8. Exception-Handling

8.1 Die Klasse MyArrayList

8.1.1 Zielsetzung

In den letzten Kapiteln (Arrays, Sortieralgorithmen und Suchverfahren) haben wir viele Methoden zum Umgang mit Arrays kennengelernt.

Und im Kapitel 7 haben wir uns dann mit der Sammlungs-Klasse **ArrayList** beschäftigt und dabei auch schon einen kurzen Blick auf das Thema ***Generics*** geworfen.

In diesem Kapitel 8 verfolgen wir mehrere Ziele gleichzeitig:

1. Erstellung einer eigenen Klasse **MyArrayList**, die ähnlich arbeiten soll wie die Klasse **ArrayList**.
2. Die Behandlung von ***Exceptions*** (Ausnahmen) mit throws und try-catch.
3. Vertiefung des Themas "***generische Datentypen***".

8.1.2 Basis-Version von MyArrayList

Beginnen wir mit der Basis-Version der Klasse. Es soll ein Objekt-Array beliebiger Länge erzeugt werden können. Wir übergeben dem Konstruktor die ***Startkapazität*** als Parameter. Ähnlich wie bei einer echten ArrayList soll die Kapazität dann ***pseudo-dynamisch*** erhöht werden können, wenn entsprechender Bedarf besteht.

Version 1a: Konventionelle Fehlermeldung

Die ArrayList-Klasse

```
public class MyArrayListSimple
{
    private Object[] elementData;
    private int size;

    public MyArrayListSimple(int startCapacity)
    // Haupt-Konstruktor
    {
        if (startCapacity <= 0)
        {
            System.out.println("Ungültige Startkapazität");
            return;
        }
        elementData = new Object[startCapacity];
        size = 0;
    }

    public MyArrayListSimple()
    // Komfort-Konstruktor
    {
        this(10);
    }
}
```

Instanzvariablen

`elementData`: das interne Array, das Referenzen auf die gespeicherten Objekte enthält.

`size`: die Anzahl der tatsächlich belegten Elemente. Ähnlich wie bei einer echten ArrayList darf hier `size` nicht mit `length` verwechselt werden.

Konstruktoren

Der Haupt-Konstruktor erstellt das interne Array mit der angegebenen Startkapazität, wenn diese mindestens 1 beträgt. Andernfalls wird eine Fehlermeldung auf der Konsole ausgegeben und der Konstruktor ***terminiert***.

Der Komfort-Konstruktor benötigt keine Angabe der Startkapazität, er ruft den Haupt-Konstruktor mit dem Wert 10 auf, sodass ein Array mit `length = 10` und `size = 0` erstellt wird.

Die Test-Klasse

```
public class TestSimple
{
 // eine fehlerhafte (F) und eine korrekte (K) Wortliste
    private MyArrayListSimple wortlisteF, wortlisteK;

    public TestSimple()
    {
        wortlisteF = new MyArrayListSimple(-3);
        if (wortlisteF == null)
            System.out.println("WorlistF existiert nicht!");
        else
            System.out.println("Wortliste -3: " + wortlisteF);

        wortlisteK = new MyArrayListSimple(4);
        if (wortlisteK == null)
            System.out.println("WorlistK existiert nicht!");
        else
            System.out.println("Wortliste +4: " + wortlisteK);
    }

    public static void main(String[] args)
    {
        TestSimple test = new TestSimple();
        System.out.println("Das Programm wird bis zum Ende ausgeführt!");
    }
}
```

Diese Testklasse überprüft nun die Arbeitsweise der ersten Basisversion unserer ArrayList. Zunächst wird bewusst versucht, eine fehlerhafte Liste mit einer Startkapazität von -3 anzulegen. Dann wird überprüft, ob die Erzeugung eines solchen Objekts gelingt. Wenn das Objekt `wortlisteF` nicht erzeugt werden kann, wird die Meldung "WortlisteF existiert nicht" in der Konsole ausgegeben. Andernfalls wird die Adresse des Objektes angezeigt.

Anschließend wird eine zweite Wortliste `wortlisteK` mit einer gültigen Startkapazität von 4 angelegt. Sollte dies nicht gelingen, wird eine entsprechende Meldung angezeigt. Bei Erfolg wird die Adresse des Objektes ausgegeben.

Konsolenausgabe

```
Ungültige Startkapazität
Wortliste -3: MyArrayListSimple@57469ffb
Wortliste +4: MyArrayListSimple@5120125d
Das Programm wird bis zum Ende ausgeführt!
```

Der Versuch, eine ungültige Liste `wortlisteF` anzulegen, wird zunächst mit einer Fehlermeldung quittiert: "Ungültige Startkapazität". Es wird aber trotzdem ein Objekt angelegt, wie die zweite Zeile der Konsolenausgabe zeigt. Allerdings handelt es sich hier um ein fehlerhaftes Objekt, mit dem man nicht arbeiten kann und das bei einem "echten" Programm gravierende Probleme bereiten könnte.

Die zweite Wortliste `wortlisteK` wird ebenfalls erzeugt, hat allerdings eine Größe von 0, sodass keine weiteren Operationen getestet werden können.

Version 1b: Exception statt Fehlermeldung

Die Array-Klasse

Der folgende Quelltext zeigt einen überarbeiteten Konstruktor der Klasse **MyArrayList**:

```
public MyArrayList(int startKapazitaet)
{
    if (startKapazitaet <= 0)
        throw new IllegalArgumentException
            ("Ungueltige Startkapazitaet: " + startKapazitaet);

    elementData = new Object[startKapazitaet];
    size = 0;
}
```

Bevor wir jetzt auf Exceptions (Ausnahmen) und die Behandlung solcher Exceptions eingehen, schauen wir uns das entsprechende Testprogramm sowie die Konsolenausgabe dieses Testprogramms näher an.

Das Testprogramm

```
public class TestBetter
{
    private MyArrayList wortlisteF, wortlisteK;

    public TestBetter()
    {
        try
        {
            wortlisteF = new MyArrayList(-3);
            if (wortlisteF == null)
                System.out.println("WorlistF existiert nicht!");
            else
                System.out.println("Wortliste -3: " + wortlisteF);

            wortlisteK = new MyArrayList(4);
            if (wortlisteK == null)
                System.out.println("WorlistK existiert nicht!");
            else
                System.out.println("Wortliste +4: " + wortlisteK);
        }
        catch (IllegalArgumentException e)
        {
            System.out.println
               ("Fehler beim Erzeugen von wortliste: " + e.getMessage());
        }
    }

    public static void main(String[] args)
    {
        TestBetter test = new TestBetter();
        System.out.println("Das Programm wird bis zum Ende ausgeführt!");
    }
}
```

Die Konsolenausgabe

```
Fehler beim Erzeugen von wortliste: Ungueltige Startkapazitaet: -3
Das Programm wird bis zum Ende ausgeführt!
```

Das ist eine sehr kurze Konsolenausgabe. Die Fehlermeldung, die durch die falsche Startkapazität verursacht wird, erscheint noch in der Konsole. Dann werden alle anderen Anweisungen des Testprogramms übersprungen, und in der zweiten Zeile wird der letzte Befehl der `main()`-Methode ausgeführt.

Der genaue Ablauf des Testprogramms

1. Der Haupt-Konstruktor von **MyArrayList** wird gestartet
2. Die Bedingung `(startKapazitaet <= 0)` ist wahr.
3. Die Exception wird mit `throw` ausgelöst.
4. Der Konstruktor wird sofort abgebrochen.
5. Das Objekt `wortlisteF` wird nicht erzeugt.
6. Die Programmausführung springt sofort in den ***catch-Block***.
7. Alle nachfolgenden Anweisungen im ***try-Block*** werden übersprungen.

Am wichtigsten ist hier die Tatsache, dass die Objekterzeugung durch die `throw`-Anweisung sofort abgebrochen wird (Schritt 4). Daher kann kein fehlerhaftes **MyArrayList**-Objekt erzeugt werden, wie es bei Version 1a (normale Fehlermeldung statt Exception) noch möglich war.

Das ist der entscheidende Vorteil: Das Programm arbeitet nicht mit einem beschädigten Objekt weiter, das später Probleme verursachen könnte. Stattdessen wird der Fehler sofort an der Stelle sichtbar, an der er entsteht.

Fehlerbehandlung mit try-catch

Wenn in einem Java-Programm eine Exception auftritt, wird die normale Programmausführung unterbrochen (auch ohne `return`-Anweisung!). Ohne besondere Maßnahmen würde das Programm dann sofort mit einer Fehlermeldung abbrechen.

Mit Hilfe eines try-catch-Konstrukts können solche Exceptions jedoch gezielt ***abgefangen*** und ***behandelt*** werden.

Allgemeine Syntax

```
try
{
   // Anweisungen, die eine Exception auslösen können
}
catch (Exceptiontyp e)
{
   // Behandlung des Fehlers
}
```

Der ***try-Block*** enthält also Anweisungen, bei denen eventuell eine Exception auftreten kann (aber nicht muss). Tritt innerhalb dieses Blocks keine Exception auf, werden alle Anweisungen normal ausgeführt, und der catch-Block wird dann übersprungen.

Ablauf beim Auftreten einer Exception

Wird dagegen eine passende Exception ausgelöst, geschieht Folgendes:

1. Die normale Programmausführung wird sofort unterbrochen, dabei werden
2. alle restlichen Anweisungen des try-Blocks übersprungen und
3. die Programmausführung springt direkt in den catch-Block.
4. Dort kann der Fehler dann behandelt werden.

Betrachten wir dazu noch einmal das try-catch-Konstrukt unseres zweiten Testprogramms:

```
try
{
   wortlisteF = new MyArrayList(-3);
   if (wortlisteF == null)
      System.out.println("WorlistF existiert nicht!");
   else
      System.out.println("Wortliste -3: " + wortlisteF);

   wortlisteK = new MyArrayList(4);
   if (wortlisteK == null)
      System.out.println("WorlistK existiert nicht!");
   else
      System.out.println("Wortliste +4: " + wortlisteK);
}

catch (IllegalArgumentException e)
{
   System.out.println("Fehler beim Erzeugen von wortliste: " + e.getMessage());
}
```

Ablauf im try-Block

Beim Aufruf

```
   wortlisteF = new MyArrayList(-3);
```

tritt im Konstruktor der Klasse **MyArrayList** eine **IllegalArgumentException** auf.
Diese Exception wird durch die `throw`-Anweisung ausgelöst:

```
throw new IllegalArgumentException
    ("Ungueltige Startkapazitaet: " + startKapazitaet);
```

Dadurch wird der Konstruktor sofort beendet. Das Objekt wird nicht erzeugt, und die Programmausführung springt direkt in den catch-Block der Testklasse. Die im try-Block folgenden Anweisungen

```
  if (wortlisteF == null)
      System.out.println("WorlistF existiert nicht!");
  else
      System.out.println("Wortliste -3: " + wortlisteF);
   // etc.
```

werden daher nicht mehr ausgeführt.

Ablauf im catch-Block

```
catch (IllegalArgumentException e)
{
   System.out.println("Fehler beim Erzeugen von wortliste: " + e.getMessage());
}
```

Im catch-Block steht die Variable `e` für das **Exception**-Objekt. Über Methoden dieses Objekts können weitere Informationen über den Fehler abgefragt werden. Besonders wichtig ist die Methode `getMessage()`. Diese Methode liefert genau den Text zurück, den wir beim Erzeugen der Exception im Konstruktor von **MyArrayList** angegeben haben:

```
throw new IllegalArgumentException
    ("Ungueltige Startkapazitaet: " + startKapazitaet);
```

In der Konsole erscheinen nun zwei Fehlermeldungen:

```
Fehler beim Erzeugen von wortliste:
Ungueltige Startkapazitaet: -3
```

Die erste Meldung wird von dem `System.out.println()`-Befehl des catch-Blocks erzeugt. Die zweite Meldung dagegen ist das Ergebnis von `e.getMessage()`.

8.1.3 Version 2: zwei add()-Methoden

Wir erweitern die Klasse **MyArrayList** jetzt um zwei `add()`-Methoden, die ein neues Objekt

- an das Ende des Arrays anhängen bzw.
- an einer bestimmten Position einfügen.

Wie bei einer echten ArrayList soll das interne Array dann auch pseudo-dynamisch wachsen können.

"Pseudo-dynamisch" heißt: Es wird einfach ein größeres Array erzeugt, die alten Elemente werden kopiert, und die alte Referenz wird auf das neue Array "umgebogen".

Version 2.1: Die erste add()-Methode: anhängen

Die add()-Methode

```
public void add(Object element)
{
    if (element == null)
        throw new IllegalArgumentException("Element ist null.");

    if (size >= elementData.length)
        grow();

    elementData[size] = element;
    size++;
}
```

Bei der `add()`-Methode wird eine **IllegalArgumentException** geworfen, und zwar dann, wenn das übergebene Objekt den Wert `null` hat.

Bevor das Objekt der Liste hinzugefügt wird, wird erst einmal überprüft, ob die Größe der Liste (`size`) größer oder gleich der Kapazität (`length`) ist. Wenn das der Fall ist, wird `grow()` aufgerufen. Anschließend wird das neue Objekt an das Ende der bisherigen Liste eingefügt.

Warum keine NullPointerException ?

Eine **NullPointerException** wird hier nicht verwendet, obwohl der fehlerhafte Wert ja durchaus `null` ist. Etwas anderes wäre es, wenn man in der `add()`-Methode tatsächlich auf Methoden des Objekts `element` zugreifen wollte, beispielsweise so:

```
String text = element.toString();
```

Dann würde nämlich bei einem `null`-Wert automatisch eine **NullPointerException** auftreten. In unserer `add()`-Methode greifen wir jedoch gar nicht direkt auf das Objekt `element` zu, sondern kopieren lediglich eine Referenz auf `element` in das entsprechende Arrayelement `elementData[size]`.

Die grow()-Methode

```
private void grow()
{
    int alteKapazitaet = elementData.length;
    int neueKapazitaet = alteKapazitaet + alteKapazitaet / 2;

    if (neueKapazitaet == alteKapazitaet)
        neueKapazitaet = alteKapazitaet + 1;

    Object[] neuesArray = new Object[neueKapazitaet];
    for (int i=0; i < alteKapazitaet; i++)
       neuesArray[i] = elementData[i];
    elementData = neuesArray;

    // elementData = Arrays.copyOf(elementData, neueKapazitaet);
}
```

Die Methode grow() berechnet zunächst die neue Kapazität aus der bisherigen Kapazität. Bei einer Liste der Länge 0 oder 1 würde die Multiplikation der alten Kapazität mit dem Faktor 1,5 zu einer neuen Kapazität von 0 oder 1 führen; die Kapazität würde also *nicht* wachsen. Aus diesem Grund wird in diesem Fall die alte Kapazität einfach um 1 erhöht.

