universität innsbruck

Java-API

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller
Universität Innsbruck

Beispiel (Arrays – Hilfsklasse)

```java
import java.util.Arrays;

public class ArraysApiExample {

    public static void main(String[] args) {
        int[] playerRanks = { 3, 5, 2, 7, 9, 4, 6, 1, 8, 8 };

        Arrays.sort(playerRanks);
        int index = Arrays.binarySearch(playerRanks, 5);

        System.out.println(Arrays.toString(playerRanks));
        System.out.println("index of 5: " + index);
    }
}
```

Ausgabe:

```
[1, 2, 3, 4, 5, 6, 7, 8, 8, 9]
index of 5: 4
```

Favor Java-API over DIY

- Vorher:
  - Eigene Implementierung (z.B. der Sort-Methode)
  - Fehleranfälligkeit
  - Längerer Code
- Nachher:
  - API-Implementierung verwendet
  - Code ist kürzer und lesbarer
  - Es wird auf eine effiziente Implementierung zurückgegriffen, die bereits ausgiebig getestet wurde.

Dokumentation der API

- Startpunkt (Java 17)
  - https://docs.oracle.com/en/java/javase/17/docs/api/index.html
- Bereitgestellte Informationen (Beispiele)
  - In welchem Paket befindet sich eine Klasse?
    - Die einzelnen Klassen eines Pakets können ausgewählt werden.
  - Für eine Klasse
    - Beschreibung der Klasse
    - Liste der Methoden
  - Für jede Methode
    - Kurzbeschreibung
    - Längere Beschreibung (Parameter, Ausnahmen etc.)

Beispiele für API-Klassen

System

- Die Klasse System bietet statische Felder und Methoden an.
- Felder:
  - `err` – Standard Error Output Stream
  - `in` – Standard Input Stream
  - `out` – Standard Output Stream
- Methoden (Beispiele):

  `arraycopy(src, srcPos, dest, destPos, length)`
  - Kopieren von `length` Elementen des `src`-Arrays beginnend bei Position `scrPos` in das `dest`-Array beginnend an Position `destPos`.

  `getProperties()`
  - Liefert die aktuellen Systemeigenschaften.

Math

- Die Klasse `Math` bietet eine Vielzahl statischer Methoden im mathematischen Kontext an.
- Beispiele:

  `exp()`, `log()`, `log10()`, `sqrt()`, `sin()`, `cos()`, `tan()`, …
  - Exponential-, Logarithmus- , Wurzel- und trigonometrische Funktionen

  `addExact()`, `subtractExact()`, `multiplyExact()`
  - Addition, Subtraktion bzw. Multiplikation mit Überlaufüberprüfung

  `min(a, b)`
  - Ermitteln des Minimums aus `a` und `b`.

  `max(a, b)`
  - Ermitteln des Maximums aus `a` und `b`.

  `ceil(a)`
  - Aufrunden von `a`.

  `floor(a)`
  - Abrunden von `a`.

BigInteger (1)

- Mit `BigInteger` können beliebig genaue Ganzzahlen erzeugt, verwaltet und für Berechnungen herangezogen werden.
- `BigInteger`-Objekte sind immutable.
- Garantierter Wertebereich:
  - Ganzzahlen im Intervall $(-2^{\texttt{Integer.MAX\_VALUE}}, 2^{\texttt{Integer.MAX\_VALUE}})$
- Konstanten:
  - `ZERO`, `ONE`, `TWO`, `TEN`
- Alle angebotenen Operationen verhalten sich als wären `BigInteger`-Werte im Zweierkomplement dargestellt.
- Angebotene Operationen (Auszug):

  `add()`, `subtract()`, `multiply()`, `divide()`, `and()`, `not()`, `negate()`, …
  - Methoden analog zu den Operatoren für primitive Ganzzahlentypen

  `min()`, `max()`, `abs()`, `sqrt()`, `pow()`, …
  - Methoden analog zu den Operationen für Ganzzahlen aus `java.lang.Math`

BigInteger (2)

- Konstruktoren (Auszug):

  `BigInteger(byte[] val)`
  - Erzeugt ein `BigInteger` basierend auf der Binärdarstellung im Zweierkomplement aus dem `byte`-Array.

  `BigInteger(String val)`
  - Erzeugt ein `BigInteger` aus dem angegebenen `String`.

- Konvertierungsmethoden (Auszug):

  `BigInteger.valueOf(long val)`
  - Erzeugt ein `BigInteger`, welches die übergebene Ganzzahl repräsentiert.

  `intValue()`, `longValue()`, `floatValue()`, `doubleValue()`, …
  - Umwandlung in einen primitiven Typ.
  - Es kann zu Verlusten der Größenordnung und Genauigkeit kommen, wie es bei einschränkenden Konvertierungen der Fall sein kann.

