gds2 Gottlieb-Daimler-Schule 2
Technisches Schulzentrum Sindelfingen
mit Abteilung Akademie für Datenverarbeitung

Inhaltsverzeichnis - Java Standardklassen

1 Standardklassen in Java

Mit dem Java-Paket jdk8 werden ca. 4000 Klassen ausgeliefert, die vom Programmierer benutzt werden können. Diese werden als Standardklassen bezeichnet.

Zu jeder dieser Klassen existiert eine ausführliche *Dokumentation*, die in einem standardisierten Format als HTML-Dateien abgelegt ist. Die Dokumentation gibt Auskunft über die verfügbaren Klassen, deren Methoden und die korrekte Benutzung der Methoden. Ebenso werden ausführliche Erklärungen und Programmierbeispiele gegeben. Ohne diese Dokumentation ist die Benutzung der Standardklassen fast unmöglich. Aus diesem Grund ist es unerlässlich, sich in die Benutzung der Dokumentation einzuarbeiten. Die meisten Entwicklungsumgebungen unterstützen direkt die Dokumentation in Form von Codevervollständigung.

2 Erzeugen von Objekten der Klasse String

Bevor mit einem Objekt einer Klasse gearbeitet werden kann, muss es erzeugt werden. Man spricht auch von der ***Instanziierung*** eines Objekts.

Dafür steht in jeder Klasse - mindestens ein - ***Konstruktor*** zur Verfügung.
Ein Konstruktor ist eine spezielle Methode,

- deren Name dem Namen der Klasse entspricht und
- die keinen Rückgabewert hat

Beispiele:

Der Konstruktor der Klasse String heißt `String()`

Der Konstruktor der Klasse Auto heißt `Auto()`

Der Konstruktor der Klasse Bruch heißt `Bruch()`

Ein neues Objekt wird in 2 Schritten erzeugt:

1. es wird eine Variable deklariert, deren Datentyp dem Namen der Klasse entspricht. Derartige Datentypen werden als **Referenztypen** bezeichnet (im Gegensatz zu den "primitiven Datentypen" int, long etc.).

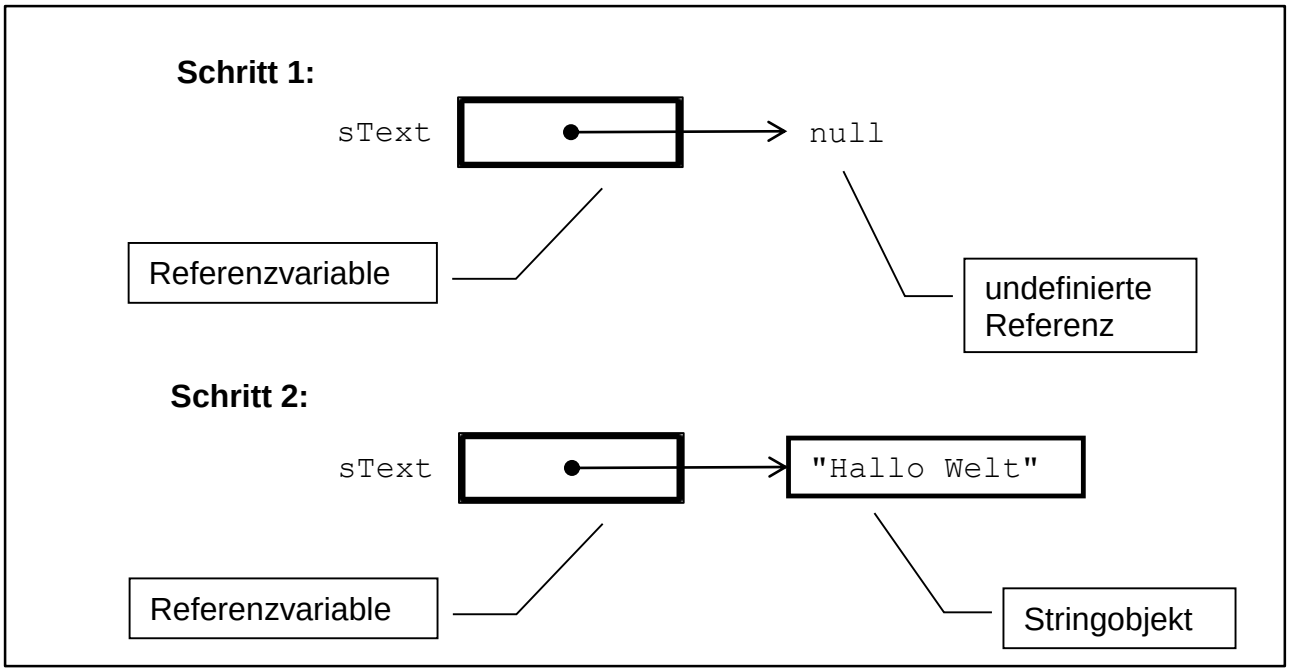
   Somit ist ***jede Klasse ein Datentyp***.

   Die deklarierte Variable wird als **Referenzvariable** bezeichnet. Sie enthält später nur die Referenz (=Adresse) des eigentlichen Objekts.
2. anschließend wird der Konstruktor der Klasse zusammen mit dem Schlüsselwort **new** aufgerufen.

