Michael Inden

Java

Die Neuerungen in Version 17 LTS, 18 und 19

dpunkt.verlag

Inhalt

4 Übungen zu den Syntaxneuerungen in JDK 11 bis 17

Mit den folgenden Aufgaben sollen die Neuerungen aus JDK 11 bis 17 anhand von Übungen in ihrer Handhabung vertieft werden. In Abschnitt 4.2 finden Sie die passenden Musterlösungen.

4.1 Aufgaben

 **Aufgabe 1 – Syntaxänderungen bei `switch`** 

Vereinfachen Sie einige herkömmliche `switch`-`case`-Konstrukte durch die neue Syntax.

Aufgabe 1a Die nachfolgende Methode soll durch die Angabe mehrerer Werte bei `case` und das Verwenden von Pfeilen kürzer und übersichtlicher werden:

```
private static void switchOld(final int number)

{

    switch (number)

    {

        case 0:
```

```
        case 1:

        case 2:

        case 3:

        case 4:

        case 5:

            System.out.println("Durchgefallen");

            break;

        case 6:

        case 7:

        case 8:

            System.out.println("Gut");

            break;

        case 9:

        case 10:

            System.out.println("Exzellent");

            break;

        default:
```

```
            System.out.println("Ungültig");

            break;

    }

}
```

Aufgabe 1b Verwenden Sie zur Vereinfachung folgender Methode die Möglichkeit, Rückgaben direkt zu spezifizieren:

```
private static int switchBonusOld(final char grade)

{

    int bonus;

    switch (grade)

    {

        case 'A':

            bonus = 2000;

            break;

        case 'B':

            bonus = 1000;

            break;

        case 'C':
```

```
            bonus = 500;
            break;
        default:
            bonus = 0;
            break;
    }
    return bonus;
}
```

 **Aufgabe 2 – Schrittweises Umformen mit `switch`** 

Der folgende Sourcecode prüft mit einem herkömmlichen `switch-case` auf gerade und ungerade Zahlen. Vereinfachen Sie ihn durch die neue Syntax.

```
private static void dumbEvenOddChecker(int value)
{
    String result;
    switch (value)
    {
        case 1:
```

```
        case 3:
        case 5:
        case 7:
        case 9:
            result = "odd";
            break;
        case 0:
        case 2:
        case 4:
        case 6:
        case 8:
        case 10:
            result = "even";
            break;
        default:
            result = "only implemented for values from 0 to
            10";
    }
```

```
    System.out.println("result: " + result);

}
```

Aufgabe 2a Nutzen Sie zunächst nur die Angabe mehrerer Werte bei `case` und die Pfeilsyntax, um die Methode kürzer und übersichtlicher zu schreiben.

Aufgabe 2b Verwenden Sie im Anschluss die Möglichkeit, Rückgaben direkt zu spezifizieren, und ändern Sie die Signatur in `String dumbEvenOddChecker(int)`. Verlagern Sie die Ausgabe an die Aufrufstelle.

Aufgabe 2c Wandeln Sie das Ganze so ab, dass bei geraden Zahlen zwischen 2 und 8 (inklusive) noch die Ausgabe »Hurra, es ist gerade und im Bereich 2 - 8« erfolgt.

Tipp Dazu müssen Sie die Spezialform »yield mit Rückgabewert« verwenden.

Aufgabe 2d Während `yield` gerade für die Rückgabe in speziellen Fällen genutzt wurde, rekapitulieren Sie diese neue Syntax für die ursprüngliche Aufgabe.

 **Aufgabe 3 – Fehlerfixing mit `switch`** 

Vereinfachen Sie die Methode `detectKeyJava11(char)` mithilfe der neuen Syntax von `switch` und korrigieren Sie gleichzeitig mögliche Flüchtigkeitsfehler. Schreiben Sie dazu eine Methode `detectKeyJava17(char)`.

