Objektorientierte Programmierung

Kapitel 5: Datentypen

Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2014/15

http://www.informatik.uni-halle.de/~brass/oop14/

Inhalt

Datentypen: Allgemeines (1)

- Ein Datentyp (kurz "Typ") bestimmt:
  - einen Name für den Datentyp, z.B. `int`,
  - eine Menge von möglichen Werten des Datentyps, z.B. die Menge der ganzen Zahlen:
    $$\mathbb{Z} = \{\ldots, -2, -1, 0, +1, +2, \ldots\}.$$
    Tatsächlich ist der Wertebereich für `int` beschränkt, siehe unten.
  - ggf. eine Möglichkeit, diese Werte im Programm mit Konstanten ("Datentyp-Literalen") aufzuschreiben, z.B. als Folge von Dezimalziffern, wie etwa `123`,
  - Operationen auf den Werten, z.B. die vier Grundrechenarten `+`, `-`, `*`, `/`.
    Die Division zweier `int`-Werte liefert in Java wieder einen `int`-Wert, und zwar wird in Richtung auf 0 gerundet, z.B. $14/3 = 4$.
    Der Operator % liefert den Divisionsrest, z.B. $14 \,\%\, 3 = 2$.

Datentypen: Allgemeines (2)

- Der Datentyp einer Variablen bestimmt auch, wie viel Speicherplatz der Compiler für die Variable reservieren muss.

  Einge ganze Zahl benötigt z.B. mehr Speicherplatz als ein einzelnes Zeichen.

- Der Datentyp einer Variablen legt fest, wie die Bits in dem Speicherbereich der Variablen zu interpretieren sind.

- Die Angabe von Datentypen hilft, Fehler zu vermeiden bzw. schon bei der Compilierung zu erkennen.

  Der Compiler stellt sicher, dass nur passende Operationen auf die Daten angewendet werden können, d.h. dass die Bits an der gleichen Stelle im Hauptspeicher nicht bei verschiedenen Befehlen völlig unterschiedlich interpretiert werden können.

- Außerdem sind Datentypen eine wichtige Dokumentation, die das Verständnis des Programms erleichtert.

Datentypen: Allgemeines (3)

Fehlervermeidung durch Datentypen:

- Je früher ein Fehler erkannt wird, desto einfacher lässt er sich beseitigen.
- Wenn der Compiler einen Fehler aufgrund von unpassenden Datentypen feststellt, ist die Korrektur schneller und sicherer, als wenn der Fehler erst zur Laufzeit (bei Ausführung des Programms) erkannt wird.

  Es gibt auch Sprachen (wie z.B. PHP), bei denen man Variablen nicht deklarieren muss, und in eine Variable nacheinander Werte von verschiedenen Datentypen speichern kann. Zum Teil gibt es dann zur Laufzeit Fehler, zum Teil finden auch fragwürdige Typ-Anpassungen statt.
- Ein besseres Verständnis des Programms führt natürlich auch dazu, dass weniger Fehler gemacht werden.

Primitive Typen vs. Referenztypen (1)

- Die Datentypen in Java gliedern sich in
  - Primitive Typen, z.B. `int`.
  - Referenz-Typen, z.B. Klassen-Typen.

    Auch Interface-Typen und Array-Typen sind Referenz-Typen (später).
- `String` ist ein spezieller Klassen-Typ (Referenz-Typ): Es gibt eine vordefinierte Klasse für Zeichenketten.

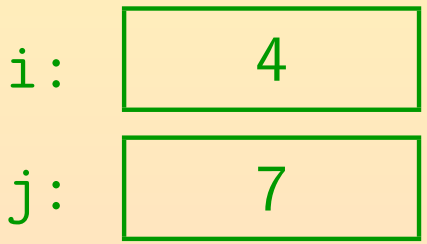
  Dies erklärt auch, warum `String` groß geschrieben wird, und `int` klein.

  Die Klasse `String` ist insofern speziell, als Zeichenketten-Konstanten wie `"abc"` vom Compiler in Objekte dieser Klasse abgebildet werden.

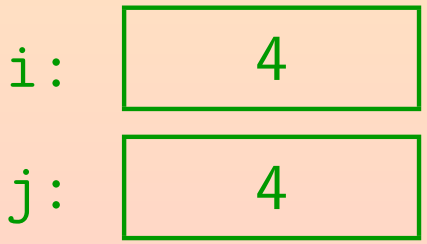
  Das kann man so nicht für selbstdefinierte Klassen haben.
- Während die Zuweisung bei primitiven Typen den ganzen Wert kopiert, speichert sie bei Referenz-Typen nur eine Referenz (d.h. Hauptspeicher-Adresse) für das Objekt (s.u.).

Primitive Typen vs. Referenztypen (2)

- Seien `i` und `j` Variablen vom Typ `int`, also einem primitiven Typ. Der Zustand sehe z.B. so aus:



- Nach der Zuweisung `j = i;` ist der Zustand dann:



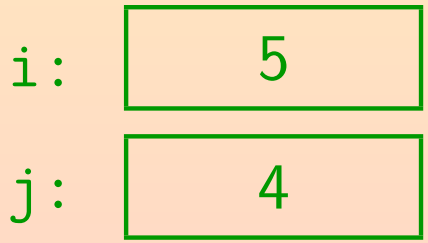
- Es wurde der Wert von der einen Variable in die andere kopiert.

Primitive Typen vs. Referenztypen (3)

- Man könnte z.B. anschließend den Wert der Variablen `i` ändern:

```
i = i + 1;
```

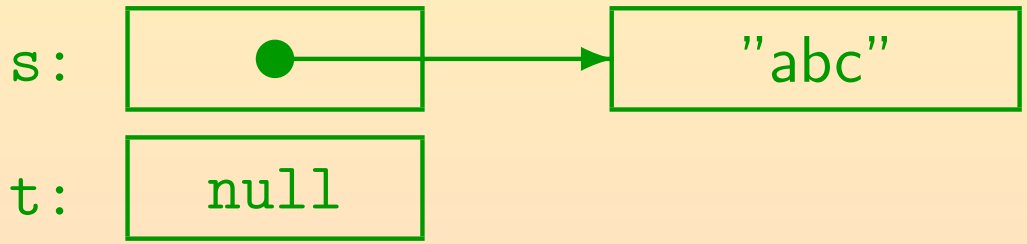
- Dies hat keinerlei Auswirkungen auf den Wert, der in der Variablen `j` gespeichert ist:



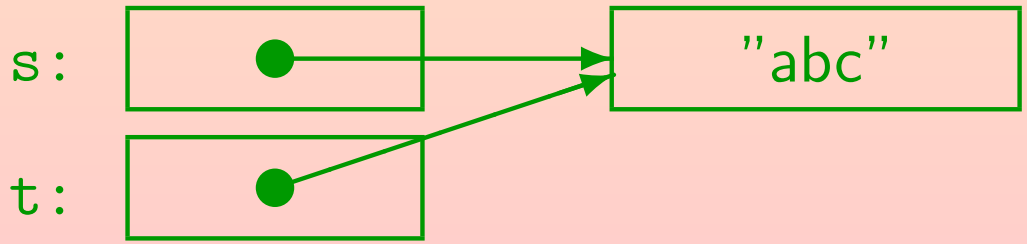
- Nach der Zuweisung `j = i;` gibt es also keinen Zusammenhang zwischen den beiden Variablen mehr, außer, dass in ihnen vorübergehend der gleiche Wert steht.

Primitive Typen vs. Referenztypen (4)

- `String` ist dagegen ein Referenztyp. Hier enthalten die Variablen nur einen Verweis auf das Zeichenketten-Objekt, was selbst an anderer Stelle im Hauptspeicher steht:



- Der spezielle Wert `null` ist die "Referenz auf nichts" (verschieden von jeder echten Referenz auf ein Objekt).
- Nach der Zuweisung `t = s;` sieht der Zustand so aus:

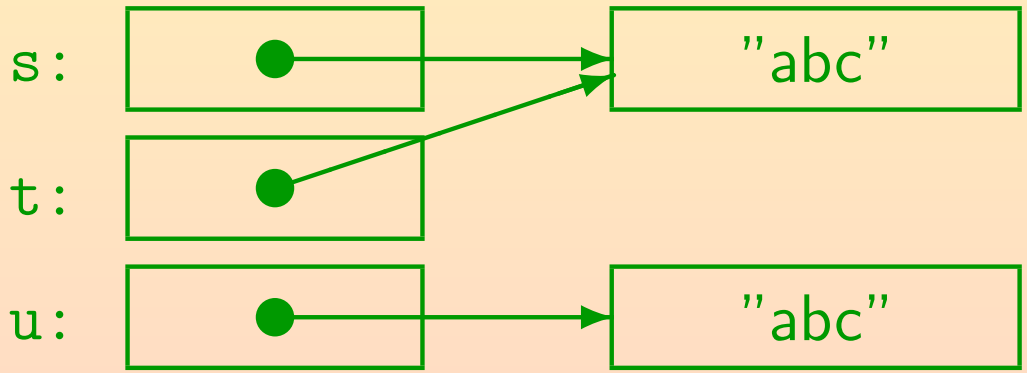


Primitive Typen vs. Referenztypen (5)

- Bei der Konkatenation von Zeichenketten entsteht ein neues Objekt. Z.B. würde nach

  ```
  String u = "ab";   u = u + "c";
  ```

  die Situation im Speicher so aussehen:



- Der Gleichheits-Operator "==" prüft nur die Referenzen, nicht die Inhalte der Objekte. Daher wäre
  - `s == t` wahr (`true`), aber
  - `s == u` falsch (`false`).

