

# Neues in der Sprache (Java 9 - 17)

## Kleinigkeiten 1 (Java 9)

```
@Deprecated(since = "11", forRemoval = true)
private static void someOldMethod()
{
    // ...
}

//<> for anonymous inner classes
Comparator<String> c = new Comparator<>() {
    @Override
    public int compare(String o1, String o2) {
        return (o1.length()-o2.length());
    }
};

//not allowed
// int _ = 1;
```

- @Deprecated ist erweitert
- „Diamond“-Operator auch für anonyme Klassen
- Unterstrich als Bezeichner ist nicht mehr möglich

## Kleinigkeiten 2 (Java 9)

```
//effective final resources in ARM
final BufferedInputStream bis = new BufferedInputStream(System.in);
BufferedOutputStream bos = new BufferedOutputStream(System.out); //effective final
try (bis; bos){
    //...
}
catch(IOException e) {
    //...
}
```

- Direkte Verwendung von finalen Variablen bei ARM

## Kleinigkeiten 3 (Java 9)

```
@SafeVarargs
public static void alpha(List<String>... args) { }
```

//@SafeVarargs // not allowed  
public void beta(List<String>... args) { } // compiler-warning

```
@SafeVarargs
public final void gamma(List<String>... args) { }
```

```
@SafeVarargs // Java 9
private void delta(List<String>... args) { }
```

- Bis Java 8: **@SafeVarargs** nur erlaubt bei statischen und finalen Methoden
- Ab Java 9: auch bei privaten Methoden und somit immer dann, wenn die Methode nicht überschrieben werden kann

## Schlüsselwort var bei Variablen (Java 10)

```
public static void main(String[] args) {  
    var name = "Stephan Karrer";  
    var list = new ArrayList<ArrayList<String>>();  
    //var x = null; //unzulässig  
    var var = 77; //zulässig!  
  
    var personAgeMapping = Map.of("Tim", 47L, "Tom", 12L, "Mike", 47L);  
    //Cast ist notwendig  
    var isAdult = (Predicate<Map.Entry<String, Long>>) entry -> entry.getValue() >= 18;  
    var firstChar = (Function<Map.Entry<String, Long>, Character>) entry ->  
                                                            entry.getKey().charAt(0);  
  
    //hier ist alles klar für den Compiler  
    var filteredPersons = personAgeMapping.entrySet().stream().  
        collect(groupingBy(firstChar,  
                            filtering(isAdult, toSet())));  
}
```

- Ist nur möglich, sofern der Compiler anhand der Zuweisung den Typ ableiten kann
- Ist nur für lokale Variablen erlaubt !
- var ist nur als Typersatz Schlüsselwort!

## var - Fallstricke

```
try {  
    throw new RuntimeException();  
} catch (RuntimeException e) { } //unzulässig: var e  
  
var counter = 9_999_999_999_999L; //muss long sein  
//hier hilft auch Cast nicht: counter = (long) 9_999_999_999_999;  
var b = (byte) 1+1; //Typ ist byte  
  
var intarray = new int[]{1, 2, 3}; //Unzulässig: var intarray = {1, 2, 3}  
  
var query = new StringBuilder("string1");  
//unzulässig: query = query.toString() + "string2";  
  
var l = new ArrayList<>();  
l.add(1); l.add("Hallo"); //keine Typsicherheit mehr  
  
var what = List.of(1, "2", LocalTime.now()); //was ist der Typ?  
System.out.println(List.of(1, "2", LocalTime.now()).getClass());
```

- Ist **var** hilfreich??

