



Java Intro

Michael Inden

Freiberuflicher Consultant und Trainer

https://github.com/Michaeli71/JAVA_INTRO

Speaker Intro



- Michael Inden, Jahrgang 1971
- Diplom-Informatiker, C.v.O. Uni Oldenburg
- ~8 ¼ Jahre **SSE** bei Heidelberger Druckmaschinen AG in Kiel
- ~6 ¾ Jahre **TPL, SA** bei IVU Traffic Technologies AG in Aachen
- ~4 ¼ Jahre **LSA / Trainer** bei Zühlke Engineering AG in Zürich
- ~3 Jahre **TL / CTO** bei Direct Mail Informatics / ASMIQ in Zürich
- **Freiberuflicher Consultant, Trainer und Konferenz-Speaker**
- Autor und Gutachter beim dpunkt.verlag

E-Mail: michael.inden@hotmail.ch

Blog: <https://jaxenter.de/author/minden>

Kurse: **Bitte spricht mich an!**

https://github.com/Michaeli71/JAVA_INTRO





Agenda



- **PART 1: Schnelleinstieg Java**
 - Erste Schritte in der JShell
 - Schnelleinstieg
 - Variablen
 - Operatoren
 - Fallunterscheidungen
 - Schleifen
 - Methoden
 - Rekursion



- **PART 2: Strings**
 - Gebräuchliche String-Aktionen
 - Suchen und Ersetzen
 - Formatierte Ausgaben
 - Einstieg Reguläre Ausdrücke
 - Mehrzeilige Strings
- **PART 3: Arrays**
 - Gebräuchliche Array-Aktionen
 - Mehrdimensionale Arrays
 - Beispiel: Flood Fill



- **PART 4: Klassen & Objektorientierung**
 - Basics
 - Textuelle Ausgaben
 - Gleichheit == / equals()
 - Klassen ausführbar machen
 - Imports & Packages
 - Information Hiding
 - Vererbung und Overloading und Overriding
 - Die Basisklasse Object
 - Interfaces & Implementierungen
 - Records



- **PART 5: Collections**
 - Schnelleinstieg Listen, Sets und Maps
 - Iteratoren
 - Generics
 - Basisinterfaces für Container
 - Praxisbeispiel Stack und Queue selbst gebaut
 - Sortierung – `sort()` + Comparator



- **PART 6: Ergänzendes Wissen**
 - Sichtbarkeits- und Gültigkeitsbereiche
 - Primitive Typen und Wrapper
 - Enums
 - ?-Operator
 - switch
 - Besonderheiten in Schleifen break und continue
 - Vererbung und Polymorphie
 - Varianten innerer Klassen



- **PART 7: Einstieg in Lambdas und Streams**
 - Syntax von Lambdas
 - Lambdas im Einsatz mit `filter()`, `map()` und `reduce()`
 - Lambdas im Einsatz mit `Collectors.groupingBy()`
 - `takeWhile()` / `dropWhile()`
- **PART 8: Datumsverarbeitung**
 - Einführung Datumsverarbeitung
 - Zeitpunkte und die Klasse `LocalDateTime`
 - Datumswerte und die Klasse `LocalDate`
 - Zeit und die Klasse `LocalTime`



- **PART 9: Exception-Handling**
 - Schnelleinstieg
 - Exceptions selbst auslösen
 - Eigene Exception-Typen definieren
 - Propagation von Exceptions
 - Automatic Resource Management
 - Checked / Unchecked Exceptions
- **PART 10: Dateiverarbeitung**
 - Verzeichnisse und Dateien verwalten
 - Daten schreiben / lesen



PART 8:

Datumsverarbeitung



Warum mit Java 8 noch ein weiteres Datums-API?



- Verarbeitung von Datumswerten und Zeit scheint einfach, ist es aber nicht
- Tatsächlich ist es sogar ziemlich kompliziert
 - Einfluss von Zeitzonen
 - Einfluss von Schaltjahren
 - Einfluss von Sommer- und Winterzeit
 - Usw.
- Beispiel “Gehe einen Monat in die Vergangenheit / Zukunft”
 - Was ist ein Monat und wie wird dieser dargestellt?
 - Monat anpassen
 - Schaltjahr berücksichtigen
 - Ggf. Jahr anpassen
 - Ggf. Uhrzeit anpassen
 - usw.

Warum mit Java 8 noch ein weiteres Datums-API?



Wurf 1: `java.util.Date` (JDK1.0)

- nur minimale Abstraktion eines `long` zum Offset 1.1.1970 00:00:00 Uhr
- Verschiedene Offsets (1900 / 1970, 0- und 1-basiert usw.)
- Verarbeitung von Datum und Zeit ist damit mühselig und fehleranfällig

```
// Mein Geburtstag: 7.2.1971
final int year = 1971;
final int month = 2;
final int day = 7;
final Date myBirthday = new Date(year, month, day);
System.out.println(myBirthday);
```



Warum mit Java 8 noch ein weiteres Datums-API?



=> Tue Mar 07 00:00:00 CET 3871

Korrektur: `new Date(year - 1900, month - 1, day)`

Warum mit Java 8 noch ein weiteres Datums-API?