Primitive Typen vs. Referenztypen (6)

- Für eine echte Gleichheitsprüfung muss man

  `s.equals(u)`

  schreiben (das liefert `true`).

  Hier wird die Methode `equals`, die in der Klasse `String` definiert ist, für das Objekt `s` dieser Klasse aufgerufen. Dabei wird das Objekt `u` als Argument ergeben. Natürlich liefert `u.equals(s)` das gleiche Ergebnis.

- Objekte vom Typ `String` sind unveränderlich, daher spielt es bis auf "`==`" keine Rolle, ob man unterschiedliche Objekte mit dem gleichen Inhalt hat, oder nur ein Objekt.

  Wenn man z.B. zwei Strings konkateniert, entsteht ein neues `String`-Objekt. Die ursprünglichen Objekte bleiben unverändert. Es gibt auch Klassen `StringBuilder` und `StringBuffer` für veränderliche Zeichenketten (später). Für gleiche String-Literale im Programm legt Java nur ein Objekt an.

Primitive Typen vs. Referenztypen (7)

- Viele Klassen bieten aber die Möglichkeit, die in einem Objekt gespeicherten Werte auch zu ändern.
- Dann macht es einen wesentlichen Unterschied, ob man
  - zwei Objekte mit gleichem Inhalt,
  - oder nur zwei Referenzen auf das gleiche Objekt hat.
- Im zweiten Fall (nur ein Objekt) würde eine Änderung des Objektes über eine Referenz sofort auch über die andere sichtbar sein (weil es das gleiche Objekt ist). Beispiel:

```
Scanner s1 = new Scanner(System.in);
Scanner s2 = s1;
s2.useRadix(16);
```

Jetzt wäre auch `s1` auf Hexadezimal-Notation umgestellt.

Primitive Typen vs. Referenztypen (8)

- Objekte haben also einen internen Zustand, der geändert werden kann (je nach Klasse).

  Dagegen bleibt die Zahl 3 immer die Zahl 3.

- Die Erzeugung eines Objektes ist ein (relativ) bewusster Vorgang (z.B. mit `new`). Es wird Hauptspeicher reserviert, das Objekt bekommt eine feste Speicheradresse (Identität).

  Ein Wert ergibt sich dagegen bei einer Rechenoperation, der gleiche Wert kann auch in verschiedenen Variablen stehen. Er "verschwindet" wieder, wenn in die Variable ein neuer Wert geschrieben wird.

- Weil Objekte groß sein können, ist es auch aus Gründen der Effizienz günstig, nur die relativ kleine Referenz (32 bit oder 64 bit) zu kopieren.

  Java bietet natürlich Möglichkeiten, ganze Objekte zu kopieren, z.B. mit der Methode `clone()`. Das will man jedoch nur relativ selten.

Inhalt

Begrenzung des Zahlbereiches (1)

- Im Gegensatz zur mathematischen Vorstellung von "ganze Zahl" ist in Java (wie in den meisten Programmiersprachen) der Wertebereich begrenzt:
  - der kleinste Wert vom Typ `int` ist $-2\,147\,483\,648 = -2^{31}$,
  - der größte Wert ist entsprechend $2\,147\,483\,647 = 2^{31} - 1$, d.h. etwas über 2 Milliarden.
  - Der Grund für die Begrenzung ist, dass zur Darstellung eines `int`-Wertes im Hauptspeicher 32 Bit (4 Byte) benutzt werden, damit kann man insgesamt 2^{32} verschiedene Werte darstellen.

    Es int einen positiven Wert weniger als negative Werte, weil auch die 0 noch mit eingerechnet werden muss. Damit sind es dann insgesamt 2^{32} verschiedene Werte.

Begrenzung des Zahlbereiches (2)

- Falls das Ergebnis einer Rechenoperation außerhalb des darstellbaren Zahlbereiches liegt,
  - gibt es in Java keinen Fehler,
  - aber das Ergebnis stimmt natürlich nicht mit dem mathematisch erwarteten Ergebnis überein.
- Wenn man z.B. $1\,000\,000$ quadriert, so erhält man $-727\,379\,968$.

  Es werden einfach die untersten 32 Bit des richtigen Ergebnisses genommen, dabei gibt es hier auch einen Überlauf in das Vorzeichen-Bit hinein.
- Als Programmierer sollte man bei zu großen Eingaben eine Fehlermeldung ausgeben (so dass der Fall nicht auftritt).

  Die Java-Bibliothek hat auch eine Klasse `BigInteger` zur Repräsentation von ganzen Zahlen beliebiger Länge.

Primitive Typen: Ganze Zahlen (1)

- Java hat insgesamt 8 verschiedene primitive Typen.
- Davon sind vier Datentypen zur Repräsentation ganzer Zahlen unterschiedlicher Größe:
  - `byte` (8 bit): Von -128 bis $+127$.
  - `short` (16 bit): Von $-32\,768$ bis $+32\,767$.
  - `int` (32 bit): Von $-2\,147\,483\,648$ bis $+2\,147\,483\,647$.
  - `long` (64 bit): Von $-9\,223\,372\,036\,854\,775\,808$ bis $+9\,223\,372\,036\,854\,775\,807$ ($>$ 9 Trillionen: $9 * 10^{18}$)

    Da $2^{10} = 1024$ etwas mehr als 10^3 ist, ist 2^{63} entsprechend mehr als $2^3 * 10^{6*3}$, d.h. $8 * 10^{18}$. In C++ ist der Datentyp `long int`, den man auch `long` abkürzen kann, üblicherweise nur 32 bit groß.
    Die genauen Zahlbereiche sind in C++ implementierungsabhängig, bei `int` ist nur garantiert, dass es mindestens 16 bit groß ist.

Primitive Typen: Ganze Zahlen (2)

- Hinweis zur Verwendung:
  - `int` ist für ganze Zahlen der normale Typ.
  - Mit den Typen `byte`, `short` kann man Speicherplatz sparen, sie sind aber für Anfänger nicht zu empfehlen.

    Berechnungen (wie Addition +) werden mit 32 Bit durchgeführt, auch wenn die Eingaben einen kleineren Typ haben. Bei der Zuweisung an eine Variable desselben Typs muss dann eine explizite Typumwandlung geschrieben werden, also z.B. "`b = (byte) (b+1);`" wenn `b` als Variable vom Typ `byte` deklariert ist. Das ist eine unnötige Komplikation. Man würde also auch zum Speichern von Prozentzahlen (die sicher in ein `byte` hineinpassen würden) eher den Datentyp `int` benutzen. Da keiner von beiden Typen genau die Zahlen 0 bis 100 erlaubt, wäre eine zusätzliche Fehlerprüfung nützlich (später).
  - `long` wird nur selten benötigt, wenn wirklich sehr große Zahlen auftreten könnten.

Primitive Typen: Zeichen

- Außerdem gehört auch der Typ `char` zu den "integral types".
  - Er dient zur Repräsentation einzelner Zeichen z.B. `'a'`.

    "`char`" stammt von engl. "character" (Zeichen). Zur Aussprache: Man kann "`char`" wie den Anfang von "character" aussprechen ("kär"), oder wie eine Mischung von "child" und "car" ("tschar"). Es gibt ein englisches Wort "char" (Aussprache "tschar") mit der Bedeutung "verkohlen", "Rotforelle", "als Putzfrau arbeiten".
  - aber man kann ihn auch für Zahlen von 0 bis 65 535 verwenden (16 bit, "unsigned").

    Kapitel 1 enthält eine ASCII-Tabelle, die die Codierung von Zeichen durch Zahlen illustriert. Java benutzt Unicode, um auch deutsche Umlaute, asiatische Schriftzeichen u.s.w. darstellen zu können. Genauer benutzt Java UTF-16. Der Unicode Standard ist inzwischen auf mehr als 16 Bit gewachsen, einige seltene Zeichen müssen als zwei UTF-16 Einheiten repräsentiert werden, und passen nicht in ein `char`.