Beispiel:

```
public class ObjektDemo
{
  public static void main (String [] arg)
  {
    // Schritt 1:
    String sText;   // Deklaration der Referenzvariablen "sText"

    // Schritt 2:
    sText = new String("Hallo Welt");
                    // Erzeugen des Stringobjekts und Zuweisen an die
                    // Referenzvariable sText.
                    // Der Inhalt der Variablen ist der Verweis auf
                    // das Objekt "Hallo Welt"
  }
}
```



Schritt 1 und 2 können auch in einem Schritt ausgeführt werden:

```
    // Schritt 1 und 2 zusammen:
    String sText = new String("Hallo Welt");
```

***Besonderheit bei Stringobjekten*:** Objekte der Klasse String können auch ohne new erzeugt werden:

```
String sText = "Hallo Welt";
```

3 Arbeiten mit Objekten und Objektmethoden

Nachdem das Objekt erzeugt ist, kann mit ihm gearbeitet werden, d.h. seine Attribute und Methoden können benutzt/aufgerufen werden.
Wie bereits erwähnt sind die Informationen zu den Methoden der Java-Dokumentation zu entnehmen.
Alle Methoden, die im Zusammenhang mit Objekten aufgerufen werden können, heißen ***Objektmethoden***[1] oder ***„nicht statische Methoden“***.

3.1 Die Schreibweise von Objektmethoden (Syntax)

Genau wie bei Klassenmethoden besteht auch der Kopf einer Objektmethode aus einem ***Namen***, einem ***Ergebnistyp*** und evtl. einer ***Parameterliste***. Der Methodenrumpf enthält wie bei Klassenmethoden den Algorithmus der Operation, die ausgeführt werden soll. Objektmethoden haben aber Zugriff auf alle Attribute der Klasse.

Zur Beachtung:

Die Unterschiede zwischen Objektmethoden und Klassenmethoden sind:

- Bei Objektmethoden ***fehlt*** das Schlüsselwort ***static***
- Objektmethoden dürfen nur mit einem Objekt aufgerufen werden (s.u. 3.2)

Es gilt die allgemeine Form:

<Zugriffsattribut> <Ergebnis-Datentyp> Methodenname (Parameterliste)

Zugriffsattribut:
Zuerst *kann* ein Zugriffsattribut angegeben werden (***z.B.*** **public**: die Methode ist öffentlich, d.h. darf von allen anderen Methoden aufgerufen werden)
Ergebnis-Datentyp:
Eine Methode *kann* einen ***Rückgabewert*** liefern. Der Rückgabewert wird an den Aufrufer zurückgeliefert. Der Datentyp dieses Rückgabewertes wird vor dem Methodennamen angegeben. Liefert eine Methode keinen Rückgabewert, so wird der Datentyp void angegeben.
Methodenname:
Über den Methodennamen wird eine Methode aufgerufen. Der Name sollte möglichst die Operation wiederspiegeln, die von der Methode ausgeführt wird.
Parameterliste:
einer Methode können Parameter übergeben werden. Parameter sind Variablen, denen beim Aufruf der Methode ein konkreter Wert zugewiesen wird. Sie werden innerhalb der runden Klammern mit Datentyp und Variablenname deklariert.
Werden mehrere Parameter erwartet, so werden sie mit ***Komma abgetrennt***.
Die Reihenfolge und der Datentyp müssen dabei genau eingehalten werden.

Beispiele:

public int length()	liefert die Anzahl Zeichen, erhält keine Parameter
public String substring (int von, int bis)	liefert einen Teil des Strings ab Position von bis Position bis

[1] Wir haben bereits mit Klassenmethoden (statischen Methoden) gearbeitet. Diese gehören zu einer Klasse, aber nicht zu einem Objekt.

Beispiele aus der Java-Dokumentation:

String(StringBuilder builder)
Allocates a new string that contains the sequence of characters currently contained in the string builder argument.

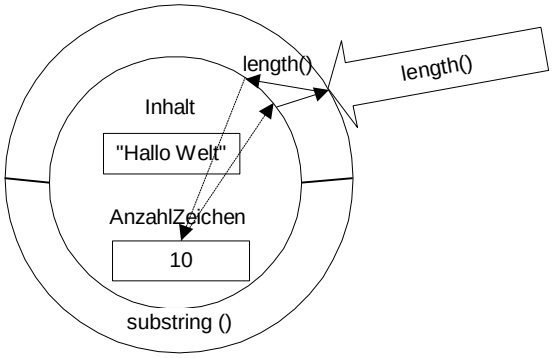
Method Summary

char	**charAt**(int index) Returns the char value at the specified index.
int	**codePointAt**(int index) Returns the character (Unicode code point) at the specified index.
int	**codePointBefore**(int index) Returns the character (Unicode code point) before the specified index.
int	**codePointCount**(int beginIndex, int endIndex) Returns the number of Unicode code points in the specified text range of this String.
int	**compareTo**(String anotherString) Compares two strings lexicographically.
int	**compareToIgnoreCase**(String str) Compares two strings lexicographically, ignoring case differences.
String	**concat**(String str) Concatenates the specified string to the end of this string.

Rückgabe-wert (Datentyp)	alphabetische Liste der Methodennamen der Klasse mit erwarteten Übergabeparametern und Kurzbeschreibung. Klick auf den Methodennamen liefert eine ausführlichere Beschreibung

3.2 Aufruf von Objektmethoden

In der Sprache der Objektorientierung heißt es: Der Aufrufer sendet eine Botschaft an das Objekt, welches dann eine Methode gleichen Namens aufruft.