Fallbeispiele zum besseren Verständnis:

Alte Kapazität K_{Alt}	Neue Kapazität $K_{Neu} = K_{Alt} + K_{Alt}/2$	$K_{Neu} == K_{Alt}$?	$K_{Alt} + 1$	Endergebnis
0	0 + 0/2 = 0	true	1	1
1	1 + 1/2 = 1,5 bzw. 1	true	2	2
2	2 + 2/2 = 3,0 bzw. 3	false	---	3
3	3 + 3/2 = 4,5 bzw. 4	false	---	4
4	4 + 4/2 = 6,0 bzw. 6	false	---	6
5	5 + 5/2 = 7,5 bzw. 7	false	---	7
6	6 + 6/2 = 9,0 bzw. 9	false	---	9

Die eigentliche Kopieraktion ist in dem obigen Quelltext in Rot dargestellt. Zunächst wird ein neues Array mit der größeren Kapazität angelegt, dann werden alle Elemente aus dem alten Array in das neue Array kopiert (for-Schleife), und schließlich wird die Referenz `elementData`, die bisher auf das alte Array verwiesen hat, auf das neue größere Array "umgebogen".

Eine einzeilige Alternative zu diesem Vierzeiler ist als Kommentar (grün markiert) ebenfalls angegeben. Wenn man die copy()-Methode der Klasse **Arrays** benutzen will, muss man diese Klasse aber zuvor über die Importanweisung einbinden:

```
import java.util.Arrays;
```

Version 2.1: Testprogramm

Für ein Testprogramm wäre es sinnvoll, wenn wir auch die aktuelle Größe (`size`) sowie die aktuelle Kapazität (`length`) der Liste überprüfen könnten. Daher ergänzen wir die Klasse **MyArrayList** noch kurz um zwei Getter-Methoden für diese Attribute:

```
public int size()
{
   return size;
}

public int capacity()
{
   return elementData.length;
}
```

Hier werden keine Parameter übergeben, somit können auch keine *falschen* Argumente übergeben werden. Das heißt, auf das Prüfen bzw. Werfen von einer **IllegalArgumentException** können wir hier verzichten.

Wir wollen die neue Methode `add()` nun testen. In dem Methoden der Testklasse werden wir Exception-Handling mit dem try-catch-Konstrukt anwenden, denn die `add()`-Methode kann ja durchaus eine **IllegalArgumentException** werfen.

Quelltext des Testprogramms siehe nächste Seite.

Die Test-Klasse

```
public class Test
// Version 2.1
{
    private MyArrayList wortliste;

    public Test()
    {
        try
        {
            wortliste = new MyArrayList(5);
            fuegeObjekteEin();
        }
        catch (IllegalArgumentException ausnahme)
        {
            System.out.println("Fehler beim Erzeugen von wortliste: " +
                ausnahme.getMessage());
        }
    }

    private void fuegeEinObjektEin(Object neu)
    {
        System.out.println("Einfügen des Objektes " + neu);

        try
        {
            wortliste.add(neu);
            System.out.printf("Neues Objekt %-20s", neu);
            System.out.printf("size = %3d, length = %3d %n%n",
                wortliste.size(),wortliste.capacity());
        }
        catch (IllegalArgumentException ausnahme)
        {
            System.out.println
            ("Fehler bei add(): " + ausnahme.getMessage() + "\n");
        }
    }

    public void fuegeObjekteEin()
    {
        fuegeEinObjektEin("1. Name");
        fuegeEinObjektEin("2. Vorname");
        fuegeEinObjektEin("3. Straße");
        fuegeEinObjektEin(null);
        fuegeEinObjektEin("4. Hausnummer");
        fuegeEinObjektEin("5. Postleitzahl");
        fuegeEinObjektEin(23);
        fuegeEinObjektEin(true);
        fuegeEinObjektEin("6. Wohnort");
        fuegeEinObjektEin("7. Landkreis");
        fuegeEinObjektEin("8. Bundesland");
        fuegeEinObjektEin("9. Land");
        fuegeEinObjektEin("10. Kontinent");
        fuegeEinObjektEin("11. Planet");
    }

    public static void main(String[] args)
    {
        Test test = new Test();
        System.out.println("Das Programm wird bis zum Ende ausgeführt!");

    }
}
```

Die Testklasse enthält drei wichtige Methoden:

1. Konstruktor

Hier wird versucht, ein Objekt `wortliste` der Klasse **MyArrayList** zu erzeugen. Die **IllegalArgumentException**, die vom Konstruktor von **MyArrayList** geworfen werden kann, wird über eine try-catch-Konstruktion abgefangen.

2. fuegeEinObjektEin(Object neu)

Es wird versucht, das Objekt `neu` in die Liste einzufügen. Zur Kontrolle und besseren Übersicht wird dann das Objekt mit dem println()-Befehl in der Konsole angezeigt. Außerdem - wieder zur Kontrolle - wird die aktuelle Größe (`size`) sowie die aktuelle Kapazität (`capacity`) der Liste ausgegeben.

3. fuegeObjekteEin()

Mit dieser Methode werden jetzt verschiedene Objekte in die Liste eingefügt, indem die private Methode fuegeEinObjektEin() aufgerufen wird. Einige dieser Objekte sind **String**-Objekte und sollten daher keine Ausnahme verursachen. Einmal wird aber `null` übergeben, was eine Ausnahme provozieren soll.

Bei der Übergabe von `23` und `true` sollten keine Fehler entstehen, denn diese beiden primitiven Datentypen werden von den neueren Java-Versionen automatisch in Objekte der ***Wrapper-Klassen*** **Integer** und **Boolean** umgewandelt und können daher problemlos in die Liste eingefügt werden (***Autoboxing***).

Ausgabe des Testprogramms:

```
Einfügen des Objektes  1. Name
Neues Objekt  1. Name            size =   1, length =   5

Einfügen des Objektes  2. Vorname
Neues Objekt  2. Vorname         size =   2, length =   5

Einfügen des Objektes  3. Straße
Neues Objekt  3. Straße          size =   3, length =   5

Einfügen des Objektes null
Fehler add(): Element ist null.

Einfügen des Objektes  4. Hausnummer
Neues Objekt  4. Hausnummer      size =   4, length =   5

Einfügen des Objektes  5. Postleitzahl
Neues Objekt  5. Postleitzahl    size =   5, length =   5

Einfügen des Objektes 23
Neues Objekt 23                  size =   6, length =   7

Einfügen des Objektes true
Neues Objekt true                size =   7, length =   7

Einfügen des Objektes  6. Wohnort
Neues Objekt  6. Wohnort         size =   8, length =  10

Einfügen des Objektes  7. Landkreis
Neues Objekt  7. Landkreis       size =   9, length =  10

Einfügen des Objektes  8. Bundesland
Neues Objekt  8. Bundesland      size =  10, length =  10

Einfügen des Objektes  9. Land
Neues Objekt  9. Land            size =  11, length =  15

Einfügen des Objektes 10. Kontinent
Neues Objekt 10. Kontinent       size =  12, length =  15

Einfügen des Objektes 11. Planet
Neues Objekt 11. Planet          size =  13, length =  15

Das Programm wird bis zum Ende ausgeführt!
```

Beim Versuch, das "Objekt" `null` einzufügen, wird die selbst erstellte Fehlermeldung ausgegeben, die teils im try-Block erzeugt wurde, teils dem Konstruktor der **IllegalArgumentException** als Parameter mitgegeben wurde. Die "Objekte" `23` und `true` werden wie erwartet problemlos in die Liste eingefügt, nämlich als Objekte der entsprechenden Wrapper-Klassen **Integer** und **Boolean**.

Was man gut erkennen kann, ist das pseudo-dynamische Wachstum der Liste. Gestartet wird mit einer Kapazität von 5, die dann nacheinander auf 7, 10 und 15 erhöht wird.

Version 2.2: Beschränkung auf "echte" Objekte

Wir wollen im nächsten Schritt die `add()`-Methode so verändern, dass keine primitiven Datentypen wie `int`, `double` oder `boolean` mehr in der Liste gespeichert werden können, sondern nur "echte" Objekte - also auch keine **Integer**-, **Double**- oder **Boolean**-Objekte.

Dazu setzen wir den `instanceof`-Operator ein:

```
    public void add(Object element)
    {
        if (element == null)
            throw new IllegalArgumentException("Element ist null.");

        if (element instanceof Integer ||
            element instanceof Double  ||
            element instanceof Boolean)
            throw new IllegalArgumentException
                ("int-, double- oder boolean-Werte sind nicht erlaubt.");

        if (size >= elementData.length)
            grow();

        elementData[size] = element;
        size++;
    }
```

Danach sollte die Konsolen-Ausgabe so aussehen:

```
Einfügen des Objektes  1. Name
Neues Objekt  1. Name            size =   1, length =   5

Einfügen des Objektes  2. Vorname
Neues Objekt  2. Vorname         size =   2, length =   5

Einfügen des Objektes  3. Straße
Neues Objekt  3. Straße          size =   3, length =   5

Einfügen des Objektes null
Fehler add(): Element ist null.

Einfügen des Objektes  4. Hausnummer
Neues Objekt  4. Hausnummer      size =   4, length =   5

Einfügen des Objektes  5. Postleitzahl
Neues Objekt  5. Postleitzahl    size =   5, length =   5

Einfügen des Objektes 23
Fehler add(): int-, double- oder boolean-Werte sind nicht erlaubt.

Einfügen des Objektes true
Fehler add(): int-, double- oder boolean-Werte sind nicht erlaubt.

Einfügen des Objektes  6. Wohnort
Neues Objekt  6. Wohnort         size =   6, length =   7

Einfügen des Objektes  7. Landkreis
Neues Objekt  7. Landkreis       size =   7, length =   7

Einfügen des Objektes  8. Bundesland
Neues Objekt  8. Bundesland      size =   8, length =  10

Einfügen des Objektes  9. Land
Neues Objekt  9. Land            size =   9, length =  10
```

```
Einfügen des Objektes 10. Kontinent
Neues Objekt 10. Kontinent       size =  10, length =  10

Einfügen des Objektes 11. Planet
Neues Objekt 11. Planet          size =  11, length =  15

Das Programm wird bis zum Ende ausgeführt!
```

Version 2.3: Die add(index)-Methode

Wie erweitern die Klasse **MyArrayList** jetzt um eine zweite `add()`-Methode, bei der die *Position* des einzufügenden Elements bestimmt werden kann:

```
public void add(int index, Object element)
// Neu in Version 2.3
{
    if (element == null)
        throw new NullPointerException("element ist null.");

    if (element instanceof Integer ||
        element instanceof Double  ||
        element instanceof Boolean)
        throw new IllegalArgumentException
           ("int-, double- oder boolean-Werte sind nicht erlaubt.");

    if (index < 0 || index > size)
        throw new IndexOutOfBoundsException("index ausserhalb: " + index);

    if (size >= elementData.length)
        grow();

    for (int i = size; i > index; i--)
        elementData[i] = elementData[i - 1];

    elementData[index] = element;
    size++;
}
```

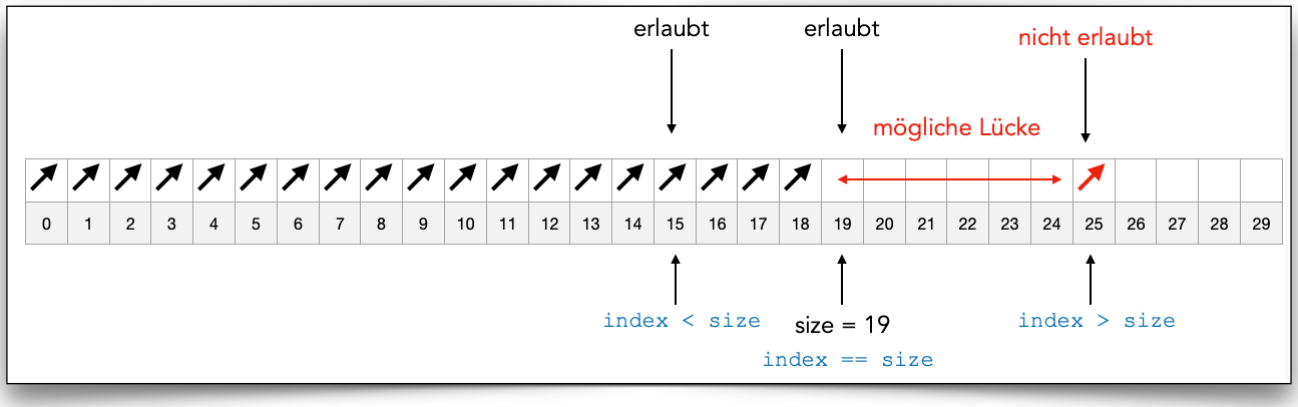
In dieser Version der `add()`-Methode begegnen wir gleich drei verschiedenen Ausnahmen.

Zunächst prüfen wir, ob das neue Element den Wert `null` hat und werfen in diesem Fall eine **NullPointerException**.

Als Zweites wird überprüft, ob das übergebene Objekt ein "echtes" Objekt ist, also kein **Integer**-, **Double**- oder **Boolean**-Objekt. Sollte das doch der Fall sein, wird eine **IllegalArgumentException** geworfen.

Schließlich wird überprüft, ob der übergebene Index innerhalb der erlaubten Werte liegt. Als Obergrenze wird hier aber nicht `elementData.length` verwendet, sondern `size`. Diese `add()`-Methode darf das neue Objekt also nur in den belegten Teil der Liste einfügen oder direkt an das Ende der Liste anhängen (`index == size`). Wenn diese Bedingungen nicht erfüllt sind, wird eine **IndexOutOfBoundsException** geworfen.

Zur IndexOutOfBoundsException:



Ein Einfügen "irgendwo" in den freien Bereich am Ende des internen Arrays ist bewusst nicht erlaubt, weil dadurch Lücken entstehen würden und `size` nicht mehr zur tatsächlichen Belegung passen würde.