Primitive Typen: Gleitkomma-Zahlen (1)

- Weitere Zahldatentypen sind die Gleitkomma-Zahlen:
  - `float` (32 bit), von engl. "floating point numbers"
  - `double` (64 bit), von engl. "double precision"
- Gleitkomma-Zahlen sind Zahlen mit Nachkomma-Stellen und werden intern mit Vorzeichen, Mantisse und Exponent dargestellt, also z.B. $1.04\text{E}5 = 1.04 * 10^5 = 10400$.

  Tatsächlich werden Exponenten zur Basis 2 verwendet (s.u.).

  Statt "Gleitkomma" kann man auch "Fließkomma" sagen.
- Beispiele für Konstanten des Datentyps `double`: `3.14`, `1E10`, `1.2E-5`.

  Man beachte, dass Werte mit "`.`" notiert werden, und nicht mit Komma.

  Das Zeichen "`E`" für den Exponenten kann auch klein geschrieben werden: "`e`".

  Konstanten des Typs `float` werden durch ein angehängtes "`f`" / "`F`" markiert.

Primitive Typen: Gleitkomma-Zahlen (2)

- Hinweise zur Verwendung von Gleitkomma-Zahlen:
  - Das Gegenteil von Gleitkommazahlen wären Festkomma-Zahlen, also Zahlen mit einer festen Anzahl von Stellen vor und nach dem Komma.

    Festkomma-Zahlen gibt es in Datenbanken, aber nicht in Sprachen wie Java oder C++, dort gibt es nur den Spezialfall "ganze Zahl". Durch entsprechende Interpretation kann man aber Festkomma-Zahlen simulieren, z.B. Centbetrag abspeichern statt Euro-Betrag.
  - Vorteil von Gleitkomma-Zahlen: Es können sehr große und sehr kleine Zahlen dargestellt werden.
  - Nachteil: Es finden automatische, nicht immer überschaubare Rundungen statt (s.u.).
  - `double` ist der normale Gleitkomma-Typ, nicht `float`.

Primitive Typen: Wahrheitswerte

- Der letzte primitive Datentyp ist der Typ `boolean` zur Repräsentation von Wahrheitswerten.

  Benannt nach George Boole, 1815-1864. In C++ heißt der Typ "`bool`".

- Er hat nur zwei mögliche Werte:
  - `true`: wahr
  - `false`: falsch
- Man erhält einen booleschen Wert z.B. als Ergebnis eines Vergleichs:
  - $1 < 2$ ist wahr,
  - $3 > 4$ ist falsch.

Primitive Typen: Übersicht

Die 8 primitiven Typen von Java:

- Ganzzahlige Typen ("integral types" $\subseteq$ "numeric types"):
  - `byte`
  - `short`
  - `int`: Normalfall für ganze Zahlen
  - `long`
  - `char`: Für einzelne Zeichen (Buchstaben, Ziffern etc.)
- Gleitkomma-Zahlen ("floating point types" $\subseteq$ "num. types"):
  - `float`
  - `double`: Normalfall für Gleitkomma-Zahlen
- `boolean`: Für Wahrheitswerte (`true`, `false`).

Ungenauigkeiten bei `float`, `double` (1)

- Der Typ `float` hat 6–7 signifikante Dezimalstellen.

  Signifikante Stelle: Von erster von 0 verschiedener Stelle bis zur letzten bei Rundung sicher korrekter Stelle.

- Man kann damit z.B. folgende Zahl darstellen:

  $$0.0000123456 = 1.23456 * 10^{-6}$$

  Die Zahl wird nicht unbedingt exakt dargestellt, da intern eine Binärdarstellung verwendet wird, und der Exponent zur Basis 2 ist (s.u.). Nur wenn man auf 6 Stellen rundet, kann man erwarten, dass die Ziffern korrekt sind.

- Folgende Zahl kann man dagegen nicht darstellen:

  $$1.0000123456$$

  Dies würde 11 signifikante Stellen benötigen.

  Speichert man diese Zahl in eine `float`-Variable, und druckt sie mit 10 Nachkommastellen aus, bekommt man: 1.0000123978. Rundet man auf 7 Nachkommastellen (8 Stellen insgesamt), ist die letzte Ziffer falsch: 1.0000124.

Ungenauigkeiten bei `float`, `double` (2)

- Intern wird die Darstellung mit Mantisse und Exponent benutzt, allerdings zur Basis 2.
- Dadurch ist schon 0.1 nicht exakt darstellbar.

  Im Binärsystem ist dieser Bruch periodisch: 0.0001100110011001100110011...
- Praktische Auswirkung: Folgende Schleife terminiert nicht:

```
double x = 0;
while(x != 1.0)
    x = x + 0.1;
```

  Addiert man 10 mal 0.1, so ist das Ergebnis nicht exakt 1.0!

  Erst bei Ausgabe mit 16 Nachkommestellen sieht man den Unterschied: 0.9999999999999999.
- Bei Gleitkommazahlen verwende man also nicht == bzw. !=, sondern >= und <=.

Ungenauigkeiten bei `float`, `double` (3)

- Übliche mathematische Gesetze gelten nicht, z.B.:
$$(A + B) + C = A + (B + C)$$
(Assoziativgesetz). Verletzt (bei `float`-Addition) für:
  - $A = +1000000$
  - $B = -1000000$
  - $C = 0.0001$
- Die Addition sehr unterschiedlich grosser Zahlen übernimmt die kleinere Zahl nur sehr ungenau oder gar nicht, was für die weitere Berechnung oft problematisch ist.

  Oben ist die rechte Variante (mit $B + C$) ungünstig. Bei `float`: Ergebnis 0.
- Die Subtraktion fast gleich grosser Zahlen reduziert die Anzahl zuverlässiger Dezimalstellen.

Ungenauigkeiten bei `float`, `double` (4)

- Wenn sich Rundungsfehler (wie im Beispiel) fortpflanzen, ist es möglich, dass das Ergebnis absolut nichts mehr bedeutet.

  Also ein mehr oder weniger zufälliger Wert ist, der weit entfernt vom mathematisch korrekten Ergebnis ist. Es gibt Bibliotheken, die mit Intervallen rechnen, so dass man am Ende wenigstens merkt, wenn das Ergebnis sehr ungenau geworden ist.

- Geldbeträge würde man z.B. mit `int` in Cent repräsentieren, und nicht mit `float`.

- Wenn Sie es wesentlich genauer wissen wollen, lesen Sie: David Goldberg: What Every Computer Scientist Should Know About Floating-Point Arithmetic.

  [http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html]

Zusammenfassung zu primitiven Typen

- Der Typ, den Sie am häufigsten benutzen werden, ist `int`.
- Implizit werden Sie auch `boolean` häufig benutzen:
  Jede Bedingung in einem `if` oder `while` braucht einen Wertausdruck, der ein Ergebnis vom Typ `boolean` liefert.

  Explizite Variablen braucht man auch manchmal, aber nicht sehr häufig.
- `double` brauchen Sie nur, wenn Sie mathematisch/numerische Berechnungen machen, z.B. als Ingenieur.
- `char` brauchen Sie, wenn Sie textuelle Eingaben verarbeiten (oder Daten aus Dateien oder dem Netz).

  Wenn Sie auf niedrigerer Ebene arbeiten, könnte eventuell `byte` wichtig werden.
- Die anderen primitiven Typen brauchen Sie nur selten.

  Wenn sie von einer Methode der Java Bibliothek geliefert werden.

Inhalt

Modell-Bildung

- Programme dienen normalerweise dazu, Aufgaben und Aktivitäten in der realen Welt zu unterstützen.

  Manchmal sind es auch gedachte Welten, wie etwa bei Spielen.

- Dazu enthalten die Programme Abbilder von Objekten der realen Welt.

- Selbstverständlich läßt man dabei alle Details weg, die für die gegebene Aufgabe nicht relevant sind (diesen Vorgang nennt man auch “Abstraktion”).

  Wenn man z.B. die Hausaufgaben-Punkte für diese Vorlesung verwalten will, kann man Studenten auf die Eigenschaften “Name”, “Vorname”, “EMail-Adresse” und “Matrikel-Nummer” reduzieren. Dinge wie die Augenfarbe oder die Hobbies sind für diese Anwendung nicht wichtig.