Die Methoden eines Objekts werden in der Form
objektname.methodenname();
aufgerufen.
Der ***Rückgabewert*** wird vom Aufrufer weiterverarbeitet, indem er ihn einer ***lokalen Variablen*** zuweist oder in einem Ausdruck direkt weiter verwendet:

Rückgabewert = objektname.methodenname();

Beispiel:

```
public class ObjektDemo
{
  public static void main (String [] arg)
  {
    int iLaenge = 0;     // Deklaration der Variablen, die den
                         // Rueckgabewert aufnimmt
    String sText;

    sText = "Hallo Welt";
    iLaenge = sText.length(); // Aufruf der Objektmethode length()
    System.out.println("Laenge des Textes: " + iLaenge);

    // Oder alternativ:
    System.out.println("Laenge des Textes: " + sText.length());

  }
}
```

4 Arbeiten mit ausgewählten Standardklassen

4.1 String

Die Klasse String stellt viele Methoden zur Verfügung, mit denen Zeichenketten bearbeitet werden können. Im Wesentlichen sei verwiesen auf die Java-Dokumentation, jedoch sollen einige Funktionsgruppen hier kurz vorgestellt werden.

4.1.1 Zeichenextraktion und Zerlegen von Strings

char charAt (int index)	liefert Zeichen an der Stelle index. Zählung beginnt bei 0
String substring (int von, int bis)	liefert Teilstring ab der Position von (inclusiv) bis bis (exclusiv)
String trim ()	entfernt führende und schließende Leerzeichen
String [] split (String regex)	trennt den String an allen Stellen, die dem regulären Ausdruck entsprechen.

4.1.2 Vergleichen von Strings

boolean equals(Object einObject)	vergleicht den Inhalt des aktuellen String-Objekts mit einObject
boolean equalsIgnoreCase(String s2)	vergleicht den Inhalt des aktuellen String-Objekts mit s2 und ignoriert Groß- und Kleinbuchstaben
int compareTo(String s2)	vergleicht den Inhalt des aktuellen String-Objekts lexikografisch mit s2; ist s2 im Lexikon „weiter hinten“, so ist das Ergebnis < 0, bei gleichen Strings = 0, sonst > 0
Int compareToIgnoreCase(String s2)	Wie compareTo(..) ohne Berücksichtigung von Groß-Klein-Schreibung
boolean contentEquals(CharSequence cs)	vergleicht den Inhalt des aktuellen String-Objekts mit einer Implementierung des Interface CharSequence, z.B. StringBuilder
boolean startsWith(String s)	vergleicht den Anfang des aktuellen String-Objekts mit s
boolean endsWith(String s)	vergleicht das Ende des aktuellen String-Objekts mit s
boolean regionMatches(int this_offset, String s2, int s2_offset, int len)	vergleicht Regionen in 2 unterschiedlichen Strings

***Achtung*:** Vergleich s1 == s2 liefert nicht das erwartete Ergebnis, da nur die Adressen (=Referenzen) verglichen werden und nicht der Inhalt (Text) der beiden Strings.

Möchte man den Inhalt zweier Strings vergleichen, so muss man deshalb eine entsprechenden Methode, z.B. equals(..) oder equalsIgnoreCase(..) verwenden!

4.1.3 Suchen und ersetzen in Strings

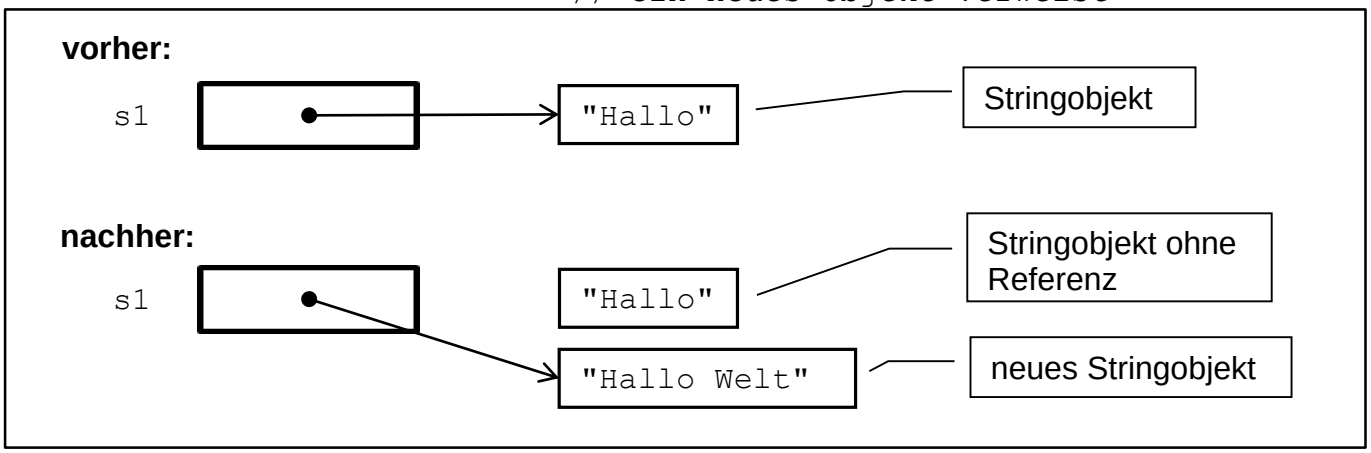
int indexOf(String s)	liefert den Index, an dem s erstmals auftaucht
int indexOf(String s, int fromIndex)	liefert den Index, an dem s erstmals auftaucht, Suche beginnt ab fromIndex
int lastIndexOf(String s)	liefert den Index, an dem s letztmals auftaucht
boolean endsWith(String s)	vergleicht das Ende des aktuellen String-Objekts mit s
boolean regionMatches(int this_offset, String s2, int s2_offset, int len)	vergleicht Regionen in 2 unterschiedlichen Strings
String toLowerCase()	wandelt in Kleinbuchstaben
String toUpperCase()	wandelt in Großbuchstaben
String replace (char alt, char neu)	ersetzt alt durch neu
String replace (String alt, String neu)	ersetzt alt durch neu

4.1.4 Besonderheiten der Klasse String

Objekte der Klasse String sind final und somit konstant, nachdem ihnen ein Wert zugewiesen wurde. Das bedeutet, dass ***eine Manipulation an einem String nicht möglich*** ist!
Dies hat zur Konsequenz, dass alle Methoden, die Strings manipulieren, immer wieder neue String-Objekte erzeugen und nicht etwa das Objekt, auf das die Methode angewendet wurde.

Beispiel:

```
String s1 = "Hallo";
s1.concat("Welt");
System.out.println (s1);      // ergibt "Hallo", da s1 nicht verändert
                              // werden kann
s1 = s1.concat(" Welt");      // alternativ: s1 = s1 + " Welt";
System.out.println (s1);      // ergibt "Hallo Welt", da s1 jetzt auf
                              // ein neues Objekt verweist[2]
```



[2] Für das „alte“ Objekt existiert jetzt keine Referenz mehr. Der Speicher wird bei Gelegenheit vom „Garbage Collector“ wieder freigegeben.

4.2 StringBuffer / StringBuilder

Da – wie erwähnt – die Klasse String final ist und keine Änderungen erlaubt, wurde mit Java 5 die Klasse StringBuilder eingeführt, die eine dynamische Vergrößerung einer Zeichenkette erlaubt. Im Unterschied zu String wird bei StringBuilder das aktuelle Objekt verändert. StringBuilder ist eine performantere und ressourcenschonendere Neuentwicklung der alten Klasse StringBuffer.

4.2.1 Einfügen

StringBuilder append (String s)	hängt s an das aktuelle StringBuilder-Objekt an
StringBuilder insert (int offset, String s)	fügt s an der Stelle offset ein

4.2.2 Löschen

StringBuilder deleteCharAt (int index)	löscht das Zeichen an Position index, der Rest wird nach vorne verschoben
StringBuilder delete (int von, int bis)	löscht die Zeichen ab Position von bis bis, der Rest wird nach vorne verschoben

4.2.3 Ändern

StringBuilder setCharAt (int index, char c)	ersetzt das Zeichen an Position index durch c
StringBuilder replace (int von, int bis, String s)	ersetzt die Zeichen im Bereich von bis bis durch s

4.2.4 Vergleichen

StringBuilder kennt keine Methode zum Vergleichen der Inhalte. Vergleiche gehen nur über die Umwandlung in die Klasse String: equals() oder contentEquals():

```
String        s1  = new String ("Hallo");
StringBuilder sb1 = new StringBuilder ("Hallo");
StringBuilder sb2 = new StringBuilder ("Hallo");

System.out.println(s1.equals(sb1));              // false
System.out.println(s1.equals(sb1.toString()));   // true
System.out.println(s1.contentEquals(sb1));       // true

String s2 = sb1.toString();
String s3 = sb2.toString();
System.out.println (s2.equals(s3));              // true
//oder dasselbe durch direkte Verkettung
System.out.println(sb1.toString().equals(sb2.toString()));  // true
```

4.2.5 Längeninformationen

int length()	liefert die Anzahl der Zeichen im StringBuilder
int capacity ()	liefert die Größe des vom aktuellen StringBuilder-Objekts belegten Puffers

4.3 StringTokenizer

Mit der Klasse StringTokenizer lassen sich Strings an definierbaren Trennzeichen (sog. "Delimiter") in einzelne Teilstrings (sog. "Tokens") trennen.

Aus dem einen String "Hallo Welt" werden die beiden Strings "Hallo" und "Welt".

Default-Trennzeichen sind Leerzeichen ' ', tab '\t', Newline '\n', Carriagereturn '\r' und Formfeed '\f'.

StringTokenizer kennt nur einzelne Zeichen als Delimiter. Soll ein String an einer Zeichenkette getrennt werden so kann die Methode split() aus der Klasse String verwendet werden.

StringTokenizer(String sText)	erzeugt einen StringTokenizer für den übergebenen String mit Default-Trennzeichen
StringTokenizer(String sText, String sDelimiter)	erzeugt einen StringTokenizer für den übergebenen String. sDelimiter ist eine Menge von Einzeltrennzeichen z.B: ";:,".
int countTokens()	liefert die Anzahl der möglichen nextToken-Aufrufe bis zum Ende
boolean hasMoreTokens()	prüft, ob noch weitere Tokens vorhanden sind
String nextToken()	liefert das nächste Token
String nextToken(String sTrenner)	liefert das nächste Token unter Verwendung des Trennzeichens sTrenner

Beispiel:

```
import java.util.*;

public class StringTokenizerDemo
{
  public static void main (String [] arg)
  {
    String sText = "Hallo Welt";
    StringTokenizer st = new StringTokenizer(sText);
    while (st.hasMoreTokens() == true)
    {
      System.out.println(st.nextToken());
    }
  }
}
```

Ausgabe:

```
Hallo
Welt
```

4.4 LocaleDate, LocalTime, LocalDateTime

4.4.1 Prinzipielles zu Datums- und Zeitwerten

In vielen Anwendungen sind Datumsangaben oder Datumsangaben plus Uhrzeit (sogenannte Zeitstempel) von großer Bedeutung. Zum Beispiel wird im Allgemeinen in Dateissytemen der Erstellungs- und Änderungszeitpunkt für jede Datei mit Datum und Uhrzeit protokolliert.

Damit ein Zeitstempel unabhängig vom Standort des Benutzers richtig interpretiert werden kann, wird er nicht in der lokalen Zeit (abhängig von der Zeitzone) gespeichert, sondern normiert als koordinierte Weltzeit (UTC). Das Betriebssystem zeigt einen Zeitstempel im Allgemeinen aber entsprechend der konfigurierten Zeitzone dem Benutzer an.

4.4.2 Datums- und Zeitwerte in Java Programmen

Seit Java 8 gibt es im Package `java.time` entsprechende Interfaces und Klassen zum Arbeiten mit Datums- und Zeitwerten. Diese Klassen lösen ältere Interfaces und Klassen wie `Calendar, GregorianCalendar` ab, die natürlich in älteren Programmen noch in Benutzung sind, aber in neuen Projekten nicht mehr verwendet werden sollten.

Im Package `java.time` gibt es die Klassen `LocalDate`, `LocalTime` und `LocalDateTime`, die zur Verarbeitung von Datums- und Zeitwerten *ohne* Zeitzonenbezug dienen. Soll bei einem Zeitstempel auch die Zeitzone berücksichtigt werden, so sollte man die Klasse `ZonedDateTime` verwenden.

Das Package `java.time` stellt außerdem in sogenannten Aufzählungen (Enumerations) *symbolische Konstanten* für Monatswerte (z.B. `Month.FEBRUARY)`, Wochentage (z.B. `DayOfWeek.MONDAY)` und Zeiteinheiten (z.B. `TimeUnit.HOURS)` zur Verfügung.

4.4.3 Die Klasse `LocalDate`

Die Klasse `LocalDate` stellt eine reine Datumsangabe dar, also eine Kombination aus Jahr, Monat und Tag ohne Zeitinformationen. `LocalDate` Objekte sind wie `String` Objekte nicht veränderbar. Man kann `LocalDate` Objekte auf verschiedene Arten erstellen und gleichzeitig initialisieren:

Erzeugen eines *LocalDate* Objekts:

- Ermitteln des aktuellen (heutigen) Datums mit der Klassenmethode `now()`:

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

- Erstellen eines `LocalDate` Objektes für ein ganz bestimmtes Datum mit der Klassenmethode *`of(…)`*:

  ```
  LocalDate petersGeburtstag = LocalDate.of(2000, Month.DECEMBER, 20);
  ```

  Wie man in dem Beispiel sieht, gibt es für die Monatsangaben spezielle symbolische Konstanten, wie `Month. DECEMBER`. Diese Konstanten sind in einem sogenannten Java Aufzählungstyp (engl. Enumeration) definiert und können in der Java API Dokumentation nachgeschaut werden.

Umwandeln eines LocalDate Objektes in einen Datums-String zur Anzeige für den Benutzer mit Hilfe der Klassenmethode `ofPattern(…)` der Klasse `DateTimeFormatter`

Je nach Anwendung gibt es verschiedene Anforderungen, wie ein Datumswert dargestellt werden soll.

Die Formatierung erfolgt in zwei Schritten:

1. Mit Hilfe der Klassenmethode `ofPattern(…)` der Klasse `DateTimeFormatter` definiert man zunächst ein bestimmtes Format, eine Art Schablone; Bsp.:

   ```
   DateTimeFormatter einFormat = DateTimeFormatter.ofPattern("d.M.yy");
   ```

2. Das in 1. erzeugte Format (die „Schablone“) kann nun verwendet werden, um ein `LocalDate` Objekt in einen formatierten Datumstring umzuwandeln; Bsp.:

   ```
   LocalDate petersGeburtstag = LocalDate.of(2000, Month.DECEMBER, 20);
   String sDatumFormatiert = petersGeburtstag.format(einFormat);

   System.out.println("Peters Geburtstag: " + sDatumFormatiert);
   ```

   Bildschirmausgabe: → **`Peters Geburtstag: 20.12.00`**

Die Klasse `DateTimeFormatter` bietet sehr vielfältige Formatiermöglichkeiten, von denen nachfolgend nur einige gezeigt werden. Die vollständigen Möglichkeiten können in der Java API Doku nachgeschaut werden.

Die Beispiele beziehen sich alle auf das `LocalDate` Objekt `petersGeburtstag`

Paramter von *ofPattern*	Rückgabe von `petersGeburtstag.format(einFormat)`
`"EEEE, dd. MMMM yyyy"`	`Mittwoch, 20. Dezember 2000`
`"EE, dd. MMM. yy"`	`Mi, 20. Dez. 00`
`"dd/MM/yy"`	`20/12/00`
`"dd. MMMM yy"`	`20. Dezember 00`
`"d.M.yy"`	`20.12.00`
`"d/M/yyyy"`	`20/12/2000`
`"yyyy-M-d"`	`2000-12-20`
`"MMMM yyyy"`	`Dezember 2000`
`"d-MMMM-yy"`	`20-Dezember-00`
`"MMM. yyyy"`	`Dez. 2000`

Extrahieren einzelner Datumsbestandteile aus einem `LocalDate` Objekt

Manchmal benötigt man aus einem Datum nur die Jahreszahl oder den Monat oder den Tag. Um aus einem LocalDate Objekt diese Werte zu ermitteln, gibt es entsprechende Objekt-Mehtoden:

Beispiele:

```
LocalDate petersGeburtstag = LocalDate.of(2000, Month.DECEMBER, 20);

int iJahr   = petersGeburtstag.getYear();         // → 2000
int iMonat  = petersGeburtstag.getMonthValue();   // → 12
Month monat = petersGeburtstag.getMonth();        // → Month.DECEMBER
int iLmonat = petersGeburtstag.lengthOfMonth ();  // → 31
int iTagIM  = petersGeburtstag.getDayOfMonth();   // → 20
int iTagIY  = petersGeburtstag.getDayOfYear();    // → 355
int iTagIW  = petersGeburtstag.getDayOfWeek().getValue();  // → 3
```

Vergleichen von `LocalDate` Objekten

- Ähnlich wie bei String Objekten können `LocalDate` Objekte nur mit Hilfe der Objektmethode **`equals`** auf Gleichheit überprüft werden.

- Möchte man zwei `LocalDate` Objekte auf größer oder kleiner vergleichen, verwendet man die Objekt Methode **`compareTo`**

Addieren und Subtrahieren von Zeiteinheiten (Tage, Monate, Jahre)

Man kann zu einem Datumswert in einem LocalDate Objekt Tage, Monate oder Jahre hinzuaddieren bzw. davon abziehen. Dabei werden selbstverständlich alle kalendarischen Regeln beachtet. Die entsprechenden Methoden finden sich in der Java API Doku.

4.4.4 Die Klasse `LocalTime`

`LocalTime` Objekte sind, wie `LocalDate` Objekte unveränderlich. Ihr Inhalt stellt einen Zeitwert bestehen aus Stunden:Minuten:Sekunden dar. Ein Zeitwert kann auf Nanosekunden genau gespeichert werden.
Beispielsweise der Wert "13:45:30,123456789" kann in einem `LocalTime` Objekt gespeichert werden. Man kann `LocalTime` Objekte auf verschiedene Arten erstellen und gleichzeitig initialisieren:

Erzeugen von *LocalTime* Objekten; Bsp:

```
LocalTime jetzt = LocalTime.now();  // aktuelle Uhrzeit

LocalTime zeitpunkt = LocalTime.of(13, 45, 30.123456789);
```

Extrahieren einzelner Zeitbestandteile aus einem `LocalTime` Objekt

Ähnlich wie bei der Klasse `LocalDate` gibt es auch hier Methoden, um den Stunden-, Minuten-, oder Sekundenwert aus einem `LocalTime` Objekt zu ermitteln.

4.5 Random

Die Klassenmethode Math.random() generiert eine Zufallszahl zwischen 0.0 und 1.0.
Die Klasse Random bietet mehr Möglichkeiten. Ein Objekt der Klasse Random liefert nahezu gleichmäßig verteilte Zufallszahlen für bestimmte Datentypen. Die Zahlen werden nach einem bestimmten Algorithmus ermittelt (Linear-Kongruenz-Algorithmus), so dass bei gleichem Ausgangswert immer die gleiche Zahlenreihe ermittelt wird. Random liefert also eigentlich Pseudo-Zufallszahlen[3].

Ausgewählte Methoden:

`Random()`	der parameterlose Konstruktor liefert ein Objekt auf Basis der aktuellen Systemzeit (in Millisekunden)
`Random(long lInit)`	liefert ein Objekt auf Basis des Initialwertes („seed“) `lInit` Das bedeutet, dass man bei gleichem `lInit` immer die selbe Folge von Pseudozufallszahlen bekommt, was für manche Simulationsanwendung wichtig ist.

`boolean nextBoolean()`
`double nextDouble()`
`float nextFloat()`
`long nextLong()`
`int nextInt()`
	liefern nahezu gleichmäßig verteilte Zufallszahlen des entsprechenden Typs

`int nextInt(int n)`	liefert einen int im Bereich 0 bis (n – 1)

Beispiel:

```
import java.util.*;
public class RandomDemo
{
  public static void main(String[] args)
  {
    int iZaehler=0;
    Random rd = new Random();
    for (iZaehler = 0;iZaehler < 10;iZaehler++)
    {
      System.out.printf("%11d - %.12f\n",
                rd.nextInt(), rd.nextDouble());
    }
  }
}
```

Ausgabe:

```
 -808739642 - 0,854149345030
-1366502272 - 0,190829420971
 1381694634 - 0,518337186275
 -408274452 - 0,382676343186
  550247221 - 0,907223614964
  342728756 - 0,393440310295
 1395501180 - 0,158298483025
  853403688 - 0,752040277467
 -127201905 - 0,560644921794
  516776618 - 0,308483110435
```

[3] Für Simulationen ist es häufig für die Vergleichbarkeit von Ergebnissen von Vorteil, wenn man immer die gleiche Folge von „Zufallszahlen“, also Pseudo-Zufallszahlen bekommt. Für andere Anwendungen, wie z.B. Spiele wäre es natürlich nachteilig, wenn sich mit Pseudo-Zufallszahlen immer der gleiche Spielablauf ergeben würde.

5 Die Klasse Formatter

Seit der J2SE 5.0 gibt es die Klasse `Formatter`, in der die Methode `format` zur Verfügung steht.
Diese bietet eine Vielzahl von Formatierungsmöglichkeiten für alle primitiven Datentypen sowie Datums- und Zeitwerten aus den Klassen `Calendar` und `Date`.

Der Konstruktor der Klasse `Formatter` ist mehrfach überladen und kann mit unterschiedlichen Ausgabezielen versorgt werden, z.B. eine Datei. Der parameterlose Konstruktor erzeugt im Objekt standardmäßig ein `StringBuilder` Objekt. Dieses kann man sich durch Aufruf von `out()` geben lassen (siehe später).
Die Objektmethode `format()` erwartet einen Formatstring, für den bestimmte Konventionen festgelegt wurden, und den oder die zu formatierenden Werte[4]:

```
public Formatter format (String formatString, Object ... args);
```

5.1 Formatstring

Der Formatstring muss folgendermaßen aufgebaut sein:

`% [Argument-Index$][Flags][Breite][.Genauigkeit]Umwandlungsform`[5]

- **`Argument-Index$:`**
  Bezug zum Argument (1$ - erstes Argument, 2$ zweites Argument etc.)

- **`Flags`**: weitere Ausgabeoptionen:

`-`	Linksbündige Ausgabe
`+`	Vorzeichen immer ausgegeben
`0`	Zahlen werden mit Nullen aufgefüllt
`,`	Zahlen werden mit Tausenderpunkten ausgegeben
`(`	Negative Zahlen werden in Klammern eingeschlossen

- **`Breite`**: Mindestanzahl auszugebender Stellen
- **`.Genauigkeit:`** bei Fließkommazahlen die Anzahl der Stellen nach dem Komma
- **`Umwandlungsform:`** zu formatierender Datentyp

`b`	Boolescher Wert
`c`	Einzelnes Zeichen
`d`	Ganzzahl in Dezimaldarstellung
`o`	Ganzzahl in Oktaldarstellung
`x`	Ganzzahl in Hexadezimaldarstellung
`X`	Dito, mit großen Buchstaben
`f`	Fließkommazahl

[4] Die uns bereits bekannte Methode `System.out.printf()` arbeitet in genau der gleichen Weise, da `printf()` intern die `Formatter` Klasse verwendet.

[5] Die eckigen Klammern haben hier eine Meta-Bedeutung und tauchen deshalb im konkreten Formatstring nicht auf. Ist etwas in eckigen Klammern eingeschlossen, so bedeutet dies, dass dieser Teil optional ist.

e	Fließkommazahl mit Exponent
E	Dito, mit großem »E«
g	Fließkommazahl in gemischter Schreibweise
G	Dito, ggfs. mit großem »E«
t*X*	Prefix für Datums/Zeitangaben (*X* s.u.)
s	Strings und andere Objekte

5.2 Ausprägungen der Klasse Formatter

Die Instanziierung eines `Formatter`-Objekts und der Aufruf der `format`-Methode ist eine Möglichkeit der formatierten Ausgabe.

Die Methode `format` steht jedoch auch in identischer Form in der Klasse `String` als Klassenmethode und in dem `PrintStream`-Objekt `System.out` zur Verfügung. Dadurch wird die Verwendung noch einfacher. In `System.out` gibt es zusätzlich noch die – funktionsgleiche – Methode `printf`:

```
System.out.printf("Hallo %4d\n",-35);       //Ausgabe am Bildschirm
System.out.format("Hallo %04d\n",35);       //Ausgabe am Bildschirm
String s1 = String.format("Hallo %d",35); //Ausgabe in einen String
```

5.3 Beispiel

```
public static void main(String[] args)
{
     Formatter formatObjekt = new Formatter();
     int iZahl = 42;
     double dZahl = 35.12646;
     String s1 = new String("Hallo Welt");
     StringBuilder sb = new StringBuilder();

     // Ausgabe mit System.out. → priftf und format sind synonym
     System.out.printf("printf       :%8.4f\n", dZahl);
     System.out.format("format       :%8.4f\n", dZahl);
     System.out.printf("printf       : %04d\n", iZahl);
     System.out.format("format       : %04d\n", iZahl);
     // Ausgabe in einen String
     s1 = String.format("String.format: %d", iZahl);
     System.out.println(s1);
     // Ausgabe über das Formatter-Objekt
     // 1. ausführlich: Formatter -> StringBuilder -> String
     formatObjekt.format("Formatter: %d\n", iZahl);
     sb = (StringBuilder) formatObjekt.out();
     s1 = sb.toString();
     System.out.println("StringBuilder -> String: " + s1);
     // 2. kurz mit println: ruft out() und toString() selber auf:
     System.out.println(formatObjekt.format("println: %d\n", iZahl));
}
```

Ausgabe des Programms:

```
printf       : 35,1265
format       : 35,1265
printf       : 0042
format       : 0042
String.format: 42
StringBuilder -> String: Formatter: 42

Formatter: 42
println: 42
```

6 Die Klasse DecimalFormat

Mit der Klasse `DecimalFormat` können Ganz- und Fließkommazahlen formatiert werden. Sie kennt ebenfalls eine Methode `format`, die aber eine Maske als Formatstring erwartet, mit der die jeweilige Zahl formatiert wird:

Formatzeichen für `DecimalFormat`:

Symbol	Bedeutung
`0`	Eine einzelne Ziffer
`#`	Eine einzelne Ziffer. Wird ausgelassen, falls führende Null.
`.`	Dezimaltrennzeichen
`,`	Tausendertrennzeichen
`E`	Aktiviert Exponentialdarstellung
`%`	Ausgabe als Prozentwert

Beispiel:

```
public static void main(String[] args)
{
     double dZahl = 77.2846439408679;
     DecimalFormat df1 = new DecimalFormat("000.00000");
     DecimalFormat df2 = new DecimalFormat("###.000");

     System.out.println(df1.format(dZahl));
     System.out.println(df2.format(dZahl));
}
```

Ausgabe:

```
077,28464        → mit führenden Nullen und gerundet
77,285           → ohne führenden Nullen und gerundet
```

7 Wrapper-Klassen

Zu jedem einfachen Datentyp (z.B. int) existiert in Java eine sogenannte Wrapper-Klasse (z.B. Integer), mit der man den Wert einer einfachen Variablen in ein unveränderliches Objekt verpacken kann. Auf dieses Objekt wird wie immer mit Hilfe einer Verweisvariablen zugegriffen.

Im package java.lang existieren folgende Wrapperklassen :

Void, Integer, Float, Boolean, Byte, Long, Double, Character, Short

Zweck der Wrapper-Klassen:

Die Wrapper-Klassen haben im Wesentlichen den folgenden Zweck:

- Objektmethoden können nur auf Objekte angewendet werden. Die Wrapper-Klassen sind notwendig, um hilfreiche Methoden auch für einfache Datentypen bereitstellen zu können. Dazu zählen z.B. Methoden zur Umwandlung von einfachen Datentypen in Strings.
- Komplexere dynamische Datenstrukturen (Collections und Generics) können nur im Zusammenhang mit Objekten verwendet werden. Möchte man Werte einfacher Datentypen darin speichern, so muss man diese mit Hilfe der entsprechenden Wrapper-Klasse erst in ein Objekt verpacken.

Einige ausgewählte Methoden am Beispiel der Klasse Integer:

Methode	**Bedeutung**

`Integer (int iWert)`
`Integer (String sWert)`
 	
`veraltet: wird nicht mehr benötigt wegen „Autoboxing“ (s.u.)`
	Der Konstruktor erzeugt ein Integer-Objekt mit dem übergebenen Wert. Lässt sich der String nicht umwandeln so wird eine NumberFormatException erzeugt. (siehe Merkblatt "Exceptions/ Fehlerbehandlung"; folgt später)

`int intValue()`	liefert den gespeicherten int-Wert zurück
`static int parseInt (String sWert)`	Versucht einen String in einen int zu konvertieren. Falls dies nicht gelingt wird eine NumberFormatException erzeugt.
`static String toString(int iWert)`	Konvertiert den übergebenen Wert in einen String
`String toString ()`	liefert den gespeicherten Wert als String zurück
`static Integer valueOf (int iWert)`	Erzeugt aus dem übergebenen iWert ein Integer-Objekt
`static Integer valueOf (String sWert)`	Erzeugt aus dem übergebenen String ein Integer-Objekt. Falls dies nicht gelingt wird eine NumberFormatException erzeugt.

Die anderen Wrapper-Klassen arbeiten nach dem gleichen Prinzip. Siehe API-Dokumentation.

Beispiel:

```
public class IntegerDemo
{
  public static void main(String[] args)
  {
    String sStrasse = new String ("Danzigerstr. ");
    int iNr = 6;
    Integer iINr = new Integer(iNr);
    sStrasse = sStrasse.concat(iINr.toString());
    System.out.println(sStrasse);
    System.out.println("Im Integer-Objekt ist der Wert "
                       +iINr.intValue() + " gespeichert.");
  }
}
```

Ausgabe:
Danzigerstr. 6
Im Integer-Objekt ist der Wert 6 gespeichert.

Autoboxing

Da die Konvertierung zwischen einfachem Datentyp und Wrapper-Klasse relativ aufwändig ist wurde in Java 5 das sog. Autoboxing eingeführt, eine kürzere Schreibweise für die Umwandlung. Damit können einfache Datentypen direkt der entsprechenden Wrapper-Klasse zugewiesen werden und umgekehrt.

Die Konvertierung eines einfachen Datentyps in ein Wrapper-Objekt heißt ***Boxing***.
Die Konvertierung eines Wrapper-Objekts in einen einfachen Datentyp heißt ***Unboxing***.

Beispiel:

	Ausführlich	**Vereinfacht mit Autoboxing**
Boxing	int i1 = 10; Integer iI1 = new Integer (i1);	int i1 = 10; Integer iI1 = i1;
Unboxing	int i2; i2 = iI1.intValue();	int i2 = iI1;