Bevor nun das neue Element an der entsprechenden Position eingefügt wird, rücken die weiter rechts stehenden Elemente je eine Position weiter nach rechts. Dann wird das neue Element eingefügt und `size` inkrementiert.

Version 2.3: Das Testprogramm

Wir ergänzen das Testprogramm aus der Version 2.2 um eine weitere Methode:

```
private void fuegeEinObjektEin(int index, Object neu)
{
    System.out.println
      ("Einfügen des Objektes " + neu + " an Position " + index);

    try
    {
        wortliste.add(index, neu);
        System.out.printf("Neues Objekt %-20s", neu);
        System.out.printf("size = %3d, length = %3d %n%n",
            wortliste.size(),wortliste.capacity());
    }
    catch (NullPointerException ausnahme)
    {
        System.out.println
        ("Fehler bei add(index): " + ausnahme.getMessage() + "\n");
    }
    catch (IllegalArgumentException ausnahme)
    {
        System.out.println
        ("Fehler bei add(index): " + ausnahme.getMessage() + "\n");
    }
    catch (IndexOutOfBoundsException ausnahme)
    {
        System.out.println
        ("Fehler bei add(index): " + ausnahme.getMessage() + "\n");
    }

}
```

Diese Methode soll die zweite `add()`-Methode testen. Für jede der drei möglichen Exceptions wurde hier ein eigener catch-Block implementiert worden. So könnte man theoretisch sehr differenziert auf die verschiedenen Ausnahmen eingehen, was wir hier aber noch nicht machen.

Bei allen drei Exceptions erscheint in der Konsole die gleiche Meldung `"Fehler bei add(index): "`. Das ist noch nicht sehr spezifisch. Aber dann wird die Methode `getMessage()` des **Exception**-Objektes `ausnahme` aufgerufen, und diese Methode liefert die spezifische Fehlermeldung der jeweiligen Exception.

Man könnte sich die Sache aber auch etwas einfacher machen, indem man nur einen einzigen catch-Block implementiert, in dem alle Unterklassen von Exception abgefangen werden:

```
catch (Exception ausnahme)
{
    System.out.println
    ("Fehler bei add(index): " + ausnahme() + "\n");
}
```

Exception ist die ***Oberklasse*** aller Exception-Klassen, und dieser catch-Block fängt nun alle ***Unterklassen*** von **Exception** auf, also auch die **NullPointerException**, die **IllegalArgumentException** und die **IndexOutOfBoundsException**.

Sollte aus irgendeinem Grund eine andere Exception von add(index) geworfen werden, würde auch diese vom catch-Block aufgefangen.

Exkurs 1: Das Prinzip der Redundanz-Vermeidung

Schaut man sich die beiden add()-Methoden der Klasse **MyArrayList** genauer an, dann stellt man fest, dass der Code nahezu identisch ist:

```
public void add(Object element)
{
   if (element == null)
     throw new IllegalArgumentException("...");

   if (element instanceof Integer ||
       element instanceof Double ||
       element instanceof Boolean)
       throw new IllegalArgumentException("...");

   if (size >= elementData.length) grow();

   elementData[size] = element;
   size++;
}
```

```
public void add(int index, Object element)
{
   if (element == null)
     throw new IllegalArgumentException("...");

   if (element instanceof Integer ||
       element instanceof Double ||
       element instanceof Boolean)
       throw new IllegalArgumentException("...");

   if (index < 0 || index > size)
       throw new IndexOutOfBoundsException("...");

   if (size >= elementData.length) grow();

   for (int i = size; i > index; i--)
      elementData[i] = elementData[i - 1];

   elementData[size] = element;
   size++;
}
```

In beiden add()-Methoden wird das übergebene Objekt überprüft, ob es den Wert `null` hat. In der neuen add()-Methode wird zusätzlich der Index überprüft, ob er innerhalb der gültigen Werte liegt. Beide Methoden testen außerdem, ob der übergebene Objekt-Parameter überhaupt ein Objekt ist und nicht etwa ein primitiver Datentyp.

Ein wichtiges Prinzip der OOP besagt nun, dass man solche ***Redundanzen*** vermeiden sollte. Eine Möglichkeit, diesen doppelten Code zu vermeiden, ist die Auslagerung eben dieses Codes in eigene Methoden, die dann von den beiden add()-Methoden aufgerufen werden:

```
private void checkIndex(int index)
{
     if (index < 0 || index > size)
         throw new IndexOutOfBoundsException
          ("Index ausserhalb der erlaubten Grenzen: " + index);
 }

 private void checkObject(Object element)
 {
    if (element == null)
         throw new IllegalArgumentException
           ("Element ist null.");

     if (element instanceof Integer ||
         element instanceof Double  ||
         element instanceof Boolean)
         throw new IllegalArgumentException
         ("int-, double- oder boolean-Werte nicht erlaubt.");
 }
```

```
    private void checkSize()
    {
        if (size >= elementData.length)
            grow();
    }
```

Die letzte Methode dient zwar nicht zur Fehlerüberprüfung, sondern verlängert die Liste, falls nötig. Aber auch diese Überprüfung wird von beiden `add()`-Methoden durchgeführt, daher könnte man sie ebenfalls in eine private Hilfsmethode auslagern.

Die beiden `add()`-Methoden vereinfachen sich durch diese Auslagerung von Code wie folgt:

```
    public void add(Object element)
    {
        checkObject(element);
        checkSize();

        elementData[size] = element;
        size++;
    }

    public void add(int index, Object element)
    {
        checkObject(element);
        checkIndex(index);
        checkSize();

        for (int i = size; i > index; i--)
            elementData[i] = elementData[i - 1];

        elementData[index] = element;
        size++;
    }
```

Insgesamt haben wir jetzt zwar deutlich mehr Quelltext produziert, dafür ist dieser Quelltext nun aber wesentlich übersichtlicher und vor allem auch leichter zu warten.

Stellen Sie sich vor, Sie hätten die Aufgabe, das übergebene Objekt oder den übergebenen Index auf weitere Fehlerquellen zu überprüfen, was den Einbau zusätzlicher Exceptions zur Folge hätte. Dann müssten Sie stets beide `add()`-Methoden modifizieren. Wenn Sie das dann bei einer der beiden Methoden vergessen, haben Sie ein großes Problem.

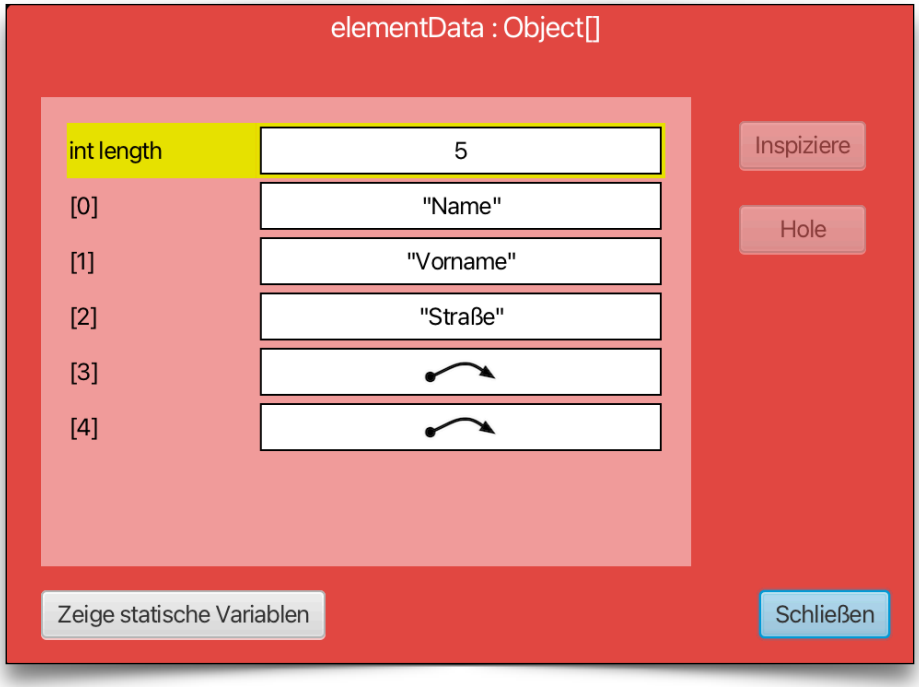
Durch die Auslagerung in private Hilfsmethoden müssen Sie diese Veränderungen aber nur an *einer* Stelle im Code vornehmen.

Exkurs 2: Eine heterogene Liste ist möglich

Bisher konnte das interne Array der Klasse **MyArrayList** beliebige Objekte von beliebigen Klassen speichern. Auch ***heterogene Listen*** können so angelegt werden, wie die folgende Methode `fuegeObjekteEin()` einer Test-Klasse zeigt:

```
public void fuegeObjekteEin()
{
   fuegeEinObjektEin("Name");
   fuegeEinObjektEin("Vorname");
   fuegeEinObjektEin("Straße");
   fuegeEinObjektEin(new Name("Anna","Maibaum"));
   fuegeEinObjektEin(new Punkt(40,60));

}
```

Wenn wir das Objekt `elementData`, also das interne Array unserer Klasse **MyArrayList**, mit dem BlueJ-Objektinspektor untersuchen, sehen wir, dass die Kapazität (`length`) der Liste den Wert 5 hat. :



Die drei ersten Array-Elemente sind Referenzen auf **String**-Objekte, das vierte ist eine Referenz auf ein Objekt der Klasse **Name**, und das fünfte Element ist eine Referenz auf ein **Punkt**-Objekt.

Natürlich müssen diese Objekte bzw. die zugehörigen Klassen vorher im Projekt vorhanden sein, damit das Testprogramm korrekt funktioniert.

8.1.4 Eine get()-Methode für MyArrayList

Als Nächstes wollen wir unsere Klasse um eine get()-Methode erweitern, damit wir die Objekte, die wir in die Liste eingefügt haben, auch wieder auslesen können.

Die Methode MyArrayList.get()

Die Methode get() lässt sich zunächst sehr einfach implementieren:

```
public Object get(int index)
{
    checkIndex(index);
    return elementData[index];
}
```

Zuerst wird der übergebene Index mit checkIndex() überprüft. Ist der Index gültig, kann das Element an dieser Position direkt zurückgegeben werden.

Beim Testen zeigt sich allerdings ein Problem: Greift man mit get() auf den Index `size` zu, bricht das Programm mit einer Exception ab. Der Fehler liegt aber nicht in get() selbst, sondern in der Prüfmethode, die wir implementiert hatten:

```
private void checkIndex(int index)
{
    if (index < 0 || index > size)
        throw new IndexOutOfBoundsException
            ("index ausserhalb: " + index);
}
```

Diese Bedingung ist für add(index, element) sinnvoll, denn dort ist `index == size` ausdrücklich erlaubt: Ein neues Element darf am Ende der Liste eingefügt werden.

Für die Methode get(index) gilt das jedoch nicht. Bei einer Liste mit 5 Elementen sind nur die Indizes 0 bis 4 gültig. Der Index 5 zeigt bereits *hinter* das letzte gespeicherte Element und ist daher nicht mehr erlaubt. Deshalb erhält get() eine zusätzliche Prüfung:

```
public Object get(int index)
{
    checkIndex(index);

    if (index == size)
        throw new IndexOutOfBoundsException
           ("index ausserhalb: " + index);

    return elementData[index];
}
```

Damit bleibt checkIndex() weiter für mehrere Methoden verwendbar, während get() den Sonderfall `index == size` zusätzlich abfängt.

Unterschiedliche Index-Prüfungen

Ein kritischer Betrachter könnte nun einwenden, ob es sinnvoll ist, in der Klasse **MyArrayList** eine Methode checkIndex() zu implementieren, die zwar für Methoden wie add() sinnvoll ist, aber nicht für die get()-Methode. Die get()-Methode musste bei unserem bisherigen Vorgehen noch eine eigene zusätzliche Index-Prüfung durchführen.

Eine bessere Lösung wäre es, zwei verschiedene Prüfmethoden zu implementieren:

```
private void checkElementIndex(int index)
{
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException
            ("index ausserhalb: " + index);
}


private void checkPositionIndex(int index)
{
    if (index < 0 || index > size)
        throw new IndexOutOfBoundsException
            ("index ausserhalb: " + index);
}
```

Die Methode checkElementIndex() wird für den Zugriff auf *vorhandene* Elemente verwendet, also zum Beispiel in get(), set() und remove().

Die Methode checkPositionIndex() wird dagegen für Einfügepositionen verwendet, also zum Beispiel in add(index, o).

Die get()-Methode würde dann so aussehen:

```
public Object get(int index)
{
    checkElementIndex(index);
    return elementData[index];
}
```

Und die add(index)-Methode sieht dann so aus:

```
    public void add(int index, Object element)
    {
        checkObject(element);
        checkPositionIndex(index);
        checkSize();

        for (int i = size; i > index; i--)
            elementData[i] = elementData[i - 1];

        elementData[index] = element;
        size++;
    }
```

8.1.5 Eigene Weiterentwicklung der Klasse

Erweitern Sie Ihre Klasse **MyArrayList** um die folgenden Methoden, die man von einer Listenklasse erwarten würde.

void set(int index, Object o)

Diese Methode ersetzt das Element an der Position `index` durch das Objekt `o`. Die übrigen Elemente der Liste bleiben an ihrer Position, und die Größe der Liste (`size`) verändert sich nicht. Achten Sie darauf, dass nur ein bereits vorhandener Index erlaubt ist. Wählen Sie die entsprechende private `check()`-Methode.

void remove(int index)

Diese Methode entfernt das Element an der angegebenen Position. Alle rechts daneben stehenden Elemente rücken um jeweils eine Position nach links, und `size` verringert sich um 1. Der durch das Aufrücken frei gewordene letzte Platz im internen Array soll dann auf `null` gesetzt werden, damit keine alte Referenz erhalten bleibt.

void clear()

Diese Methode entfernt alle Elemente aus der Liste. Nach dem Aufruf ist die Liste leer, alle bisherigen Referenzen werden auf `null` gesetzt. Die Kapazität der Liste (also `length`) ändert sich durch `clear()` jedoch nicht.

boolean isEmpty()

Diese Methode liefert den Wert `true`, wenn die Liste leer ist, also kein Element enthält. Andernfalls liefert sie `false`.

void trimToSize()

Diese Methode verkleinert das interne Array so, dass seine Länge genau der aktuellen Anzahl gespeicherter Elemente entspricht.

boolean contains(Object o)

Liefert `true`, wenn ein zu `o` gleiches Objekt in der Liste enthalten ist, sonst `false`.

int indexOf(Object o)

Liefert den Index des ersten zu `o` gleichen Elements zurück oder `-1`, falls kein solches Element in der Liste enthalten ist.