Komponenten eines Objektes

- Wenn man ein Objekt hat, z.B. das Objekt `s` der Klasse `Student`, kann man im wesentlichen zwei Dinge damit tun:
  - Man kann auf Eigenschaften des Objektes (Attribute, Felder) zugreifen (im Objekt gespeicherte Variablen):
    ```
    System.out.println(s.matrikelnr);
    ```
  - Man kann das Objekt bitten, eine Aktion durchzuführen, indem man eine Methode für das Objekt aufruft:
    ```
    s.sendeEMail("Willkommen zur Vorlesung OOP");
    ```
- Um unkontrollierte Änderungen zu vermeiden, sind direkte Zugriffe auf die im Objekt gespeicherten Variablen oft verboten. Für wichtige Eigenschaften des Objektes werden dann zumindest Methoden zum Abfragen (Lesen) angeboten:
  ```
  System.out.println(s.getMatrikelnr());
  ```

Klassen (1)

- Man kann Klassen als Blaupausen/Schablonen zur Konstruktion von Objekten verstehen.
- In der Klasse ist festgelegt,
  - welche Variablen welcher Datentypen im Objekt existieren,
  - welche Methoden die Objekte dieser Klasse anbieten, und was geschieht, wenn die Methode aufgerufen wird.

    Für jede Methode ist in der Klasse Java Programmcode definiert, der bei Aufruf der Methode ausgeführt wird.
  - was davon von außen zugreifbar ist.

    Als Anwender der Klasse braucht uns der Rest nicht zu interessieren, und wird in der API-Dokumentation auch nicht genannt. Man darf sich aber nicht wundern, wenn das Objekt offensichtlich Daten speichern muss, aber in der API-Dokumentation keine oder nicht ausreichend "Fields" (Variablen, Attribute, Felder) genannt werden.

Klassen (2)

- Eine Klasse ist also eine Zusammenfassung von ähnlichen Objekten, die also insbesondere die gleichen Attribute haben und die gleichen Methoden anbieten.

  Beispiel: Alle Studenten für die Hausaufgaben-Punkte-Datenbank.

  Eine Klasse ist also ein Begriff, mit denen man über eine Menge von gleichartigen Dingen der realen Welt reden kann (auch Form der Abstraktion).

- Während die primitiven Typen fest in Java eingebaut sind, und nicht erweitert werden können, kann man eigene Klassen definieren, und damit die Menge der Datentypen erweitern.

  Tatsächlich ist Programmierung in Java immer die Definition von neuen Klassen (mit Attributen und Methoden darin).

- Die Datentypen im Programm sollten die Begriffe der Anwendung in natürlicher Weise widerspiegeln.

Methoden-Aufruf: Beispiel `String` (1)

- In Java gibt es eine große Menge vordefinierter Klassen, die die Entwickler bei Sun/Oracle bereits programmiert haben (natürlich in Java), und die man in eigenen Programmen nutzen kann.
- Während die Definition eigener Klassen in dieser Vorlesung erst später kommt, ist die Verwendung vordefinierter Klassen schon von Anfang an nötig (z.B. für Ein- und Ausgabe).
- Z.B. ist auf folgender Seite die Klasse `String` dokumentiert:
  [http://docs.oracle.com/javase/7/docs/api/java/lang/String.html]

  > Die Klasse `String` für Zeichenketten gehört zum Paket `java.lang`.

- Im Abschnitt "Method Summary" sind die von dieser Klasse angebotenen Methoden aufgelistet.

Methoden-Aufruf: Beispiel `String` (2)

- Z.B. findet sich dort der Eintrag:

```
char   charAt(int index)
       Returns the char value at the specified index.
```

- In Klammern sind die Parameter angegeben (Eingabewerte der Funktion). Im Beispiel ist es nur einer (Datentyp `int`).
- Links (vor dem Methodenname) steht der Ergebnis-Typ: Diese Funktion liefert ein Zeichen (Wert vom Typ `char`).
- Wenn man auf den Namen der Methode klickt, kommt man zu einer ausführlicheren Beschreibung.

  Dort ist erläutert, dass `index` die Position des ausgewählten Zeichens in der Zeichenkette ist, von 0 an gezählt (für das erste Zeichen), bis zur Länge -1 (für das letzte).

Methoden-Aufruf: Beispiel `String` (3)

- Man kann die Methode `charAt` z.B. wie in folgendem Programmstück aufrufen:

```
String s = "abc";
char c = s.charAt(0);
System.out.println(c);
```

- Dies gibt "a" aus, das erste Zeichen der Zeichenkette.

  Informatiker fangen oft bei 0 an zu zählen. Vielleicht liegt das an den internen Bitmustern, bei denen "alle Bits 0" der natürliche erste Wert ist. Im Zusammenhang mit Zeichenketten und Arrays (s.u.) hängt es auch mit der Adress-Rechnung in der Programmiersprache C zusammen: Wenn man die Hauptspeicher-Adresse der Zeichenkette hat, ist dort gleich das erste Zeichen gespeichert (in C). Das zweite Zeichen ist dann an der nächsten Adresse gespeichert (+1). Für Java ist das nicht mehr so relevant, aber es erleichtert C-Programmierern den Wechsel (die interne Adress-Rechnung könnte so auch in Java etwas schneller gehen).

Methoden-Aufruf: Beispiel `String` (4)

- Beim Aufruf `s.charAt(0)` hat der Programmcode in der Methode `charAt`
  - sowohl Zugriff auf das Objekt `s`, für das die Methode aufgerufen wird (Zeichenkette),
  - als auch auf den Parameter-Wert `0` (Position).
- In einer nicht objektorientierten Sprache hätte `charAt` zwei Parameter.

  Ebenso auch, wenn man in mathematischen Funktionen denkt.
- Bei der objektorientierten Notation unterscheidet man das Objekt, für das die Methode aufgerufen wird, von den restlichen Parametern.

  Beim Aufruf `s.equals(t)` ist das etwas komisch, weil `s` und `t` beides Strings sind, und der Vergleich auf Gleichheit zwei gleichberechtigte Eingabewerte hat.

Methoden-Aufruf: Beispiel `String` (5)

- Auch folgender Aufruf ist möglich:

```
System.out.println("abc".charAt(1));
```

- Dies gibt "`b`" aus, das zweite Zeichen der Zeichenkette.
- `"abc"` ist ja ein Objekt der Klasse `String`. Allgemein kann links vom "`.`" ein beliebiger Wertausdruck stehen, der ein solches Objekt berechnet:

```
String s = "ab";
char c = (s + "c").charAt(2);
System.out.println(c);
```

- Dies gibt "`c`" aus, das dritte Zeichen der Zeichenkette, die hier mit einer String-Konkatenation berechnet wird.

Methoden-Aufruf: Beispiel `String` (6)

- Selbstverständlich kann auch der Wert für den Parameter der Methode mit einem Wertausdruck berechnet werden:

```
int i = 1;
System.out.println("abc".charAt(i+1));
```

- Es kommt nur auf den an die Methode übergebenen Wert an, der ist wieder 2, daher wird "`c`" gedruckt.
- Natürlich muss der übergebene Wert eine gültige Position in der Zeichenkette sein, sonst bekommt man eine `IndexOutOfBoundsException` (Laufzeit-Fehler).
- Folgende Methode (ohne Parameter) liefert die Anzahl `chars`:

```
int  length()
     Returns the length of this string.
```

Methoden-Aufruf: Beispiel `String` (7)

- Mit folgender Schleife kann man die Position des Zeichens "`b`" bestimmen:

```
String s = "abc";
int i = 0;
while(i < s.length()) {
    if(s.charAt(i) == 'b')
        System.out.println(i);
    i = i + 1;
}
```

- Im Beispiel würde `s.length()` den Wert 3 liefern.
- Der Rumpf der Schleife wird also drei Mal durchlaufen, und zwar für `i=0`, `i=1` und `i=2`.

Im Beispiel wird `1` ausgedruckt. Vielleicht hätte man besser `i+1` drucken sollen.

Methoden-Aufruf: Beispiel `String` (8)

- Tatsächlich muss man dies nicht unbedingt selbst programmieren, denn es gibt auch folgende Methode:

  ```
  int   indexOf(int ch)
        Returns the index within this string of the
        first occurrence of the specified character.
  ```

  Kommt das Zeichen nicht vor, wird -1 geliefert.

- Der Typ des Parameters ist `int` und nicht `char`, weil auch Unicode "Code Points" jenseits der "Basic Multilingual Plane" möglich sind (die mehr als 16 Bit benötigen).

- Man darf beim Aufruf aber auch ein Zeichen angeben: Der kleinere Typ `char` (16 Bit) wird automatisch in den größeren Typ `int` (32 Bit) umgewandelt:

  ```
  System.out.println(s.indexOf('b'));
  ```

Methoden-Aufruf: Beispiel `String` (9)

- Hier noch die Methode für den Gleichheitstest:

  `boolean` **`equals`**`(Object anObject)`

  `Compares this string to the specified object.`

  Es sind beliebige Objekte als Argument erlaubt, dass schließt `String`-Objekte ein. Wenn das Argument kein `String` ist, ist das Ergebnis aber `false`.

- Da das Resultat ein Wahrheitswert ist, kann man z.B. schreiben (wenn `antwort` eine Variable vom Typ `String` ist):

```
if(antwort.equals("ja"))
    System.out.println("Richtig!");
else
    System.out.println("Leider falsch.");
```

Methoden-Aufruf: `StringBuilder` (1)

- Es gibt auch Methoden, die keinen Wert liefern.