Wurf 2: `java.util.Calendar` (JDK1.1)

- ist besser gelungen und bietet eine bessere Abstraktion (Konstanten für Monate, Addition von Zeitwerten usw.)
- Verarbeitung wird deutlich leichter, vor allem Berechnungen
- ABER: Es ist immer noch Einiges ziemlich kompliziert, etwa wenn man nur mit Zeitangaben oder Datumswerten rechnen möchte

Alternative: JODA-Time

- Probleme auch bei SUN / Oracle im Bewusstsein, aber es passierte nichts
- Abhilfe für JDK 7 versprochen, aber erst für JDK 8 adressiert
- Zwischenzeitlich: Joda-Time





Wurf 3: JSR 310 – Neuer (dritter) Wurf eines Datums-APIs im JDK

- Viel ist besser gelungen als die Vorgänger
- basiert auf der erfolgreichen JodaTime-Bibliothek (von S.Colebourne)

Designziele:

- Klarheit und Verständlichkeit, “Works-as-expected”
- Fluent Interface, sprechende Methodennamen, Method-Chaining
- Immutable, somit automatisch Thread-Safe



Klarheit und Verständlichkeit, analog zu Denkweise von Menschen

// Varianten von `LocalDate`: Datum ohne Uhrzeit und Zeitzone

```
LocalDate today = LocalDate.now();  
LocalDate jan23 = LocalDate.parse("2014-01-23");  
LocalDate feb7 = LocalDate.of(2014, 2, 7);  
LocalDate mar24 = LocalDate.of(2014, Month.MARCH, 24);
```

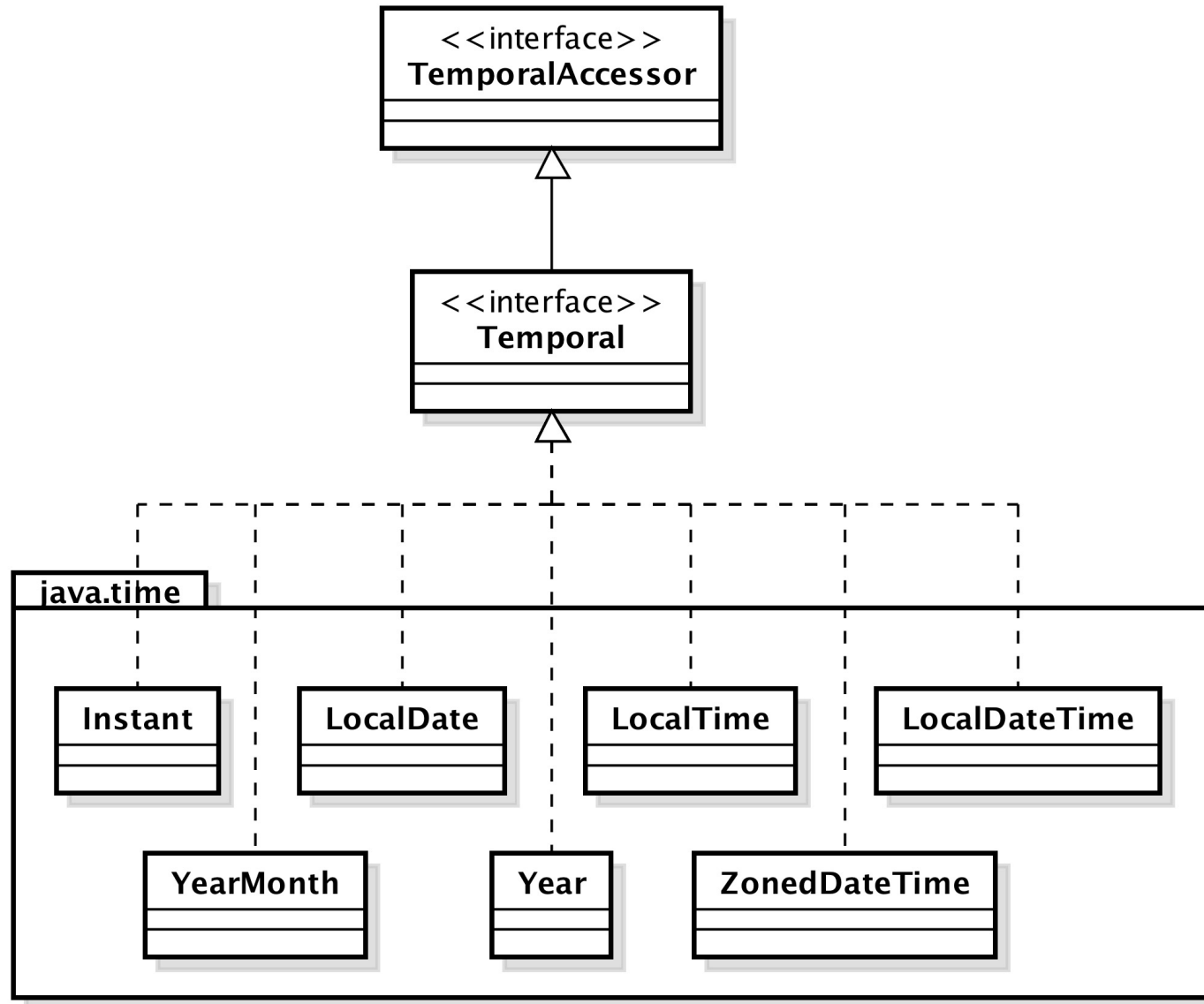
// Zeitangabe ohne Datum

```
LocalTime now = LocalTime.now();  
LocalTime at_15_30 = LocalTime.parse("15:30");  
LocalTime at_12_11_10 = LocalTime.of(12, 11, 10);
```

// Kombination aus Datum und Zeit

```
LocalDateTime nowWithTime = LocalDateTime.now();  
LocalDateTime feb8_at_10_11 = LocalDateTime.parse("2014-02-08T10:11:12");
```