Hinweis zu den beiden letzten Methoden

Jedes Objekt in Java besitzt eine Methode `equals()`. Mit ihr kann geprüft werden, ob zwei Objekte als gleich angesehen werden sollen. Der Aufruf `a.equals(b)` liefert den Wert `true`, wenn die beiden Objekte `a` und `b` gleich sind.

Achtung: `equals()` prüft die *inhaltliche* Gleichheit. Mit dem Operator `==` wird dagegen nur geprüft, ob zwei Referenz-Variablen auf genau dasselbe Objekt *verweisen*.

Beispiel:

```
String s1 = new String("Hallo");
String s2 = new String("Hallo");
```

Hier liefert `s1 == s2` den Wert `false`, weil beide Variablen auf *unterschiedliche* Objekte im Speicher verweisen.

Dagegen liefert `s1.equals(s2)` den Wert `true`, weil beide String-Objekte *inhaltlich* gleich sind.

8.1.6 MyArrayList wird generisch

Einleitung und Begründung

In den bisherigen Versionen unserer Klasse **MyArrayList** konnten wir beliebige Objekte im internen Array `elementData` speichern. Das war möglich, weil dieses Array als **Object**-Array angelegt war. Auf diese Weise konnten in einer einzigen Liste beispielsweise **String**-Objekte, **Name**-Objekte oder auch **Punkt**-Objekte gemeinsam gespeichert werden. Eine solche Liste nennt man ***heterogen***.

Dieses Vorgehen hatte jedoch auch Nachteile. Da in einer Liste Objekte unterschiedlicher Klassen liegen können, ist beim Auslesen eines Elements zunächst unklar, welchen Typ es hat. Dies erschwert die Arbeit mit der Liste und erhöht die Fehleranfälligkeit. Typumwandlungen sind häufig erforderlich, und genau dabei können Laufzeitfehler auftreten.

Aus diesem Grund werden wir MyArrayList jetzt *generisch* machen.

Eine generische Klasse wird nicht mehr für beliebige Objekte formuliert, sondern für Objekte eines bestimmten Typs. Man kann dann zum Beispiel mit

```
MyArrayList<String> wortliste = new MyArrayList<>();
```

eine Liste für Wörter anlegen, in der nur **String**-Objekte gespeichert werden können.

Mit

```
MyArrayList<Punkt> punktliste = new MyArrayList<>();
```

kann man entsprechend eine Liste anlegen, in der nur Objekte der Klasse **Punkt** gespeichert werden können.

Da der Typ der gespeicherten Elemente von Anfang an eindeutig festgelegt ist, kann der Compiler bereits beim Übersetzen des Programms viele Fehler erkennen. Bei einer Anweisung wie

```
punktliste.add("Mein Name");
```

würde der Compiler die Übersetzung mit einer Fehlermeldung abbrechen, weil in die Liste `punktliste` ein Objekt vom Typ **String** eingefügt werden soll, obwohl dort nur **Punkt**-Objekte erlaubt sind.

Der Quelltext der generischen Klasse, Teil 1

Betrachten wir nun den Anfang des modifizierten Quelltextes der Klasse MyArrayList. Im Vergleich zur bisherigen Version wurde die Klasse **MyArrayList** an den entscheidenden Stellen so umgebaut, dass sie ***generisch*** ist.

Das Ziel dieser Änderung ist, dass ein Objekt der Klasse **MyArrayList** nicht mehr *beliebige* Objekte speichern soll, sondern nur noch Objekte eines vorher festgelegten Typs.

```
import java.util.Arrays;

public class MyArrayList<T> // Änderung 1
{
    // Instanzvariablen
    // ========================================================
    private T[] elementData; // Änderung 2
    private int size;

    // Zwei Konstruktoren
    // ========================================================
    public MyArrayList(int startKapazitaet)
    {
        if (startKapazitaet <= 0)
            throw new IllegalArgumentException
            ("Ungueltige Startkapazitaet: " + startKapazitaet);

        elementData = (T[]) new Object[startKapazitaet]; // Änderung 3
        size = 0;
    }

    public MyArrayList()
    {
        this(10);
    }
```

Eigentlich wurde der Quelltext nur an drei Stellen verändert (rot markiert und kommentiert). Wir wollen diese Änderungen nun genauer betrachten.

1. Änderungen in der Kopfzeile

In der Kopfzeile der Klasse steht jetzt

```
public class MyArrayList <T>
```

Dabei ist `<T>` der sogenannte ***Typ-Parameter*** der Klasse. Ein solcher Typ-Parameter steht als ***Platzhalter*** für einen konkreten Datentyp, der erst beim Erstellen eines **MyArrayList**-Objekts festgelegt wird. Wenn wir später zum Beispiel schreiben

```
MyArrayList<String> wortliste = new MyArrayList<>();
```

dann steht `T` in dieser Liste für **String**. Bei

```
MyArrayList<Punkt> punktliste = new MyArrayList<>();
```

steht `T` entsprechend für **Punkt**.

2. Änderungen an elementData

In der bisherigen Version von **MyArrayList** war die Instanzvariable `elementData` so deklariert:

```
private Object[] elementData;
```

Statt dessen schreiben wir jetzt:

```
private T[] elementData;
```

Auch das ist eine wichtige Änderung. Das interne Array soll nun nicht mehr allgemein ein **Object**-Array sein, sondern ein Array für Elemente des Typs `T`.

Wenn `T` also **String** ist, dann wird `elementData` wie ein **String**-Array behandelt. Wenn `T` dagegen **Punkt** ist, ist `elementData` quasi ein **Punkt**-Array.

3. Änderungen im Konstruktor

Im Konstruktor steht jetzt:

```
elementData = (T[]) new Object[startKapazitaet];
```

Das sieht zunächst vielleicht etwas merkwürdig aus. Warum wird hier nicht einfach

```
elementData = new T[startKapazitaet];
```

geschrieben?

Der Grund ist: **In Java kann man kein Array eines Typ-Parameters direkt erzeugen**. Eine Anweisung wie

```
new T[startKapazitaet];
```

ist nicht erlaubt. Deshalb wird zunächst ein Object-Array erzeugt:

```
new Object[startKapazitaet]
```

und dieses Array wird dann mit `([T])` in ein Array vom Typ `T[]` umgewandelt.

Der Quelltext der generischen Klasse, Teil 2

Ein Großteil der Methoden von **MyArrayList** musste nicht verändert werden, beispielsweise die Methode

```
public int size()
{
    return size;
}
```

oder die Methode

```
private void checkElementIndex(int index)
// Überprüft den Index für vorhandene Elemente.
// Sinnvoll für get(), set() und remove()
{
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException
        ("index ausserhalb: " + index);
}
```

Aber die folgende Methode wurde wieder angepasst:

```
private void checkObject(T element) // wurde angepasst
{
    if (element == null)
        throw new IllegalArgumentException
          ("Element ist null.");
}
```

In der nicht-generischen Version sah die Methode noch so aus:

```
private void checkObject(Object element)
{
    if (element == null)
        throw new IllegalArgumentException("Element ist null.");

    if (element instanceof Integer ||
        element instanceof Double ||
        element instanceof Boolean)
        throw new IllegalArgumentException
          ("int-, double- oder boolean-Werte sind nicht erlaubt.");

}
```

Wenn wir beispielsweise ein Objekt des Typs `MyArrayList<String>` anlegen, dann verhindert bereits der Compiler, dass eine `int`- oder **Integer**-Variable eingefügt wird. Eine zusätzliche Laufzeitprüfung auf **Integer**, **Double** oder **Boolean** ist dann überflüssig.

Die Prüfung auf `null` ist dagegen weiterhin sinnvoll, wenn `null`-Werte in der Liste grundsätzlich nicht *als Elemente* erlaubt sein sollen - ausgenommen sind hier natürlich die `null`-Werte, die beim Erzeugen oder Vergrößern der Liste automatisch angelegt werden.

Bei den folgenden Methoden von **MyArrayList** muss lediglich das Wort `Object` durch den Buchstaben `T` ersetzt werden:

- `public void add(Object element)`
- `public void add(int index, Object element)`
- `public Object get(int index)`
- `public void set(int index, Object element)`
- `public boolean contains(Object element)`
- `public int indexOf(Object element)`

Die Methode `set()` wurde beispielsweise wie folgt verändert:

Nicht-generische Version:

```
public void set(int index, Object element)
{
    checkObject(element);
    checkElementIndex(index);
    elementData[index] = element;
}
```

Generische Version:

```
public void set(int index, T element)
{
    checkObject(element);
    checkElementIndex(index);
    elementData[index] = element;
}
```

Den kompletten Quelltext der generischen Version von **MyArrayList** können Sie sich hier herunterladen:

MyArrayList.java

TestMyArrayList.java (von ChatGPT erzeugt)

Name.java

Punkt.java

8.2 Exception-Handling in Java

In dem sehr praxisorientiertem Abschnitt 8.1 haben wir wir Java-Exceptions am Beispiel der selbst erstellen Klasse **MyArrayList** kennengelernt. Nun wollen wir das Thema etwas allgemeiner betrachten.

Bei den Ausnahmen, die zu einem Programmabbruch führen können, gibt es zwei Varianten.

1. ***Kontrollierte Ausnahmen***: Das sind Exceptions, die vom Programm behandelt werden *müssen*.
2. ***Nicht kontrollierte Ausnahmen***: Das sind Exceptions, die vom Programm zwar behandelt werden *können*, aber nicht zwingend behandelt werden *müssen*.

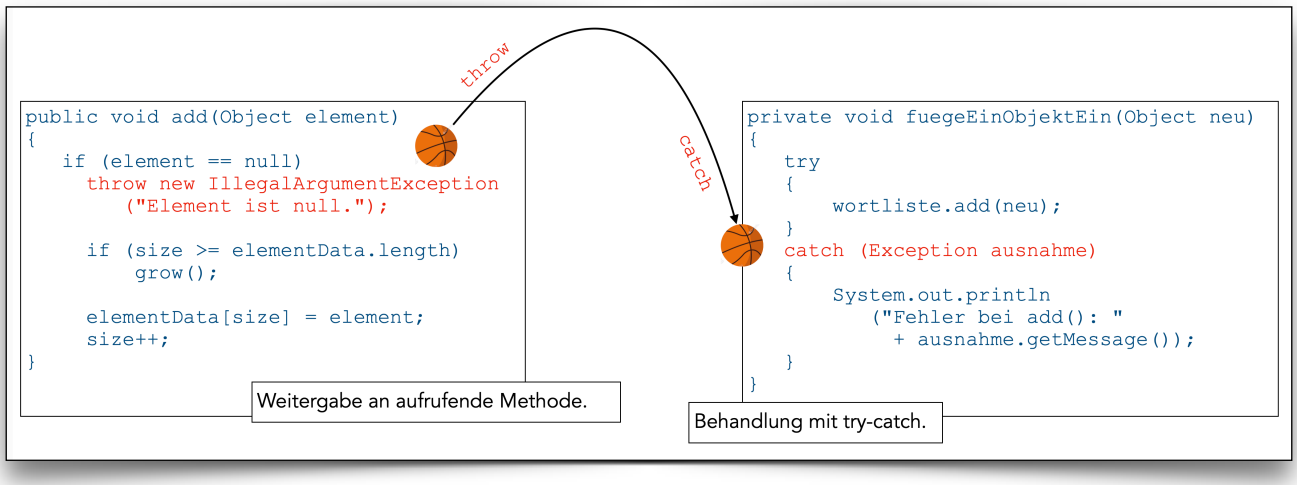
Schwerpunkt dieses Abschnitts 8.2 werden die ***nicht kontrollierten Ausnahmen*** sein, wie wir sie bereits bei der Erstellung der Klasse **MyArrayList** kennengelernt haben. Zunächst aber wollen wir recht kurz über die ***kontrollierten Ausnahmen*** sprechen. Kontrollierte Ausnahmen werden uns begegnen, sobald wir anfangen, mit Dateien zu arbeiten.

Was ist eine Exception?

In Java wird in einer Ausnahmesituation ein Exception-Objekt erzeugt und "geworfen" (throw). Wenn das Programm diese Exception nicht behandelt, beendet die JVM (Java Virtual Machine) das Programm und gibt eine Fehlermeldung aus.

In dem Buch von D. ABTS, *Grundkurs Java* von 2020, ist auf Seite 127 sehr übersichtlich dargestellt, wie das Grundprinzip dieses Ausnahme-Mechanismus aussieht. Wir übernehmen diese Übersicht einmal komplett als Zitat:

- *Das Laufzeitsystem erkennt eine Ausnahmesituation bei der Ausführung einer Methode, erzeugt ein Objekt einer bestimmten Klasse (Ausnahmetyp) und löst damit eine Ausnahme aus.*
- *Die Ausnahme kann entweder in derselben Methode abgefangen und gleich behandelt werden oder sie kann an die aufrufende Methode weitergereicht werden.*
- *Die Methode, an die die Ausnahme weitergereicht wurde, hat nun ihrerseits die Möglichkeit, diese entweder abzufangen und zu behandeln oder ebenfalls weiterzureichen.*
- *Wird die Ausnahme nur immer weitergereicht und in keiner Methode behandelt, bricht das Programm mit einer Fehlermeldung ab.*



Hier sehen wir das Weitergeben an die aufrufende Methode und die Behandlung mit try-catch in der aufrufenden Methode noch einmal an einem bekannten Beispiel.

8.2.1 Kontrollierte Ausnahmen

Kontrollierte Ausnahmen (checked exceptions) sind solche, die vom Compiler ausdrücklich überwacht werden.

Das heißt, wenn eine Methode eine solche kontrollierte Ausnahme auslösen kann, zwingt der Compiler den Entwickler, sich darum zu kümmern. Die Methode muss die Ausnahme entweder

- mit einem `try-catch`-Block behandeln oder
- mit dem Schlüsselwort `throws` in der Methodensignatur deklarieren und an die aufrufende Methode weitergeben, die sich dann um die Fehlerbehandlung kümmert.

Wird eine solche kontrollierte Ausnahme nicht behandelt, kann der Code nicht kompiliert werden.