  `intValueExact()`, `longValueExact()`, …
  - Umwandlung in einen primitiven Ganzzahltyp.
  - Bei Informationsverlust wird eine `ArithmeticException` geworfen.

BigDecimal (1)

- Mit `BigDecimal` können beliebig genaue Dezimalzahlen erzeugt, verwaltet und für Berechnungen herangezogen werden.
- `BigDecimal`-Objekte sind immutable.
- Interne Repräsentation: $unscaledValue \cdot 10^{-scale}$
  - Wobei $unscaledValue$ vom Typ `BigInteger` und $scale$ vom Typ `int` ist.
  - Beispiele:

    $123456 \cdot 10^{-5} \rightarrow 1{,}23456$
    - $unscaledValue = 123456$
    - $scale = 5$

    $56 \cdot 10^{3} \rightarrow 56000$
    - $unscaledValue = 56$
    - $scale = -3$
- Konstanten:
  - `ZERO`, `ONE`, `TEN`

BigDecimal (2)

- Angebotene Operationen (Auszug):

  `add()`, `subtract()`, `multiply()`, `divide()`
  - Grundrechenarten

  `min()`, `max()`, `abs()`, `sqrt()`, `pow()`, `negate()`, …
  - Weitere arithmetische Operationen

- `BigDecimal` erlaubt vollständige Kontrolle des Rundungsverhaltens.
  - Kann ein Ergebnis bei nicht angegebenen Rundungsverhalten nicht exakt dargestellt werden kommt es zu einer `ArithmeticException`.
    - Beispiel: $1/3 = 0,\overline{3}$
  - Rundungsverhalten wird im allgemeinen mit einem `MathContext`-Objekt angegeben.
    - Bei den Methoden `divide()` und `setScale()` kann auch direkt eine der Aufzählungskonstanten aus `RoundingMode` angegeben werden.
  - `RoundingMode`-Aufzählungskonstanten:
    - `UP`, `DOWN`, `CEILING`, `FLOOR`, `HALF_UP`, `HALF_DOWN`, `HALF_EVEN`

Beispiel – Runden (1)

```java
import java.math.BigDecimal;

import static java.math.BigDecimal.ONE;

public class BigDecimalExample1 {

    public static void main(String[] args) {
        BigDecimal three = new BigDecimal(3);
        System.out.println(ONE.divide(three));
    }
}
```

Ausgabe:

```
Exception in thread "main" java.lang.ArithmeticException: Non-terminating
decimal expansion; no exact representable decimal result.
   at java.base/java.math.BigDecimal.divide(BigDecimal.java:1766)
   at BigDecimalExample1.main(BigDecimalExample1.java:8)
```

Beispiel – Runden (2)

```
import java.math.BigDecimal;
import java.math.MathContext;
import java.math.RoundingMode;

import static java.math.BigDecimal.ONE;

public class BigDecimalExample2 {

    public static void main(String[] args) {
        BigDecimal three = new BigDecimal(3);
        System.out.println(ONE.divide(three, 5, RoundingMode.HALF_UP));
        System.out.println(ONE.divide(three, MathContext.DECIMAL32));
    }
}
```

Ausgabe:

```
0.33333
0.3333333
```

BigDecimal – Konstruktoren

- Konstruktoren (Auszug):
  - `BigDecimal(double val)`
    - Erzeugt ein `BigDecimal` aus der exakten Gleitkommadarstellung des angegebenen double-Werts.
  - `BigDecimal(long val)`
    - Erzeugt ein `BigDecimal` aus dem angegebenen `long`-Wert.
  - `BigInteger(String val)`
    - Erzeugt ein `BigDecimal` aus dem angegebenen `String`.

```
import java.math.BigDecimal;

public class BigDecimalExample1 {

    public static void main(String[] args) {
        System.out.println(new BigDecimal( 1.001));
        System.out.println(new BigDecimal("1.001"));
    }
}
```

Ausgabe:

```
1.00099999999999988986587595718447118997573852539062 5
1.001
```

BigDecimal – Konvertierungsmethoden

- Konvertierungsmethoden (Auszug):

`intValue()`, `longValue()`, `floatValue()`, `doubleValue()`, …

- Umwandlung in einen primitiven Typ.
- Es kann zu Verlusten der Größenordnung und Genauigkeit kommen, wie es bei einschränkenden Konvertierungen der Fall sein kann.

`intValueExact()`, `longValueExact()`, …

- Umwandlung in einen primitiven Ganzzahltyp.
- Bei Informationsverlust wird eine `ArithmeticException` geworfen.

`toBigInteger()`

- Umwandlung in ein `BigInteger`.

`toBigIntegerExact()`

- Umwandlung in ein `BigInteger`.
- Gibt es Nachkommastellen, wird eine `ArithmeticException` geworfen.

`BigDecimal.valueOf(long val)`

- Erzeugt ein `BigDecimal`, welches die übergebene Ganzzahl repräsentiert.

`BigDecimal.valueOf(long unscaledVal, int scale)`

- Erzeugt ein `BigDecimal`, welches die übergebene Ganzzahl unter Berücksichtigung von `scale` repräsentiert.

Date-Time API (java.time)

- Mit Java 8 wurde die neue Date-Time API in Java eingeführt.
- Die Klassen der alten Date-Time API sollten in neuem Code nicht mehr verwendet werden.
- Die Date-Time API bietet neben Klassen, die ein konkretes Datum bzw. eine konkrete Uhrzeit repräsentieren unter anderem auch Klassen zum
  - Darstellen von Zeitpunkten und Zeitspannen,
  - Formatieren und Parsen von Datum und Zeit,
  - Arbeiten mit zeitzonenabhängigen Daten und Zeiten sowie
  - Rechnen mit Objekten der Date-Time API.
- Alle Klassen aus `java.time` sind immutable.

Angebotene Klassen (Auszug)

`LocalDate`
- Speichert ein Datum ohne Zeitzone.

`LocalTime`
- Speichert eine Zeit ohne Zeitzone.

`LocalDateTime`
- Speichert Datum und Zeit ohne Zeitzone.

`ZonedDateTime`
- Speichert Datum, Zeit und Zeitzone.

`DayOfWeek`
- Aufzählungstyp der Wochentage repräsentiert. (`MONDAY`, …, `SUNDAY`)

`Month`
- Aufzählungstyp der Monate repräsentiert. (`JANUARY`, …, `DECEMBER`)

`ChronoUnit`
- Aufzählungstyp der standardisierte Zeiteinheiten repräsentiert.
- `NANOS`, …, `HOURS`, …, `DECADES`, …, `MILLENIA`, `ERAS`, `FOREVER`

`TemporalAdjusters`
- Sammlung von Hilfsmethoden zum Rechnen mit Objekten der Date-Time API.

Namensschema angebotener Methoden

`of*()` – statisch
- Erzeugen eines neuen Objekts.

`parse*()` – statisch
- Erzeugen eines neuen Objekts basierend auf einer String-Repräsentation.

`get*()`
- Liefert einen Wert.

`is*()`
- Fragt den Status eines Objekts ab.

`with*()`
- Liefert eine Kopie des Objekts mit geänderter Eigenschaft zurück.

`plus*()`
- Liefert eine Kopie des Objekts mit aufsummierter Eigenschaft zurück.

`minus*()`
- Liefert eine Kopie des Objekts mit reduzierter Eigenschaft zurück.

`to*()`
- Konvertiert ein Objekt in einen anderen Typ.

`at*()`
- Kombiniert ein Objekt mit einem anderen Objekt (z.B. `date.atTime(time)`).

Date-Time API - Beispiel (1)

```
import java.time.LocalDateTime;
import static java.time.temporal.ChronoUnit.*;

public class TimeApiExample1 {

    public static void main(String[] args) {
        LocalDateTime departure = LocalDateTime.now();
        LocalDateTime arrival = departure.plusHours(2).plusMinutes(40);
        System.out.println("departure: " + departure);
        System.out.println("arrival: " + arrival);

        long hoursBetween = HOURS.between(departure, arrival);
        long minutesBetween = MINUTES.between(departure, arrival);
        System.out.println("difference (hours): " + hoursBetween);
        System.out.println("difference (minutes): " + minutesBetween);
    }
}
```

Ausgabe:

```
departure: 2022-04-01T10:18:27.940580
arrival: 2022-04-01T12:58:27.940580
difference (hours): 2
difference (minutes): 160
```

Date-Time API - Beispiel (2)

```
import java.time.LocalDate;
import static java.time.DayOfWeek.MONDAY;
import static java.time.Month.MARCH;
import static java.time.temporal.TemporalAdjusters.firstInMonth;

public class TimeApiExample2 {

    public static LocalDate getSummerSemesterStart(int year) {
        LocalDate date = LocalDate.of(year, MARCH, 1);
        return date.with(firstInMonth(MONDAY));
    }
}
```

Quellen

- Christian Ullenboom: **Java ist auch eine Insel: Einführung, Ausbildung, Praxis**, Rheinwerk Verlag, 16. Auflage, 2022 (Java 17)
- Michael Inden: **Der Weg zum Java-Profi: Konzepte und Techniken für die professionelle Java-Entwicklung**, dpunkt.verlag, 5. Auflage, 2021
- James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, Gavin Bierman: **The Java® Language Specification (***Java SE 17 Edition***)**, Oracle, 2021