In einem nächsten Schritt schauen Sie bitte genauer hin und versuchen Sie das Ganze maximal zu vereinfachen.

```
static void detectKeyJava11(final char key)

{

    switch (key)

    {
```

```
        case '0':
            System.out.println("You pressed: 0");
            break;
        case '1':
            System.out.println("You pressed: 1");
            break;
        case '2':
            System.out.println("You pressed: 2");
        case '3':
            System.out.println("You pressed: 3");
        case '4':
            System.out.println("You pressed: 4");
            break;
        case '5':
        case '6':
        case '7':
        case '8':
```

```
        case '9':
            System.out.println("You pressed: " + key);
            break;
        default:
            System.out.println("Not allowed!");
    }
}
```

Überprüfen Sie Ihre Arbeiten mit den `char`-Werten für `2` und `7`. Erwartet werden folgende Ausgaben:

```
You pressed: 2
You pressed: 7
```

 **Aufgabe 4 – Text Blocks** 

Vereinfachen Sie den folgenden Sourcecode mit herkömmlichen Stringkonkatenationen, die über mehrere Zeilen gehen, und nutzen Sie dazu die neu eingeführte Syntax der Text Blocks.

```
String multiLineStringOld = "THIS IS\n" +
        "A MULTI\n" +
        "LINE STRING\n" +
        "WITH A BACKSLASH \\n";
```

```
System.out.println(multiLineStringOld);

String multiLineHtmlOld = "<html>\n" +

        "    <body>\n" +

        "        <p>Hello, world</p>\n" +

        "    </body>\n" +

        "</html>";

System.out.println(multiLineHtmlOld);

String jsonTextBlockOld = "{\n" +

                          "    \"version\": \"Java 17\",\n" +

                          "    \"feature\": \"text
                          blocks\",\n" +

                          "    \"attention\": \"very
                          cool!\"\n" +

                          "}\n";

System.out.println(jsonTextBlockOld);
```

Bonus Betreiben Sie etwas ASCII-Art und zeichnen Sie einen Tee- oder Kaffeebecher oder ein Glas Bier mithilfe von Text Blocks.

 Aufgabe 5 – Text Blocks mit Platzhaltern 

Vereinfachen Sie den folgenden Sourcecode mit einem herkömmlichen String, der über mehrere Zeilen geht, und nutzen Sie die neu eingeführte Syntax inklusive der Möglichkeit, Platzhalter anzugeben und mit Werten zu ersetzen:

```
String multiLineStringWithPlaceholdersOld =

       String.format("HELLO \"%s\"!\n" +

                     "  HAVE %s\n" +

                     "  NICE \"%s\"!",

                     "WORLD", "A", "DAY");

System.out.println(multiLineStringWithPlaceholdersOld);
```

Produzieren Sie folgende Ausgaben mit der neuen Syntax:

```
HELLO "WORLD"!

  HAVE A

  NICE "DAY"!
```

 Aufgabe 6 – Record-Grundlagen 

Gegeben seien zwei einfache Klassen, die reine Datencontainer darstellen und somit lediglich ein öffentliches Attribut bereitstellen. Wandeln Sie diese in Records um:

```
public class Square
```

```
{

    public final double sideLength;

    public Square(final double sideLength)

    {

        this.sideLength = sideLength;

    }

}

public class Circle

{

    public final double radius;

    public Circle(final double radius)

    {

        this.radius = radius;

    }

}
```

Welche Vorteile ergeben sich – außer der kürzeren Schreibweise – durch den Einsatz von Records statt eigener Klassen?