Das ist ein wichtiger Unterschied zu den nicht kontrollierten Ausnahmen. Wenn wir in den Quelltext beispielsweise den folgenden Code schreiben

```
for (int i=0; i<array.length+3; i++)
   System.out.println(array[i]);
```

dann wird der Compiler weder eine Fehlermeldung ausgeben noch eine Behandlung dieser Ausnahme verlangen. Erst zur Laufzeit tritt dann eine **ArrayIndexOutOfBoundsException** auf, und das Programm bricht ab, wenn die Ausnahme nicht abgefangen wurde. Diese Ausnahme muss also nicht zwingend behandelt werden.

Typische kontrollierte Ausnahmen

IOException

Allgemeiner Ein-/Ausgabefehler, zum Beispiel wenn aus einer Datei gelesen werden soll, die aus irgendeinem Grund nicht lesbar ist, oder wenn auf der Festplatte ein Schreibfehler vorliegt, die Netzwerkverbindung während des Lesens/Schreibens abbricht und so weiter.

FileNotFoundException

Das Objekt **FileNotFoundException** ist eine Unterklasse von **IOException**. Eine solche Ausnahme tritt beispielsweise auf, wenn eine Datei, auf die zugegriffen werden soll, gar nicht existiert oder wenn keine Zugriffsrechte für die Datei vorhanden sind.

SQLException

Diese Ausnahme bezieht sich auf Fehler bei Zugriffen auf eine Datenbank. Ursache hierfür können falsche Zugangsdaten, fehlerhafte SQL-Anweisungen oder ein nicht erreichbarer Datenbank-Server sein.

Weitere kontrollierte Ausnahmen

Es gibt noch eine ganze Reihe weiterer kontrollierter Ausnahmen wie **ClassNotFoundException**, **InterruptedException**, **ParseException** oder **MalformedURLException**.

Gemeinsames Merkmal all dieser Exceptions ist, dass sie meistens bei Dateioperationen, Netzwerkzugriffen, Datenbankzugriffen oder bei der Thread-Programmierung auftreten.

Eine Methode muss solche kontrollierten Ausnahmen entweder selbst behandeln oder mit `throws` an die aufrufende Methode weitergeben, die sich dann um die Fehlerbehandlung kümmern muss.

8.2.2 Nicht kontrollierte Ausnahmen

Wie bereits zu Beginn des Abschnitts 8 gesagt, haben wir mit den nicht kontrollierten Ausnahmen bereits jede Menge Erfahrungen gemacht, als wir die Klasse **MyArrayList** entwickelten.

1. IllegalArgumentException

Codebeispiel 1

```
public MyArrayList(int startKapazitaet)
{
    if (startKapazitaet <= 0)
        throw new IllegalArgumentException
        ("Ungueltige Startkapazitaet: " + startKapazitaet);

    elementData = (T[]) new Object[startKapazitaet];
    size = 0;
}
```

Dieses Beispiel kommt aus der Klasse ***MyArrayList***, die wir hier entwickelt haben. Die **IllegalArgumentException** dient dazu, unzulässige Argumente oder Parameterwerte kenntlich zu machen.

Codebeispiel 2

```
public class Kreis
{
    private double radius;

    public Kreis(double radius)
    {
        if (radius <= 0)
        {
            throw new IllegalArgumentException("Der Radius muss positiv sein.");
        }

        this.radius = radius;
    }

    public double getFlaeche()
    {
        return Math.PI * radius * radius;
    }
}
```

Wenn hier also für den Radius des Kreises ein negativer Wert übergeben wird, dann wird eine **IllegalArgumentException** geworfen. Die JVM würde einen negativen Wert für `radius` durchaus akzeptieren und das Programm nicht automatisch beenden.

Daher unterscheidet sich die **IllegalArgumentException** von Exceptions wie **ArithmeticException**, **ArrayIndexOutOfBoundsException** oder **NullPointerException**, die in typischen Fehlersituationen oft *automatisch* zur Laufzeit entstehen.

Codebeispiel 3

```
public void setGewicht(double gewicht)
{
    if (gewicht < 3)
        throw new IllegalArgumentException
        ("Gewicht ist unter 3 kg!");
    if (gewicht > 130)
        throw new IllegalArgumentException
        ("Gewicht ist über 130 kg!");

    this.gewicht = gewicht;
}
```

Codebeispiel 4

```
public double berechneBMI(double gewicht, double groesse)
{
    if (gewicht <= 0)
        throw new IllegalArgumentException
        ("Gewicht muss größer als 0 sein.");

    if (groesse <= 0)
        throw new IllegalArgumentException
        ("Die Größe muss größer als 0 m sein.");

    if (groesse > 2.5)
        throw new IllegalArgumentException
        ("Die Größe muss kleiner als 2,5 m sein.");

    return gewicht / (groesse * groesse);
}
```

Mögliche Klausuraufgabe

Eine mögliche Aufgabe in der Klausur könnte darin bestehen, dass Sie den Quelltext der Klasse **Waage** erhalten und die Methoden dann um Exceptions und try-catch-Blöcke erweitern müssen.

2. IndexOutOfBoundsException

Diese Exception wird verwendet, wenn man mit Listen oder ähnlichen Datenstrukturen arbeitet und ein Index außerhalb des zulässigen Bereichs liegt. Bei Listen wie **ArrayList** tritt in solchen Fällen meist eine **IndexOutOfBoundsException** auf, bei richtigen Arrays dagegen eine **ArrayIndexOutOfBoundsException**. Diese Ausnahme ist eine Unterklasse der **IndexOutOfBoundsException**.

Codebeispiel 1

```
    private void checkElementIndex(int index)
    {
        if (index < 0 || index >= size)
            throw new IndexOutOfBoundsException
            ("index ausserhalb: " + index);
    }
```

Dies ist eine private Hilfsmethode aus der Klasse **MyArrayList**. Der Parameter `index` darf einerseits nicht negativ sein, denn Array-Elemente mit negativen Indizes gibt es nicht. Andererseits darf `index` auch nicht den Wert `size` oder einen noch größeren Wert haben. Beide Bedingungen werden in der if-Anweisung geprüft. Ist eine von ihnen erfüllt, wird eine **IndexOutOfBoundsException** geworfen.

Codebeispiel 2

```
public class DemoArray
{
    public static void main(String[] args)
    {
        int[] zahlen = new int[3];

        zahlen[0] = 10;
        zahlen[1] = 20;
        zahlen[2] = 30;

        System.out.println(zahlen[3]);
    }
}
```

Ein Array-Element mit dem Index 3 existiert nicht, dieser Index liegt außerhalb des gültigen Bereichs 0 ... 2. Die JVM erzeugt hier dann eine **ArrayIndexOutOfBoundsException**.

Aufgabe für zwischendurch

Was ist an dem folgenden Code falsch - und warum?

```
public class Kreis
{
    private double radius;

    public Kreis(double radius)
    {
        if (radius <= 0)
            throw new IndexOutOfBoundsException
            ("ungültiger Wert: " + radius);

        this.radius = radius;
    }
}
```

Das wäre übrigens auch eine schöne kleine **Klausuraufgabe**.

Wir wollen die in der Aufgabe gestellte Frage gleich beantworten:

Dieses Beispiel ist zwar technisch möglich, aber inhaltlich nicht sinnvoll. Ein negativer oder zu kleiner Radius hat nichts mit einem Index zu tun.

Dieses Beispiel verdeutlicht, dass man im Code nicht wahllos Exceptions werfen sollte, sondern immer die passende Exception für den erwarteten Fehler wählen sollte. In diesem Fall wäre eine **IllegalArgumentException** die bessere Wahl.

Codebeispiel 3

```
public class TextDemo
{
    public char zeichenAnPosition(String text, int index)
    {
        if (index < 0 || index >= text.length())
            throw new IndexOutOfBoundsException
            ("ungueltiger Index: " + index);

        return text.charAt(index);
    }
}
```

Dieses Beispiel ist sinnvoller als das aus der Zwischendurch-Aufgabe. Der Parameter `index` gibt eine Position innerhalb eines **Strings** an. Ist der Index negativ oder größer bzw. gleich der String-Länge, dann liegt er außerhalb des gültigen Bereichs. In diesem Fall wird deshalb eine **IndexOutOfBoundsException** geworfen.

Codebeispiel 4

```
public class MatrixDemo
{
    private int[][] matrix;

    public MatrixDemo()
    {
        matrix = new int[3][4];
    }

    public int getWert(int zeile, int spalte)
    {
        if (zeile < 0 || zeile >= matrix.length)
            throw new IndexOutOfBoundsException
            ("ungueltige Zeile: " + zeile);

        if (spalte < 0 || spalte >= matrix[0].length)
            throw new IndexOutOfBoundsException
            ("ungueltige Spalte: " + spalte);

        return matrix[zeile][spalte];
    }
}
```

Hier sehen wir die Anwendung der **IndexOutOfBoundsException** auf ein zweidimensionales Array, bei dem zwei Indizes geprüft werden müssen. Daher kann die **IndexOutOfBoundsException** auch zweimal geworfen werden, jeweils mit einer sinnvollen Fehlermeldung.

Warum wird bei diesem Beispiel keine ArrayIndexOutOfBoundsException verwendet?

Eine **ArrayIndexOutOfBoundsException** tritt immer dann auf, wenn bei einem *tatsächlichen* Array-Zugriff ein ungültiger Index verwendet wird, also etwa bei `zahlen[15]` in einem Array aus nur 10 Elementen. Diese Exception wird dann automatisch von der JVM ausgelöst, was in der Regel einen Programmabsturz zur Folge hat.

In der Methode `getWert()` werden die beiden Indizes aber bewusst selbst geprüft. Die Methode ist eine öffentliche Getter-Methode, die auf eine Datenstruktur zugreift. Für solche Methoden ist die I**ndexOutOfBoundsException** die bessere Wahl.

3. ArithmeticException

Eine **ArithmeticException** tritt bei ganzzahliger Division durch 0 auf; andere Anwendungsfälle für diese Exception sind sehr selten. Betrachten wir einmal die folgende for-Schleife:

```
for (int i = 10; i > -10; i--)
   x = y / i;
```

Sobald die Laufvariable `i` bei ihrem Abstieg den Wert `0` erreicht hat, versucht das Programm, durch 0 zu dividieren. Dabei löst die JVM eine **ArithmeticException** aus.

Die folgende Variante löst ebenfalls eine **ArithmeticException** aus, diesmal jedoch ausdrücklich mit `throw`:

```
for (int i = 10; i > -10; i--)
{
   if (i == 0)
      throw new ArithmeticException("Division durch 0!");
   x = y / i;

}
```

Hier wird die Exception also nicht erst durch die Division verursacht, sondern bereits vorher absichtlich ausgelöst.

Das gilt übrigens nur für die ganzzahlige Division, also etwa bei `int` oder `long`. Bei `double` oder `float` führt eine Division durch 0 nicht zu einer **ArithmeticException**, sondern zu speziellen Werten wie `Infinity` oder `NaN`.

Vergleichen wir einmal an einem einfachen Beispiel, welche Vorteile uns das Exception-Handling bringt.

Bisher

Bisher, also bevor wir uns mit Exceptions beschäftigt hatten, mussten wir eine Methode wie `quotient()` so implementieren:

```
public double quotient(double zaehler, double nenner)
{
   if (nenner == 0)
   {
      System.out.println("Division durch 0!");
      return 0;
   }
   else
      return zaehler / nenner;
}
```

Der Zähler wird hier als Parameter übergeben und kann durchaus auch einmal den Wert 0 haben - die Methode `quotient()` muss mit dieser Möglichkeit rechnen und darauf vorbereitet sein. In diesem Fall wird dann eine Fehlermeldung in der Konsole ausgegeben und

die Methode mit `return 0` verlassen. Irgendein `double`-Wert muss auf jeden Fall mit `return` zurückgegeben werden, da die Signatur der Methode dies ausdrücklich verlangt.

Jetzt

Die selbe Methode würden wir jetzt folgendermaßen implementieren:

```
public double quotient(double zaehler, double nenner)
{
   if (nenner == 0)
      throw new ArithmeticException("Division durch 0!");
   return zaehler / nenner;
}
```

Wenn der Nenner den Wert 0 haben sollte, wird eine **ArithmeticException** geworfen, um die sich dann die aufrufende Methode kümmern muss - entweder weiterreichen oder selbst mit try-catch behandeln.

Übrigens ist der Ausdruck `nenner == 0` nicht optimal für `double`-Werte, aber das ist ein anderes Thema und soll hier nicht vertieft werden.

Seltene Fälle von ArithmeticException

In den allermeisten Fällen tritt eine **ArithmeticException** auf, wenn bei einer ganzzahligen Division durch 0 geteilt wird. Das ist der typische und mit Abstand häufigste Fall.

Es gibt aber auch seltenere Situationen, in denen eine **ArithmeticException** geworfen werden kann, nämlich dann, wenn bei einer ganzzahligen Rechnung ein Überlauf ausdrücklich erkannt werden soll. Das betrifft jedoch nicht die normalen Rechenoperatoren wie +, * oder ++, denn diese lösen bei int-Werten in Java normalerweise keine Exception aus. Stattdessen entsteht einfach ein falscher, "umgesprungener" Wert.

Eine **ArithmeticException** wegen Überlaufs erhält man nur dann, wenn man spezielle Methoden der Klasse **Math** verwendet, zum Beispiel `Math.incrementExact()` oder `Math.addExact()`. Diese Methoden prüfen, ob das Ergebnis noch im zulässigen Wertebereich des Datentyps int liegt.

```
public class OverflowDemo
{
    public static void main(String[] args)
    {
        int x = Integer.MAX_VALUE;
        x = Math.incrementExact(x);

        System.out.println(x);
    }
}
```

In diesem Beispiel hat `x` bereits den größtmöglichen int-Wert. Eine weitere Erhöhung ist daher nicht mehr möglich. Die Methode `Math.incrementExact()` erkennt diesen Überlauf und wirft eine ArithmeticException.

4. NullPointerException

Wenn man größere Programme mit vielen Klassen und Objekten schreibt, kommt eine **NullPointerException** recht häufig vor. Eine solche Exception entsteht immer dann, wenn man über eine Referenz auf ein Objekt zugreifen möchte, diese Referenz aber den Wert `null` hat. In diesem Fall existiert also kein Objekt, auf das tatsächlich zugegriffen werden könnte.