## var für Lambda-Parameter (Java 11)

```
@interface SomeAnnotation { }

public class LambdaVarExample
{
    public static void main(final String[] args)
    {
        ToIntFunction<String> ftyped = (String s) -> s.length();
        ToIntFunction<String> fnottyped = s -> s.length();

        ToIntFunction<String> fwithvar = (var s) -> s.length();

        Function<String, String> trimmer = (@SomeAnnotation var str) -> str.trim();
    }
}
```

- Wenn **var** verwendet wird, dann für alle Parameter !
- Dadurch kann annotiert werden

## Pattern Matching for instanceof (Java 16)

```
//bisher
Object o1 = "Hallo";
if (o1 instanceof String) {
    int len = ((String)o1).length();    }

//neu
if (o1 instanceof String s) {
    int len = s.length();    }
```

- Es ist kein expliziter Cast nötig
- Nennt sich Typ-Pattern: Prädikat spezifiziert den Typ, ergänzt um eine Pattern-Variable
- Der Gültigkeitsbereich der Variablen ist auf den true-Zweig bzgl. des Typ-Patterns begrenzt



## Switch: neues case-Label (Java 14)

```
String s = "zweig2";
switch (s) {
    case "zweig1", "zweig2" -> System.out.println(1);
    //case "zweig3", "zweig2" -> System.out.println(2); //not allowed
    case "zweig3", "zweig4" -> { int i = 2;
                                System.out.println(2); }
    default -> throw new IllegalArgumentException();
}
```

- Kein Fall-Through bei Verwendung des neuen Labels ( case ... → )
- Zweige können kombiniert werden, wobei Konstanten-Duplikate nicht erlaubt sind
- Mehrere Anweisungen im Zweig müssen als Block geschrieben werden
  - Gültigkeit von Variablen im Block ist lokal für den Zweig !
- default-Zweig ist wie bisher optional

## Switch-Expression (Java 14)

```
var i = switch (s) {      // i ist polymorph (Object, Serializable, ...)
    case "zweig1", "zweig2" -> "String1";    //String
    case "zweig3", "zweig4" -> 2;            //int
    default -> throw new IllegalArgumentException();
}; //Ausdruck abgeschlossen mit ;

System.out.println(switch (s) {
    case "zweig1", "zweig2" -> "String1";
    case "zweig3", "zweig4" -> 2;
    default -> throw new IllegalArgumentException();} );
```

- Der Zweig kann auch aus einem Ausdruck bestehen, dann muss aber auch jeder Zweig einen Wert (oder Exception) liefern
- Default-Zweig ist jetzt Pflicht !
- Auch hier kann der Compiler den Typ des Ergebnisses ableiten

## Switch-Expression: yield

```
int n = switch (s) {  
    case "zweig1", "zweig2" -> { int k = "Hallo".length();  
                                yield k;  
                                }  
    case "zweig3", "zweig4" -> 2;  
    default -> 1;  
};
```

- Ist ein Block für die Berechnung des Rückgabewerts notwendig, so muss der Wert per **yield** geliefert werden

## Switch-Expression mit dem bisherigen case-Zweig

```
String s = "hey";  
int result = switch (s) {  
    case "Foo":  
        yield 1;  
    case "Bar":  
        yield 2;  
    default:  
        System.out.println("Neither Foo nor Bar, hmmm...");  
        yield 0;  
}; // "Neither Foo nor Bar, hmmm..."
```

- Hier muss dann **yield** benutzt werden
- Wir haben wieder Fall-Through und **break** ist nicht erlaubt !

## Pattern Matching für Switch (Preview Java 17)

```
public static double getPerimeter(Shape shape){  
    return switch (shape) {  
        case Rectangle r -> 2 * r.length() + 2 * r.width();  
        case Circle c    -> 2 * c.radius() * Math.PI;  
        default          -> throw new  
                            IllegalArgumentException("Unrecognized shape");  
    };  
}
```

- case-Zweige für die Typ-Unterscheidung

## Text Block (Java 15)

```
String query = ""  
    SELECT "EMP_ID", "LAST_NAME" FROM "EMPLOYEE_TB"  
    WHERE "CITY" = 'INDIANAPOLIS'  
    ORDER BY "EMP_ID", "LAST_NAME";  
    "";
```

- Ein Textblock ist eine Zeichenkette, die auch Zeilenumbrüche und sonstigen Whitespace ohne spezielle Kennzeichnung enthalten darf.
- Als Begrenzer kommen """" zum Einsatz, wobei der öffnende Begrenzer auf einer eigenen Zeile stehen muss. Der eigentliche Inhalt beginnt nach dem Zeilenumbruch und endet vor dem schließenden Begrenzer.
- Ein einfaches " ist kein Sonderzeichen und die Nutzung von `\n` ist nicht notwendig (aber erlaubt). Ansonsten können die üblichen Sonderzeichen (`\t`, `\s`, ...) verwendet werden.
- Ein Textblock ist ebenfalls eine String-Konstante !