 Aufgabe 7 – Records 

Ergänzen Sie im nachfolgend gezeigten Record eine Prüfung der beiden Namensbestandteile sowie des Geburtstags, sodass die Namen jeweils länger als zwei Zeichen sind und das Datum in der Vergangenheit liegt. Erstellen Sie zusätzlich zwei Methoden, die eine JSON- und eine XML-Ausgabe erzeugen – oftmals sollte eine derartige Transformation nicht vom Objekt selbst implementiert werden, hier als Übung aber schon:

```
record Person(String firstName, String lastName, LocalDate
birthday) {}
```

Wir konstruieren ein Objekt und geben dieses als XML und JSON aus:

```
Person mike = new Person("Michael", "Inden",
LocalDate.of(1971, 2, 7));

System.out.println(mike.toXml());

System.out.println(mike.toJSON());
```

Folgendes sollte auf der Konsole erscheinen:

```
<Person>

    <firstName>Michael</firstName>

    <lastName>Inden</lastName>

    <birthday>1971-02-07</birthday>

</Person>

{
```

```
    "firstName" : "Michael",

    "lastName" : "Inden",

    "birthday" : "1971-02-07"

}
```

 **Aufgabe 8 – `instanceof`-Grundlagen** 

Gegeben seien folgende Zeilen mit einem `instanceof` sowie einem Cast:

```
Object obj = "BITTE MICHAEL";

if (obj instanceof String)

{

    final String str = (String)obj;

    if (str.contains("BITTE"))

    {

        System.out.println("It contains the magic word!");

    }

}
```

Vereinfachen Sie das Ganze mit den Neuerungen aus Java 17.

Aufgabe 9 – `instanceof` und OO-Design

Vereinfachen Sie den Sourcecode mithilfe der Syntaxneuerungen bei `instanceof` und danach mithilfe der Besonderheiten bei Records.

Aufgabe 9a Gegeben seien die folgenden beiden Records:

```
record Square(double sideLength) {};

record Circle(double radius) {};
```

Für diese wird deren Fläche nicht besonders elegant (später mehr dazu) berechnet:

```
public double computeAreaOld(final Object figure)

{

    if (figure instanceof Square)

    {

        final Square square = (Square)figure;

        return square.sideLength * square.sideLength;

    }

    else if (figure instanceof Circle)

    {

        final Circle circle = (Circle)figure;

        return circle.radius * circle.radius * Math.PI;
```

```
    }

    throw new IllegalArgumentException("figure is not a
    recognized figure");

}
```

Vereinfachen Sie diese Methode durch den Einsatz der neuen Syntax von `instanceof`.

Aufgabe 9b Zwar haben wir durch `instanceof` sicher eine Verbesserung bezüglich Lesbarkeit und Anzahl Zeilen erzielt, jedoch deuten mehrere derartige Prüfungen auf einen Verstoß gegen das Open Closed Principle, eines der SOLID-Prinzipien guten Entwurfs, hin. Was wäre ein objektorientiertes Design? Die Antwort ist in diesem Fall einfach: Oftmals lassen sich `instanceof`-Prüfungen vermeiden, indem ein Basistyp eingeführt wird. Vereinfachen Sie das Ganze durch ein Interface `BaseFigure` und nutzen Sie dieses passend.

4.2 Lösungen

 Lösung 1 – Syntaxänderungen bei `switch` 

Die Aufgabe bestand darin, einige herkömmliche `switch`-`case`-Konstrukte durch die neue Syntax zu vereinfachen.

Lösung 1a In dieser Aufgabe lassen sich die `cases` durch eine kommaseparierte Angabe, die Pfeilsyntax und das Weglassen von `break` wie folgt vereinfachen:

```
private static void switchNew(final int number)

{

    switch (number)

    {
```

```
        case 0, 1, 2, 3, 4, 5 ->
        System.out.println("Durchgefallen");

        case 6, 7, 8 -> System.out.println("Gut");

        case 9, 10 -> System.out.println("Exzellent");

        default -> System.out.println("Ungültig");

    }

}
```