Die Ursache ist oft, dass ein Objekt nicht erzeugt wurde, eine Referenz-Variable nie einen sinnvollen Wert erhalten hat oder dass ein Array zwar angelegt wurde, seine einzelnen Elemente aber noch nicht mit Objekten gefüllt wurden.

Codebeispiel 1

```
public class DemoNull
{
    public static void main(String[] args)
    {
        String text = null;
        System.out.println(text.length());
    }
}
```

Wie jede Variable vom Typ **String** ist auch `text` eine ***Referenzvariable***. Sie kann also auf ein konkretes Objekt im Speicher verweisen. In diesem Beispiel wurde `text` jedoch ausdrücklich auf `null` gesetzt. Die Variable verweist daher auf kein einziges Objekt.

Der Aufruf von `text.length()` führt deshalb zu einem Laufzeitfehler: Es wird eine **NullPointerException** geworfen.

Bei diesem Beispiel sieht man die Fehlerursache sofort. Als Nächstes schauen wir uns ein Beispiel an, bei dem die Fehlerursache nicht ganz so augenfällig ist.

Codebeispiel 2

```
public class NullDemo
{
    public static void main(String[] args)
    {
        Person[] personen = new Person[3];

        personen[0] = new Person("Meier");
        personen[1] = new Person("Schulze");

        for (int i = 0; i < personen.length; i++)
        {
            System.out.println(personen[i].getName());
        }
    }
}
```

Hier wurde ein Array für drei Objekte der Klasse **Person** angelegt. Dabei ist aber zunächst nur das Array selbst erzeugt worden. Die einzelnen Array-Elemente enthalten anfangs noch den Wert `null`.

Anschließend werden nur die Elemente `personen[0]` und `personen[1]` mit echten Person-Objekten belegt. Das dritte Element `personen[2]` bleibt dagegen null.

Im letzten Durchgang der for-Schleife wird daher versucht, `getName()` für `personen[2]` aufzurufen. Da dort aber gar kein Objekt vorhanden ist, entsteht eine **NullPointerException**.

Codebeispiel 3

```
public class Konto
{
    private String inhaber;

    public void ausgabe()
    {
        System.out.println(inhaber.toUpperCase());
    }
}
```

In diesem Beispiel wurde die Instanzvariable `inhaber` zwar deklariert, aber nirgends mit einem String-Objekt belegt. Sie hat daher automatisch den Wert `null`.

Wenn die Methode `ausgabe()` aufgerufen wird, versucht das Programm, mit `toUpperCase()` auf ein **String**-Objekt zuzugreifen. Da inhaber aber `null` ist, wird eine **NullPointerException** ausgelöst.

Dieses Beispiel zeigt, dass der Fehler auch bei Instanzvariablen auftreten kann, nicht nur bei lokalen Variablen oder Array-Elementen.

Codebeispiel 4

```
public class NullRueckgabe
{
    public static String liefereText(boolean ok)
    {
        if (ok)
            return "Hallo";

        return null;
    }

    public static void main(String[] args)
    {
        String s = liefereText(false);
        System.out.println(s.length());
    }
}
```

Hier liefert die Methode liefereText() nicht in jedem Fall ein echtes **String**-Objekt zurück. Wenn der Parameter `ok` den Wert false hat, wird stattdessen `null` zurückgegeben.

In der main()-Methode wird dieses Ergebnis in der Variablen `s` gespeichert. Anschließend wird sofort s.length() aufgerufen. Da `s` in diesem Fall aber `null` ist, entsteht wieder eine **NullPointerException**.

Dieses Beispiel ist besonders wichtig, weil solche Fehler in größeren Programmen oft schwer zu finden sind. Die fehlerhafte Stelle liegt dann nämlich nicht unbedingt dort, wo die Exception auftritt, sondern oft schon in einer zuvor aufgerufenen Methode.

5. IllegalStateException

Auch diese Exception kann von den Entwicklern einer Anwendung bewusst eingesetzt werden, wenn ein Objektzustand den Methodenaufruf eigentlich nicht zulassen sollte. Ein typisches Beispiel ist eine Klasse **Motor** mit den Methoden starten() und stoppen(). Wenn der Motor schon läuft, sollte die Methode starten() nicht mehr aufgerufen werden können. Wird sie dennoch aufgerufen, sollte eine entsprechende Fehlermeldung erscheinen.

Codebeispiel

```
public class Motor
{
    private boolean gestartet;

    public void starten()
    {
        if (gestartet)
        {
            throw new IllegalStateException("Der Motor läuft bereits.");
        }

        gestartet = true;
   }

    public void stoppen()
    {
        if (!gestartet)
        {
            throw new IllegalStateException("Der Motor läuft noch nicht.");
        }

        gestartet = false;
   }
}
```

6. InputMismatchException

Eine **InputMismatchException** tritt auf, wenn ein Programm mit einem Scanner einen Wert eines bestimmten Datentyps einlesen möchte, die Benutzereingabe aber nicht zu diesem Datentyp passt.

Das ist zum Beispiel der Fall, wenn mit **Scanner**.nextInt() eine ganze Zahl eingelesen werden soll, der Benutzer aber einen Text oder eine Kommazahl eingibt.

Die Ausnahme zeigt also an, dass die Eingabe nicht dem erwarteten Format entspricht.

Weiter werden wir an dieser Stelle nicht auf die **InputMismatchException** eingehen. Wenn wir uns in einem späteren Kapitel mit der Klasse **Scanner** und ihren Verwandten beschäftigt haben, werden wir auch noch einmal auf diese Exception zurückkommen.

Merke:

Laufzeitfehler

- werden vom Compiler nicht erkannt und nicht verhindert,
- treten erst während der Ausführung des Programms (zur Laufzeit) auf und
- führen zum Programmabbruch, wenn sie nicht korrekt abgefangen werden.

Merke:

Die wichtigsten Laufzeitfehler sind:

- **NullPointerException**: Es wird auf eine Referenz mit dem Wert null zugegriffen.
- **ArrayIndexOutOfBoundsException**: Es wird ein Array-Element mit einem ungültigen Index aufgerufen.
- **ArithmeticException**: Es wird versucht, eine ganze Zahl durch 0 zu teilen.

8.3 Aufgaben

Aufgabe 8.1

Teil a

Implementieren Sie eine Methode

```
public int summe(int n)
```

die die Summe 1 + 2 + ... + n ermittelt und zurückgibt. Falls n kleiner als 3 ist, soll die Methode eine **IllegalArgumentException** auslösen.

Teil b

Eine Methode

```
public void testeSumme()
```

soll `summe()` dann mit verschiedenen Werten für n testen: 10, 4, 2, -1.

Die von `summe()` geworfene Exception soll mit einem try-catch-Konstrukt abgefangen werden.

Aufgabe 8.2

Teil a

Implementieren Sie eine Klasse mit einem int-Array `zahlen[100]`, bei dem die Elemente die Werte 1, 2, ..., 100 haben.

Schreiben Sie für diese Klasse dann eine Methode

```
public void tausche(int i, int j)
```

die die beiden Elemente an Position i und j miteinander vertauscht.

Diese Methode soll folgende **IllegalArgumentExceptions** werfen:

- falls i oder j < 0 sind
- falls i = j ist
- falls i oder j >= zahlen.length sind.

Beim Erzeugen dieser Exceptions sollen sinnvolle Fehlermeldungen übergeben werden.

Teil b

Schreiben Sie für `tausche()` eine Testmethode, die per Zufallsgenerator je zwei Werte für i und j generiert und dann `tausche()` damit aufruft. Stellen Sie sicher, dass auch unzulässige Werte ausgelöst werden.

Diese Testmethode soll die aufgefangenen Exceptions mit einem try-catch-Konstrukt abfangen.

Aufgabe 8.3

Wenn Sie diese Aufgabe lösen wollen, müssen Sie sich etwas mit der Klasse **String** auskennen, vor allem, was die Umwandlung von Strings in Zahlen angeht. Lesen Sie in der Fachliteratur nach, welche Methoden **String** dafür anbietet.

Teil a

Implementieren Sie eine Methode

```
public int leseGanzzahl(String text)
```

die den übergebenen String in eine ganze Zahl umwandelt und zurückgibt.

Die Methode soll dabei folgende Exceptions auslösen können:

- falls text == null: **IllegalArgumentException**
- falls text ein leerer String ist: **IllegalArgumentException**
- falls text keine gültige Zahl darstellt (zum Beispiel "12a" oder "fünf"): **NumberFormatException**

Beim Erzeugen der **Exception**-Objekte sollen sinnvolle Fehlermeldungen übergeben werden.

Teil b

Schreiben Sie eine Testmethode

```
public void testeLeseGanzzahl()
```

die `leseGanzzahl()` mit folgenden Werten testet:

- `42`
- `0`
- `"42"`
- `"  17  "` (also mit Leerzeichen vor und nach der Zahl)
- `""` (leerer String)
- `"fünf"`
- `null`

Fangen Sie diese Exceptions mit try-catch ab und geben Sie die Fehlermeldungen wie üblich in der Konsole aus.

Aufgabe 8.4

Teil a

Implementieren Sie eine Klasse, die ein int-Array `werte[20]` besitzt.

Füllen Sie das Array im Konstruktor mit Zufallszahlen im Bereich -5 .. +5.

Schreiben Sie dann eine (sinnfreie) Methode

```
public int hundertDurchX(int index)
```

die folgenden Quotienten berechnet und zurückgibt:

```
100 / werte[index];
```

Die Methode soll folgende Exceptions auslösen können:

Falls `index` ungültig ist: **IndexOutOfBoundsException**

Falls `werte[index] == 0`: **ArithmeticException**

Teil b

Schreiben Sie eine passende Testmethode, die `hundertDurchX()` mit zufälligen Indizes aufruft. Stellen Sie sicher, dass auch tatsächlich ungültige Indizes vorkommen.

Fangen Sie die Exceptions mit try-catch ab und geben Sie die Fehlermeldungen aus, ohne dass das Programm abbricht.

Teil c

Informieren Sie sich in der Fachliteratur, wie man hier eine try-catch-Konstruktion mit *zwei* catch-Blöcken sinnvoll einsetzen kann und probieren Sie das auch praktisch aus.

Aufgabe 8.5

Diese Aufgabe ist eine Erweiterung der Klausuraufgabe 3 aus der letzten Klausur.

Vorgaben

Eine Java-Klasse **Konto** soll folgende **Attribute** haben, die durch geeignete `private` Instanzvariablen realisiert werden müssen: kontonummer, kundennummer und guthaben.

Folgende **Methoden** werden benötigt:

- Ein Konstruktor, der die Kontonummer und die Kundennummer als Parameter übernimmt und das Guthaben auf 0,0 Euro setzt.
- Je eine Methode zum Einzahlen / Abheben eines bestimmten Betrags.
- Eine Methode, die den aktuellen Kontostand (Guthaben) zurückliefert.

Die Methode zum Abheben von Geld soll den tatsächlich abgehobenen Betrag als Wert (zur Kontrolle) zurückliefern.

Fallbeispiel zum Abheben von Geld:

Auf dem Konto befindet sich ein Guthaben von 3.500 Euro, es sollen aber 5.000 Euro abgehoben werden, also mehr Geld als vorhanden ist. Überziehungen sind bei der Klasse Konto nicht vorgesehen, daher können nur die vorhandenen 3.500 Euro abgehoben werden. Die Methode würde in diesem Fall den Wert 3500 zurückliefern.

Aufgabenstellung

Implementieren Sie die Klasse **Konto** mit den benötigten Instanzvariablen, dem Konstruktor und den erforderlichen Methoden.

Wenden Sie bei der Implementierung der Methoden Ihre frisch erworbenen Kenntnisse über Exceptions an.

Schreiben Sie eine Testklasse, die die Methoden der Klasse **Konto** testet, dabei sollen die von Konto geworfenen Exceptions mit try-catch-Konstrukten behandelt werden.

Aufgabe 8.6

ChatGPT hat folgendes Programm generiert:

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class Lottozahlen
{
    private int[] lottozahlen;
    private Scanner sc;

    public Lottozahlen()
    {
        lottozahlen = new int[6];
        sc = new Scanner(System.in);
        testen();
    }

    public int getInt()
    {
        try
        {
            System.out.print("Bitte int eingeben: ");
            int x = sc.nextInt();
            return x;
        }
        catch (InputMismatchException e)
        {
            sc.nextLine(); // falsche Eingabe aus dem Puffer entfernen
            throw new IllegalArgumentException("Keine int-Zahl eingegeben.");
        }
    }

    public void einlesen()
    {
        for (int i = 0; i < 6; i++)
        {
            try
            {
                lottozahlen[i] = getInt();
            }
            catch (IllegalArgumentException e)
            {
                System.out.println("Keine Lottozahl");
                i--; // diese Position nochmal neu eingeben
            }
        }
    }

    public void ausgeben()
    {
        for (int i = 0; i < 6; i++)
        {
            System.out.printf("%6d", lottozahlen[i]);
        }
        System.out.println();
    }

    public void testen()
    {
        einlesen();
        ausgeben();
    }

    public static void main(String[] args)
    {
        new Lottozahlen();
    }
}
```

Das Einlesen von Zahlen über die Klasse **Scanner** ist neu für Sie, aber recht einfach zu implementieren; die entsprechenden Zeilen sind rot hervorgehoben.

Aufgabenstellung

Erklären Sie, wie das Programm arbeitet und welche Rolle Exceptions dabei spielen.

Gehen Sie dabei vor allem auf die Zusammenarbeit der Methoden `getInt()` und `einlesen()` ein, der Hauptaspekt sollte dabei auf dem Exception-Handling liegen.

Zusatzaufgabe für Experten (nicht klausurrelevant)

Ergänzen Sie das Programm so, dass keine Zahl doppelt eingelesen werden kann. Lassen Sie für diesen Fall eine entsprechende Exception erzeugen und behandeln.

Aufgabe 8.7

Diese Aufgabe ist nicht klausurrelevant, aber für fortgeschrittene Kursteilnehmer sehr interessant und herausfordernd.

Die Aufgabe sollte in Ruhe zu Hause bearbeitet werden.

Betrachten Sie den folgenden Quelltext:

```
public class List<E>
{
    private class Node
    {
        private E value;
        private Node next;

        public Node(E element)
        {
            value = element;
            next = null;
        }

        public E getValue()
        {
            return value;
        }

        public Node getNext()
        {
            return next;
        }

        public void setNext(Node next)
        {
            this.next = next;
        }
    }

    private Node first;
    private int size;

    public List()
    {
        first = null;
        size = 0;
    }

    public boolean empty()
    {
        return size == 0;
    }

    public int size()
    {
        return size;
    }

    public void append(E element)
    {
        insert(size, element);
    }
```

```
    public void insert(int index, E element)
    {
        Node newNode = new Node(element);

        if (index == 0)
        {
            newNode.setNext(first);
            first = newNode;
            size++;
            return;
        }

        Node temp = first;

        for (int i = 0; i < index - 1; i++)
        {
            temp = temp.getNext();
        }

        newNode.setNext(temp.getNext());
        temp.setNext(newNode);
        size++;
    }

    public E get(int index)
    {
        Node temp = first;

        for (int i = 0; i < index; i++)
        {
            temp = temp.getNext();
        }

        return temp.getValue();
    }

    public void remove(int index)
    {
        if (index == 0)
        {
            first = first.getNext();
            size--;
            return;
        }

        Node temp = first;

        for (int i = 0; i < index - 1; i++)
        {
            temp = temp.getNext();
        }

        temp.setNext(temp.getNext().getNext());
        size--;
    }
}
```

Diesen Quelltext müssen Sie nicht abtippen, sondern können ihn hier herunterladen.

ChatGPT hat für diese Klasse ein umfangreiches Testprogramm erstellt, das Sie sich ebenfalls hier herunterladen können.

Teilaufgaben:

1. Analysieren Sie den Quelltext und erläutern Sie, wie die einzelnen Methoden arbeiten.
2. Die Methoden enthalten noch keine Exceptions. Ergänzen Sie alle kritischen Methoden um die entsprechenden Ausnahmen.

Hinweis: Achten Sie bei den Exceptions darauf, dass bei `insert()` der Bereich von `0` bis `size` erlaubt ist, bei `get()` und `remove()` dagegen von `0` bis `size-1`.

8.4 Erzeugen eigener Exception-Klassen

Dieses Thema werden wir erst behandeln, wenn wir uns mit dem Konzept der Vererbung auskennen. Am Ende des Vererbungs-Skriptes finden Sie dann einen Anhang, in dem an Beispielen erläutert wird, wie man eigene Exception-Klassen erstellen kann.

8.5 Exkurs: Dateneingabe in Java

Dieser Exkurs wurde nur der Vollständigkeit wegen ergänzt, das Thema ist nicht klausurrelevant, soll Ihnen aber bei der Bewältigung einiger praktischer Aufgaben helfen.

8.5.1 Rückblick

Wir haben wir bisher Daten in ein laufendes Java-Programm eingegeben?

Machen wir uns das noch einmal am Beispiel des Waage-Projektes klar.

Die Klasse Waage verfügte über zwei Getter-Methoden

```
public double getGewicht()
public double getGroesse()
```

mit denen das Gewicht in kg und die Körpergröße in cm ermittelt werden konnte. Allerdings wurden diese Daten nicht über die Konsole, eine Dialogbox, eine Editbox oder Ähnliches in das laufende Programm eingegeben, sondern ganz einfach mithilfe von Parametern. Hier die Version ohne Exception-Handling:

```
public double getGewicht(double gewicht)
{
   if (gewicht >= 3 && gewicht <= 130)
      return gewicht;
   else
     return -1;
}
```

In diesem Exkurs werden wir bessere Möglichkeiten kennenlernen, mit denen man solche Daten in ein laufendes Java-Programm einlesen kann. Zunächst beschäftigen wir uns mit der Klasse **Scanner**, mit der man Strings, Zahlen oder andere Daten direkt in die Konsole eingeben kann. Danach werfen wir einen kurzen Blick auf die Methoden der Klasse **JOptionPane**, mit denen man echte Dialogboxen erzeugen kann. Am Ende erstellen wir dann eine richtige Java-Anwendung, mit denen man die Daten sehr komfortabel einlesen kann.

8.5.2 Einlesen von Zahlen über ein Scanner-Objekt

Die Klasse **Scanner** ist eine einfache und weit verbreitete Methode, um Benutzereingaben in Java zu erfassen. Sie ermöglicht es Programmen, Daten über die Tastatur einzulesen.

Erzeugung eines Scanner-Objekts

```
Scanner taste = new Scanner(System.in);
```

Mit dieser Anweisung wird ein Objekt der Klasse **Scanner** erzeugt. Dieses Objekt ist mit der Standardeingabe **System**.`in` verbunden, also in der Regel mit der Tastatur. Über das Scanner-Objekt `taste` kann das Programm nun Benutzereingaben einlesen und in verschiedene Datentypen umwandeln.

Beispiele für das Einlesen von Werten:

```
short ganzzahl1   = taste.nextShort();
int ganzzahl2.    = taste.nextInt();
long ganzzahl3    = taste.nextLong();
float kommazahl1  = taste.nextFloat();
double kommazahl2 = taste.nextDouble();
String wort       = taste.next();
String zeile      = taste.nextLine();
```

Die jeweiligen Methoden lesen das nächste passende Eingabe-Element ein und wandeln es in den gewünschten Datentyp um. Dabei liest `next()` nur ein *einzelnes* Wort bis zum nächsten Leerzeichen ein.

Soll eine ganze Zeile eingelesen werden, muss man stattdessen die Methode `nextLine()` verwenden.

Einlesen von Zahlen

Das folgende Programm soll die Breite und Länge eines Zimmers in Metern einlesen und daraus die Fläche berechnen und ausgeben.

Solange gültige `double`-Werte eingegeben werden, funktioniert das Programm problemlos. Dabei muss in der hier verwendeten Umgebung ein Komma statt eines Punktes eingegeben werden. Wird jedoch schon bei der ersten Eingabe keine gültige Zahl eingegeben, zum Beispiel `3.14` oder ein Text, erscheint eine Fehlermeldung. Anschließend wird auch die zweite Eingabe noch als fehlerhaft interpretiert.

Das Programm:

```
import java.util.Scanner;
import java.util.InputMismatchException;

public class Fliesenleger
{
    public static void main(String[] args)
    {
       double f = flaeche();
       System.out.printf("Das Zimmer hat eine Fläche von %6.2f qm%n",f);

    }

    public static double flaeche()
    {
        Scanner scanner = new Scanner(System.in);

        double breite =
            liesDouble(scanner, "Bitte die Breite des Zimmers in m eingeben: ");
        System.out.println("Eingelesene Zahl: " + breite + "m");

        double laenge =
            liesDouble(scanner, "Bitte die Länge des Zimmers in m eingeben: ");
        System.out.println("Eingelesene Zahl: " + laenge);

        scanner.close();

        return breite * laenge;
    }

    public static double liesDouble(Scanner scanner, String meldung)
    {
        try
        {
           System.out.println(meldung);
           return scanner.nextDouble();
        }
        catch(InputMismatchException e)
        {
           System.out.println("Fehler beim Einlesen einer Zahl!" );
        }
        return -1;
    }
}
```

Zunächst müssen die Klassen **Scanner** und **InputMismatchException** importiert werden. Die main()-Methode ruft dann die Methode flaeche() auf. Diese liest die beiden benötigten Werte ein, berechnet daraus die Fläche des Zimmers und gibt das Ergebnis an main() zurück. Dort wird die Fläche schließlich ausgegeben.

Die Methode flaeche()

In der Methode flaeche() wird als Erstes ein Objekt `scanner` der Klasse **Scanner** erzeugt:

```
Scanner scanner = new Scanner(System.in);
```

Dem Konstruktor wird dabei `System.in` übergeben, also die Standard-Eingabe.

Anschließend werden nacheinander die Breite und die Länge des Zimmers eingelesen. Dazu wird zweimal die Methode liesDouble() aufgerufen. Dieser Methode werden jeweils zwei Argumente übergeben: das Scanner-Objekt `scanner` und ein **String** mit der Eingabeaufforderung.

Sobald beide Werte eingelesen wurden, wird der Scanner geschlossen. Danach berechnet die Methode die Fläche und gibt sie zurück.

Die Methode liesDouble()

In der Methode liesDouble() wird die eigentliche Zahleneingabe durchgeführt. Im try-Block wird zunächst die Eingabeaufforderung ausgegeben. Danach liest die Anweisung `scanner.nextDouble()` den nächsten Wert ein und gibt ihn als `double` zurück.

Tritt dabei ein Fehler auf, so wirft nextDouble() eine **InputMismatchException**. Das ist zum Beispiel der Fall, wenn statt einer Zahl ein Text oder eine Zahl im falschen Format eingegeben wurde.

Diese Exception wird im catch-Block abgefangen. Statt eines Programmabbruchs wird dann nur eine Fehlermeldung ausgegeben:

```
catch(InputMismatchException e)
{
   System.out.println("Fehler beim Einlesen einer Zahl!" );
}
```

Am Ende der Methode steht noch die Anweisung return -1;. Inhaltlich ist das nur eine Notlösung. Der Compiler verlangt nämlich, dass eine Methode mit dem Rückgabetyp `double` in jedem Fall einen Wert zurückliefert. Deshalb wird hier im Fehlerfall der Wert `-1` zurückgegeben.

Problem bei einer falschen ersten Eingabe

Hier zunächst die Konsolenausgabe bei korrekter Eingabe:

```
Bitte die Breite des Zimmers in m eingeben:
4,5
Eingelesene Zahl: 4.5m
Bitte die Länge des Zimmers in m eingeben:
6,8
Eingelesene Zahl: 6.8
Das Zimmer hat eine Fläche von  30,60 qm
```

Und hier die Ausgabe, wenn die erste Zahl falsch eingegeben wurde, also 4.5 statt 4,5:

```
Bitte die Breite des Zimmers in m eingeben:
4.5
Fehler beim Einlesen einer Zahl!
Eingelesene Zahl: -1.0m
Bitte die Länge des Zimmers in m eingeben:
Fehler beim Einlesen einer Zahl!
Eingelesene Zahl: -1.0
Das Zimmer hat eine Fläche von   1,00 qm
```

Bei einer falschen Eingabe versucht `scanner.nextDouble()`, die eingegebenen Zeichen als `double`-Zahl zu interpretieren. Das gelingt jedoch nicht. Daher wird eine **InputMismatchException** geworfen, und der catch-Block wird ausgeführt.

Dort wird zwar eine Fehlermeldung ausgegeben, aber die fehlerhafte Zeichenkette bleibt weiterhin im ***Eingabepuffer*** des Scanners stehen. Und genau das ist die Ursache für das Problem.

Wenn nämlich die zweite Zahl eingelesen werden soll, versucht der Scanner erneut, den noch immer im Puffer befindlichen fehlerhaften Text als `double`-Zahl zu lesen. Das misslingt natürlich wieder. Deshalb wird ein zweites Mal eine **InputMismatchException** geworfen, und die Fehlermeldung erscheint noch einmal.

Die Methode `liesDouble()` liefert in beiden Fehlerfällen den Wert -1 zurück. Aus diesen beiden Rückgabewerten berechnet die Methode `flaeche()` dann die Fläche 1,00 qm.

Problembehebung, Teil 1

Das Problem mit dem nicht gelöschten Eingabepuffer kann man recht leicht beheben, indem wir den catch-Block wie folgt erweitern:

```
catch(InputMismatchException e)
{
    System.out.println("Fehler beim Einlesen einer Zahl!" );
    scanner.nextLine();
}
```

Der Aufruf `scanner.nextLine()` liest den restlichen Inhalt der aktuellen Eingabezeile ein. Dadurch wird die fehlerhafte Zeichenkette aus dem Eingabepuffer entfernt. Die zweite Zahl kann danach korrekt eingelesen werden, auch wenn bei der ersten Eingabe ein Fehler aufgetreten ist.

Ganz gelöst ist das Problem damit aber noch nicht. Betrachten wir diese Konsolenausgabe:

```
Bitte die Breite des Zimmers in m eingeben:
12.4
Fehler beim Einlesen einer Zahl!
Eingelesene Zahl: -1.0m
Bitte die Länge des Zimmers in m eingeben:
5,7
Eingelesene Zahl: 5.7
Das Zimmer hat eine Fläche von  -5,70 qm
```

Die erste Zahl wurde hier (bewusst) falsch eingegeben, die zweite dagegen korrekt. Dass die zweite Zahl korrekt eingelesen werden konnte, ist bereits ein Fortschritt gegenüber der ersten Version, denn dort konnte nach einer fehlerhaften ersten Eingabe die zweite Zahl nicht mehr gelesen werden.

Trotzdem bleibt ein Problem: Für die Breite wurde der Wert `-1.0` zurückgegeben. Deshalb berechnet das Programm eine negative Fläche, was natürlich keinen sinnvollen Wert darstellt.

Problembehebung, Teil 2

Bei einer Falscheingabe beendet die Methode `liesDouble()` in der bisherigen Version die Eingabe sofort und liefert den Ersatzwert `-1` zurück.

Besser wäre es, dem Benutzer die Möglichkeit zu geben, die *Eingabe zu wiederholen*. Genau das erreicht die folgende Änderung:

```
public static double liesDouble(Scanner scanner, String meldung)

{
    while (true)
    {
        try
        {
            System.out.print(meldung);
            return scanner.nextDouble();
        }
        catch (InputMismatchException e)
        {
          System.out.println
               ("Fehler: Bitte eine gültige Zahl eingeben.");
           scanner.nextLine();
         }
    }
}
```

Die while-Schleife sorgt dafür, dass die Methode so lange aktiv bleibt, bis tatsächlich eine gültige `double`-Zahl eingegeben wurde. Dann wird die Endlos-Schleife mit dem return-Befehl beendet. Bei jeder falschen Eingabe wird dagegen eine Fehlermeldung ausgegeben, der Eingabepuffer mit `scanner.nextLine()` geleert und anschließend ein neuer Eingabeversuch gestartet.

```
Bitte die Breite des Zimmers in m eingeben: 5.6
Fehler: Bitte eine gültige Zahl eingeben.
Bitte die Breite des Zimmers in m eingeben: fünf
Fehler: Bitte eine gültige Zahl eingeben.
Bitte die Breite des Zimmers in m eingeben: FünfKommaSechs
Fehler: Bitte eine gültige Zahl eingeben.
Bitte die Breite des Zimmers in m eingeben: 5,6
Eingelesene Zahl: 5.6m
Bitte die Länge des Zimmers in m eingeben: 7,8
Eingelesene Zahl: 7.8
Das Zimmer hat eine Fläche von  43,68 qm
```

Erst nach der Eingabe eines korrekten Zahlenwerts - hier `5,6` - wird die while-Schleife mit dem `return`-Befehl beendet. Auf diese Weise erhält das Programm nur noch gültige Zahlen, und falsche Eingaben führen nicht mehr zu unsinnigen Ergebnissen.

8.5.3 Einlesen von Zahlen über Dialogboxen

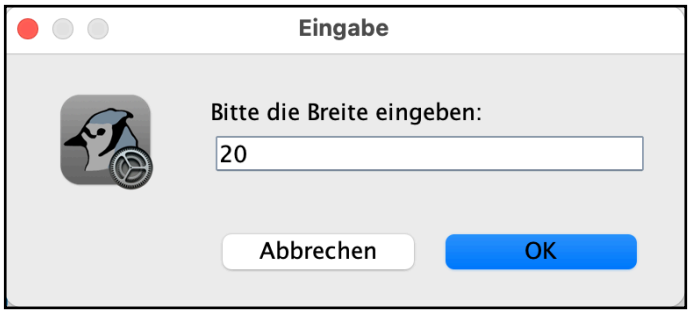
Das Einlesen von Daten mit der Klasse **Scanner** ist stark textorientiert und wirkt etwas aus der Zeit gefallen. Moderner und vor allem benutzerfreundlicher ist die Eingabe über Dialogfenster der Klasse **JOptionPane**. Das folgende, von einer KI erzeugte Programm zeigt beispielhaft, wie sich solche Dialogfenster in Java einsetzen lassen.

Das Beispiel-Programm (KI-generiert)

```
import javax.swing.JOptionPane;

public class RechteckDialog
{
    public static void main(String[] args)
    {
        try
        {
            double breite  = liesDouble("Bitte die Breite eingeben:");
            double hoehe   = liesDouble("Bitte die Höhe eingeben:");

            double flaeche = breite * hoehe;

            JOptionPane.showMessageDialog(
                null,
                "Breite: " + breite + "\n
                 Höhe: " + hoehe + "\n
                 Fläche: " + flaeche,
                "Ergebnis",
                JOptionPane.INFORMATION_MESSAGE
            );
        }
        catch (NumberFormatException e)
        {
            JOptionPane.showMessageDialog(
                null,
                "Sie haben keine gültige Zahl eingegeben.",
                "Fehler",
                JOptionPane.ERROR_MESSAGE
            );
        }
        catch (IllegalArgumentException e)
        {
            JOptionPane.showMessageDialog(
                null,
                e.getMessage(),
                "Abbruch",
                JOptionPane.WARNING_MESSAGE
            );
        }
    }
```

```
    public static double liesDouble(String meldung)
     {
         String eingabe = JOptionPane.showInputDialog(null, meldung);

         if (eingabe == null)
         {
             throw new IllegalArgumentException
             ("Die Eingabe wurde abgebrochen.");
         }

         return Double.parseDouble(eingabe);
     }
}
```

Dieses von ChatGPT generierte Programm fragt die Breite und Höhe eines Rechtecks mithilfe von zwei Dialogboxen ab. Hier der Screenshot der Dialogbox für die Eingabe der Breite:



Wenn man den Wert als `int`- oder `double`-Zahl eingegeben und auf den OK-Button geklickt hat, erscheint die selbe Dialogbox noch einmal, um die Höhe des Rechtecks abzufragen.

Wenn man nach der zweiten Eingabe den OK-Button betätigt, erscheint ein kleines Fenster mit den Ergebnissen:

Hier kann man die eingegebenen Daten sowie die berechnete Fläche des Rechtecks sehen.

Programm-Analyse

Als Erstes muss die Klasse **JOptionPane** importiert werden:

```
import javax.swing.JOptionPane;
```

In der main()-Methode wird im try-Block versucht, die Breite und die Höhe des Rechtecks mithilfe der Methode liesDouble() einzulesen:

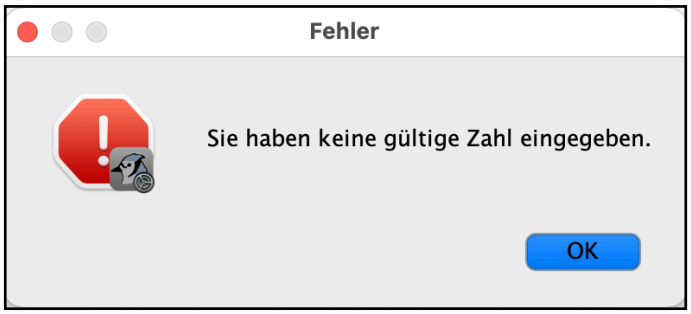
```
double breite  = liesDouble("Bitte die Breite eingeben:");
double hoehe   = liesDouble("Bitte die Höhe eingeben:");
```

Diese Methode kann verschiedene Exceptions auslösen. Daher besitzt das try-catch-Konstrukt in main() auch mehrere catch-Blöcke, jeweils einen für einen bestimmten Fehlertyp.

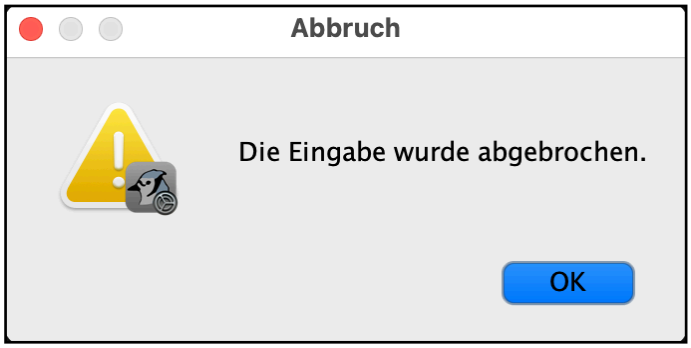
Wird in die Dialogbox keine gültige Zahl eingegeben, sondern zum Beispiel ein beliebiger Text, dann wird eine **NumberFormatException** geworfen. In diesem Fall wird der erste catch-Block ausgeführt:

```
        catch (NumberFormatException e)
        {
            JOptionPane.showMessageDialog(
                null,
                "Sie haben keine gültige Zahl eingegeben.",
                "Fehler",
                JOptionPane.ERROR_MESSAGE
            );
        }
```

Dieser catch-Block für die **NumberFormatException** erzeugt ein kleines Fenster mit einer entsprechenden Fehlermeldung:



Wird die Eingabe dagegen abgebrochen, etwa durch den Button `Abbrechen` oder durch Schließen des Fensters, dann wird der zweite catch-Block ausgeführt:



Hier wurde der zweite catch-Block ausgeführt:

```
        catch (IllegalArgumentException e)
        {
            JOptionPane.showMessageDialog(
                null,
                e.getMessage(),
                "Abbruch",
                JOptionPane.WARNING_MESSAGE
            );
        }
```

Die Methode showMessageDialog

Die Methode showMessageDialog() benötigt vier Parameter. Der erste Parameter gibt an, zu welchem Fenster der Dialog gehören soll. In diesem Beispiel wird `null` übergeben, da das Programm kein eigenes Anwendungsfenster besitzt. Der zweite Parameter ist die angezeigte Nachricht. Der dritte legt den Fenstertitel fest, und der vierte bestimmt den Typ des Dialogs, also zum Beispiel Information, Warnung oder Fehlermeldung.

Kommen wir nun zur eigentlichen Kernmethode des Programms, nämlich liesDouble():

```
    public static double liesDouble(String meldung)
    {
        String eingabe =
            JOptionPane.showInputDialog(null, meldung);

        if (eingabe == null)
        {
            throw new IllegalArgumentException
                ("Die Eingabe wurde abgebrochen.");
        }

        return Double.parseDouble(eingabe);

    }
```

Hier wird die Methode showInputDialog() der Klasse **JOptionPane** aufgerufen. Der erste Parameter - hier also `null` - gibt wieder das Anwendungsfenster an, zu dem die Dialog-

box gehören soll. Da wir in diesem Programm kein eigenes Anwendungsfenster haben, wird hier `null` eingetragen.

Der zweite Parameter ist die Meldung bzw. die Eingabeaufforderung, die in der Dialogbox erscheinen soll, also zum Beispiel `"Bitte die Breite eingeben:"`.

Eine solche Dialogbox liefert aber keine `int`- oder `double`-Zahl zurück, sondern immer einen **String**. In unserem Beispiel wird dieser Rückgabewert in der Variablen `eingabe` gespeichert.

Anschließend wird geprüft, ob `eingabe` den Wert `null` hat. Das ist genau dann der Fall, wenn der Benutzer die Eingabe abbricht. Dann wird bewusst eine **IllegalArgumentException** geworfen, die dann von der aufrufenden `main()`-Methode im zweiten catch-Block behandelt wird.

Liegt kein Abbruch vor, wird der String mit **Double**.`parseDouble()` in eine `double`-Zahl umgewandelt. Ist der eingegebene Text keine gültige Zahl, so entsteht dabei eine **NumberFormatException**. Diese wird im ersten catch-Block der `main()`-Methode behandelt.

Wenn also weder ein Abbruch vorliegt noch eine falsche Zahl eingegeben wurde, liefert die Methode den eingelesenen Wert als `double` zurück. Dieser wird in `main()` gespeichert, sodass anschließend die Fläche des Rechtecks berechnet werden kann.

8.5.4 Weitere Möglichkeiten zur Dateneingabe

Neben **Scanner** und **JOptionPane** gibt es in Java auch noch die Klasse **Console** für Eingaben in textorientierten Konsolenprogrammen. Da sie jedoch in vielen Entwicklungsumgebungen nicht zuverlässig funktioniert, wird sie im Anfangsunterricht oft nicht verwendet.

Eine weitere Möglichkeit ist das Einlesen über die Kommandozeilenargumente

```
String[] args
```

Das ist technisch ebenfalls eine Form der Eingabe, aber keine interaktive Benutzereingabe und für den Anfangsunterricht ebenfalls nur bedingt geeignet.

8.5.5 Eingabe über Edit-Boxen

Hier ein von ChatGPT generierter Quelltext, der eine kleine Anwendung erzeugt, in die man die Breite und Höhe eines Rechtecks eingeben kann, das dann gezeichnet wird. Allerdings nur dann, wenn beide double-Zahlen korrekt eingegeben wurden.

ChatGPT wurde angewiesen, Exception-Handling in das Programm einzubauen, so dass Fehleingaben korrekt behandelt werden und der User die Möglichkeit hat, falsche Eingaben zu korrigieren.

Hier der komplette Quelltext des Programms. Eine Erläuterung findet hier nicht statt, da dieses Thema nicht klausurrelevant ist. Wer sich in Java-Anwendungen einarbeiten möchte, kann dies gerne machen. Viele Bücher sind dabei eine große Hilfe, und KIs können einem auch viel beibringen, wenn man systematisch dabei vorgeht.

```
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class RechteckFenster extends JFrame
{
   private JTextField breiteFeld;
   private JTextField hoeheFeld;
   private Zeichenflaeche zeichenflaeche;

   private double rechteckBreite = -1;
   private double rechteckHoehe  = -1;

   public RechteckFenster()
   {
      super("Rechteck zeichnen");

      setSize(1024, 768);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setLayout(null);

      JLabel breiteLabel = new JLabel("Breite in Pixeln:");
      breiteLabel.setBounds(20, 20, 120, 25);
      add(breiteLabel);

      breiteFeld = new JTextField();
      breiteFeld.setBounds(145, 20, 80, 25);
      add(breiteFeld);

      JLabel hoeheLabel = new JLabel("Höhe in Pixeln:");
      hoeheLabel.setBounds(250, 20, 120, 25);
      add(hoeheLabel);

      hoeheFeld = new JTextField();
      hoeheFeld.setBounds(365, 20, 80, 25);
      add(hoeheFeld);

      JButton okButton = new JButton("OK");
      okButton.setBounds(470, 20, 80, 25);
      add(okButton);

      zeichenflaeche = new Zeichenflaeche();
      zeichenflaeche.setBounds(0, 60, 1024, 708);
      add(zeichenflaeche);
```

```
      okButton.addActionListener(e -> eingabenVerarbeiten());

      setLocationRelativeTo(null);
      setVisible(true);
   }

   private void eingabenVerarbeiten()
   {
      double breite;
      double hoehe;

      try
      {
         breite = liesDouble(breiteFeld, "Breite");
         pruefeBereich(breite, "Breite");

         hoehe = liesDouble(hoeheFeld, "Höhe");
         pruefeBereich(hoehe, "Höhe");
      }
      catch (NumberFormatException e)
      {
         JOptionPane.showMessageDialog(
            this,
            "Bitte geben Sie eine gültige Zahl ein.",
            "Fehlerhafte Eingabe",
            JOptionPane.ERROR_MESSAGE
         );
         return;
      }
      catch (IllegalArgumentException e)
      {
         JOptionPane.showMessageDialog(
            this,
            e.getMessage(),
            "Ungültiger Wert",
            JOptionPane.ERROR_MESSAGE
         );
         return;
      }

      rechteckBreite = breite;
      rechteckHoehe  = hoehe;
      zeichenflaeche.repaint();
   }

   private double liesDouble(JTextField feld, String name)
   {
      String text = feld.getText().trim();
      return Double.parseDouble(text);
   }

   private void pruefeBereich(double wert, String name)
   {
      if (wert < 50)
      {
         throw new IllegalArgumentException(
            name + " ist zu klein. Erlaubt sind Werte von 50 bis 500."
         );
      }

      if (wert > 500)
      {
         throw new IllegalArgumentException(
            name + " ist zu groß. Erlaubt sind Werte von 50 bis 500."
         );
      }
   }
```

```
   private class Zeichenflaeche extends JPanel
   {
      public Zeichenflaeche()
      {
         setBackground(Color.WHITE);
      }

      @Override
      protected void paintComponent(Graphics g)
      {
         super.paintComponent(g);

         if (rechteckBreite > 0 && rechteckHoehe > 0)
         {
            g.setColor(Color.RED);

            int x = 50;
            int y = 50;
            int b = (int)Math.round(rechteckBreite);
            int h = (int)Math.round(rechteckHoehe);

            g.fillRect(x, y, b, h);
         }
      }
   }

   public static void main(String[] args)
   {
      SwingUtilities.invokeLater(() -> new RechteckFenster());
   }
}
```