  Ihre ganze Aufgabe besteht in einer Änderung des internen Zustands des Objektes, oder einer Ein-/Ausgabe.

- Die Klasse `String` hat keine solchen Methoden, aber die Klasse `StringBuilder` für änderbare Zeichenketten.

  [http://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html]

- Man kann sich auf folgende Art ein Objekt dieser Klasse anlegen, das mit einer gegebenen Zeichenkette initialisiert ist:

```
StringBuilder s = new StringBuilder("abc");
```

  In dem Objekt `s` ist jetzt die Zeichenkette `"abc"` gespeichert, aber man kann die in dem Objekt gespeicherte Zeichenkette ändern. Wäre `s` ein `String`, so könnte man nur die Variable auf andere Objekte zeigen lassen, aber das jeweilige Objekt selbst könne man nicht ändern.

Methoden-Aufruf: StringBuilder (2)

- Die Klasse `StringBuilder` bietet eine Methode, um das Zeichen an einer Position zu ändern:

```
void  setCharAt(int index, char ch)
      The character at the specified index is
      set to ch.
```

- Man könnte diese Methode z.B. so aufrufen:

```
s.setCharAt(1, 'x');
```

- Die Methode hat zwei Parameter, es müssen also zwei Werte angegeben werden: Eine ganze Zahl (`int`) und ein Zeichen (`char`). Sonst Fehlermeldung.

```
Str.java:4: error: method setCharAt in class AbstractStringBuilder
                   cannot be applied to given types;
        s.setCharAt(1);
          ^
  required: int,char
  found: int
  reason: actual and formal argument lists differ in length
```

Methoden-Aufruf: StringBuilder (3)

- Da diese Methode keinen Wert liefert (Rückgabetyp `void`), bekommt man eine Fehlermeldung, wenn man z.B. versucht, den Wert des Methodenaufrufs auszudrucken:

```
System.out.println(s.setCharAt(1, 'x'));
```

```
Str.java:5: error: 'void' type not allowed here
        System.out.println(s.setCharAt(1, 'x'));
                                          ^
```

- Da es keine Werte des Typs `void` gibt, kann man natürlich auch keine Variable von diesem Typ deklarieren:

```
void x; // Falsch!
```

Formal ist `void` gar kein Typ. Dieses Schlüsselwort kann nur alternativ zu einem Datentyp verwendet werden, um den Ergebniswert einer Methode zu charakterisieren.

Methoden-Aufruf: StringBuilder (4)

- Man kann den aktuellen Inhalt eines `StringBuilder`-Objektes mit dem üblichen Ausgabe-Befehl anzeigen:

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

- Dabei wird implizit folgende Methode aufgerufen:

```
String  toString()
        Returns a string representing the data
        in this sequence.
```

  Man kann diese Methode auch für eigene Klassen definieren, um anzugeben, wie Objekte der Klasse ausgedruckt werden sollen.

- Wenn die Änderung mit `s.setCharAt(1, 'x')` durchgeführt ist, wird "`axc`" ausgegeben.

Methoden-Aufruf: `StringBuilder` (5)

- Um die Zeichenkette im Objekt zu verlängern, gibt es verschiedene append-Methoden:

```
StringBuilder  append(char c)
               Appends the string representation of
               the char argument to this sequence.
```

Das Zeichen wird also hinten an die Zeichenkette angehängt.

- Von dieser Funktion gibt es Varianten für verschiedene Argument-Datentypen (sie ist "überladen"), z.B.

```
StringBuilder  append(int i)
               Appends the string representation of
               the int argument to this sequence.
```

Es gibt u.a. auch Varianten für `String`, `double` und `boolean`.

Methoden-Aufruf: `StringBuilder` (6)

- Da die Funktion das Objekt selbst zurückgibt, kann man Ketten schreiben:

```
s.append(' ').append(10).append('!');
```

- Dies ist so geklammert zu verstehen ("von links"):

```
((s.append(' ')).append(10)).append('!');
```

- Zum Beispiel ist `s.append(' ')` wieder das Objekt `s`, die Aufrufe werden von links nach rechts nacheinander ausgeführt. Man kann aber auch einzelne Aufrufe schreiben:

```
s.append(' ');
s.append(10);
s.append('!');
```

Der zurückgelieferte Wert wird hier einfach vergessen.

Aufgabe/Beispiel (Referenz-Typen)

- Was gibt dieses Programm aus?

```
class RefTest {
    public static void main(String[] args) {
        StringBuilder s1 =
                new StringBuilder("abc");
        StringBuilder s2 = s1;
        s1.setCharAt(2, '!');
        System.out.println(s2);
    }
}
```

Untertypen / Subklassen (1)

- Natürlich sind `String` und `StringBuilder` in vieler Hinsicht ähnlich: beides sind Zeichenketten.

  Der Unterschied ist nur, dass die Zeichenkette in einem `String`-Objekt nicht änderbar ist, die in einem `StringBuilder`-Objekt aber schon.

- Es ist kein Zufall, dass beide eine Methode `charAt(i)` haben, die das Zeichen an Position `i` liefert.

- Beide sind Untertypen des allgemeineren Typs `CharSequence`.

- D.h. die Objekte der Klassen `String` und `StringBuilder` sind Teilmengen der Objekte des Typs `CharSequence`.

- Weiteres Beispiel für Untertypen / Subklassen:
  Die Menge der Rechtecke ist eine Teilmenge der Menge der Vierecke (`Rechteck` ist eine Subklasse von `Viereck`).

Untertypen / Subklassen (2)

- Weil Strings auch zum allgemeineren Typ `CharSequence` gehören, ist folgende Zuweisung legal:

```
CharSequence s = "abc";
```

  Anschließend kann man auf `s` aber nur noch die für `CharSequence` definierten Methoden verwenden. Der Compiler weiß nicht mehr, dass in `s` tatsächlich ein spezielleres Objekt gespeichert ist. Bei Bedarf kann man (mit aller Vorsicht) eine Typ-Umwandlung zurück machen, später mehr.

- Substitutionsprinzip: Man kann ein Objekt eines spezielleren Typs (Unterklasse) überall verwenden, wo ein Objekt des allgemeineren Typs (Oberklasse) erlaubt ist.

- Insbesondere kann man jede Methode, die für eine Oberklasse definiert ist, auch für Objekte der Subklasse aufrufen.

  Man sagt auch: Sie Subklasse "erbt" die Methoden der Oberklasse.

Objekt-Erzeugung, Konstruktoren (1)

- Man erzeugt ein Objekt (etwas vereinfacht) mit
  - dem Schlüsselwort "`new`",
  - gefolgt von dem Klassennamen, z.B. `StringBuilder`,
  - und einer Argumentliste, z.B. "`()`".
- Damit man das neu erzeugte Objekt verwenden kann, muss man es sich in eine Variable speichern.

  Sonst ist es nicht mehr zugreifbar und wird von Java automatisch gelöscht.
- Man braucht eine Variable für (Referenzen auf) Objekte der Klasse `StringBuilder`:

```
StringBuilder s;
```

- Dieser Variablen weist man das neu erzeugte Objekt zu:

```
s = new StringBuilder();
```

Objekt-Erzeugung, Konstruktoren (2)

- Alternativ: Deklaration und Initialisierung in einer Anweisung:

```
StringBuilder s = new StringBuilder();
```

- Wenn ein neues Objekt erzeugt wird, wird automatisch ein Stück Programmcode der Klasse, ein sogenannte "Konstruktor" ausgeführt.
- Dieser hat die Aufgabe, die im Objekt gespeicherten Variablen zu initialisieren.
- Dazu verwendet er Werte, die bei der Objekterzeugung übergeben werden, z.B.

```
StringBuilder s = new StringBuilder("abc");
```

- Eine Klasse kann mehrere Konstruktoren mit unterschiedlichen Parameterlisten anbieten (überladene Konstruktoren).

Objekt-Erzeugung, Konstruktoren (3)

- Konstruktoren sind recht ähnlich zu Methoden, sind aber formal keine Methoden. Sie heißen immer so wie die Klasse, und haben keinen Rückgabetyp.
- Die Klasse `StringBuilder` bietet mehrere Konstruktoren an, u.a. einen ohne Parameter:

  StringBuilder()

  `Constructs a string builder with no characters in it and an initial capacity of 16 characters.`

  Konstruktoren stehen in der Klassenbeschreibung oben: "Constructor Summary".
- Wir haben aber auch den folgenden Konstruktor verwendet:

  StringBuilder(String str)

  `Constructs a string builder initialized to the contents of the specified string.`

Statische Methoden und Attribute (1)

- Normale Methoden müssen für ein Objekt der Klasse aufgerufen werden.