Lösung 1b Auch bei der Ermittlung des Bonus wenden wir die zuvor genutzten Vereinfachungen an. Zusätzlich lässt sich mit `switch` direkt ein Wert zurückgeben, wodurch das Ganze kurz und übersichtlich wird.

```
private static int switchBonusNew(final char grade)

{

    return switch (grade)

    {

        case 'A' -> 2000;

        case 'B' -> 1000;

        case 'C' -> 500;

        default -> 0;
```

```
    };

}
```

Prüfung Zum Ausprobieren definieren wir folgende `main()`-Methode:

```
public static void main(String[] args)

{

    switchNew(7);

    switchNew(10);

    System.out.println("Bonus A: " + switchBonusNew('A'));

    System.out.println("Bonus C: " + switchBonusNew('C'));

    System.out.println("Bonus E: " + switchBonusNew('E'));

}
```

Als Ergebnis erhalten wir folgende Ausgabe:

```
Gut

Exzellent

Bonus A: 2000

Bonus C: 500

Bonus E: 0
```

Lösung 2 – Schrittweises Umformen mit `switch`

Die Aufgabe bestand darin, Sourcecode zum Prüfen auf gerade und ungerade Zahlen mit einem herkömmlichen `switch-case` durch die neue Syntax zu vereinfachen.

Lösung 2a Zunächst nutzen wir nur die Angabe mehrerer Werte bei `case` und die Pfeilsyntax, um die Methode kürzer und übersichtlicher zu schreiben.

```
private static void dumbEvenOddChecker2a(int value)

{

    // In Eclipse früher Initialisierung mit = "" nötig, mit
    Java-Compiler nicht

    String result;

    switch (value)

    {

       case 1, 3, 5, 7, 9 -> result = "odd";

       case 0, 2, 4, 6, 8, 10 -> result = "even";

       default -> result = "only implemented for values from
       0 to 10";

    }

    System.out.println("result: " + result);

}
```

Lösung 2b Durch die Möglichkeit, Rückgaben direkt zu spezifizieren, kann man dann noch auf die künstliche Hilfsvariable `result` sowie die Zuweisungen innerhalb der `cases` verzichten. Damit erreichen wir recht kompakten Sourcecode:

```
private static String dumbEvenOddChecker2b(int value)

{

    return switch (value)

    {

        case 1, 3, 5, 7, 9 -> "odd";

        case 0, 2, 4, 6, 8, 10 -> "even";

        default -> "only implemented for values from 0 to
        10";

    };

}
```

Lösung 2c Wandeln Sie das Ganze so ab, dass bei geraden Zahlen zwischen 2 und 8 (inklusive) noch die Ausgabe »Hurra, es ist gerade und im Bereich 2 - 8« erfolgt. Dazu müssen Sie die Spezialform »yield mit Rückgabewert« verwenden.

```
private static String dumbEvenOddChecker2cSpecialCase(int
value)

{

    return switch (value)
```

```
    {

        case 1, 3, 5, 7, 9 -> "odd";

        case 0, 2, 4, 6, 8, 10 ->

        {

            if (value > 1 && value < 9)

                System.out.println("Hurra, es ist gerade und
                im Bereich 2 - 8");

            yield "even";

        }

        default -> "only implemented for values from 0 to
        10";

    };

}
```

Lösung 2d Während `yield` gerade für die Rückgabe in speziellen Fällen genutzt wurde, verwenden wir nun diese neue Syntax für die ursprüngliche Aufgabe:

```
private static String dumbEvenOddChecker2dWithYield(int
value)

{

    return switch (value)
```

```
    {
        case 1, 3, 5, 7, 9:
            yield "odd";
        case 0, 2, 4, 6, 8, 10:
            yield "even";
        default:
            yield "only implemented for values from 0 to 10";
    };
}
```

Prüfung Zum Ausprobieren definieren wir folgende `main()`-Methode:

```
public static void main(String[] args)
{
    int value = 6;
    dumbEvenOddChecker2a(value);
    String result1 = dumbEvenOddChecker2b(value);
    System.out.println("result1: " + result1);
    String result2 = dumbEvenOddChecker2cSpecialCase(value);
```

```
    System.out.println("result2: " + result2);

    String result3 = dumbEvenOddChecker2dWithYield(value);

    System.out.println("result3: " + result3);

}
```

Als Ergebnis erhalten wir folgende Ausgabe:

```
result: even

result1: even

Hurra, es ist gerade und im Bereich 2 – 8

result2: even

result3: even
```

 **Lösung 3 – Fehlerfixing mit `switch`** 

Gegeben war ein Programmfragment für eine Methode `detectKeyJava11(char)`, das mithilfe der neuen Syntax von `switch` vereinfacht werden soll. Bei der Umformung entdeckt man vermutlich die (bewusst) eingebauten Flüchtigkeitsfehler, nämlich das vergessene `break` für die Fälle 2 und 3, die jeweils zu einem Fall Through führen:

```
case '2':

    System.out.println("You pressed: 2");

case '3':
```

```
    System.out.println("You pressed: 3");

case '4':

    System.out.println("You pressed: 4");

    break;
```

Bei einer Prüfung mit dem Wert 2 kommt es dann unerwartet zu diesen Ausgaben:

```
You pressed: 2

You pressed: 3

You pressed: 4
```

Lösung Mithilfe der neuen Syntax wandeln wir das Ganze zunächst einmal wie folgt ab und korrigieren automatisch die zuvor fehlerhafte Behandlung für die Fälle 2 und 3:

```
static void detectKeyJava17(final char key)

{

    switch (key)

    {

        case '0' -> System.out.println("You pressed: 0");

        case '1' -> System.out.println("You pressed: 1");
```

```
        case '2' -> System.out.println("You pressed: 2");

        case '3' -> System.out.println("You pressed: 3");

        case '4' -> System.out.println("You pressed: 4");

        case '5', '6', '7', '8', '9' ->

            System.out.println("You pressed: " + key);

        default -> System.out.println("Not allowed!");

    }

}
```

Es ist immer eine gute Idee, den Sourcecode in möglichst kleinen Schritten zu vereinfachen, idealerweise unterstützt durch die automatischen Refactorings von IDEs. Im obigen Beispiel wird dann die Duplikation mit minimalen Anpassungen für die ersten 5 Fälle offensichtlich und dass es danach bereits eine allgemeingültige Lösung gibt.

Im nachfolgenden Schritt sollte man genauer hinschauen und basierend auf den gerade genannten Erkenntnissen das Ganze maximal vereinfachen.

```
static void detectKeyJava17Improved(final char key)

{

    switch (key)

    {

        case '0', '1', '2', '3', '4', '5', '6', '7', '8',
        '9'->
```

```
            System.out.println("You pressed: " + key);

        default -> System.out.println("Not allowed!");

    }

}
```

Prüfung Zum Ausprobieren definieren wir folgende `main()`-Methode:

```
public static void main(String[] args)

{

    detectKeyJava17('2');

    detectKeyJava17('7');

    detectKeyJava17Improved('2');

    detectKeyJava17Improved('7');

}
```

Als Ergebnis erhalten wir folgende Ausgabe:

```
You pressed: 2

You pressed: 7

You pressed: 2

You pressed: 7
```

Lösung 4 – Text Blocks

In dieser Aufgabe ging es darum, herkömmliche Stringkonkatenationen, die über mehrere Zeilen gehen, durch die neu eingeführte Syntax zu vereinfachen:

```
public static void main(final String[] args)

{

    String multiLineString = """

                          THIS IS

                          A MULTI

                          LINE STRING

                          WITH A BACKSLASH \

                          """;

    System.out.println(multiLineString);

    String multiLineHtml = """

           <html>

               <body>

                   <p>Hello, world</p>

               </body>
```

```
            </html>""";

    System.out.println(multiLineHtml);

    String jsonTextBlock = """

    {

        "version": "Java17",

        "feature": "text blocks",

        "attention": "very cool!"

    }

    """;

    System.out.println(jsonTextBlock);

}
```

Prüfung Als Ergebnis erhalten wir folgende Ausgabe:

```
THIS IS

A MULTI

LINE STRING

WITH A BACKSLASH \

<html>
```

```
    <body>

        <p>Hello, world</p>

    </body>

</html>

{

    "version": "Java17",

    "feature": "text blocks",

    "attention": "very cool!"

}
```

Bonus Als Bonus sollte eine kleine Spaßaufgabe gelöst werden, nämlich das Zeichnen eines Tee- oder Kaffeebechers oder eines Glases Bier mithilfe von Text Blocks.

```
var coffeePot = """

                ~~ ~~ ~~

             /=|~~~~~~~~|

             | | :   :  |

             |=|   : :  |

               | :  :   |
```

```
                ___""";
```

 **Lösung 5 – Text Blocks mit Platzhaltern** 

In dieser Aufgabe waren Platzhalter zu ersetzen. Anstatt dafür jedoch die statische Methode `String.format()` zu nutzen, lässt sich nun die Methode `formatted()` gewinnbringend verwenden, die einfacher in der Handhabung ist:

```
public static void main(final String[] args)

{

    String multiLineStringWithPlaceHolders = """

        HELLO "%s"!

          HAVE %s

          NICE "%s"!""".formatted("WORLD", "A", "DAY");

    System.out.println(multiLineStringWithPlaceHolders);

}
```

Prüfung Als Ergebnis erhalten wir folgende Ausgabe:

```
HELLO "WORLD"!

  HAVE A

  NICE "DAY"!
```

 Lösung 6 – Record-Grundlagen 

Diese Aufgabe zeigt wie viel Sourcecode sich bei der Implementierung eines reinen Datencontainers einsparen lässt, indem Records für die Definition genutzt werden.

```
record Square(double sideLength) {}

record Circle(double radius) {}
```

Darüber hinaus erhält man als positive Begleiterscheinungen noch kontraktkonforme Implementierungen für die Methoden `equals()` und `hashCode()` sowie auch eine menschenlesbare Ausgabe für `toString()`. Nichts davon haben die ursprünglichen Klassen geboten. Was auf den ersten Blick wie ein Schönheitsmakel aussieht, ist in Wirklichkeit weit mehr: Würden die ursprünglichen Klassen in Listen oder Maps genutzt, stieße man recht schnell auf unterschiedlichste Probleme. Wenn Sie an Hintergründen und Details interessiert sind, dann möchte ich Sie für eine fundierte Behandlung der Thematik an mein Buch »Der Weg zum Java-Profi« [2] verweisen.

 Lösung 7 – Records 

Es war ein einfacher Record zur Modellierung einer Person samt Vorname, Nachname und Geburtstag gegeben, der um Parameterprüfungen sowie Methoden, die eine JSON- und eine XML-Ausgabe erzeugen, ergänzt werden sollte. Für Ersteres nutzen wir die spezielle Kurzschreibweise und führen dort einfache Prüfungen aus. Für die XML- und JSON-Ausgaben stützen wir uns auf Text Blocks und Platzhalter ab, wie wir es zuvor schon kennengelernt haben. Beachten Sie unbedingt, nicht versehentlich die statische Methode `format()` aufzurufen:

```
record Person(String firstName, String lastName, LocalDate
birthday)

{
```

```
    public Person

    {

        if (firstName.length() < 3 || lastName.length() < 3)

        {

            String errorMsg = "first and last name must be at
            least 3 chars long

                 ";

            throw new IllegalArgumentException(errorMsg);

        }

        if (birthday.isAfter(LocalDate.now()))

        {

            String errorMsg = "birthday can not be in the
            future";

            throw new IllegalArgumentException(errorMsg);

        }

    }

    String toXml()

    {

        return """
```

```
            <Person>

                <firstName>%s</firstName>

                <lastName>%s</lastName>

                <birthday>%s</birthday>

            </Person>

            """.formatted(firstName, lastName, birthday);

            /* Achtung nicht: """.format(firstName,
            lastName, birthday);*/

}

String toJSON()

{

    return """

            (

                "firstName" : "%s",

                "lastName" : "%s",

                "birthday" : "%s"

            )
```

```
                """.formatted(firstName, lastName, birthday);

    }

}
```

Prüfung Zum Ausprobieren definieren wir folgende `main()`-Methode:

```
public static void main(String[] args)

{

    Person mike = new Person("Michael", "Inden",
    LocalDate.of(1971, 2, 7));

    System.out.println(mike);

    System.out.println(mike.toXml());

    System.out.println(mike.toJSON());

    checkNameTooShort();

}

private static void checkNameTooShort()

{

    try

    {
```

```
        Person toShort = new Person("Mi", "In",
        LocalDate.of(1971, 2, 7));

        System.out.println(toShort);

        System.out.println(toShort.toXml());

        System.out.println(toShort.toJSON());

    }

    catch (Exception e)

    {

        e.printStackTrace();

    }

}
```

Als Ergebnis erhalten wir folgende Ausgabe (gekürzt):

```
Person[firstName=Michael, lastName=Inden, birthday=1971-02-
07]

<Person>

    <firstName>Michael</firstName>

    <lastName>Inden</lastName>

    <birthday>1971-02-07</birthday>
```

```
</Person>

(

    "firstName" : "Michael",

    "lastName" : "Inden",

    "birthday" : "1971-02-07"

)

java.lang.IllegalArgumentException: first / last name must be
at least 3 chars

      long
```

 **Lösung 8 – `instanceof`-Grundlagen** 

Das Pattern Matching bei `instanceof` erlaubt es, nach dem Cast eine Variable anzugeben, die mit dem Wert aus `obj` typsicher befüllt wird, sofern der Typ passt:

```
Object obj = "BITTE MICHAEL";

if (obj instanceof String str)

{

    if (str.contains("BITTE"))

    {

        System.out.println("It contains the magic word!");
```

```
    }

}
```

Darüber hinaus können bei Bedarf weitere Prüfungen angefügt werden. Das ermöglicht ziemlich kompakte Ausdrücke:

```
Object obj = "BITTE MICHAEL";

if (obj instanceof String str && str.contains("BITTE"))

{

    System.out.println("It contains the magic word!");

}
```

 **Lösung 9 – `instanceof` und OO-Design** 

In dieser Aufgabe sollten Besonderheiten von Records und bei Typprüfungen mit `instanceof` genutzt werden.

Lösung 9a In einem ersten Schritt kann man die Neuerung bei `instanceof` nutzen, um auf Casts zu verzichten:

```
public double computeArea(final Object figure)

{

    if (figure instanceof Square square)

    {

        return square.sideLength * square.sideLength;
```

```
    }

    else if (figure instanceof Circle circle)

    {

        return circle.radius * circle.radius * Math.PI;

    }

    throw new IllegalArgumentException("figure is not a
    recognized figure");

}
```

Das stellt für die Flächenberechnung eine Verbesserung dar. Mehrfache Abfragen mit `instanceof` sind aber ein Indikator für einen Verstoß gegen das Open Closed Principle, eines der SOLID-Prinzipien guten OO-Entwurfs, die ich detailliert in meinem Buch »Der Weg zum Java-Profi« [2] behandle und im nachfolgenden Praxistipp einführend vorstelle.

Schnelleinstieg: Open Closed Principle

Ein Paradebeispiel aus der realen Welt für das ***Open Closed Principle*** (OCP) sind HiFi-Verstärker. Diese können jede beliebige Quelle verstärken, solange diese über Cinch-Buchsen angeschlossen werden kann. Auf diese Weise lassen sich alte Kassettenrekorder, MiniDisc-Player, CD-Player, aber auch moderne Blu-Ray-Player anschließen und deren Ton wiedergeben.

Das OCP zielt auf die ***leichte Erweiterbarkeit*** und ***korrekte Kapselung*** sowie ***Trennung von Zuständigkeiten*** ab. Wenn man es extrem sehen möchte, sollte sich eine Klasse nach ihrer Fertigstellung nur noch dann ändern müssen, wenn komplett neue Anforderungen oder Funktionalitäten zu integrieren sind oder aber Fehler korrigiert werden müssen. Dahingegen sollten Änderungen an der eigenen Klasse nicht dadurch erforderlich sein, dass sich andere Klassen ändern.

Schauen wir uns ein Beispiel an, um das OCP ein wenig besser zu verstehen. Dazu betrachten wir eine Spieleapplikation, die verschiedene Bonuselemente, wie Extraleben oder Zusatzausrüstungen, als Anreiz anbietet. Ihre Aufgabe als Entwickler ist es nun, ein neues Level zu gestalten und dort neue Arten von Bonuselementen zu integrieren. Wünschenswert wäre es, wenn dies möglichst einfach realisierbar wäre, etwa wenn man lediglich neue Klassen für die neuen, speziellen Bonuselemente erstellen müsste. Das wäre beispielsweise dann möglich, wenn die Bonuselemente alle ein gemeinsames Interface `BonusElement` erfüllen oder eine (abstrakte) Basisklasse implementieren würden. In diesem Fall sollte die restliche Applikation kaum oder im besten Fall gar nicht von Änderungen bzw. Erweiterungen der Bonuselemente betroffen sein. Eine gemeinsame Basis aus Interface und/oder abstrakter Basisklasse und der Möglichkeit zur Definition von Spezialisierungen sind eine Variante, das OCP umzusetzen. Ein Verstoß gegen das OCP besteht dann, wenn die beschriebene Erweiterung an diversen Stellen zu Änderungen führen würde.

Lösung 9b Wie schon erkannt, sind die `instanceof`-Prüfungen noch ungünstig. Zudem verstößt es gegen gutes OO-Design, wenn Funktionalitäten nicht innerhalb von Klassen mithilfe von Methoden gekapselt sind. Wir beginnen mit einem Interface `BaseFigure` und einer abstrakten Methode `double calcArea()`. Die einzelnen Berechnungen verschieben wir jetzt in die Records:

```
interface BaseFigure

{
```

```
    double calcArea();

}

record Square(double sideLength) implements BaseFigure

{

    @Override

    public double calcArea()

    {

        return sideLength * sideLength;

    }

}

record Circle(double radius) implements BaseFigure

{

    @Override

    public double calcArea()

    {

        return radius * radius * Math.PI;

    }
```

```
}

public double computeArea(final Object figure)

{

    if (figure instanceof Square square)

    {

        return square.calcArea();

    }

    else if (figure instanceof Circle circle)

    {

        return circle.calcArea();

    }

    throw new IllegalArgumentException("figure is not a
    recognized figure");

}
```

Nun können wir das gemeinsame Basisinterface und Polymorphie nutzen und die Methode `computeArea()` rein auf den Eingabetyp `BaseFigure` umbauen und dort einfach die Methode aufrufen – sämtliche Spezialbehandlungen und Typabfragen lassen sich so vermeiden:

```
public double computeArea(final BaseFigure figure)

{
```

```
    return figure.calcArea();

}
```

Besser noch: Wir halten das Open Closed Principle ein und könnten nun problemlos eine weitere Figurenklasse, etwa für Rechtecke, hinzufügen, ohne dafür den eigentlichen Algorithmus anpassen zu müssen. Hier erkennt man sehr gut die Vorteile eines gelungenen OO-Designs:

```
record Rectangle(double width, double height) implements
BaseFigure

{

    @Override

    public double calcArea()

    {

        return width * height;

    }

}
```

Index