Geschichte

- 1992 bei Firma Sun entworfen.
- Ursprünglich für Consumer-Geräte gedacht.
- Ab 1995 in Netscape-Browser integriert und als Internet-Sprache beworben.
- Seit 2008 Hauptsprache für Android-Apps.
- Java wird als Sprache für Business-Anwendungen positioniert.
- Oracle kauft Sun 2010.
- 2010 verklagt Oracle Google wegen Java-Nutzung.
- 2018 beginnt Oracle, Lizenzgebühren für die Java-Runtime zu fordern.

Pro und Contra

Vorteile

- voll objektorientiert
- automatische Speicherverwaltung
- seit Version 8 auch funktionale Elemente
- weit verbreitet im Business-Umfeld

Nachteile

- viel Boilerplate-Code zu schreiben
- langsam
- schwierige Lizenzsituation
- Kompatibilitätsprobleme von Version zu Version

Auf dem absteigenden Ast?

- Java Applets (Java im Browser) gibt es nicht mehr
- Java Webstart (Desktopprogramme einfach installieren und aktualisieren) gibt es nicht mehr
- JavaFx (für moderne Benutzeroberflächen) ist gescheitert
- Lizenzgebühren führen zu Verunsicherung
- Google empfiehlt inzwischen Kotlin statt Java für die Android-Programmierung

. . . aber:

- Java ist seit 2001 immer unter den ersten drei Plätzen im TIOBE-Index.
- Programme für die öffentliche Verwaltung müssen oft in Java geschrieben werden.
- Es gibt umfangreiche Bibliotheken für viele Gebiete.
- Bei Geschäftsanwendungen im Serverbereich ist Java stark vertreten.

Java-Code nutzen

Bei Java wird der Quellcode in Code für die JVM (*Java Virtual Machine*) übersetzt (1).
Die JVM interpretiert dann diesen Zwischencode (2).

1. `javac hallo.java`
2. `java hallo.class`

Vorteile

- Für eine neue Plattform muss nur die JVM angepasst werden.
- Es gibt auch andere Sprachen für die Java-JVM.

Nachteile

- Die Java-VM muss immer mit geladen werden.
- Versionsunterschiede machen Probleme.

Kompatibilität

Compile-Time

Programm	*Compiler*	*läuft?*
alt	neu	ja
neu	alt	nein

Runtime

Programm	*JVM*	*läuft?*
alt	neu	ja
neu	alt	nein

Programm	*Library*	*läuft?*
alt	neu	meist
neu	alt	nein

Speicherverwaltung

- Lokale Variablen mit primitiven Datentypen werden auf dem Stack angelegt.
- Arrays, komplexe Datentypen, Objekte, Instanzvariablen und statische Variablen landen auf dem Heap.
- Die Java-Runtime kümmert sich um das Löschen nicht mehr benötigter Variablen und Objekte: Von Zeit zu Zeit räumt die *Garbage Collection* den Heap auf.

Java schreiben

Eigentlich kann man einen Texteditor verwenden, üblich ist aber eine IDE (*Integrated Develpoment Environment*).
Beispiele:

- Eclipse
- Netbeans
- IntelliJ IDEA
- Android Studio

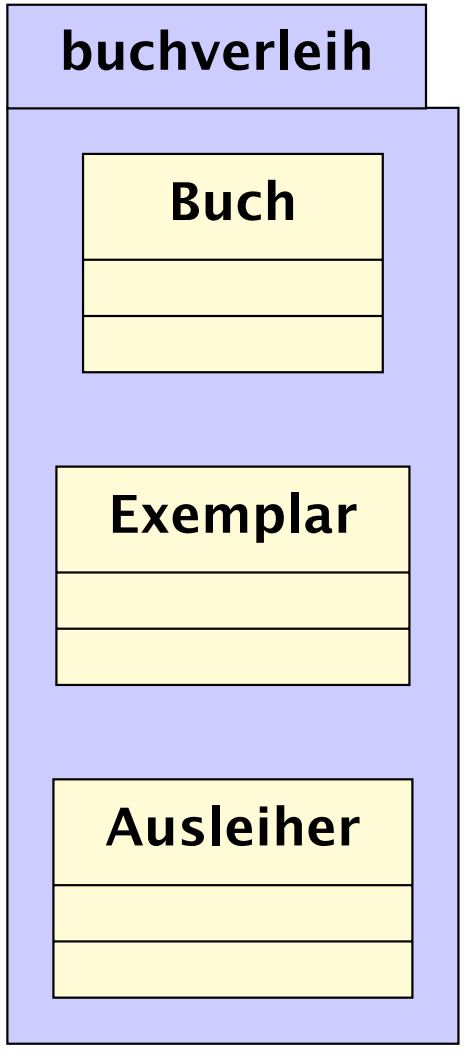
Aus der IDE kann ein Programm auf Knopfdruck übersetzt, gestartet und debuggt werden.

Dateien

- Java-Quellcode bekommt für jede Klasse eine einzelne Datei (*.java).
- Code für die JVM liegt pro Klasse in einer Datei (*.class).
- Mehrere Klassen können in ein JAR (*Java Archive*) gepackt und auch aus dem Archiv gestartet werden (*.jar).
- Für die Ausführung werden benötigt:
  - JVM für das Ziel-Betriebssystem,
  - eigene Klassen,
  - zusätzlich eingebundene Bibliotheken.
- Ein Installer-Generator (z.B. IzPack) erleichtert die Weitergabe.

Packages

- Klassen leben meist in einem Package.
- Ein Package gibt den enthaltenen Klassen einen weltweit eindeutigen Namensraum.
- Packagenamen bildet man aus dem wortweise umgedrehten Domainnamen: `www.les-pf.de` ergibt `package de.lespf;`
- Für weniger Tipparbeit kann man fremde Packages importieren: `import javax.swing.*;`
- Die Datei `package-info.java` unterstützt bei der Dokumentation.



Module

- Module sind relativ neu (ab Java 9).
- Sie bieten eine Package-übergreifende Gliederung.
- Die Moduldefinition beschreibt Abhängigkeiten zu anderen Modulen, definiert Sichtbarkeiten und mehr.

Wir definieren im Unterricht keine Module.

Jeder Vergleich hinkt...

Adresse	*Java*
Ort	Modul
Straße	Package
Haus	Klasse
Zimmer	Variable
Bewohner	Methode

Programme

- Jede Java-Klasse besteht aus einer Datei.
- Ein Java-Programm besteht aus mindestens einer Klasse.
- Klassen können einzeln oder in einem jar-Archiv vorliegen.
- Um als Programm zu laufen, braucht eine Klasse eine `main()`-Methode.

Kommentare

- Kommentare mit `//` am Anfang laufen bis zum Zeilenende.
- Kommentare mit `// TODO` am Anfang kann man sich in einer Taskliste (To-Do-Liste) anzeigen lassen.
- Kommentare mit `/*` am Anfang laufen bis zu `*/`.
- Es ist guter Stil, Zeilen im Kommentar mit einem `*` anzufangen.
- Kommentare mit `/**` am Anfang sind Javadoc-Kommentare.
  Sie werden speziell behandelt (Tooltips, Dokumentation).
- Nutzen Sie es, wenn Ihnen die IDE die Erstellung von Javadoc-Kommentaren anbietet.

Annotationen

- Annotationen stehen sprachlich zwischen Kommentaren und Schlüsselwörtern.
- Sie werden zum Teil vom Java-Übersetzer ausgewertet, zum Teil von Java-Software.
- Annotationen stehen meist am Anfang einer Zeile, sie beginnen mit einem @-Zeichen und werden groß geschrieben.

Beispiel

- Wenn eine Methode nicht mehr verwendet werden soll, enthält sie die Annotation `@Deprecated`.
- So kann Software umgeschrieben werden, bevor es die Methode nicht mehr gibt.

Datentypen

Primitive Datentypen

Zahlen	int, float, double
Sonstiges	boolean, char
Container	Array

Objekte

Es gibt sehr viele vorgefertigte Klassen, zudem können Sie selbst Klassen erstellen. Eine kleine Auswahl:

Wrapper-Klassen	verpacken primitive Typen: Integer, Double
Text	String, StringBuilder

Primitive Datentypen

int	Ganze Zahl, 32 Bit
double	Fließkommazahl, ca. 16 Stellen Genauigkeit
boolean	Wahrheitswert: true oder false
char	ein einzelnes Zeichen, z.B. 'Ä'

Variablen

- Variablen sind *Behälter* für Werte.
- In Java sind Variablen typisiert: Ein Text passt nicht in eine Integer-Variable.
- Variablen müssen vor der ersten Benutzung deklariert werden.
- Variablen leben immer innerhalb von Klassen. In Java gibt es keine globalen Variablen.
- Primitive Daten werden direkt in Variablen abgelegt.
- Primitive Variablen werden mit 0 initialisiert, wenn Sie nichts anderes angeben.

Objektvariablen

- Objektvariablen enthalten nur einen Zugriffsschlüssel auf das Objekt, nicht das Objekt selbst.
- Objektvariablen werden mit `null` initialisiert.
- Sie *müssen* einer Objektvariablen ein bereits erzeugtes Objekt zuweisen, bevor Sie sie nutzen können.

Falsch

```
Kunde k;
k.setKundennname("Fred");
// erzeugt Exception
```

Richtig

```
Kunde k;
k = new Kunde();
k.setKundennname("Fred");
```

Lebensdauer

- Variablen in Objekten leben so lange wie die umgebenden Objekte.
- Variablen in Methoden existieren nur während eines Methodenaufrufes.
- statische Variablen leben so lange wie das Programm läuft.

Sichtbarkeit

public	Allgemein
private	Nur innerhalb der Klasse
protected	Innerhalb der Klasse und innerhalb von Unterklassen.

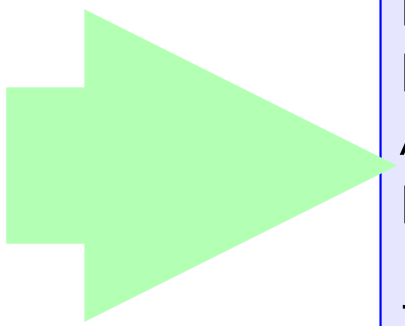
Für den Zugriff auf nicht öffentliche Variablen schreibt man Getter- und Setter-Methoden.

```
private String kundenname;
String getKundenname() {
  return kundenname;
}
void setKundenname(String neuername) {
  kundenname = neuername;
  return;
}
```

Wozu Getter?

- Getter und Setter können die Schnittstelle erhalten, auch wenn sich die Umsetzung ändert.

```
private int netto;
private int steuer;
private int brutto;
public int getBrutto() {
   return brutto;
}
```



```
private int netto;
private int steuer;
// brutto wird berechnet
public int getBrutto() {
   return netto + steuer;
}
```

Wozu Setter?

- Getter und Setter können die Schnittstelle erhalten, auch wenn sich die Umsetzung ändert.
- Setter können Werte prüfen, bevor sie übernommen werden.

```
public int rabatt;
// in Prozent,
//von 0 bis 100
...
// anderswo, 2*falsch:
rabatt = -10;
rabatt = 2000;
```



```
private int rabatt;
public boolean setRabatt(int neu) {
   if((neu < 0) || (neu > 100)) {
      return false;
   }
   rabatt = neu;
   return true;
}
```

Lokale Variablen

- Variablen innerhalb von Methoden (lokale Variablen) sind niemals nach außen sichtbar.
- Bei gleichem Namen werden lokale Variablen bevorzugt angesprochen.
- Namensgleichheiten lassen sich mit der Punktschreibweise ausräumen.

```
private int rabatt;
public void setRabatt(int rabatt) {
  /* this macht klar: die Klassenvariable
   * wird gesetzt */
  this.rabatt = rabatt;
}
```

static und final

- statische Variablen in Methoden behalten ihren Wert zwischen Methodenaufrufen.
- statische Variablen in Klassen gibt es pro Klasse nur ein Mal – alle Objekte der Klasse nutzen sie gemeinsam.
- statische Methoden in Klassen kann man direkt über die Klasse aufrufen, auch ohne dass es ein Objekt gibt.
- Finale Variablen können nach der ersten Zuweisung nicht mehr geändert werden.

Arrays

- Ein Array ist ein Container für Variablen eines bestimmten Datentyps.
- Die Größe ist festgelegt.
- Die Reihenfolge der Variablen ist festgelegt.
- Mittendrin Einfügen, Löschen, Sortieren ist nur mit Aufwand möglich.
- Java bietet viele Collection-Klassen an, die flexibler sind.

Arrays definieren

```
int[] besondereZahlen = {6, 23, 28, 42};
1   2  3                  4  5
```

1. Datentyp und weitere Schlüsselwörter wie üblich.
2. Nach dem Datentyp zwei eckige Klammern, dazwischen die Größe des Array.
3. Dann der Name des Arrays.
4. Primitive Werte kann man gleich zuweisen.
5. Wenn die Werte direkt zugewiesen werden, kann Java die Größe selbst ermitteln.

Arrays definieren

- Wenn Sie nicht alle Inhalte angeben, müssen Sie die Größe festlegen.
  ```
  static double[12] koordinaten;
  ```
- Sie können Deklaration und Definition eines Arrays trennen:
  ```
  int[] kundennummern;
  anzahlKunden = 10;
  kundennummern = new int[anzahlKunden];
  ```
- Objektarrays müssen Objekte erhalten, eine Arraydefinition erzeugt kein Objekt.
  ```
  Kunde[] kunden = new Kunde[10];
  kunde[0] = new Kunde();
  ```

Array-Elemente ansprechen

```
System.out.print(besondereZahlen[2]);
System.out.print(kunden[3].getKundenname());
```

- Man nennt den Namen des Arrays, dann in eckigen Klammern die Nummer des Elementes, um ein Element anzusprechen.
- Array-Elemente werden ab 0 durchnummeriert.
  Die höchste Nummer ist 1 kleiner als die Größe des Arrays.
- Der Zugriff auf Elemente außerhalb des Arrays führt zum Absturz.
- Auf Objekte kann man erst zugreifen, wenn man dem Element ein Objekt zugewiesen hat.
- Die Eigenschaft `.length` liefert die Größe des Array, sagt aber nichts darüber, ob alle Elemente belegt sind.

Array-Elemente ansprechen

```
int[] besondereZahlen = {6, 23,  28, 42};
//                        ↑   ↑    ↑   ↑
// Element Nr.            0   1    2   3
```

```
System.out.print(besondereZahlen[0]); // druckt 6
System.out.print(besondereZahlen[3]); // -> 42
```

Methoden

- Methoden gibt es in Java nur innerhalb von Klassen.
- Methoden können genau *einen* Wert zurückliefern.
- Methoden gleichen Namens sind möglich, wenn sich die *Signatur* unterscheidet. (*overloading*)
- Unterklassen können Methoden der Oberklasse überschreiben. (*overriding*)
- Für Methoden gelten die selben Sichtbarkeitsregeln wie für Variable.
- Statische Methoden können ohne ein Objekt benutzt werden.

Methoden-Signatur

```
public static void main(String[] args)
1      2      3    4    5         6
```

1. Öffentlich sichtbare Methode
2. Ohne Klasse nutzbar
3. Liefert keinen Rückgabewert
4. main-Methode ist Einstiegspunkt in's Programm
5. Erwartet ein Array von Strings
6. Dieses Array kann als `args` angesprochen werden.

Methoden-Gerüst

- Der Funktionsbody steht in geschweiften Klammern. Rücken Sie ein!
- Der Funktionsaufruf endet mit `return`, gefolgt vom Rückgabewert.
- Bei `void`-Funktionen (keine Wertrückgabe) kann das `return` entfallen.

```
public String getAnrede(boolean vertraulich) {
  if(vertraulich == true) {
    return "Du";
  }
  return "Sie";
  // alles ab hier wird ignoriert.
}
```

Standard-Methoden

- Die Java-Bibliothek bietet sehr viele Methoden.
- Open Source-Projekte erweitern die Möglichkeiten zusätzlich.

Abstrakte Methoden

- Eine Abstrakte Methode besteht nur aus einer Methodensignatur mit dem zusätzlichen Schlüsselwort `abstract`.
- Eine abstrakte Methode bildet ein Gerüst, das vom Programmierer mit Leben gefüllt werden muss.
- Abstrakte Methoden standardisieren das Verhalten von Klassen.
- Abstrakte Methoden sind ein wichtiges Element von Interfaces.
- Enthält eine Klasse eine abstrakte Methode, wird sie ebenfalls abstrakt.

printf

- `printf` gibt Text mit Hilfe eines Format-Strings formatiert aus.
- `printf` gehört zu `java.io.PrintStream` und ist daher breit verwendbar.
- Der Format-String enthält Format-Codes. Sie beginnen mit einem Prozent-Zeichen (%).
- Spezielle Codes regeln die genaue Formatierung.
- Die Liste der Codes ist lang.
- Alternative: `DecimalFormat` mit Excel-artigen Formatangaben.

printf

```
double preis = 12,077; String waehrung="Euro";
System.out.printf(

    "Preis: %7.2f %s%n",  preis, waehrung);
1   2        3     4  5    6
```

1. Wir schreiben in die Standard-Ausgabe.
2. Im Format-String kann normaler Text enthalten sein.
3. `%7.2f` bedeutet: Kommazahl mit insgesamt sieben Stellen, davon zwei nach dem Komma.
4. `%s` bedeutet: Ein String.
5. `%n` fängt eine neue Zeile an.
6. Die einzusetzenden Werte folgen nach dem Format.

Ausgabe: `Preis:␣␣␣12,21␣Euro`

printf

Format-Codes	
%d	Ganze Zahl
%f	Kommazahl
%s	String (Text)
%%	Prozentzeichen
%n	neue Zeile

Breitenangaben	
%-12	links ausgerichtet, zwölfstellig
%05	rechtsbündig, führende Nullen, fünf Stellen

- Dies ist ein kleiner Ausschnitt aus den Codes. Eine vollständige Liste finden Sie in der Java-Hilfe zu `Formatter`.
- Man kann auch Argumente mehrmals oder in anderer Reihenfolge verwenden.
  `%3$s` druckt das dritte Argument als String.
- Längenangaben werden überschritten, falls es nötig ist.

Klassen

- Jede Klasse liegt in einer Datei, die den Klassennamen trägt.
- Klassen können unterschiedliche Sichtbarkeitsstufen haben: `public`, package
- Klassen können auch innerhalb anderer Klassen existieren. Sie sind dann privat.
- Klassen enthalten *Eigenschaften* (Variablen, Properties) und *Methoden*.
- Eigenschaften und Methoden können unterschiedliche Sichtbarkeitsstufen haben: `public`, `protected` oder `private`.

Klassendefinition

```
class Kunde {
  public final int kundennummer;
  // Der Konstruktor erzeugt Objekte:
  public Kunde(int nummer) {
    kundennummer = nummer;
    ladeAusDatenbank();
  }
  private boolean ladeAusDatenbank() {
    ...
    return true;
  }
}
```

Innere Klassen

- In Java können Klassen innerhalb von anderen Klassen definiert werden.
- Diese inneren Klassen haben Zugriff auf alles in der äußeren Klasse – auch private Attribute und Methoden.
- Innere Klassen werden für Ereignishandler in der GUI-Programmierung benötigt.
- Mit etwas Glück lassen sie sich durch funktionale Programmierung ersetzen.

Klassen und Objekte

Klassen sind Bauvorlagen für Objekte.

Klasse

- Klassen sind abstrakt. Sie stehen für eine Gruppe von Dingen, nicht für ein Ding.
- Nur statische Eigenschaften und Methoden sind nutzbar.

Objekt

- Objekte sind konkret. Ein Objekt steht für ein bestimmtes Ding.
- Alle Eigenschaften und Methoden sind nutzbar.

Vererbung

- Eine Unterklasse enthält *alle* Attribute und Methoden der Oberklasse.
- Zusätzlich kann sie weitere Attribute und Methoden anbieten.
- Eine Unterklasse kann Methoden der Oberklasse durch eigene Methoden ersetzen (overriding).
- Wo im Programm ein Objekt der Oberklasse erwartet wird, kann auch eines der Unterklasse eingesetzt werden (Liskovsches Substitutionsprinzip).
- Mit Vererbung bildet man *ist-ein*-Beziehungen ab, keine *hat-ein*-Beziehungen.

```
public class Azubi extends Mitarbeiter ...
```

Mehrfachvererbung

- Mehrfachvererbung (mehrere Oberklassen) führt zu konzeptionellen Schwierigkeiten.
- Deswegen ist sie in Java nicht möglich.
- Interfaces sind ein sicherer Ersatz für Mehrfachvererbung.

Methoden überschreiben

- Wenn Sie eine Methode der Oberklasse überschreiben (ersetzen), gehört eine Annotation zum guten Ton.
- Hierfür schreiben Sie `@Override` in die Zeile direkt oberhalb der Methodendefinition.
- Als Bonus werden Sie gewarnt, falls Sie nicht wirklich überschreiben (weil die Signatur nicht übereinstimmt).
- Finale Methoden können nicht überschrieben werden.
- Wenn Sie die Oberklassenmethode doch brauchen, können Sie sie mit `super.Methodenname()` aufrufen.

Methoden überschreiben

```
class Azubi extends Mitarbeiter { // 1
  @Override
  int berechneWochenstunden() {
    if(kalender.isSchulferien()) { // 2
       return super.berechneWochenstunden();
    } else { // 3
       // Schulzeit berücksichtigen
       ...
    }
```

1. Azubi ist eine Unterklasse von Mitarbeiter
2. In den Ferien wird die Arbeitszeit wie bei anderen Mitarbeitern berechnet.
3. Schulzeit zählt zur Arbeitszeit.

Konstruktoren und Vererbung

- Der Konstruktor der Unterklasse ruft automatisch als erstes den Konstruktor der Oberklasse auf.
- Bei Vererbung über mehrere Stufen ergibt das eine Aufrufkaskade.
- Möchten Sie dem Oberklassen-Konstruktor Parameter übergeben, dann müssen Sie ihn explizit aufrufen – gleich am Anfang Ihres Konstruktors.

Der Unterklassenkonstruktor ersetzt den Oberklassenkonstruktor nicht – er ergänzt ihn.

private und final

- Unterklassen können genauso wenig auf private Attribute und Methoden der Oberklasse zugreifen wie externe Klassen.
- Man kann Elemente der Oberklasse als `protected` statt `private` deklarieren, damit sie auch in Unterklassen erreichbar sind.
- Von als `final` getaggten Klassen können keine Unterklassen abgeleitet werden.
  Beispiel: `String` ist final.

Typecast

- Mit `instanceof` kann zur Laufzeit abgefragt werden, ob ein Objekt zu einer bestimmten Klasse gehört.
- Mit einem *Cast* kann ein Objekt seiner eigentlichen Klasse zugeordnet werden, wenn es als Objekt einer Oberklasse daherkommt.

```
class Azubi extends Mitarbeiter...
ArrayList<Mitarbeiter> liste = new ArrayList();
liste.add(new Azubi(...));
// geht, weil ein Azubi auch Mitarbeiter ist
Mitarbeiter m = liste.get(0);
if(m instanceof Azubi) {
  Azubi a = (Azubi) m; // Cast
...
```

Abstrakte Klasse

- Aus einer abstrakten Klasse können keine Objekte erzeugt werden. Hierfür müssen Unterklassen geschaffen werden.
- Klassen werden abstrakt, wenn sie mit dem Schlüsselwort `abstract` deklariert sind oder wenn sie mindestens eine als `abstract` deklarierte Methode haben.
- Beispiel: *Bankkonto*.
  - Ein allgemeines Bankkonto gibt es nicht: Ein Bankkonto ist ein Girokonto, ein Sparkonto, ein Anlagekonto....
  - Allen Bankkonten gemeinsam sind ein Kontoinhaber und eine Kontonummer.

```
public abstract class Bankkonto ...
```

Parametrisierte Klasse

- Eine *parametrisierte* oder *generische* Klasse wird durch eine Typangabe konkret. Beispiel:
  - Eine Liste enthält im Idealfall Objekte *eines* Typs.
  - Anstatt IntListe, StringListe, BooleanListe usw. wird eine generische Liste `Liste<Element>` erstellt.

```
ArrayList<String> namensliste =
          new ArrayList<String>();
/* in neueren Java-Versionen kann das zweite
 * »String« entfallen: ... = new ArrayList<>();*/
```

Interface

- Ein Interface ist eine abstrakte Klasse mit weiteren Einschränkungen.
  - Attribute sind immer `public` und `final`, also Konstanten.
  - Ein *klassisches* Interface enthält nur abstrakte Methoden.
  - Seit Java 8 kann ein Interface auch statische Methoden haben oder Default-Implementierungen der Methoden vorgeben.
- Marker-Interfaces haben keine Methoden.

Interface

- Interfaces werden für Querschnittsfunktionen (z.B. Sortierbarkeit) verwendet.
- Mit Interfaces wird das Problem der Mehrfachvererbung umgangen.
- Wenn Sie ein Interface *implementieren*, erfüllen Sie Ihren Teil eines Vertrages.
- Im Gegenzug können Sie Features nutzen, die das Interface benötigen.
- Ob eine Klasse ein Interface implementiert, kann man mit `instanceof` abfragen.

Comparable

- Wenn Java sortieren soll, müssen die zu sortierenden Objekte das Interface `Comparable` implementieren.
- `Comparable` enthält nur eine Methode: `compareTo()`.
- `compareTo()` erhält ein Objekt der selben Klasse und vergleicht es mit sich selbst.
- Eine Integer-Zahl wird zurückgeliefert:
  - kommt vor dem anderen (kleiner): eine negative Zahl
  - kommt nach dem andern(größer): eine positive Zahl
  - gleich: 0
- Viele Standard-Klassen (z.B. `String`) implementieren Comparable.

```
class QuartettSpielkarte implements Comparable
...
```

Comparable implementieren I

Beispiel: Mitarbeiter sollen nach Namen sortiert werden können.

```
class Mitarbeiter implements Comparable { // 1
  @Override // 2
  public int compareTo(Mitarbeiter kollege) {
    String meinName = getNachname(); // 3
    String seinName = kollege.getNachname()); // 4
    return meinName.compareTo(seinName); // 5
  }
```

1. Die Klasse implementiert das Interface Comparable.
2. Wir überschreiben die abstrakte Funktion compareTo.
3. Wir benötigen unseren Nachnamen
4. und den des Kollegen.
5. Wir liefern den Stringvergleich zurück.

Comparable implementieren II

```
@Override
public int compareTo(Mitarbeiter kollege) {
  int ergebnis = getNachname.compareTo( // 1
     kollege.getNachname()); // 2
  if(ergebnis != 0) // 3
    return ergebnis;
  else ...// 4
}
```

1. Wir speichern das Ergebnis in `ergebnis`.
2. Da wir die beiden Strings nur für den Vergleich benötigen, brauchen wir sie nicht Variablen zuzuweisen.
3. Wenn die Nachnamen verschieden sind, liefern wir das Vergleichsergebnis zurück.
4. Wenn sie gleich sind, vergleichen wir die Vornamen (folgt).

Comparable implementieren

Beispiel: Mitarbeiter sollen sortiert werden können, zuerst nach Nachnamen, bei Gleichheit nach dem Vornamen.

```
class Mitarbeiter implements Comparable {
  @Override
  public int compareTo(Mitarbeiter kollege) {
    int ergebnis = getNachname().compareTo(
          kollege.getNachname());
    if(ergebnis != 0)
      return ergebnis;
    else
      return getVorname().compareTo(
          kollege.getVorname());
  }
```

Enum – Überblick

- Ein `enum` ist eine vollständige Aufzählung von möglichen Werten.
- Enums (kurz für *Enumeration*, Aufzählung) machen Code lesbarer und verhindern Fehler.
- Java-Enums können weit mehr als die in vielen anderen Sprachen, wir beschränken uns hier auf den minimalen Standard.

Enum – Verwendung

- Enum-Werte werden komplett groß geschrieben, da sie Konstanten sind.
- Wenn ein Enum definiert wird, wird gleichzeitig ein Datentyp definiert.
- Beim Speichern sollten Enums in eine feste Zahl konvertiert werden.

Enum – Beispiel

Mit enum

```
// Enum definieren
public enum MwstSatz {
  MWST_VOLL,
  MWST_ERMAESSIGT,
  MWST_FREI
}
// bei einem Artikel
MwstSatz mwstsatz =
MWST_VOLL;
// Steuerberechnung
switch(mwstsatz) {
case MWST_VOLL:
  return preis * 119 /19;
...
```

Ohne enum

```
// Steuersätze:
//1 ist voll,
//2 ist ermäßigt,
//3 ist frei
// hoffentlich wird das
// überall beachtet
...
// bei einem Artikel
int mwstsatz = 1;
// Steuerberechnung
switch(mwstsatz) {
case 1:
  return preis * 119 /19;
...
```

Klassen nutzen

Normalerweise spricht man Klassen nicht direkt an – man nutzt Objekte. Ausnahmen:

- Statische Methoden werden über die Klasse aufgerufen, sie benötigen kein Objekt. Beispiel: Methoden von `java.Math`
  ```
  groesser = Math.max(x, 5.2);
  ```
- Sonderfall: *Factory-Methoden* erzeugen Objekte oder liefern bereits vorhandene Objekt-Instanzen zurück.
  ```
  Calendar kalender = Calendar.getInstance();
  ```
- Methoden der Oberklasse werden mit `super` aufgerufen, wenn wir sie überschrieben haben. Sie sind jedoch nicht objektlos, denn das aufrufende Objekt wird verwendet.
  ```
  return super.getPersonennummer();
  ```

Objekte erzeugen

- Klassen enthalten einen *Konstruktor*. Seine Aufgabe ist es, das Objekt einzurichten.
- Der Konstruktor trägt in Java den Namen der Klasse.
- Er ist eine Methode ohne Rückgabewert.
- Der Konstruktor wird aufgerufen, wenn man ein Objekt der Klasse mit new erzeugt.
- Mehrere Konstruktoren mit unterschiedlicher Signatur sind möglich (overloading).

Objekte erzeugen

```
private Kunde kunde = new Kunde(123);
1       2     3     4     5     6
```

1. Eine private Variable wird definiert.
2. Sie hat den Datentyp Kunde
   (sie zeigt auf ein Objekt der Klasse Kunde).
3. Der Variablenname ist kunde.
4. Es wird ein Objekt erzeugt (new) und der Variablen zugewiesen (=).
5. Der Konstruktor der Klasse Kunde heißt Kunde().
6. Der Konstruktor erhält das Argument 123.

Bootstrapping

- `main` ist eine statische Methode.
- `main` muss *ein* Objekt erzeugen und dort eine Methode aufrufen.
- Dieses Objekt kann der selben Klasse angehören.

```
public class Kasse {
private void starteGUI()
...
public static void main(int[] args)
{
  Kasse kasse = new Kasse();
  kasse.starteGUI();
...
```

Objekte verbinden

- Beziehungen gehen normalerweise nur in *eine* Richtung.
- In Java gibt es keinen Unterschied zwischen einer Aggregation und einer Assoziation.
- Beziehungen zwischen Objekten werden über Objektvariablen realisiert.
- Bei Eins-zu-viele-Beziehungen werden Collections, z.B. `ArrayList` eingesetzt.
- Teile einer Komposition werden nur von *einer* Variablen referenziert.

Object

- `Object` ist in Java die Oberklasse aller Klassen.
- Sie enthält drei Methoden, die Sie in Ihren Klassen oft überschreiben werden:
  - `equals()`,
  - `hashCode()` und
  - `toString()`
- Wenn Sie `toString()` für Ihre Klasse überschreiben, erhalten Sie zum Beispiel mit `System.put.print()` eine brauchbare Ausgabe.

Strings

- `String` ist in Java kein primitiver Datentyp, sondern ein Objekt.
- Strings genießen im Vergleich zu anderen Objekten sprachliche Erleichterungen.
- Der Datentyp `String` eignet sich nicht für umfangreiche Manipulationen. Verwenden Sie dafür einen `StringBuilder`.

Einstieg

definieren	`String s1 = "Hallo";`
	`String s2 = new String ("Leute");`
verbinden	`String s3 = s1 + ", " + s2;`
Länge finden	`int laenge = s3.length(); // 12`
Ausschneiden	`String s4 = s1.substring(0, 3); // Hal`
Großschreibung	`String s5 = s1.toUpperCase(); // HALLO`

Strings vergleichen

```
String s1 = "Hallo", s2 = "HALLO", s3 = "Leute";
```

- Test auf Gleichheit:
  ```
  boolean passt = s1.equals(s2); // false
  ```
- Test auf Gleichheit, Großschreibung egal:
  ```
  passt = s1.equalsIgnoreCase(s2); // true
  ```
- Sortierreihenfolge ermitteln:
  ```
  int nachher = s1.compareTo(s3); // < 0
  ```
- Anfang vergleichen:
  ```
  passt = s1.startsWith("Ha"); // true
  ```

Konvertierungen

- Jedes Objekt hat eine `toString()`-Funktion.
- Für primitive Datentypen muss der Objekt-Wrapper herhalten:
  ```
  String drei = Integer.toString(3); // "3"
  ```
- Strings kann man mit `parse...`-Funktionen der Wrapper in primitive Typen wandeln:
  ```
  String drei = "3";
  int zahl = Integer.parseInt(drei);
  ```

StringBuilder

- `StringBuilder` bietet Zusammensetzen, Ausschneiden, Ersetzen von Text.
- `StringBuilder` ist kein `String`. Typumwandlung mit `toString()`.

```
StringBuilder sb = new StringBuilder("Hallo");
sb.append("Leute"); // HalloLeute
sb.insert(5, ", "); // Hallo, Leute
sq.replace(1, 5, "i"); // Hi, Leute
sb.delete(1, 5); // Heute
System.out.println(sb.toString());
```

System

- System enthält zwei PrintStreams, die Standardausgabe und den Standard-Fehlerkanal:
  `System.out` und `System.err`
- Die Standardeingabe ist ein InputStream:
  `System.in`
- `System.exit()` beendet das Programm.
- Mit `System.setProperty()` und `System.getProperty()` können Sie Einstellungen Ihres Programmes verwalten.
- `System.arraycopy()` kopiert ein Array.

System.arraycopy()

- Das Ziel-Array muss existieren und groß genug sein.
- Die Datentypen beider Arrays müssen kompatibel sein.
- Es wird eine *shallow copy* gemacht.
- Falls das Ursprungsarray nicht mehr benötigt wird, sollten alle Einträge auf `null` gesetzt werden.

```
Getraenk[] drinks1 = new Getraenk[10];
drinks1[0] = new Getraenk(1);
Getraenk[] drinks2 = new Getraenk[10];
System.arraycopy(drinks1, 0, drinks2, 0, 1);
drinks1[0].setArtikelname("Fred");
assert(drinks1[0].getArtikelname().equals(
    drinks2[0].getArtikelname()));
```

Zwei Arten Streams

- Seit Anfang kennt Java Streams zum Lesen und Schreiben von Daten.
- Mit Java 8 kam Stream-Verarbeitung als Werkzeugkasten Listen dazu.
- Leider heißen beide Konzepte gleich, obwohl sie total verschieden sind.
- In diesem Abschnitt geht es um I/O-Streams.

Gestapelte Ströme

- *Connection streams* arbeiten direkt auf Ressourcen, zum Beispiel einer Datei. In ihnen fließen Low-Level-Daten (Bytes).
  Sie können zum Beispiel mit einer Datei, einem String oder einer Netzwerkverbindung verknüpft sein.
- *Chained streams* setzen auf *connection streams* auf und bringen die Daten auf eine höhere Ebene.
  Sie können zum Beispiel gezippte Daten ein- oder auspacken, Integer- und String-Werte lesen oder schreiben.

Typische Streams

- Die Klasse `System` bringt drei geöffnete Streams mit (siehe dort).
- Aus einer Datei lesen:
```
FileInputStream rein =
        new FileInputStream("daten.csv");
```
- Gepufferte Streams sind meistens besser:
```
BufferedInputStream in =
        new BufferedInputStream(rein);
```
- Streams müssen nach Gebrauch geschlossen werden:
```
rein.close();
```

PrintStream

- PrintStream bietet einem Ausgabestream bequeme Druckmöglichkeiten.
- `PrintStream(`*`Datei`*`)` öffnet einen Stream in eine Datei.
- `print()` gibt einen Wert aus.
- `println()` gibt einen Wert und ein Zeilenende aus.
- `printf()` gibt formatierte Werte aus.
- `flush()` gibt den Ausgabepuffer aus.
- `close()` schließt den Stream.

java.util.Scanner

- Scanner zerlegt einen Stream z.B. in Text und Zahlen.
- Das Locale wird berücksichtigt.
- Trennzeichen ist normalerweise Leerraum.
- Mustererkennung ist möglich.

- `close()` beendet die Bearbeitung.
- `hasNext()` prüft, ob es noch etwas gibt.
- `hasNextInt()` prüft, ob eine ganze Zahl folgt.
- `next()` holt das nächste String-Token.
- `nextInt()` liefert die nächste int-Zahl.

Scanner-Beispiel

```
import java.util.Scanner; // 1 fügt Eclipse ein.
Scanner scanner = new Scanner(System.in); // 2
System.out.print("Eingabe: ");
while (scanner.hasNext()) { // 3
  String s = scanner.next(); // 4
  if(s.equalsIgnoreCase("ende")) {
    break; // 5
  }
  System.out.println(s);
  System.out.print("Eingabe: ");
}
scanner.close(); // 6
```

Scanner-Beispiel erklärt

1. `import java.util.Scanner;`
   Das Package muss importiert werden.
2. `Scanner scanner = new Scanner(System.in);`
   Der Scanner liest aus der Standardeingabe `System.in`.
3. `while (scanner.hasNext()) {`
   Solange die Eingabedatei geöffnet ist, wird gelesen.
4. `String s = scanner.next();`
   Die Eingabe wird in durch Leerraum getrennte Texte zerlegt.
5. `if(s.equalsIgnoreCase("ende")) break;`
   Schleifenende, wenn man »Ende« eintippt.
6. `scanner.close();`
   Den Scanner schließen, wenn Sie fertig sind.

java.io.File und java.nio.file

- Für neue Projekte werden die nio-Klassen empfohlen, wir bleiben bei `File`.
- `File` bietet Zugriff auf Datei- und Pfadnamen und Attribute.
- Auch Löschen und Umbenennen wird angeboten.
- Streams können auf Dateinamen oder auf Files geöffnet werden.
- JFileChooser erwartet ein File, keinen Dateinamen.

```
File vorgabe = new File("c:\autoexec.bat");
if(vorgabe.exists()) ...
```

File API

exists	prüft auf Vorhandensein
isFile	prüft, ob Datei
isDirectory	prüft, ob Verzeichnis
delete	löscht die Datei
renameTo	ändert den Namen
getCanonicalPath	liefert den offiziellen Pfad zur Datei
getParent	liefert den Namen der nächsthöheren Ebene
list	Liefert die Dateinamen in einem Verzeichnis

Sammelklassen

- Sammelklassen (Container, Collections) verwalten Sammlungen von Objekten.
- Arrays wurden vorab besprochen, weil sie keine Objekte sind.
- Es gibt grob drei Arten von Collections:

  List Elemente liegen in einer festen Reihenfolge vor, Duplikate sind möglich.

  Set Duplikate sind nicht erlaubt. Sets gibt es mit und ohne automatische Sortierung.

  Map Eine Map enthält Paare, zu einem Schlüssel gehört genau ein Wert. Maps gibt es mit und ohne automatische Sortierung.

- Von jeder Art gibt es mehrere Ausführungen.

Sammelklassen benutzen

- Sammelklassen nehmen nur Objekte auf, keine primitiven Typen (int, boolean...)
- Primitive Typen können Sie in Wrapper-Klassen einpacken (Integer, Boolean...). In neueren Java-Versionen geschieht das automatisch (Autoboxing).
- Wo es auf eine Reihenfolge ankommt, müssen Elemente sortierbar sein. Dafür müssen Ihre Klassen `Comparable` implementieren.
- Sammelklassen sind parametrisierbar. So werden nur Objekte des richtigen Typs aufgenommen.
- Wenn Sie nicht parametrisieren möchten, können Sie mit `instanceof` und Casts arbeiten.

Inhalte tauglich machen

Wenn Sie kompliziertere Objekte als Strings oder Integer-Objekte in Sammelklassen verwalten möchten, müssen Sie in den meisten Fällen

- die von `Object` geerbte Methode `equals()` überschreiben,
- die von `Object` geerbte Methode `hashCode()` überschreiben,
- das Interface `Comparable` implementieren.

`equals()` und `hashCode()` sind wichtig, um Duplikate zu erkennen. `Comparable` wird zum Sortieren benötigt.

Gemeinsame Methoden

Sammelklassen verfügen über gemeinsame Methoden.
Beispiele:

isEmpty()	Prüfen, ob leer.
size()	Anzahl der Elemente ermitteln.
remove()	Ein Objekt entfernen.
clear()	alle Elemente entfernen.

Sammlungen durchlaufen

- Sammelklassen, die das Interface `Iterable` implementieren, können Sie mit einer modernen `for`-Schleife bequem durchsuchen.
- Bei Maps können Sie wahlweise durchlaufbare Sammlungen
  - der Schlüssel (`keySet()`),
  - der Werte (`values()`),
  - von Schlüssel-Wert-Paaren (`entrySet()`)

  erhalten.

```
ArrayList<String> namensliste ...
for(String name : namensliste) {
  if name.equals("Fred") ...
}
```

Arten von Listen

- Schneller Zugriff auf ein Element: `ArrayList`
- Schnelles Einfügen mittendrin: `LinkedList`
- Schnelles Anhängen/Entfernen am Ende: `Queue`, `Deque`, `ArrayList`
- Schnelles Einfügen am Anfang: `Deque`
- Schnelles Entfernen am Anfang: `Queue`, `Deque`

Das sind nicht alle Listen-Arten, die Java anbietet. Wir besprechen nur `ArrayList`.

ArrayList

Nachteile von Arrays

- Feste Größe.
- Einfügen oder Entfernen von Elementen ist mühsam.
- Auch für viele andere Operationen müssen Schleifen geschrieben werden.

Abhilfe

- `Arrays` unterstützt Array-Bearbeitung (z.B. sortieren, suchen, kopieren).
- `ArrayList` ist ein verbesserter Ersatz für `array`.

ArrayList

- `ArrayList` ist eine *parametrisierbare* Klasse. Bei der Definition wird ein Datentyp in spitzen Klammern angegeben.
- `ArrayList` beseitigt nicht nur die offensichtlichen Schwächen von `array`, es wird weit mehr geboten.
- Der Zugriff auf Elemente läuft über Funktionsaufrufe, nicht über eckige Klammern.
- `ArrayList` gehört zum Package `java.util`.

Methoden (Auswahl)

- `add(Element e)` fügt ein Element hinten an.
- `add(int platz, Element e)` fügt ein Element zwischendrin ein.
- `set(int platz, Element e)` ersetzt ein Element.
- `get(int platz)` liefert ein Element zurück.
- `remove(int platz)` entfernt ein Element.
- `clear()` entfernt alle Elemente.
- `size()` liefert die Anzahl der Elemente.
- `toArray()` liefert ein Array zurück.

ArrayList-Beispiel

```
import java.util; // fügt Eclipse ein.
ArrayList<String> namen = new ArrayList<>();
namen.add("Fred");
namen.add("Otto"); // Fred Otto
namen.add(1, "Erik"); // Fred Erik Otto
System.out.println(namen.get(1)); // Erik
for(String name: namen) {
  System.out.println(name); // Fred Erik Otto
}
namen.remove(1); // Fred Otto
namen.clear();
```

Set und Map

- Ein *Set* ist eine Sammlung von unterschiedlichen *Objekten*.
- Eine *Map* ist eine Sammlung von unterschiedlichen *Schlüsseln*. Zu jedem Schlüssel gehört ein *Wert*.
- `Set` und `Map` sind abstrakte Klassen. Es gibt Unterklassen für verschiedene Zwecke. Die wichtigsten:
  - sehr schneller Zugriff (`HashSet`, `HashMap`)
  - Einfügereihenfolge bleibt erhalten (`LinkedHashSet`, `LinkedHashMap`)
  - automatisch sortiert (`TreeSet`, `TreeMap`)

Set

- Ein Set nimmt jeden Wert garantiert nur einmal an.
- Prüfen auf einen bestimmten Wert geht sehr schnell.
- Wichtige Methoden:

add()	Objekt einfügen, sofern noch nicht drin
contains()	Prüfen, ob ein Objekt enthalten ist.
remove()	Ein Objekt entfernen.

Set verwenden

```
HashSet<String> namensSet = new HashSet<>();
namensSet.add("Fred");
namensSet.add("Fred"); // folgenlos
if(!namensSet.contains("Yasemin")) {
  // kommt nicht vor...
}
namensSet.remove("Fred");
```

Map

- Eine Map speichert Schlüssel und Werte.
- Ein Set nimmt jeden Schlüssel garantiert nur einmal an, Werte können mehrmals vorkommen.
- Prüfen auf einen bestimmten Schlüssel geht sehr schnell.
- Wichtige Methoden:

  `put()` Ein Schlüssel/Wert-Paar einfügen oder updaten.

  `containsKey()` Prüfen, ob ein Schlüssel enthalten ist.

  `get()` Den Wert zu einem Schlüssel erhalten.

  `remove()` Ein Schlüssel/Wert-Paar entfernen.

- Properties sind eine wichtige Anwendung von `Map`.

Map verwenden

```
HashMap<String,Integer> altersMap =
    new HashMap<>();
altersMap.put("Fred", Integer.valueOf(18));
altersMap.put("Fred", Integer.valueOf(19));
if(altersMap.containsKey("Fred")) {
  Integer alter = altersMap.get("Fred");
  // ergibt 19.
}
altersMap.remove("Fred");
```

Die Klasse Collections

`Collections` ist eine Hilfsklasse, die Sammelklassen bearbeitet. Wichtige Methoden:

`sort()`	sortiert eine Liste,
`min()`	liefert das kleinste Element,
`max()`	liefert das größte Element,
`reverse()`	dreht die Reihenfolge um,
`addAll()`	fügt zwei Listen zu einer zusammen.

Für viele Methoden müssen die Objekt in der Liste das Interface `Comparable` implementieren.

Entscheidungen und Verzweigungen

- Entweder-oder: `if`
  - Verkettetes if
  - Verschachteltes if
  - Kurzform `? :`
- Auswahl aus einer Liste: `switch`

Einfaches if

Einfach

```
if(Bedingung ist erfüllt) {
    // Code ...
}
```

Mit Sonst-Teil

```
if(Bedingung ist erfüllt) {
    // Code ...
}else {
    // anderer Code ...
}
```

Wenn nur eine Code-Zeile folgt, können die geschweiften Klammern entfallen.

Verkettetes if

```
if(Bedingung ist erfüllt) {
    // Code
}else if(nächste Bedingung {
    // nächster Code
...
}else {
    // »Sonst«-Code. }
```

- Beliebige viele `else if`-Terme können folgen.
- Der `else`-Teil am Schluss ist optional.

Verschachteltes if

```
if(Bedingung ist erfüllt) {
    // vielleicht Code
    if(Bedingung 2) {
        // vielleicht mehr Code
    }
    // vielleicht mehr Code
}else {
    // Code
}
```

- Logische Verknüpfungen in der Bedingung (`||` und `&&`) können verschachteltes `if` oftmals ersetzen.
- Der `else`-Teil ist wie immer optional.

Inline-if

- `if` ist ein Blockbefehl, das heißt, Sie können es nicht innerhalb eines Ausdrucks verwenden.
- Für die Verwendung innerhalb von Ausdrücken gibt es das `? :`-Konstrukt.
  Bedingung ? *dann* : *sonst*
- Vor dem Fragezeichen steht ein Vergleich. Trifft er zu, wird der Wert vor dem Doppelpunkt verwendet, sonst der dahinter.

```
int anzahl = 2; // oder eine andere Zahl.
System.out.print("Ich habe ");
System.out.print(anzahl == 0? "nichts": "etwas");
System.out.println(" gefunden.");
```

Listenauswahl mit switch

- `switch()` wählt eine aus mehreren Alternativen aus.
- Man muss die Werte nicht sortieren, es ist aber besser, das zu tun.
- Die Alternativen sind int-, enum- oder String-Werte.
- Ein zu testender String darf nicht `null` sein.
- Mehrere Werte können sich den selben Code teilen.
- Wenn `break` weggelassen wird, wird der folgende Code ebenfalls berücksichtigt.
- Mit `default` fängt man alles ab, was nicht passt.

Numerische Auswahl

```
int i = 9;
switch(i) {
case 1:
  ergebnis = "Januar";
  break;
case 2:
  ergebnis = "Februar";
  break;
case 3:
case 4:
case 5:
  ergebnis = "Frühling";
  break;
default:
  ergebnis = "sonstwann";
  break;
}
```

- Für i gleich 1 oder 2 gibt es individuelle Ergebnisse.
- Für i von 3 bis 5 wird »Frühling« verwendet.
- Für alle anderen Zahlen gibt es das Ergebnis »sonstwann«.
- `default` nicht vergessen.

Textauswahl mit switch

```
... anredekurz = "f";
if(anredekurz == null)
  return; // 1
switch(anredekurz) {
case "f":
case "F": // 2
case "w":
  anredelang = "Frau";
  break;
case "m":
case "M": // 2
  anredelang = "Herrn";
  break;
default: // 3
  anredelang = "";
  break;
}
```

1. Vor dem switch auf *null* prüfen.
2. Groß- und Kleinschreibung beachten.
   Alternative: Text vor dem switch mit `toLowercase()` normalisieren.
3. `default` nicht vergessen.

Moderner Switch

- Seit Java 14 gibt es eine zusätzliche neue `switch`-Syntax.
- Mehrere Werte können mit Kommas getrennt aufgezählt werden.
- Es gibt kein Fallthrough.
- Ein Switch kann einen Wert zurückliefern.
- Im Unterricht bleiben wir beim klassischen Switch.

```
... anredekurz = "f";
if(anredekurz == null)
  return; // 1
anredelang = switch(anredekurz) {
case "f", "F"  -> "Frau";
case "m", "M"-> "Herrn";
default -> "";
}
```

Schleifen

- Klassische Zählschleife:
  `for(`*`Start ;Schleifenbedingung ; Inkrement`*`)`
- Zählschleife über Collection:
  `for(`*`Variable : Collection`*`)`
- Allgemeine Schleife:
  `while(`*`Schleifenbedingung`*`)`

Klassische for-Schleife

```
for(int i = 0; i < 10; i++)  {
1   2   3       4         5  6
```

1. Wir zeigen hier den typischen Fall.
2. Die Zählvariable kann im Schleifenkopf deklariert werden, dann ist sie nur innerhalb der Schleife sichtbar.
3. Sie erhält den Startwert 0.
4. Die Schleife läuft, solange die Zählvariable kleiner als 10 ist.
5. In jedem Schleifendurchgang wird die Zählvariable um 1 erhöht.
6. Es folgt ein Block mit dem Schleifenkörper.

Klassische for-Schleife

- Die klassische for-Schleife gibt es in vielen Programmiersprachen.
- Sie ist sehr flexibel.
  - Sie können die Schleife mit `break;` vorzeitig verlassen.
  - Sie können mit `continue;` zum nächsten Schleifendurchlauf springen.
  - Sie können die Schleifenvariable innerhalb der Schleife ändern.
  - Sie können die Methode, in der die Schleife liegt, selbstverständlich auch mit `return` beenden.
  - Alle drei Teile des Schleifenkopf sind optional.

Klassische und moderne Schleife

```
final int[] besondereZahlen = {6, 23, 28, 42};
int summe = 0;
```

Klassisch

```
for(int i = 0; i < besondereZahlen.length; i++) {
    System.out.println(besondereZahlen[i]);
    summe += besondereZahlen[i];
}
```

Modern

```
for(int dieZahl : besondereZahlen) {
    System.out.println(dieZahl);
    summe += dieZahl;
}
```

Moderne for-Schleife

```
final int[] besondereZahlen = {6, 23, 28, 42};
int summe = 0;

for(var wert  : besondereZahlen) {
1    2    3      4 5
    summe += wert;
}
```

1. Das Schlüsselwort `for` gilt für beide Arten von Schleifen.
2. Java kann den Datentyp hier selbst ermitteln.
3. Die Variable hat den selben Datentyp wie die Liste, die sie durchläuft.
4. Dann ein Doppelpunkt
5. und der Name der Liste.

Moderne for-Schleife

Vorteile

- Die Formulierung ist knapp und eindeutig.
- Startwert, Inkrement, Bedingung sind automatisch richtig.
- Direkter Zugriff auf den Wert, nicht über Index.
- Funktioniert auch mit Listen, die keinen Index haben.

Nachteile

- Der Index ist nicht verfügbar.
- Sie können nur über *eine* Collection iterieren.
- Die Syntax unterschiedet sich von Sprache zu Sprache.

while-Schleife

- Mit `while()` bilden Sie eine allgemeine Schleife.
- Im Schleifenkopf geben Sie nur die Schleifenbedingung an.
- Startwerte setzen Sie vor der Schleife.
- Falls es etwas zu inkrementieren gibt, machen Sie es innerhalb der Schleife.
- Auch in `while()`-Schleifen können Sie `continue` oder `break` machen.

```
while(nochEinDatensatz() == true) {
  if(bearbeiteNaechstenDatensatz() == false)
    break;
}
```

Ausnahmen

- Java wirft bei Fehlern *Ausnahmen* (Exceptions).
- Mit einem `try`-`catch`-Konstrukt können sie behandelt werden.
- *Checked Exceptions* müssen Sie behandeln.
- *Checked Exceptions* dürfen Sie riskieren.

Exceptions definieren

- Es gibt eine Menge vordefinierter Exception-Klassen.
- Für spezielle Zwecke können Sie eine eigene Klasse ableiten:

```
class KundeNichtDaException
  extends Exception { }
```

Exceptions werfen

- Klassen, die *Checked Exceptions* werfen können, sind mit `throws` gekennzeichnet.
```
class Kunde throws KundeNichtDaException {
```
- Im Fehlerfall kann man dann eine Exception werfen:
```
if(ladeKundeAusDB() == false)
  throw new KundeNichtDaException();
```
- Sie können einer Exception einen Fehlertext mitgeben:
```
KundeNichtDaException e = new
    KundeNichtDaException(
        "Kundennummer ist falsch.");
```
- Der Methodenaufruf endet an dieser Stelle und die aufrufende Methode muss den Fehler behandeln.

Exceptions weiterreichen

- Wenn Sie einen Fehler an dieser Stelle nicht behandeln möchten, können Sie ihn weiterreichen.
- Hierzu deklarieren Sie ebenfalls `throws`. Sie brauchen in Ihrer Klasse kein zusätzliches `throws` anbringen.
- Spätestens in `main()` müssen Sie ihn dann doch berücksichtigen.

Exceptions abfangen

- Gefährlicher Code wird in einen `try`-Block eingeschlossen.
- Ein `catch`-Block übernimmt die Fehlerbehandlung.
- Sie können für verschiedene Fehlerklassen mehrere `catch`-Blöcke einsetzen.

```
try {
    Kunde kunde = new Kunde(123);
} catch (KundeNichtDaException e) {
    System.out.println("Kundennummer ungültig.");
    return;
}
```

Aufräumarbeiten

- Ein `finally`-Block wird auf alle Fälle ausgeführt, egal ob ein Fehler geworfen wurde oder nicht.
- Er wird auch ausgeführt, wenn der `try`-Block mit `return` verlassen wird.

```
try {
    oeffneDatei();
    liesEineZeile();
    return;
} catch (DateiException e) {
    System.out.println("Lesefehler");
} finally {
    schliesseDatei();
}
```

Try with Resources

- Wenn eine Klasse das Interface `AutoCloseable` implementiert, gibt es seit Java 7 eine Alternative zu `finally`.
- Das Öffnen der Datei geschieht in runden Klammern direkt nach dem Wort `try`.
- Java schließt die Ressource, wenn das `try-catch`-Konstrukt verlassen wird.

Daten speichern

- nativ mit ObjectOutputStream
- als Text/CSV
- als XML mit XMLEncoder
- als JSON mit Jackson
- in eine relationale Datenbank mit dem Persistence API

Hinweis: Die folgenden Beispiele verwenden »try with resources«. Dadurch werden Streams nach Gebrauch automatisch geschlossen.

Native Serialisierung

Vorteile

- Fest in Java eingebaut
- Einfach anzuwenden

Nachteile

- Daten sind nicht mit Fremdsoftware lesbar
- Konvertierung bei Softwareänderung ist aufwändig

Native Serialisierung I: Objekte

```
// class Kunde implements Serializable {
  private String nachname;
  private transient int loginZeit;
  ...
}
```

- Alle zu speichernden Klassen müssen `Serializable` implementieren. Dazu muss kein Code geschrieben werden.
- Attribute, die nicht gespeichert werden sollen, werden als `transient` getaggt.

Native Serialisierung II: Sichern

```
Kunde kunde = new Kunde();
try (
  FileOutputStream fos =
        new FileOutputStream("/tmp/test.ser");
  ObjectOutputStream oos =
        new ObjectOutputStream(fos);
) {
  oos.writeObject(kunde);
}catch(IOException e) {
  System.out.println("Gescheitert");
}
```

Native Serialisierung III: Laden

```
try (
  FileInputStream fis =
        new FileInputStream("/tmp/test.ser");
  ObjectInputStream ois =
        new ObjectInputStream(fis);
){
  Object ob = ois.readObject();
  Kunde kunde = (Kunde) ob;
}catch(IOException e) {
  System.out.println("Gescheitert");
}
```

- Beim Laden wird der Konstruktor nicht aufgerufen.
- Datentyp wird von *Object* zum eigentlichen Typ gecastet.

Native Serialisierung IV: Versionierung

- Sie können die automatisch vergebene serialVersionUID mit dem Kommandozeilentool servialver auslesen:
  `serialver de.lespf.bkwi.Kunde`
- Setzen Sie in der zu sichernden Klasse diese serialVersionUID:
  `static final long serialVersionUID = 1234567L;`
- Wenn Sie die Klasse inkompatibel ändern, ändern Sie auch die serialVersionUID.
- Damit alte Dateien dann noch gelesen werden können, schreiben Sie ein Konvertierungsprogramm.

Serialisierung als XML

Vorteile

- XMLEncoder ist Bestandteil von Java.
- Bearbeiten durch Fremdsoftware ist möglich.

Nachteile

- Alle Objekte müssen JavaBeans sein.

- Serializable muss nicht implementiert sein.
- Für alle privaten Attribute braucht es öffentliche Getter und Setter.
- Es muss einen argumentlosen öffentlichen Konstruktor geben.

Serialisierung als XML I: Sichern

```
try (
  FileOutputStream fos =
        new FileOutputStream("/tmp/test.xml");
  XMLEncoder enc =
        new XMLEncoder(fos);
) {
  enc.writeObject(kunde);
}catch(Exception e) {
  System.out.println("Gescheitert");
}
```

Serialisierung als XML II: Laden

```
try (
  FileInputStream fis =
        new FileInputStream("/tmp/test.xml");
  XMLDecoder dec =
        new XMLDecoder(fis);
) {
  Object ob = dec.readObject();
  Kunde kunde = (Kunde) ob;
}catch(Exception e) {
  System.out.println("Gescheitert");
}
```

- Damit alte Dateien nach inkompatiblen Änderungen noch gelesen werden können, schreiben Sie ein Konvertierungsprogramm.
- Das können Sie auch mit einem XML-Prozessor statt in Java machen.

Serialisierung als Text/CSV

Vorteile

- Bearbeitung durch Fremdsoftware ist möglich.

Nachteile

- Keine eingebaute Unterstützung.
- Schreiben und Lesen von CSV ist aufwändig.

Serialisierung als JSON

Vorteile

- JSON ist leichtgewichtig und verbreitet.
- Die Bibliothek Jackson integriert JSON-Unterstützung in Java.

Nachteile

- Es gibt keine offizielle Standard-JSON-Bibliothek für Java.
- JSON-Datenqualität ist schwierig sicherzustellen.

Objektrelationale Abbildung

- Objekte (Java) und Tabellen (Datenbank) müssen aneinander angepasst werden.
- Das Jakarta Persistence API (JPA) ist die Standard-Programmierschnittstelle hierfür.
- Wichtigste Implementation ist Hibernate.

Wegen des großen Umfangs und der Wichtigkeit des Themas gibt es eine gesonderte Präsentation zu Java und Datenbanken.

Properties

- Properties sind ein *Key-Value-Store*. Zu einem Schlüssel kann ein Wert abgefragt oder gesetzt werden.
- Schlüssel und Wert müssen Strings sein. Für andere Zwecke können Sie eine `HashMap` verwenden.
- Properties eignen sich zum Beispiel für:
  - Systemeinstellungen
  - Einstellungen eines Programmes
  - Übergabe von vielen Argumenten an eine Methode

Properties setzen und auslesen

Erstellen

```
Properties prop = new Properties();
```

Abfragen Abfrage mit Standardwert (»blue«)

```
String meineFarbe =
prop.getProperty("Farbe", "blue");
```

Setzen

```
prop.setProperty("Farbe", meineFarbe);
```

Properties-Dateien

- Properties-Dateien sind einfache Textdateien.
- Am Anfang jeder Zeile steht ein Schlüssel, dann folgt ein Gleichheitszeichen, anschließend der Wert.
- Schlüssel und Werte müssen als Text vorliegen, andere Formate sind nicht möglich.
- Properties-Dateien eignen sich nicht für umfangreiche Daten.

Properties-Dateien nutzen

Zum Öffnen/Speichern wird ein Stream übergeben, kein Dateiname.

Einfache Textdatei

```
   Öffnen prop.load(eingabestream);
Speichern prop.store(ausgabestream, "Kommentar");
```

XML

```
   Öffnen prop.loadFromXML(eingabestream);
Speichern prop.storeToXML(ausgabestream,
          "Kommentar");
```

Spezialitäten

- Properties können in einer Schleife durchstöbert werden:
```
Set<String> keys = prop.stringPropertyNames();
for(String key : keys) {
    System.out.println(key);
}
```
- Man kann zusätzlich Default-Properties einbinden, die nicht verändert oder gespeichert werden:
```
Properties prop = new Properties(defaultProp);
```
- Properties können als Liste »ausgedruckt« werden:
```
prop.list(System.out);
```

System-Umgebung auslesen

- Die Klasse `System` hält Properties mit System-Eigenschaften.
- System-Properties können auch auf der Java-Kommandozeile mit `-D` gesetzt werden.
- Auslesen über `System.getProperty("Eigenschaft");`
- Ergänzend können Umgebungsvariablen mit `System.getenv()` ausgelesen werden.

Beispiel

```
String benutzer =
    System.getProperty("user.name");
String betriebssystem =
    System.getProperty("os.name");
```

Einstellungen speichern

- Nahezu jedes Programm benötigt Zugriff auf Einstellungen, die ein Programmende überstehen sollen.
- Windows bietet die Registry an, andere Betriebssysteme haben vergleichbare Angebote.
- Properties-Dateien sind der Java-Weg; betriebssystem-unabhängig, ohne spezielle Werkzeuge zu bearbeiten.
- Programmeinstellungen sollten nur *einmal* vorliegen.
- Damit man sie nicht an viele Methoden übergeben muss, kann man sie als *Singleton* implementieren.

Singleton-Beispiel

```
class Einstellungen {
  private static Einstellungen einstell;
  private void Einstellungen() {...}
  public Einstellungen getInstance() {
    if(einstell == null) {
      einstell = new Einstellungen();
    }
    return einstell;
  }
}
```

Java-Umgebungen

- Ein Computer benötigt eine *Java Virtual Machine* (JVM), um Java-Programme auszuführen, dazu noch Java-Bibliotheken.
- Ein *Java Development Kit* (JDK) enthält neben der Laufzeitumgebung auch vieles, was man zum Entwickeln von Java-Programmen benötigt.
- Ein *Java Runtime Environment* (JRE) enthält nur, was zum Ausführen von Java-Programmen nötig ist.

Versionierung

- Offizielle Java-Releases erscheinen seit Java 11 im Halbjahresrhythmus.
- Alle paar Jahre erscheint eine *Long-Term-Support-Version* (LTS).
- Java 17 ist die aktuelle LTS-Version.
- Es ist sinnvoll, Programme auszuliefern, die mit einer LTS-Version funktionieren.
- Bei Oracle gibt es gegen Bezahlung längeren Support für ältere Java-Runtimes.

Anbieter

- Oracle Java enthält lizenzpflichtige Teile. Softwareentwicklung ist kostenlos möglich, für die Runtime können beträchtliche Gebühren anfallen.
- Es gibt mehrere Initiativen, die kostenlos nutzbare JDK-Distributionen anbieten, zum Teil auch als LTS-Version. Beispiele:
  - Adoptium
  - Amazon Corretto
  - SAP Machine

Java-Programme ausführen

- Zuerst müssen die `.java`-Dateien mit `javac` in `.class`-Dateien kompiliert werden.
- Eine `.class`-Datei kann durch die JVM ausgeführt werden; damit alle weiteren benötigten Klassen gefunden werden, ist oft Fummelei nötig.
- Meist packt man die benötigten Klassen zusammen in ein jar-Archiv.
- Ein jar-Archiv kann durch die JVM ausgeführt werden.
- Eclipse bietet einen Export als ausführbares jar an.

Jar-Dateien

- Jar-Dateien enthalten Java-Klassen, Ressourcen und eine Manifest-Datei.
- Jar-Dateien sind technisch gesehen Zip-Dateien.
- Das JDK-Werkzeug zur Jar-Verwaltung nutzt nicht Zip-Befehle, sondern tar-Befehle.

Java-Programme verteilen

- Die JRE muss installiert werden.
- Java-Programme müssen nicht installiert werden. Geben Sie einfach eine Jar-Datei weiter.
- Wenn Voraussetzungen geschaffen werden müssen (z.B. Datenbankclient konfigurieren, Eintrag im Startmenü), benutzen Sie einen Installer wie IZPack.