  Und entsprechend normale Attribute für ein Objekt der Klasse abgefragt werden (falls Sie überhaupt von außen zugreifbar sind).

- Es gibt aber auch Methoden, für die kein Objekt benötigt wird ("statische Methoden", "Klassen-Methoden").
- Da unterschiedliche Klassen Methoden gleichen Namens haben können, muss man dann wenigstens den Namen der Klasse schreiben:

```
double x = Math.sqrt(9.0);
```

- Die Klasse `Math` mit mathematischen Funktionen ist auf folgender Seite dokumentiert:
  [http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html]

Statische Methoden und Attribute (2)

- Funktionen, die ohne Objekt aufgerufen werden können, sind mit dem Schlüsselwort "`static`" gekennzeichnet:

```
static double sqrt(double a)
    Returns the correctly rounded
    positive square root of a
    double value.
```

- Entsprechend werden auch Konstanten und Variablen, die der Klasse zugeordnet sind, und nicht einem einzelnen Objekt, mit "`static`" markiert:

```
static double PI
    The double value that is closer than
    any other to pi, the ration of the
    circumference of a circle to its
    diameter.
```

Konstanten

- Obwohl `PI` eigentlich eine Variable in der Klasse `Math` ist, kann man ihr nichts zuweisen:

```
Math.PI = 3.0; // Geht nicht!
```

```
Test.java:3: error: cannot assign a value to final variable PI
```

- Wenn man sich die Detailinformation zu `Math.PI` anschaut, steht dort

```
public static final double PI
```

Das Schlüsselwort "`final`" bedeutet, dass ihr ein Mal in der Klasse ein Wert gegeben wird, und dann keine weiteren Zuweisungen möglich sind (es ist der finale/endgültige Wert).

D.h. es handelt sich um eine Konstante. Die meisten von außen zugreifbaren Attribute sind so vor Veränderungen geschützt.

Klassen ohne Konstruktor

- Man kann kein Objekt der Klasse `Math` erzeugen.

```
Test.java:3: error: Math() has private access in Math
            Math x = new Math();
                     ^
```

- In der API-Spezifikation sind keine Konstruktoren aufgelistet.

  Tatsächlich muss intern ein Konstruktor deklariert werden, aber der ist von außen nicht zugreifbar: "privat" (und wird intern nicht benutzt). Dies wird in einem späteren Kapitel noch erklärt. Im Prinzip wäre es möglich, dass eine Klasse zwar keine direkte Objekterzeugung mit `new` erlaubt (die einen zugreifbaren Konstruktor erfordert), aber statische Methoden anbietet, die Objekte liefern. Das ist bei `Math` aber nicht der Fall: Es gibt einfach keine Objekte dieser Klasse. Es ist einfach eine Sammlung von Bibliotheksfunktionen.

- Da alle Methoden und Attribute `static` sind, braucht man auch kein Objekt dieser Klasse.

Beispiel: Ausgabe-Anweisung (1)

- Unsere erste Anweisung war:

```
System.out.println("Hello, world!");
```

- `System` ist eine Klasse, dokumentiert unter:
  [http://docs.oracle.com/javase/7/docs/api/java/lang/System.html]
- "`out`" ist ein statisches Attribut dieser Klasse.
  Das Objekt, auf die die Referenz in diesem Attribut zeigt, gehört selbst zur Klasse `PrintStream` im Paket `java.io`.
  [http://docs.oracle.com/javase/7/docs/api/java/io/PrintStream.html]
- `println` ist eine Methode der Klasse `PrintStream`.
  Sie wird für das Objekt `System.out` aufgerufen.

Beispiel: Ausgabe-Anweisung (2)

- `println` ist eine recht stark überladene Methode. Es gibt Varianten für unterschiedliche Datentypen, oben wurde die mit einem Zeichenketten-Argument verwendet:

  ```
  void  println(String x)
        Prints a String and then terminate the line.
  ```

- Es gibt auch Varianten für Argumente u.a. vom Typ `int`, `double`, `char`, `boolean`.

- Wenn man will, kann man das Objekt für den Ausgabestrom auch in einer Variable speichern:

  ```
  java.io.PrintStream p = System.out;
  p.println("Hello, World!");
  ```

  Weil die Klasse nicht zum Paket `java.lang` gehört, muss man den Paketnamen voranstellen (oder eine `import`-Deklaration verwenden).

Inhalt

Arrays: Allgemeines (1)

- Arrays sind eine Zusammenfassung von
  - n Variablen gleichen Typs,
  - wobei die einzelne Variable über eine Zahl, den Index, identifiziert wird (zwischen 0 und $n-1$).
- Objekte enthalten auch mehrere Variablen, aber
  - die Komponenten können unterschiedlichen Typ haben,
  - werden über Namen identifiziert,
  - sind meist nur indirekt über Methoden zugreifbar.
- Die Nützlichkeit von Arrays beruht darauf, dass die konkrete Variable für Lese- und Schreibzugriffe über eine Berechnung ausgewählt werden kann.

  Das würde mit einzelnen Variablen `x0`, `x1`, `x2`, ... nicht gehen.

Arrays: Allgemeines (2)

- Der Element-Typ (engl. “component type”) eines Arrays kann ein beliebiger Typ sein, also ein primitiver Typ oder ein Referenz-Typ (z.B. eine Klasse).

  Ein Array-Typ ist selbst ein Referenz-Typ, man kann also auch Arrays von Arrays definieren (später).

- Der Array-Typ über einem Element-Typ `T` wird `T[]` geschrieben.

- Z.B. deklariert man so eine Variable `a`, die auf ein Array von `int`-Werten verweist:

```
int[] a;
```

  Zur Erleichterung für ehemalige C- und C++-Programmierer kann man auch “`int a[];`” schreiben. Die beiden Notationen sind äquivalent.

Arrays: Allgemeines (3)

- Mathematisch gesehen ist der Wert eines Arrays eine Abbildung vom Indexbereich auf den Wertebereich des Element-Datentyps, im Beispiel:

$$\{0, 1, 2, 3, 4\} \rightarrow \texttt{int}: \{-2^{31}, \ldots, 2^{31} - 1\}$$

- Man kann sie als Tabelle darstellen (Beispiel):

Index	Inhalt
0	27
1	42
2	18
3	73
4	56

Arrays: Allgemeines (4)

- In meinem Englisch-Deutsch Wörterbuch stehen für “array” u.a. “Ordnung”, “Schlachtordnung”, “Phalanx”, “stattliche Reihe”, “Menge”.

  Also alles nicht besonders hilfreich für den informatischen Fachbegriff.

- Der informatische Fachbegriff wird gelegentlich mit “Feld” übersetzt.

  Das ist etwas problematisch, weil es nichts mit dem “field” in Records zu tun hat, und dieses Wort in der Java-Spezifikation für Attribute verwendet wird.

- Meist sagt man auch auf Deutsch “Array”.

- Ein Array entspricht mathematisch auch einem Vektor.

  In Java gibt es die Klasse “`java.util.Vector<T>`” für Arrays mit änderbarer Größe (und Element-Typ `T`).

Arrays: Deklaration und Erzeugung (1)

- Mit der Variablen-Deklaration wird das Array noch nicht angelegt, sondern nur Speicherplatz für eine Referenz reserviert (Arrays sind Objekte).

  Das ist ein Unterschied zu Pascal, C, C++: Dort deklariert man das Array mit einer Größe, und es wird gleich entsprechend Speicherplatz reserviert.

- Man erzeugt ein Array mit dem `new`-Operator, wobei man hier die Größe des Arrays angeben muss, d.h. die Anzahl der Komponenten-Variablen:

```
a = new int[5];
```

- Oder zusammen mit der Deklaration von a:

```
int[] a = new int[5];
```

  Die doppelte Angabe des Komponenten-Typs ist nicht ganz redundant (bei Subklassen müssen die beiden Typen nicht übereinstimmen, später mehr).

Arrays: Deklaration und Erzeugung (2)

- Die Größe eines Arrays ist nicht nachträglich änderbar.
- Die Variable a ist aber nicht ein eine bestimmte Größe gebunden, sie kann später auch auf andere `int`-Array-Objekte mit anderer Größe zeigen.

  Wenn das ursprünglich angelegte Array später zu klein sein sollte, kann man ein größeres anlegen, den Inhalt des alten Arrays in das neue kopieren, und dann das neue Array der Variablen a zuweisen.
- Man kann die Größe abfragen mit dem Attribut `length`:

  ```
  System.out.println(a.length);
  ```

  Dieses Attribut kann abgefragt, aber nicht geändert werden (`final`).
- Die Array-Größe `0` ist möglich.

  Manchmal muss man einer Methode ein Array übergeben, braucht bei einem speziellen Aufruf aber vielleicht keine Werte. Negative Größen gehen nicht.