JSR-310: Date And Time API Overview





- Ein Datumswert besteht aus diversen Bestandteilen. Zugriff mit `.-Notation`:

```
final LocalDate now = LocalDate.now();
```

```
System.out.println("Today: " + now);
```

```
System.out.println("DayOfWeek: " + now.getDayOfWeek());
```

```
System.out.println("DayOfMonth: " + now.getDayOfMonth());
```

```
System.out.println("DayOfYear: " + now.getDayOfYear());
```

```
System.out.println("Month: " + now.getMonth());
```

```
System.out.println("LengthOfMonth: " + now.lengthOfMonth());
```

```
System.out.println("Days in Month: " +  
now.getMonth().length(now.isLeapYear()));
```

```
System.out.println("LengthOfYear: " + now.lengthOfYear());
```

Today: 2021-10-11

DayOfWeek: MONDAY

DayOfMonth: 11

DayOfYear: 284

Month: OCTOBER

Length Of Month: 31

Days in Month: 31

Length Of Year: 365

Year: 2021



DEMO

LocalDateExample



Fluent API

```
LocalDate jan15 = LocalDate.parse("2015-01-15");
```

```
LocalDate myStartAtSwisscom = jan15.plusDays(5);  
myStartAtSwisscom = myStartAtSwisscom.minusYears(1);  
System.out.println(myStartAtSwisscom);           // 2014-01-20
```

```
LocalDate jan15_2015 = LocalDate.of(2015, Month.JANUARY, 15);  
System.out.println(jan15_2015.getDayOfWeek());   // THURSDAY
```

```
LocalDate feb7_2015 = jan15_2015.withMonth(2).withDayOfMonth(7);  
System.out.println(feb7_2015.getDayOfYear());    // 38
```

```
LocalDate feb7_2015 = jan15_2015.withMonth(Month.FEBRUARY).withDayOfMonth(7);
```



- **Period** – Datumsbasierter Zeitabschnitt: Monate, Wochen, Tage, ...
- **Duration** – Zeitbasierte Bereiche: Stunden, Minuten, Sekunden, ...

```
final LocalDateTime christmasEve = LocalDateTime.of(2016, 12, 24, 17, 30, 00);  
final LocalDateTime silvester = LocalDateTime.of(2016, 12, 31, 23, 59, 59);
```

```
final Period week = Period.between(christmasEve.toLocalDate(),  
  
silvester.toLocalDate());  
System.out.println("a week: " + week); // a week: P7D  
System.out.println("period: " + Period.of(1, 2, 7)); // period: P1Y2M7D
```

```
final Duration sevenDays = Duration.ofDays(7);  
System.out.println("sevenDays: " + sevenDays); // sevenDays: PT168H
```

```
final Duration duration = Duration.between(christmasEve, silvester);  
System.out.println("duration: " + duration); // duration: PT174H29M59S
```

JSR-310: TemporalAdjusters & Lesbarkeit



// STATISCHE IMPORTS vs. QUALIFIZIERTE REFERENZIERUNG

```
import static java.time.Month.AUGUST;  
import static java.time.DayOfWeek.SUNDAY;  
import static java.time.temporal.TemporalAdjusters.firstInMonth;  
import static java.time.temporal.TemporalAdjusters.lastInMonth;
```

// FRIDAY 2015-08-14

```
LocalDate midOfAugust = LocalDate.of(2015, AUGUST, 14);
```

// MONDAY 2015-08-31

```
LocalDate lastOfAugust = midOfAugust.with(TemporalAdjusters.lastDayOfMonth());
```

// WEDNESDAY 2015-08-05

```
LocalDate firstWednesday = lastOfAugust.with(firstInMonth(DayOfWeek.WEDNESDAY));
```

// SUNDAY 2015-08-30

```
LocalDate lastSunday = lastOfAugust.with(lastInMonth(SUNDAY));
```



- Die Art und Weise, wie Datum und Uhrzeit dargestellt werden, variiert von Land zu Land.
 - Während man in Deutschland ein `dd.mm.YYYY` nutzt (mit den Platzhaltern `d` für Tage, `m` für Monate, `Y` für Jahre), so ist in den USA `mm/dd/YYYY` üblich, während in Großbritannien eher `dd/mm/YYYY` gebräuchlich ist.
 - Gebräuchliche und wesentliche Platzhalter sind folgende:
 - `Y` -- Jahr aus dem Bereich 1 bis 9999
 - `m` -- Monat im Bereich [01, 02, ..., 11, 12]
 - `d` -- Tag aus dem Bereich [01, 02, ..., 30, 31]
 - `H` -- Stunde aus dem Bereich [00, 01, ..., 22, 23]
 - `M` -- Minute aus dem Bereich [00, 01, ..., 58, 59]
 - `S` -- Sekundenangabe aus dem Bereich [00, 01, ..., 58, 59]
-



```
final LocalDate date = LocalDate.now();

System.out.println("original date: " + date);
final DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy MM dd");

final String text = date.format(formatter);
System.out.println("as text: " + text);

final LocalDate parsedDate = LocalDate.parse(text, formatter);
System.out.println("parsed date: " + parsedDate);
```

```
original date: 2017-06-04
as text: 2017 06 04
parsed date: 2017-06-04
```

JSR-310: Vordefinierte Formatierungen



```
final LocalDate date = LocalDate.now();
System.out.println("original date: " + date);

final DateTimeFormatter formatter1 = DateTimeFormatter.BASIC_ISO_DATE;
final DateTimeFormatter formatter2 = DateTimeFormatter.ISO_DATE;
final DateTimeFormatter formatter3 =
    DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM);

System.out.println("BASIC_ISO_DATE: " + date.format(formatter1));
System.out.println("ISO_DATE: " + date.format(formatter2));
System.out.println("ofLocalizedDate: " + date.format(formatter3));
```

```
original date:    2017-06-04
BASIC_ISO_DATE:  20170604
ISO_DATE:        2017-06-04
ofLocalizedDate: 04.06.2017
```

JSR-310: Formatierung und Zeitzonen



```
final LocalDateTime ldt = LocalDateTime.of(2016, 7, 14, 5, 25, 45);
final String pattern = "'Datum:' dd.MM.yyyy ' / Uhrzeit:' HH:mm";
final DateTimeFormatter formatter = DateTimeFormatter.ofPattern(pattern);
System.out.println("formattedDate " + formatter.format(ldt));
```

```
final String zonedDateTime = "2007-12-03T10:15:30+01:00[Europe/Paris]";
final ZonedDateTime zdt = ZonedDateTime.parse(zonedDateTime);
System.out.print(zdt + " as LocalDateTime " + zdt.toLocalDateTime());
System.out.println(" / ZoneId " + zdt.getZone());
System.out.println(" / ZoneOffset " + zdt.getOffset());
```

```
formattedDate Datum: 14.07.2016 / Uhrzeit: 05:25
```

```
2007-12-03T10:15:30+01:00[Europe/Paris] as LocalDateTime 2007-12-03T10:15:30 /
ZoneId Europe/Paris / ZoneOffset +01:00
```



DEMO

CalendarPrinter



Date API in Java 9





- `datesUntil()` – erzeugt einen `Stream<LocalDate>` zwischen zwei `LocalDate`-Instanzen und erlaubt es, optional eine Schrittweite vorzugeben:

```
public static void main(final String[] args)
{
    final LocalDate myBirthday = LocalDate.of(1971, Month.FEBRUARY, 7);
    final LocalDate christmas = LocalDate.of(1971, Month.DECEMBER, 24);

    System.out.println("Day-Stream");
    final Stream<LocalDate> daysUntil = myBirthday.datesUntil(christmas);
    daysUntil.skip(150).limit(4).forEach(System.out::println);

    System.out.println("\n3-Month-Stream");
    final Stream<LocalDate> monthsUntil =
        myBirthday.datesUntil(christmas, Period.ofMonths(3));
    monthsUntil.limit(3).forEach(System.out::println);
}
```



- Start 7. Februar => Sprung um 150 Tage in die Zukunft => 7. Juli
- **Day-Stream:** Tageweise Iteration begrenzt auf 4
- **Month-Stream:** Monatsweise Iteration begrenzt auf 3
=> Vorgabe einer alternativen Schrittweite, hier Monate:

```
Day-Stream  
1971-07-07  
1971-07-08  
1971-07-09  
1971-07-10
```

```
3-Month-Stream  
1971-02-07  
1971-05-07  
1971-08-07
```

Klasse Duration



- **divideBy()** – teilen durch die übergebene Einheit
- **truncateTo()** – abschneiden auf übergebene Einheit

```
public static void main(String[] args)
{
    final Duration tenDaysSevenHoursThirdMinutes = Duration.ofDays(10).
                                                            plusHours(7).
                                                            plusMinutes(30);

    // JDK 9: divideBy(Duration)
    final long wholeDays = tenDaysSevenHoursThirdMinutes.dividedBy(Duration.ofDays(1));
    final long wholeHours = tenDaysSevenHoursThirdMinutes.dividedBy(Duration.ofHours(1));
    final long wholeMinutes = tenDaysSevenHoursThirdMinutes.dividedBy(Duration.ofMinutes(15));
    System.out.println("wholeDays:      |" + wholeDays);
    System.out.println("wholeHours:      " + wholeHours);
    System.out.println("whole15Minutes: " + wholeMinutes);

    // JDK 9: truncatedTo(TemporalUnit)
    System.out.println("truncatedTo(DAYS):      " + tenDaysSevenHoursThirdMinutes.truncatedTo(ChronoUnit.DAYS));
    System.out.println("truncatedTo(HOURS):     " + tenDaysSevenHoursThirdMinutes.truncatedTo(ChronoUnit.HOURS));
    System.out.println("truncatedTo(MINUTES):  " + tenDaysSevenHoursThirdMinutes.truncatedTo(ChronoUnit.MINUTES));
}
```


Klasse Duration



```
public static void main(String[] args)
{
    final Duration tenDaysSevenHoursThirdMinutes = Duration.ofDays(10).
                                                            plusHours(7).
                                                            plusMinutes(30);

    // JDK 9: divideBy(Duration)
    final long wholeDays = tenDaysSevenHoursThirdMinutes.dividedBy(Duration.ofDays(1));
    final long wholeHours = tenDaysSevenHoursThirdMinutes.dividedBy(Duration.ofHours(1));
    final long wholeMinutes = tenDaysSevenHoursThirdMinutes.dividedBy(Duration.ofMinutes(15));
    System.out.println("wholeDays: " + wholeDays);
    System.out.println("wholeHours: " + wholeHours);
    System.out.println("whole15Minutes: " + wholeMinutes);

    // JDK 9: truncatedTo(TemporalUnit)
    System.out.println("truncatedTo(DAYS): " + tenDaysSevenHoursThirdMinutes.truncatedTo(ChronoUnit.DAYS));
    System.out.println("truncatedTo(HOURS): " + tenDaysSevenHoursThirdMinutes.truncatedTo(ChronoUnit.HOURS));
    System.out.println("truncatedTo(MINUTES): " + tenDaysSevenHoursThirdMinutes.truncatedTo(ChronoUnit.MINUTES));
}
```

```
wholeDays:      10
wholeHours:     247
howMany15Minutes: 990
truncatedTo(DAYS): PT240H
truncatedTo(HOURS): PT247H
truncatedTo(MINUTES): PT247H30M
```

Klasse Duration



- **toXXX()** – wandelt in die entsprechende Einheit
- **toXXXPart()** – extrahiert den Teil der entsprechenden Einheit

```
public static void main(final String[] args)
{
    final Duration tenDaysSevenHoursThirdMinutes = Duration.ofDays(10).
                                                                plusHours(7).
                                                                plusMinutes(30);

    // JDK 9:toXXX() und toXXXPart()
    System.out.println("toDays():           " + tenDaysSevenHoursThirdMinutes.toDays());
    System.out.println("toDaysPart():        " + tenDaysSevenHoursThirdMinutes.toDaysPart());
    System.out.println("toHours():          " + tenDaysSevenHoursThirdMinutes.toHours());
    System.out.println("toHoursPart():      " + tenDaysSevenHoursThirdMinutes.toHoursPart());
    System.out.println("toMinutes():        " + tenDaysSevenHoursThirdMinutes.toMinutes());
    System.out.println("toMinutesPart():    " + tenDaysSevenHoursThirdMinutes.toMinutesPart());
}
```

Klasse Duration



```
public static void main(final String[] args)
{
    final Duration tenDaysSevenHoursThirdMinutes = Duration.ofDays(10).
                                                                plusHours(7).
                                                                plusMinutes(30);

    // JDK 9:toXXX() und toXXXPart()
    System.out.println("toDays():           " + tenDaysSevenHoursThirdMinutes.toDays());
    System.out.println("toDaysPart():        " + tenDaysSevenHoursThirdMinutes.toDaysPart());
    System.out.println("toHours():         " + tenDaysSevenHoursThirdMinutes.toHours());
    System.out.println("toHoursPart():     " + tenDaysSevenHoursThirdMinutes.toHoursPart());
    System.out.println("toMinutes():       " + tenDaysSevenHoursThirdMinutes.toMinutes());
    System.out.println("toMinutesPart():  " + tenDaysSevenHoursThirdMinutes.toMinutesPart());
}
```

```
toDays():           10
toDaysPart():       10
toHours():           247
toHoursPart():      7
toMinutes():        14850
toMinutesPart():    30
```



Exercises Part 8

https://github.com/Michaeli71/JAVA_INTRO





PART 9:

Exception Handling



- **Fehler können beim Programmieren eigentlich immer und überall auftreten:**

```
jshell> 7 / 0
| Exception
java.lang.ArithmeticException: / by zero
|         at (#15:1)
```

```
jshell> String[] names = { "Tim", "Tom", "Mike" }
names ==> String[3] { "Tim", "Tom", "Mike" }
```

```
jshell> names[42]
| Exception java.lang.ArrayIndexOutOfBoundsException: Index 42 out of bounds for
length 3
|         at (#17:1)
```

- **Wenn als Folge des Fehlers eine Exception (oftmals mit Namensendung Error oder Exception) auftritt, dann stoppt die Programmausführung.**
- **Wichtig ist eine geeignete Reaktion darauf**



- **Exceptions in Java mit try-catch behandeln**
 - **Für verschiedene Arten von Problemen verschiedene Typen von Exceptions. Vier recht gebräuchliche vordefinierte Typen sind diese:**
 - **IllegalArgumentException** – Mit einer `IllegalArgumentException` können falsche Belegungen von Parametern ausgedrückt werden.
 - **NullPointerException** – Sind Eingabewerte null, so kann man darauf mit einer `NullPointerException` reagieren.
 - **IllegalStateException** – Sind benötigte Daten nicht korrekt initialisiert, so kann dies über eine `IllegalStateException` kommuniziert werden.
 - **UnsupportedOperationException** – Auf eine fehlende Implementierung kann mittels einer `UnsupportedOperationException` hingewiesen werden.
-



- **Exceptions mit try-catch**

```
try
{
    // Hier können Exceptions auftreten
}
catch (Exception-Typ1 e)
{
    // Hier können Fehlersituationen behandelt werden
}
catch (Exception-Typ2 e)
{
    // Hier können Fehlersituationen behandelt werden
}
```


Auf mehrere Exceptions reagieren



- Auf mehrere Exceptions reagieren (Multi Catch)

```
var names = List.of("Tim", "Tom", "Mike");
for (int i = 0; i < 5; i++)
{
    try
    {
        int value = Integer.valueOf(names.get(i));
    }
    catch (NumberFormatException | ArithmeticException ex)
    {
        System.out.println("can't parse to integer");
    }
    catch (ArrayIndexOutOfBoundsException aioobe)
    {
        System.out.println("wrong index");
    }
}
```

```
can't parse to integer
can't parse to integer
can't parse to integer
wrong index
wrong index
```



- **Unspezifisch auf mehrere Exceptions reagieren**

```
List<String> names = List.of("Tim", "Tom", "Mike");
try
{
    System.out.println("INVALID INDEX: " + names.get(42));
}
catch (Exception ex)
{
    System.out.println(("an unspecified error occurred. seams to be wrong index"));
}
```

- **Ergebnis**

an unspecified error occurred. seams to be wrong index

- **ABER: Mit der gezeigten Art lassen sich Fehlersituationen nicht unterscheiden und somit kann man nicht adäquat auf unterschiedliche Probleme reagieren.**



- **Folgende grundsätzliche Struktur**

```
try
{
    // Hier können Exceptions auftreten
}
catch (Exception e)
{
    // Hier werden Exceptions abgearbeitet, sofern der catch-Block vorhanden ist.
    // Ansonsten muss die Exception in der Methodensignatur aufgeführt werden,
    // falls es eine Checked Exception (vgl. Abschnitt 11.4) ist.
}
finally
{
    // Wird immer durchlaufen, ist allerdings optional
}
```

Der letzte Wille – abschließende Aktionen und der finally-Block



- **Beispiel**

```
String[] names = { "Tim", "Tom", "Mike" };  
try  
{  
    System.out.println("INVALID INDEX: " + names[42]);  
}  
catch (ArrayIndexOutOfBoundsException aioobe)  
{  
    System.out.println("wrong index");  
}  
finally  
{  
    System.out.println("ALWAYS EXECUTED");  
}
```

- =>

wrong index

ALWAYS EXECUTED



- Mittlerweile haben wir schon mehrmals gesehen, dass bei der Abarbeitung von Programmen in gewissen Fehlersituationen automatisch Exceptions ausgelöst werden.
- Aber auch wir als Programmierer können selbst Exceptions auslösen.
- Dazu dient das Schlüsselwort **throw** in Kombination mit einem **Ausnahmetyp**.

```
void ensureValueInRange(int value, int lowerBound, int upperBound)
{
    if (value < lowerBound || value > upperBound)
        throw new IllegalArgumentException("out of bounds");
}
```

Eigene Exception-Typen definieren



- Neben der Verwendung vordefinierter Exceptions problemlos möglich, eigene Typen von Exceptions zu definieren.

```
public class CustomerNotFoundException extends Exception
{
    public CustomerNotFoundException(String message)
    {
        super(message);
    }

    public CustomerNotFoundException(String message, Throwable throwable)
    {
        super(message, throwable);
    }
}
```

- Sinnvoller als eine derart pure Exception ist es natürlich, dort weitere Informationen bereitzuhalten
-

Propagation von Exceptions



- In `func2()` wird eine `IllegalStateException` ausgelöst. In keiner der Methoden findet eine Fehlerbehandlung statt:

```
public static void main(String[] args)
{
    func1();
}

static void func1()
{
    func2();
}

static void func2()
{
    throw new IllegalStateException("propagate me");
}
```

- Aufruf: `main()` → `func1()` → `func2()` und Rückpropagation



ARM

Automatic Resource Management



// I/O ohne ARM

```
public static String readFirstLine(final String path) // throws IOException
{
    BufferedReader br = null;
    try
    {
        br = new BufferedReader(new FileReader(path));
        return br.readLine();
    }
    catch (final IOException ex)
    {
        // handle or rethrow
    }
    finally
    {
        // Diese manuellen Aufräumarbeiten werden durch ARM überflüssig
        try
        {
            if (br != null)
                br.close();
        }
        catch (final IOException ioe)
        {
            // ignore
        }
    }
    return "";
}
```



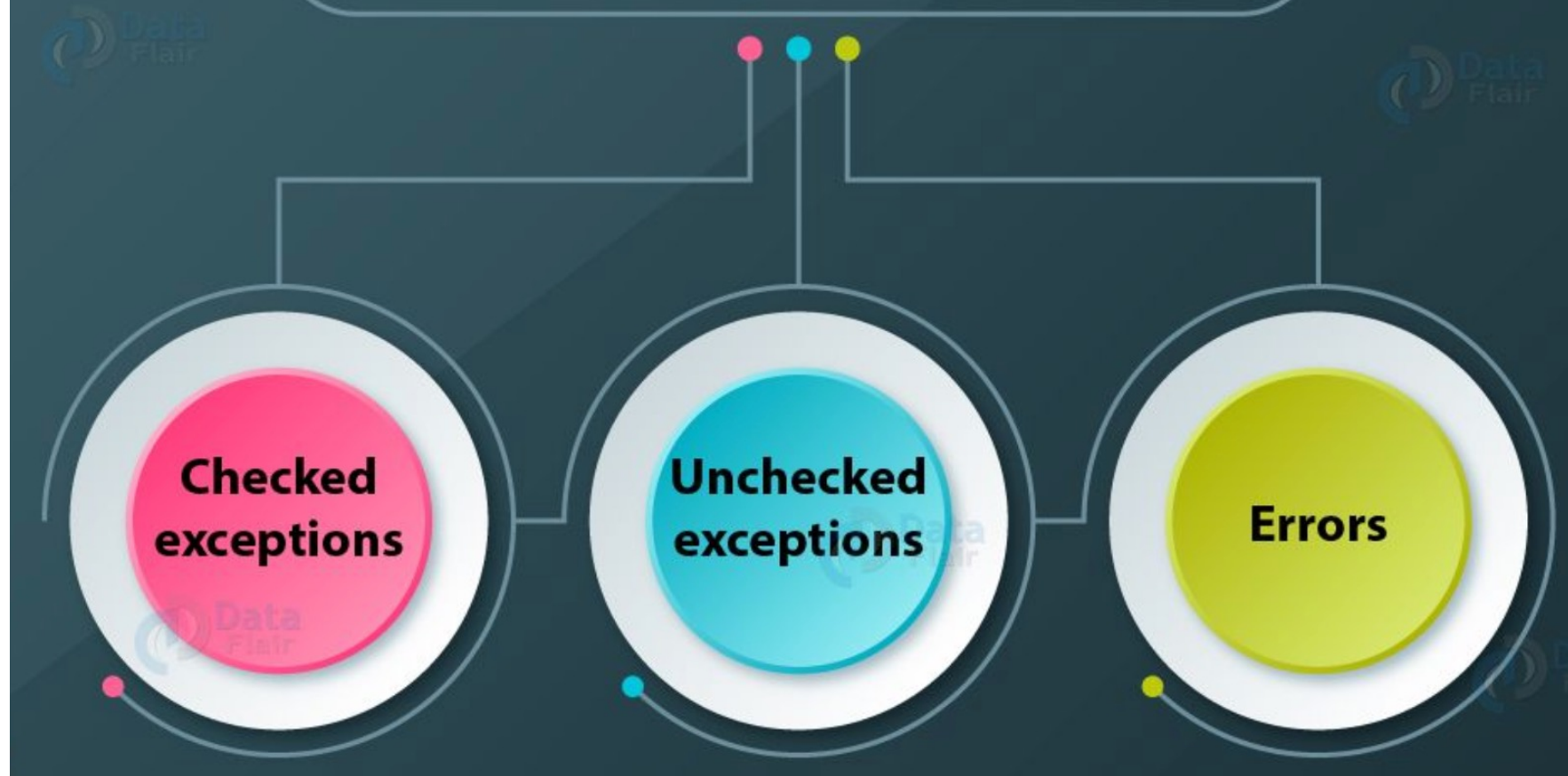
ARM– Automatic Resource Management (try-with-resources)



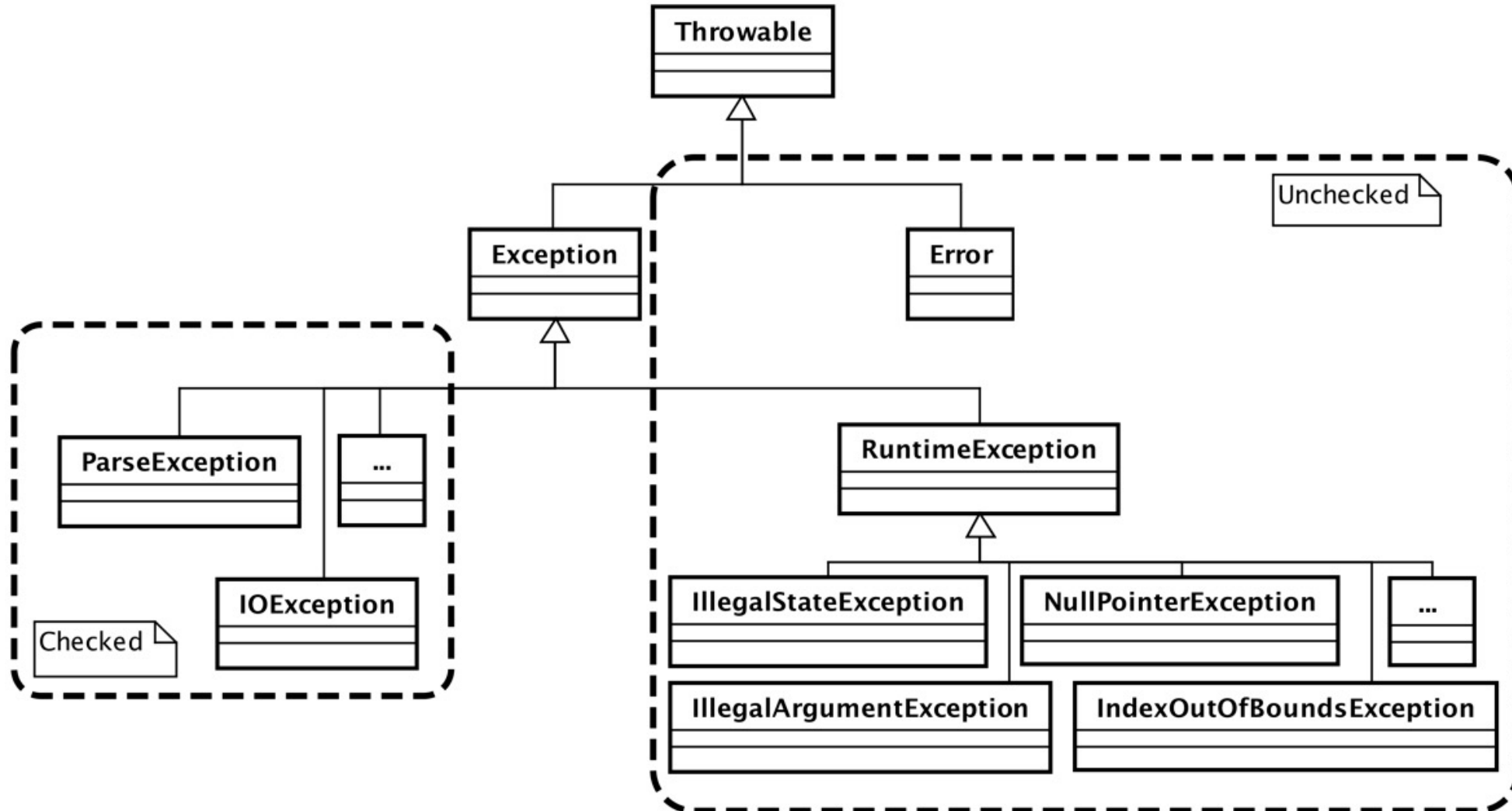
```
public static String readFirstLine(final String path)
{
    // Spezielle Angabe der Ressourcenvariablen
    try (final FileReader fr = new FileReader(path);
        final BufferedReader br = new BufferedReader(fr))
    {
        return br.readLine();
    }
    catch (final IOException ex)
    {
        // handle or rethrow
    }
    return "";
}
```



Categories of Exceptions



Checked vs Unchecked Exceptions





PART 10:

Dateiverarbeitung





- **Ein wichtiger Bestandteil vieler Anwendungen ist die Verarbeitung von Informationen aus Dateien**
 - **Java bietet zur Ein- und Ausgabe in den Packages `java.io` und `java.nio` einen objektorientierten Zugang und mehrere Möglichkeiten zum Erstellen, Lesen, Aktualisieren und Löschen von Dateien.**
 - **bei der Kommunikation und Ein- und Ausgabe immer auch zu Fehlern oder Problemen möglich => Exception Handling schon thematisiert**
 - **Beispiele bieten nur rudimentäre Fehlerbehandlung, um diese kurz zu halten**
-



- **Beispiele bieten nur rudimentäre Fehlerbehandlung, um diese kurz zu halten**
- **Beispiele nutzen folgende Verzeichnisstruktur als Ausgangsbasis**

```
files-examples-dir
|-- example-data.csv
|-- example-file.txt
|-- rename-dir
|-- subdir1
`-- subdir2
```



- **Beispiele sollen folgende Verzeichnisstruktur als Ausgangsbasis nutzen:**

```
files-examples-dir
|-- example-data.csv
|-- example-file.txt
|-- rename-dir
|-- subdir1
`-- subdir2
```

- **Neue Verzeichnisse mit der Methode `createDirectory()` erstellen:**

```
jshell> Files.createDirectory(Path.of("files-examples-dir"))
$255 ==> files-examples-dir
```

```
jshell> Files.createDirectory(Path.of("files-examples-dir/rename-dir"))
$256 ==> files-examples-dir/rename-dir
```




- **Neue Verzeichnisse mit der Methode `createDirectory()` erstellen:**

```
jshell> Files.createDirectory(Path.of("files-examples-dir/subdir1"))  
$257 ==> files-examples-dir/subdir1
```

```
jshell> Files.createDirectory(Path.of("files-examples-dir/subdir2"))  
$258 ==> files-examples-dir/subdir2
```

- **Neue Dateien mit der Methode `createFile()` erstellen:**

```
var newCsvFile = Files.createFile(Path.of("files-examples-dir/example-data.csv"))  
newCsvFile ==> files-examples-dir/example-data.csv
```

```
jshell> var newTxtFile = Files.createFile(Path.of("files-examples-dir/example-  
file.txt"))  
newTxtFile ==> files-examples-dir/example-file.txt
```



- **Das aktuelle Verzeichnis lässt sich mit `list()` wie folgt auslesen:**

```
jshell> Files.list(Path.of(".")).forEach(System.out::println)
```

```
jshell> Files.list(Path.of("files-examples-dir")).forEach(System.out::println)
files-examples-dir/rename-dir
files-examples-dir/subdir2
files-examples-dir/example-data.csv
files-examples-dir/subdir1
files-examples-dir/example-file.txt
```

- **Das durch Path-spezifizierte Verzeichnis als Liste auslesen:**

```
static List<Path> listDirectory(Path dir) throws IOException
{
    try (Stream<Path> content = Files.list(dir))
    {
        return content.toList();
    }
}
```



- **Datei oder Verzeichnis? (isDirectory() / isRegularFile())**

```
var dirContent = listDirectory(Path.of("."));

for (var path : dirContent)
{
    if (Files.isDirectory(path))
    {
        System.out.println(path + " is a directory");
    }
    if (Files.isRegularFile(path))
    {
        System.out.println(path + " is a file");
    }
}
```



- **Größe für Pfad bzw. Datei mit `size()` bestimmen:**

```
jshell> Files.size(Path.of("files-examples-dir"))  
$267 ==> 224
```

- **Existenzprüfung? (`exists()`)**

```
jshell> Files.exists(Path.of("UnknownFile.txt"))  
$263 ==> false
```

- **Absoluten Pfad bestimmen:**

```
jshell> Path absolute = Path.of(".").toAbsolutePath()  
absolute ==> /Users/michaelinden/.
```



DEMO

DirectoryTreeWithPath



- In Java 11 wurde die Verarbeitung von Strings im Zusammenhang mit Dateien erleichtert.
- Es ist nun einfach möglich, Strings in eine Datei zu schreiben bzw. daraus zu lesen.
- Dazu bietet die Utility-Klasse `Files` die Methoden `writeString()` und `readString()`.

```
final Path destDath = Path.of("ExampleFile.txt");
```

```
Files.writeString(destDath, "1: This is a string to file test\n");
```

```
Files.writeString(destDath, "2: Second line");
```

```
final String line1 = Files.readString(destDath);
```

```
final String line2 = Files.readString(destDath);
```

```
System.out.println(line1);
```

```
System.out.println(line2);
```

=>

```
2: Second line
```

```
2: Second line
```



- **Korrektur 1:** APPEND-Mode

```
Files.writeString(destDath, "2: Second line", StandardOpenOption.APPEND);
```

=>

```
1: This is a string to file test
2: Second line
1: This is a string to file test
2: Second line
```

- **Korrektur 2:** String nur einmal lesen

```
final String content = Files.readString(destDath);
content.lines().forEach(System.out::println);
```

=>

```
1: This is a string to file test
2: Second line
```



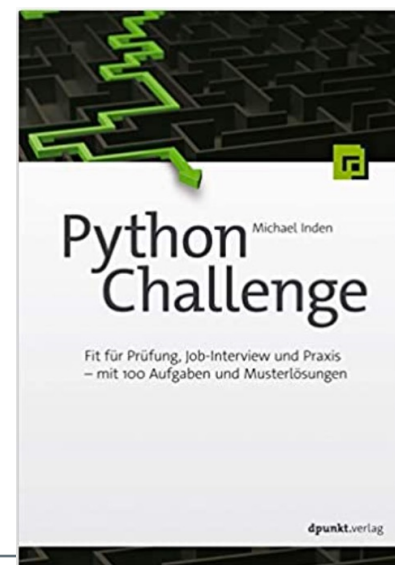
Exercises Part 10

https://github.com/Michaeli71/JAVA_INTRO





Questions?





Thank You