# Wie verarbeitet der Compiler einen Text Block

Auszug aus dem JEP 378: Text Blocks

## *Compile-time processing*

*A text block is a constant expression of type String, just like a string literal. However, unlike a string literal, the content of a text block is processed by the Java compiler in three distinct steps:*

*Line terminators in the content are translated to LF (\u000A). The purpose of this translation is to follow the principle of least surprise when moving Java source code across platforms.*

*Incidental white space surrounding the content, introduced to match the indentation of Java source code, is removed.*

*Escape sequences in the content are interpreted. Performing interpretation as the final step means developers can write escape sequences such as \n without them being modified or deleted by earlier steps.*

# Incidental Whitespace

- Im Standardfall wird die Zeile mit den wenigsten Leerzeichen als Maß genommen: Somit führen „Fall1“ und „Fall2“ zu identischen Strings
- Will man mehr führende Leerzeichen haben, kann man den schließenden Delimiter entsprechend positionieren: „Fall3“ (resultiert in einem Zeilenumbruch, den man wiederum „escapen“ kann)
- Nachfolgende Leerzeichen werden standardmäßig entfernt
- Es gibt auch entsprechende Methoden in der Klasse String, z.B. ***indent()***
- Leerzeichenbehandlung kann komplex werden !

## Fall 1

```
String html = "<html>\n" +  
              "    <body>\n" +  
              "        <p>Hello, world</p>\n" +  
              "    </body>\n" +  
              "</html>\n";
```

## Fall 2

```
String html = ""  
.....<html>  
.....    <body>  
.....        <p>Hello, world</p>  
.....    </body>  
.....</html>  
....."";
```

## Fall 3

```
String html = ""  
.....    <html>  
.....        <body>  
.....            <p>Hello, world</p>  
.....        </body>  
.....    </html>  
....."";
```

siehe: Programmer's Guide to Text Blocks von Jim Laskey  
und JEP 378 Text Blocks



# Records (Java 16)

```
record Point(int x, int y) { }
```

- Ein Record entspricht einer Klasse mit finalen Instanzattributen, passendem Konstruktor, lesenden Zugriffsoperationen und überschriebenen Standardmethoden
- Damit hat man immutable Value-Objekte
- Benutzt werden Records analog zu den bisherigen Klassen:

```
Point p = new Point(4,5);
```

```
class Point {  
    private final int x;  
    private final int y;  
  
    Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    int x() { return x; }  
    int y() { return y; }  
  
    public boolean equals(Object o) {  
        if (!(o instanceof Point)) return false;  
        Point other = (Point) o;  
        return other.x == x && other.y == y;  
    }  
  
    public int hashCode() {  
        return Objects.hash(x, y);  
    }  
  
    public String toString() {  
        return String.format("Point[x=%d, y=%d]", x, y);  
    }  
}
```

## Regeln für Records

- Records erben von der Basisklasse ***java.lang.Record*** (analog Enums), sind final und nie abstrakt.
- Es sind keine weiteren expliziten Instanzattribute und Instanzinitialisierer möglich.
- Die generierte Methodik kann angepasst (überschrieben) werden, sofern der Kontrakt eingehalten wird.
- Es können mit Ausnahme von nativen Methoden weitere Instanzmethoden hinzugefügt werden.
- Es können statische Attribute, Methoden und Initialisierer hinzugefügt werden.
- Interfaces können implementiert werden.
- Record-Klassen können innere Klassen verwenden, insbesondere Records. Innere Record-Klassen sind implizit static.
- Annotationen und Generics können verwendet werden.
- Records sind serialisierbar (sofern als Serializable markiert), allerdings ohne Anpassungsmöglichkeiten.
- Zyklische Anhängigkeiten zwischen Record-Klassen sind nicht möglich (Henne-Ei-Problem).
- Vorsicht: equals ist nur flach implementiert und muss gegebenenfalls überschrieben werden.

# Records mit explizitem Konstruktor

- Der Konstruktor kann auch explizit geschrieben werden, um die Initialisierung anzupassen (es muss 1:1 die Signatur verwendet werden).
- Die Argumente und die initialisierende Zuweisung dürfen mit der besonderen Variante kompakter kanonischer Konstruktor entfallen.

```
public record Point(int x, int y) {  
    public Point(int x, int y) {  
        if (x <= 0 || y <= 0) {  
            throw new java.lang.IllegalArgumentException();  
        }  
        this.x = x;  
        this.y = y;  
    }  
}
```

```
public record Point(int x, int y) {  
    //compact canonical constructor  
    public Point {  
        if (x <= 0 || y <= 0) {  
            throw new java.lang.IllegalArgumentException();  
        }  
    }  
}
```

## Records mit angepassten Instanzmethoden

```
public record Point(int x, int y) {  
  
    //compact canonical constructor  
    public Point {  
        if (x <= 0 || y <= 0) {  
            throw new java.lang.IllegalArgumentException();  
        }  
    }  
    @Override //optional  
    public int x() { return this.x < 100 ? this.x : 100; }  
    public int y() { return this.y < 100 ? this.y : 100; }  
}
```

- Vorsicht: Obiges Beispiel ist „bad practice“, da jetzt beim Lesen des Zustands nicht der korrekte Wert zurückgeliefert wird.
  - Kopie via gelesener Werte verletzt „equals“-Kontrakt.

## Records mit statischen Elementen

```
record Rectangle(double length, double width) {  
  
    // Static field  
    static double goldenRatio;  
  
    // Static initializer  
    static {  
        goldenRatio = (1 + Math.sqrt(5)) / 2;  
    }  
  
    // Static method  
    public static Rectangle createGoldenRectangle(double width) {  
        return new Rectangle(width, width * goldenRatio);  
    }  
}
```

## Records: Interfaces und Generics

```
//kein sonderlich sinnhaftes Beispiel
import java.util.function.Function;

public record GenRecord<T> ( T t) implements Function<T, String>{

    @Override
    public String apply(T t) {
        return t.toString();
    }

    public static void main(String[] args) {
        GenRecord<Point> g = new GenRecord<Point>(null);
        System.out.println(g.apply(new Point(1,1)));
    }
}
```

- Records können Typ-Parameter verwenden und Interfaces implementieren.

## Records als innere Klassen

```
List<Merchant> findTopMerchants(List<Merchant> merchants, int month) {  
  
    // Local record  
    record MerchantSales(Merchant merchant, double sales) {}  
  
    return merchants.stream()  
        .map(merchant -> new MerchantSales(merchant, computeSales(merchant, month)))  
        .sorted((m1, m2) -> Double.compare(m2.sales(), m1.sales()))  
        .map(MerchantSales::merchant)  
        .collect(toList());  
}
```

- Innere Record-Klassen sind stets implizit **static**. D.h. sie haben keinen Zugriff auf die umgebende Instanz. Dies gilt auch im lokalen Fall.

# Records und Annotationen

- Records und ihre Elemente können annotiert werden.
- Die Annotationen werden je nach Anwendbarkeit (@Target) auf die Konstruktoren, Methoden und Attribute propagiert.

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface GreaterThanZero {
}
```

```
record Point(
    @GreaterThanZero int x,
    @GreaterThanZero int y) { }
```

Entspricht:

```
class Point {
    private final @GreaterThanZero int x;
    private final @GreaterThanZero int y;

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    //...
```



## Statische Methoden in Interfaces (ab Java 8)

```
public interface InterfaceWithStatic {  
  
    int MINVALUE = 10;  
    int MAXVALUE = 1_000_000;  
  
    int getValue();  
  
    public static boolean isValidValue(int i) {  
        return (MINVALUE <= i) & (i <= MAXVALUE);  
    }  
}
```

- Statische Methoden in Interfaces sind auscodiert und müssen als solche gekennzeichnet werden.
  - Ist unkritische Erweiterung, da kein Vererbungsproblem
  - Werden stets anhand des Interface referenziert (keine Referenzierung anhand einer implementierenden Instanz möglich)

## Factory-Method-Pattern mit statischen Methoden

