oder eine davon abgeleitete Klasse beschränkt.

Als Gegenstück gibt es auch Beschränkungen in die andere Richtung:

<T super MeineKlasse>, d. h. **MeineKlasse** selbst oder eine ihrer Vorfahren

Generics/3-bounds/Pair.java

```
7   public class Pair<T extends Number>
8   {
21    public double getDSum() { return elem0.doubleValue() + elem1.doubleValue(); }
22    // Number enthält doubleValue() als abstrakte Methode,
23    // alle davon abgeleiteten Wrapperklassen implementieren diese
27      Pair<Integer> p = new Pair<Integer>(3, 4);
28      Pair<? extends Number> p2 = p;
29      System.out.println("Summe: " + p2.getDSum());
```


7.4 Generische Methoden

Bei **generischen Methoden** wird der formale Typparameter zwischen den Spezifizierern wie **public** oder **static** und dem Ergebnistyp angegeben

Anwendungsfälle:

1. Objekt- oder Klassenmethode in einer nicht-generischen Klasse
2. Klassenmethode (statische Methode) in einer bereits generischen Klasse, denn die Typvariable der Klasse gilt nur in allen Objektmethoden

Comparable ist ein Interface, das eine Vergleichsmethode **int compareTo(T o)** enthält. Diese Methode liefert als Ergebnis einen Wert < 0, = 0 oder > 0 für „kleiner als“, „gleich“ oder „größer als“.

Generics/4-genMeth/Pair.java

```
7   public class Pair<T extends Comparable<T>>
8   {
21    // Supremum, hier als komponentenweises Maximum
22    public static <T extends Comparable<T>> Pair<T> sup(Pair<T> p1, Pair<T> p2)
23    {
24      return new Pair<T>(p1.get0().compareTo(p2.get0()) > 0 ?
25                         p1.get0() : p2.get0(),
26                         p1.get1().compareTo(p2.get1()) > 0 ?
27                         p1.get1() : p2.get1());
28    }
34      Pair<Integer> ps = sup(p1, p2);
```

8 Collections

Inhalt

Inhalt/Ziele: Als Collections werden Datenstrukturen bezeichnet, die dazu dienen, Mengen von Daten aufzunehmen und zu verarbeiten.

Alle Collections sind generisch implementiert und damit typsicher.

8.1 Das Collection Framework

Das Collection Framework besteht aus gut 40 Klassen und Interfaces im Paket **java.util**.

Im wesentlichen werden als Datenstrukturen drei Grundformen über Interfaces bereitgestellt:

- **List** (Liste mit sequentiellem und wahlfreiem Zugriff)
- **Set** (Menge von Elementen mit typischen Mengenoperationen)
- **Map** (Abbildung von Elementen eines Typs auf Elemente eines anderen Typs)

Die folgende Tabelle enthält die wichtigsten Implementierungen dieser Interfaces:

implementiert als	List	Set	Map
Array veränderlicher Größe	ArrayList, Vector		
doppelt verkettete Liste	LinkedList		
Hashtabelle		HashSet	HashMap
balancierter Baum		TreeSet	TreeMap
Hashtabelle + dopp. verk. Liste		LinkedHashSet	LinkedHashMap

Das folgende Beispiel zeigt das Anlegen und Bearbeiten zweier unterschiedlicher Listen:

Collections/Listen/1/ListenBsp.java

```
11  static void fillList(List<Integer> list)
12  {
13    for (int i = 0; i < 10; ++i)
14      list.add(i);  // Autoboxing int → Integer
15    list.remove(3);
16  }
17
18  static void doubleElements(List<Integer> list)
19  {
20    for (int i = 0; i < list.size(); i++)
21      list.set(i, 2 * list.get(i)); // Achtung: für LinkedList Aufwand O(i) pro set/get
22  }
23
24  static <E> void printList(List<E> list)
25  {
26    for (int i = 0; i < list.size(); i++)
27      System.out.println(list.get(i));
28    System.out.println("---");
29  }
30
31  public static void main(String[] args)
32  {
33    // Erzeugen der LinkedList
34    LinkedList<Integer> list1 = new LinkedList<Integer>(); // Java-6-Stil
35    fillList(list1);
36    doubleElements(list1);
37    printList(list1); // →  0 2 4 8 10 12 14 16 18
38
39    // Erzeugen der ArrayList
40    ArrayList<Integer> list2 = new ArrayList<>();          // Java-7-Stil
41    fillList(list2);
42    printList(list2); // →  0 1 2 4 5 6 7 8 9
43
44    // Test von removeAll
45    list2.removeAll(list1);
46    printList(list2); // →  1 5 6 7 9
47
48    var list3 = new ArrayList<Integer>();                  // Java 10
49  }
```


Quelltextanalyse

- Eine Collection vom Typ **List** ist eine geordnete Menge von Objekten, auf die entweder sequentiell oder wahlfrei über ihren Index zugegriffen werden kann.
- Während **fillList()** und **doubleElements()** spezielle Methoden für **Integer**-Listen sind, ist **printList()** eine generische Methode.

8.1.1 Iteratoren

Mit Hilfe eines **Iterators** können die Elemente einer Collection durchlaufen werden. Das Interface **Iterator<E>** fordert hierfür die Methoden

- **boolean hasNext()**,
- **E next()** und
- **void remove()** zum Löschen des zuletzt geholten Elements.

Collections/Listen/2-Iterator/ListenBsp.java

```
18  static <E> void printList(List<E> list)
19  {
20    for (Iterator<E> it = list.iterator(); it.hasNext(); )
21      System.out.println(it.next());
22    System.out.println("---");
23  }
```

Noch eleganter lassen sich Collections mit einer for-each-Schleife durchlaufen.

Collections/Listen/3-for-each/ListenBsp.java

```
18  static <E> void printList(List<E> list)
19  {
20    for (E e : list)
21      System.out.println(e);
22    System.out.println("---");
23  }
```

Quelltextanalyse

- Alle von **List** und **Set** abgeleiteten Collections implementieren das Interface **Iterable<T>**.
- Dieses Interface beinhaltet die Methode **Iterator<T> iterator()**.

Ein Beispiel zu **HashMap**s folgt in 11.7.

8.2 Collections nach altem Stil

Alle Collections können auch ohne generischen Typ (und damit ohne Typsicherheit) verwendet werden.

Collections/alt/ListenBsp.java

```
11  @SuppressWarnings({ "rawtypes", "unchecked" })
12  static void fillList(List list)
13  {
14    for (int i = 0; i < 10; ++i)
15      list.add(i * i);  // wird per Autoboxing nach Integer umgewandelt
16    list.add("Hallo");
17  }
18
19  @SuppressWarnings("rawtypes")
20  static void printList(List list)
21  {
22    for (Iterator it = list.iterator(); it.hasNext(); )
23      {
24        Object o = it.next();
25        if (o instanceof Number)
26          {
27            Number n = (Number)o;
28            String größe = (n.doubleValue() < 10) ? "kleine" : "große␣";
29            System.out.print("eine␣" + größe + "␣Zahl:␣␣␣");
30          }
31        else
32          System.out.print("anderer␣Typ:␣␣␣␣␣␣␣␣");
33        System.out.println(o);
34      }
35  }
36
37  @SuppressWarnings("rawtypes")
38  public static void main(String[] args)
39  {
40    LinkedList list1 = new LinkedList();
41    fillList(list1);
42    printList(list1);
43  }
```

Quelltextanalyse

- Untypisierte Collections speichern ausschließlich Referenzen auf **Object**.
- Beim Lesen aus einer solchen Collection sind daher Cast-Operationen nötig.

9 Lambda-Ausdrücke

Inhalt

Inhalt/Ziele: Lambda-Ausdrücke ermöglichen eine Kurzschreibweise für Klassen mit funktionaler Schnittstelle, d. h. mit genau einer Methode. Lambda-Ausdrücke sind mit Java 8 neu eingeführt worden.

9.1 Einleitung

Ausgangspunkt ist ein Interface für Operationen mit ganzen Zahlen:

Lambda/Einleitung/IntOp.java

```
6  @FunctionalInterface
7  public interface IntOp
8  {
9    int f(int i);
10 }
```

Die Annotation **@FunctionalInterface** ist optional. Ist sie vorhanden, prüft der Compiler, ob es tatsächlich genau eine abstrakte Methode gibt.

Die folgende Testklasse bietet eine statische Methode, um Parameter und Ergebnis einer Ganzzahloperation auszugeben:

Lambda/Einleitung/Test.java

```
6  public class Test
7  {
10   public static void druckeIntOp(IntOp op, int i)
11   { System.out.println("" + i + "␣wird␣abgebildet␣auf␣" + op.f(i)); }
```

Beispieloperation: Quadrieren einer Zahl – hierzu benötigen wir eine Klasse, die das Interface **IntOp** passend implementiert.

Lambda/Einleitung/Quadrierer.java

```
6  public class Quadrierer implements IntOp
7  {
8    @Override public int f(int i) { return i * i; }
9  }
```

```
18     druckeIntOp(new Quadrierer(), 4);
```


9.1.1 Anonyme Klasse

Die Klasse **Quadrierer** wird nur an einer einzigen Stelle benötigt und instantiiert. Wir können stattdessen eine **anonyme Klasse** definieren, die das Interface **IntOp** implementiert.

```
19     IntOp quadop = new IntOp()
20       { // Beginn der anonymen Subklasse von IntOp
21         @Override public int f(int i) { return i * i; }
22       }; // Ende der anonymen Subklasse von IntOp
23     druckeIntOp(quadop, 4);
```

- Eine anonyme Unterklasse der Klasse **Kl** wird erzeugt, indem bei der Instantiierung **new Kl(…)** das sonst folgende Semikolon durch einen Klassenrumpf **{ … }** ersetzt wird.
- Auf dieselbe Art können wir mit anonymen Klassen Interfaces implementieren.
- Die Vererbung bzw. Implementierung geschieht hier ohne die Schlüsselwörter **extends** bzw. **implements**.

Die anonyme Klasse kann sogar direkt bei Ihrer Verwendung erzeugt werden:

```
24     druckeIntOp(new IntOp() { @Override public int f(int i) { return i * i; } },
25                 4);
```


9.1.2 Lambda-Ausdrücke

Ein **Lambda-Ausdruck** ermöglicht eine noch kompaktere Schreibweise für Klassen, die nur eine Funktion implementieren.

```
26     druckeIntOp((int i) -> i * i, 4);
```

Der Compiler weiß hier aus dem Zusammenhang, welches Interface implementiert wird und wie der Rückgabetyp aussieht.

Auch die Angabe der Parametertypen kann entfallen:

```
27     druckeIntOp((i) -> i * i, 4);
```

Bei nur einem Parameter können sogar die Klammern entfallen:

```
28     druckeIntOp(i -> i * i, 4);
```

Attribute der umgebenden Klasse sowie Variablen aus dem umgebenden Block, die **final** sind oder es ohne weitere Änderungen am Code sein könnten, lassen sich ohne Übergabe als Parameter verwenden.

```
8    private static int v = 7;
29     int k = 5; // wird nie verändert, könnte auch final sein
30     druckeIntOp(i -> i * k, 4);
31     v = 8;     // ist static, trotz Änderung im Lambda-Ausdruck verwendbar
32     druckeIntOp(i -> i * v, 4);
```


Lambda-Ausdrücke dürfen auch aus mehreren Anweisungen bestehen. Dann sind Blockklammern und ein **return** nötig.

```
33     druckeIntOp(i ->
34                 {
35                   int i3 = i * i * i;
36                   return i3 / 7;
37                 }, 4);
```

Enthält ein Lambda-Ausdruck lediglich einen Methodenaufruf, der alle Parameter weitergibt, so lässt sich dieser auch als **Methoden-Referenz** schreiben.

```
13   public static int qfkt(int i)
14   { return i * i; }
38     druckeIntOp(i -> qfkt(i), 4);          // mit Lambda-Ausdruck
39     druckeIntOp(Test::qfkt, 4);            // äquivalent mit Methoden-Referenz
```

9.2 Verwendung mit Collections

Das Collection-Framework wurde mit Java 8 so erweitert, dass an vielen Stellen Lambda-Ausdrücke verwendet werden können.

Beispiel: Löschen von Einträgen aus einer Liste, die eine bestimmte Bedingung erfüllen

Lambda/Collections/CollLambdaBsp.java

```
13    ArrayList<Integer> l1 = new ArrayList<>();
14    for (int i = 0; i < 100; i++)
15      l1.add(i);
16    // löschen aller Einträge, die durch 2 oder 3 teilbar sind
17    l1.removeIf(i -> i % 2 == 0 || i % 3 == 0);
```

Beispiel: Ausführen einer Operation auf jedem Element

```
18    l1.forEach(System.out::println);                    // Ausgabe der Elemente
19    l1.forEach(i -> System.out.println(i * i));         // Ausgabe derer Quadrate
```

Mit Hilfe von Streams lassen sich Operationen auf Collections verketten – und zwar so, dass nicht nach jeder Operation eine Liste mit Zwischenergebnissen gespeichert wird, sondern die Elemente weitergereicht werden.

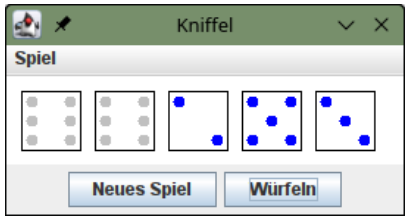
```
21    Random r = new Random();
22    System.out.print("Länge von l2 (≥ 5): ");
23    int l2länge = new Scanner(System.in).nextInt();
24    ArrayList<Integer> l2 = new ArrayList<>(l2länge);
25    for (int i = 0; i < l2länge; i++)
26      l2.add(r.nextInt(l2länge / 5));
27    // zählen verschiedener gerader Zahlen
28    long n = l2.stream().filter(i -> i % 2 == 0).distinct().count();
29    System.out.println("Anzahl verschiedener gerader Zahlen in l2: " + n);
```


10 Swing

Inhalt

Inhalt/Ziele: Erstellung von graphischen Applikationen mit Swing. In diesem Abschnitt wird eine Anwendung erstellt, mit der man ein einfaches Kniffelspiel simulieren kann. Einhergehend mit der Entwicklung des Kniffelspiels erfolgt ein Crash-Kurs zu objektorientierten graphischen Benutzerschnittstellen **(Graphical User Interface, GUI)**.

10.1 Kniffel: Spielregeln



- Beim Kniffeln kann man fünf Würfel bis zu dreimal hintereinander werfen. Ziel ist es, möglichst günstige Augenkombinationen zu erreichen (z. B. nur „Sechser“).
- Wir klicken zuerst auf die Taste „Würfeln“.
- Dann markieren wir mit der Maus die Würfel, die wir festhalten möchten. Durch nochmaliges Anklicken können wir die Markierung wieder aufheben.
- Wir klicken erneut auf die Taste „Würfeln“. Diesmal ändern sich nur die Würfel mit den blauen Augen.
- Wir können jetzt noch einmal markieren und ein letztes Mal würfeln.
- Nun steht unser Endergebnis fest. Mit der Taste „Neues Spiel“ können wir unser Glück erneut versuchen.