Array-Zugriff: Grundlagen

- Man kann auf die `i`-te Komponente des Arrays `a` zugreifen mit dem Wertausdruck `a[i]`.
- Dies ist eine Variable des Element-Typs (im Beispiel `int`). Daher sind sowohl Schreib-Zugriffe möglich:

  ```
  a[i] = 100;
  ```

  als auch Lese-Zugriffe:

  ```
  System.out.println(a[i]);
  ```

- Der Index `i` muss zwischen `0` und `a.length-1` liegen, im Beispiel also zwischen `0` und `4`.

  Das sind `5` verschiedene Werte, entsprechend der Array-Größe `5`.

- Sonst erhält man eine `ArrayIndexOutOfBoundsException`.

  Laufzeit-Fehler: Der Array Index liegt außerhalb der Grenzen.

Array-Zugriff: Grenzen-Verletzung (1)

- Java prüft also den Index bei jedem Zugriff auf eine eventuelle Verletzung der Array-Grenzen.

  C und C++ tun das nicht: Es wird einfach die in einem späteren Kapitel angegebene Formel zur Berechnung der Speicheradresse angewendet, und auf die zufällig dort im Hauptspeicher liegenden Daten zugegriffen. Das ist besonders bei Schreibzugriffen gefährlich, da ganz andere Variablen und sogar Rücksprung-Adressen verändert werden können. "Buffer Overflows" haben schon oft Hackern Tor und Tür geöffnet. Es ist also sehr wichtig, so zu programmieren, dass Arraygrenzen-Verletzungen nicht vorkommen können. In Java natürlich auch, dort würde der Fehler aber sicher gemeldet, wenn er vorkommt (und das Programm normalerweise beendet, auf keinen Fall aber Variablen in unvorhersehbarer Weise verändert, oder gar die Integrität des Laufzeitsystems kompromittiert). Der Preis, der für die zusätzliche Sicherheit gezahlt werden muss, ist eine leichte Verlangsamung durch die zusätzlichen Tests.

Array-Zugriff: Grenzen-Verletzung (2)

- Laufzeitfehler (wie die Array-Grenzen-Verletzung) können von den Eingabe-Daten abhängen, müssen also nicht bei jedem Test bemerkt werden.

  Z.B. reicht die Arraygröße vielleicht für kleine Eingaben, aber nicht für große. Der Benutzer erwartet dann mindestens eine anständige Fehlermeldung, nicht einfach eine `ArrayIndexOutOfBounds`-Exception. Es sollte auch eine Konstante geben, mit der man das Array leicht vergrößern kann. Noch besser wäre es natürlich, wenn sich das Programm automatisch anpasst.

- Dies ist ein Unterschied zu Fehlern, die der Compiler findet: Diese hängen nicht von der Eingabe ab und sind daher nicht zu übersehen.

  Da Laufzeitfehler nicht sicher durch Testen gefunden werden hilft nur Nachdenken: Man braucht eigentlich einen mathematischen Beweis für jeden Array-Zugriff im Programm, dass der Index sicher innerhalb der Grenzen liegt.

Array-Zugriff: Index-Berechnung

- Der Index kann über einen beliebigen arithmetischen Ausdruck berechnet werden:

```
a[n*2+k]
```

- Der Wertausdruck im Innern der `[...]` muss den Typ `int` haben.

  Die kleineren ganzzahligen Typen `byte`, `short`, `char` gehen auch, sie werden automatisch in `int` umgewandelt. Aber z.B. `long`, `float` gehen nicht.

- In Java beginnt der Indexbereich eines Arrays immer mit `0`.

  Das ist von C übernommen und hängt dort mit der Berechnung der Hauptspeicher-Adresse für ein Array-Element zusammen (späteres Kapitel).
  In Pascal gibt man dagegen Untergrenze und Obergrenze des Indexbereiches explizit an: "`a: array[1..5] of integer;`".

Arrays: Initialisierung

- Die Komponenten eines Arrays werden automatisch initialisiert (auf 0, `null`, bzw. `false`, je nach Typ).

  Java stellt sicher, dass man nicht auf uninitialisierte Variablen zugreifen kann. Das kostet etwas Laufzeit, bringt aber mehr Sicherheit. Normale (lokale) Variablen werden nicht automatisch initialisiert, aber hier prüft der Compiler, dass man auf sie nicht zugreift, bevor man keinen Wert zugewiesen hat.

- Z.B. kann man sich nach

  ```
  int[] a = new int[3];
  ```

  darauf verlassen, dass in `a[0]`, `a[1]` und `a[2]` der Wert `0` steht.

Arrays: Referenztypen

- Array-Typen sind Referenztypen.
- Wie bei Objekten vergleicht `==` nur die Referenz.

  Und nicht den Inhalt, d.h. die einzelnen Array-Elemente.

- Aufgabe: Was gibt dieses Programmstück aus?

```
(1)  class Test {
(2)      static public void main(String[] args) {
(3)          int[] a = new int[3];
(4)          a[0] = 27;
(5)          int[] b = a;
(6)          b[0] = 32;
(7)          System.out.println(a[0]);
(8)      }
(9)  }
```

Programm-Beispiele (1)

- Typische Schleife über ein Array (hier zum Ausgeben):

```
int i = 0;
while(i < a.length) {
    System.out.println(a[i]);
    i = i + 1;
}
```

- Es gibt immer die drei Teile:
  - Deklaration und Initialisierung der Laufvariablen:
    `int i = 0;`
  - Bedingung für weiteren Durchlauf (Begrenzung):
    `i < a.length`
  - Weiterschalten der Laufvariable (Inkrement, Schritt):
    `i = i+1;`

Programm-Beispiele (2)

Prüfen, ob Element in Array:

- Boolesche Variable vor der Schleife auf `false` setzen, falls gefunden, in der Schleife auf `true`:

```
boolean gefunden = false;
int i = 0;
while(i < a.length) {
    if(a[i] == gesuchterWert)
        gefunden = true;
    i = i + 1;
}
if(gefunden)
    System.out.println("Wert wurde gefunden.");
else
    System.out.println("Nicht gefunden.");
```

Strings und Arrays

- `String` und `StringBuilder` verwenden intern Arrays vom Typ `char[]`, um die Zeichen in der Zeichenkette zu verwalten.
- Der Methoden-Aufruf `s.charAt(i)` ist in der Wirkung sehr ähnlich zum Array-Zugriff `s[i]`.

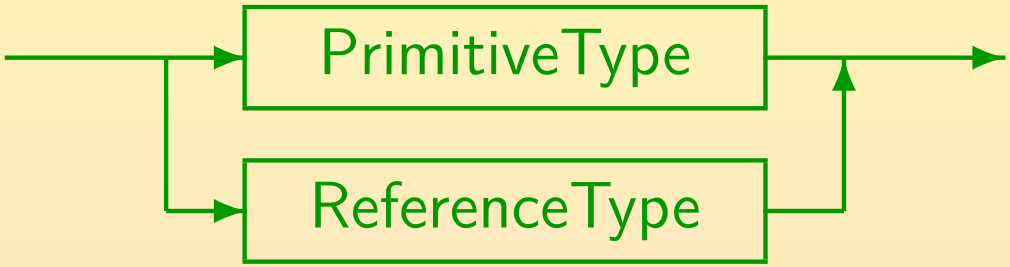
  Wenn `s` ein `String` ist, und nicht ein Array, wäre `s[i]` aber ein Syntaxfehler.
- Man beachte, dass die Länge
  - bei `String` und `StringBuilder` mit dem Methodenaufruf `s.length()` bestimmt wird,
  - bei Arrays dagegen mit dem Attribut-Zugriff `a.length`.

    Mögliche Erklärung: Man wollte bei `String` und `StringBuilder` die gleiche Schnittstelle, aber bei `StringBuilder` ist die Länge variabel, deswegen ging dort ein konstantes Attribut (`final`) nicht. Ein normales Attribut, das auch von außen geändert werden kann, wäre aber unsicher.