```
// aus java.util

public interface Comparator<T> {
    //...

    static <T> Comparator<T> comparingDouble (
        ToDoubleFunction<? super T> keyExtractor)      { //...

    static <T extends Comparable<? super T>> Comparator<T>
        naturalOrder() { //...
```

- Das Factory-Method-Pattern kann nun via Interface umgesetzt werden!

## Private Methoden in Interfaces (ab Java 9)

```
public interface InterfaceWithStatic {  
  
    int MINVALUE = 10;  
    int MAXVALUE = 1_000_000;  
  
    int getValue();  
  
    public static boolean isValidValue(int i) {  
        return validate(i);  
    }  
  
    private static boolean validate(int i) {  
        return (MINVALUE <= i) & (i <= MAXVALUE);  
    }  
}
```

- Konsequente Erweiterung, da ebenfalls kein Vererbungsproblem

## Default Methoden in Interfaces (ab Java 8)

```
public interface InterfaceWithDefault {  
  
    int MINVALUE = 10;  
    int MAXVALUE = 1_000_000;  
  
    int getValue();  
  
    default int getRandomValue() {  
        return getFixValue() + new Random().nextInt(MINVALUE);  
    }  
  
    private int getFixValue() {  
        return 1;  
    }  
}
```

- Löst ein Problem bei Erweiterung eines vorhandenen Interface um neue Methoden:  
Bisher implementierende Klassen bekommen Default-Implementierungen via Vererbung verpasst.
- Die Default-Implementierung ist in der Regel nicht optimal und sollte überschrieben werden.

## Jetzt existiert Mehrfachvererbung!

```
public class BrownEyes {  
    public String getColor() {  
        return "brown";  
    }  
}
```

```
public interface BlueEyes {  
    default String getColor() {  
        return "blue";  
    }  
}
```

```
public class WhatEyes extends BrownEyes implements BlueEyes{  
    private String c = BlueEyes.super.getColor();  
    public static void main(String[] args) {  
        System.out.println( new WhatEyes().getColor()); //brown  
        System.out.println( new WhatEyes().c); //blue  
    }  
}
```

- Ist kein Fehler: Elternklasse hat Vorrang
- Mittels entsprechendem super kann der jeweilige Elternteil angesprochen werden !

## Mehrfachvererbung mit Konflikt

```
public interface GreyEyes {  
    default String getColor() {  
        return "grey";  
    }  
}
```

```
public interface BlueEyes {  
    default String getColor() {  
        return "blue";  
    }  
}
```

```
public class WhatEyes2 implements BlueEyes, GreyEyes{  
  
    @Override  
    public String getColor() {  
        return BlueEyes.super.getColor();  
    }  
}
```

- Bei konkurrierenden Interfaces muß der Konflikt durch Überschreiben gelöst werden !

## Sealed Interfaces (ab Java 17, preview ab 15)

```
sealed interface AIf permits B1, B2If, B3If {}  
├── final class B1 implements AIf {}  
├── sealed interface B3If extends AIf permits C1 {}  
│   └── final class C1 implements B3If {}  
└── non-sealed interface B2If extends AIf {}
```

- **sealed** ist auch für abstrakte Klassen und Interfaces anwendbar
- Für Interfaces wird dadurch sowohl Implementierung als auch Vererbung eingeschränkt.

## Sealed: Records als Kindklassen

```
package com.example.expression;

public sealed interface Expr
    permits ConstantExpr, PlusExpr, TimesExpr, NegExpr { ... }

public record ConstantExpr(int i) implements Expr { ... }
public record PlusExpr(Expr a, Expr b) implements Expr { ... }
public record TimesExpr(Expr a, Expr b) implements Expr { ... }
public record NegExpr(Expr e) implements Expr { ... }
```

- Da Records final sind, können sie als erlaubte Kindklassen von sealed Klassen und Interfaces benutzt werden.