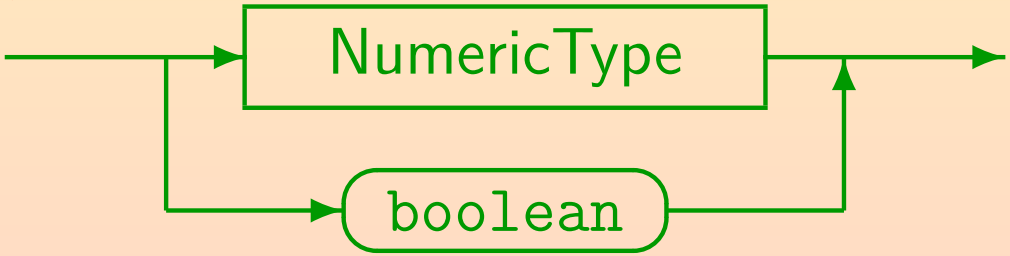
Inhalt

Syntaxdiagramme für Datentypen (1)

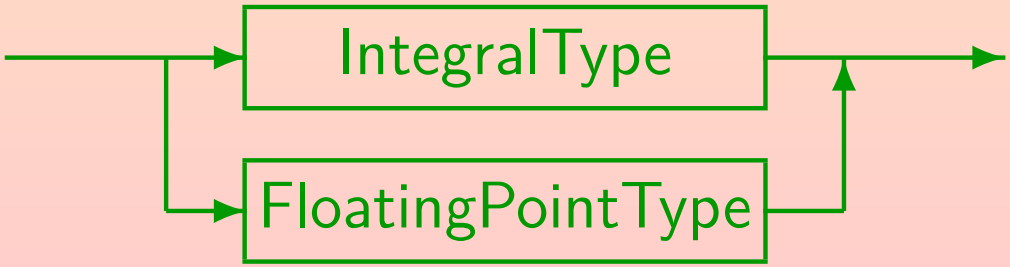
- Type:



- PrimitiveType:

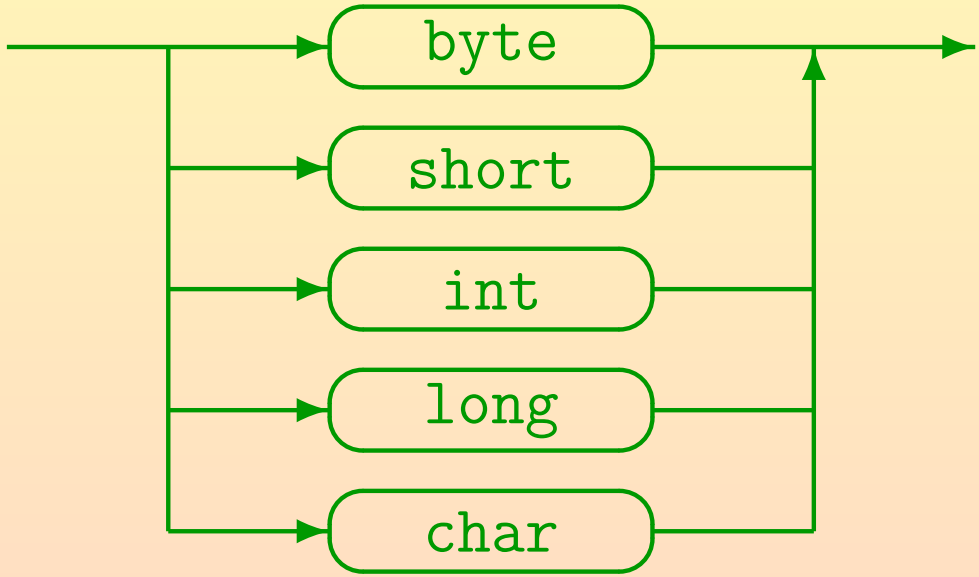


- NumericType:

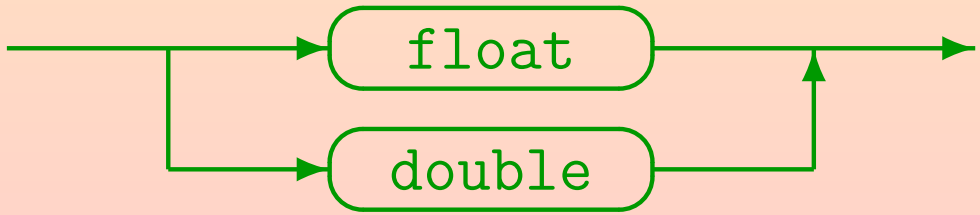


Syntaxdiagramme für Datentypen (2)

- IntegralType:



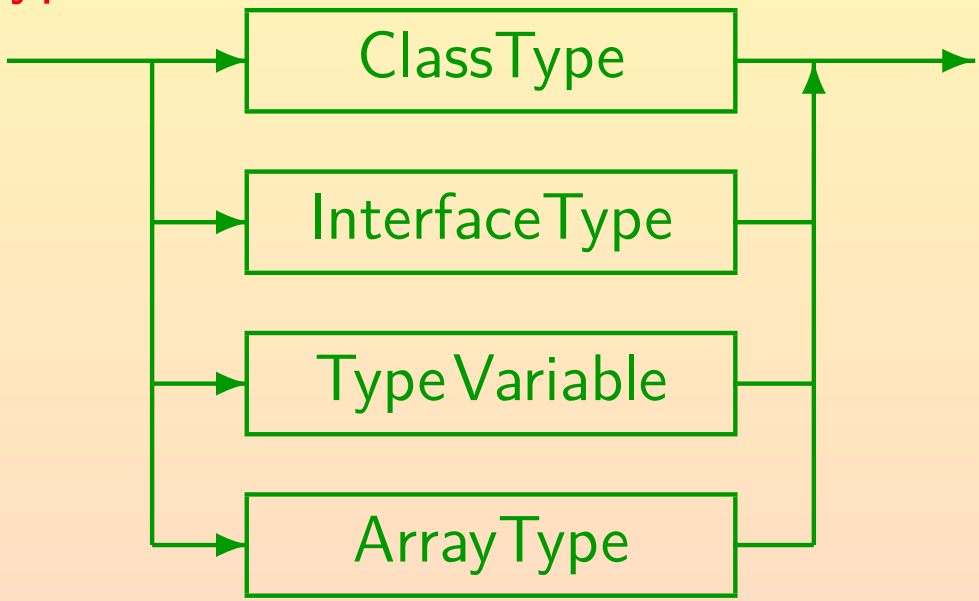
- FloatingPointType:



Die Syntaxdiagramme entsprechen Grammatikregeln in der "Java Language Specification" [http://docs.oracle.com/javase/specs/].

Syntaxdiagramme für Datentypen (3)

- ReferenceType:



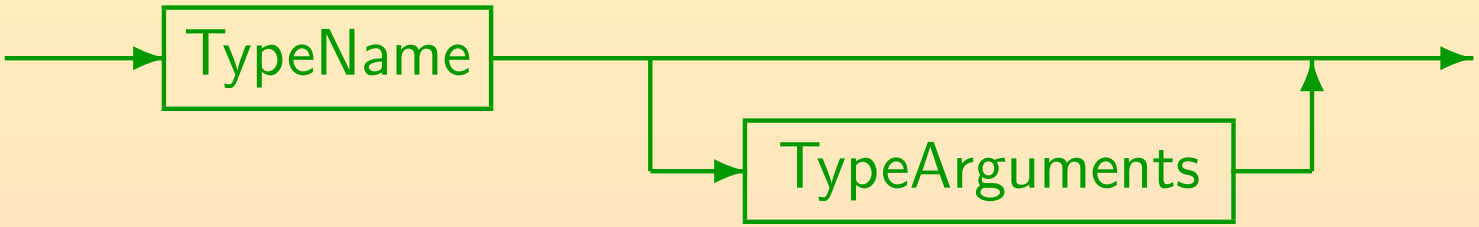
Wir haben bisher nur über die Fälle "ClassType" und "ArrayType" gesprochen. Die anderen Fälle werden in späteren Kapiteln behandelt.

- TypeVariable:

Dies wird erst später relevant.

Syntaxdiagramme für Datentypen (4)

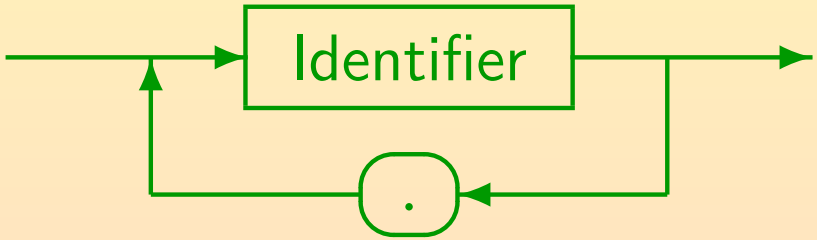
- ClassType / InterfaceType:



Die "TypeArguments" werden auch erst später besprochen. Im Moment ist dies einfach nur ein "TypeName" (man muss also den horizontalen Pfeil nach rechts nehmen). Während ich sonst die Regeln aus der "Java Language Specification" übernommen habe, ist der "TypeName" eine Vereinfachung. Dort gibt es noch den Umweg über einen "TypeDeclSpecifier" zur Behandlung von lokalen Klassen. Das brauchen Sie nicht zu verstehen.

Syntaxdiagramme für Datentypen (5)

- TypeName:



Wir haben bei "`java.io.PrintStream`" gesehen, dass ein Typname durch Voranstellen des Paketnamens vor den eigentlichen Klassennamen aus mehreren Teilen bestehen kann.

- ArrayType: