


- Die Methode `get` liefert eine Referenz vom Typ `Object`. Daraus resultieren die bereits bekannten Nachteile
- Mit J2SE 5.0 wurden daher **typsichere Collections** eingeführt
- Diese werden auch als generische Sammlungen (*generics*) bezeichnet
- Bei der Erzeugung eines Listenobjekts muss der Typ `T` vorgegeben werden (in spitzen Klammern hinter dem Listentyp)

```
List<T> liste = new ArrayList<T>()
```


- Der Compiler prüft dann, ob nur Objekte des erlaubten Typs `T` in die Sammlung aufgenommen werden

– Beispiel: Eine Liste nur für Angestellte

In die Liste kann nun kein Auto aufgenommen werden



```
List<Angestellter> liste = new ArrayList<Angestellter>();  
liste.add(new Angestellter("Mittelmeier", 5000.00));  
liste.add(new Vorstand("Obermeier", 100000.00, 10000.00));  
  
System.out.println(liste.get(0).berechneJahreszahlung());
```



Keine Typkonvertierung mehr erforderlich

- Häufig wird eine komplette Sammlung durchlaufen, um alle Objekte der Sammlung zu besuchen
- Für diese Aufgabe existieren spezielle Objekte
- Ein Objekt, das es ermöglicht, Sammlungen linear zu durchlaufen, wird als **Iterator** bezeichnet
- Jede konkrete Collection ist in der Lage, ein spezielles Iterator-Objekt zu erzeugen
- Die Methoden, die ein Iterator zur Verfügung stellen muss, werden über das Interface `Iterator` beschrieben
- Über diese Schnittstelle können wir alle konkreten Iterator-Objekte ansprechen
- Damit sind wir unabhängig vom konkreten Aufbau einer Sammlung

– Die Methoden der Iterator-Schnittstelle:

Methode	Bedeutung
<code>hasNext()</code>	liefert <code>true</code> , wenn noch ein nicht geliefertes Objekt in der Collection ist
<code>next()</code>	liefert eine Referenz auf das nächste Objekt
<code>remove()</code>	das zuletzt geholte Objekt wird aus der Sammlung gelöscht

– Verwendung eines Iterators

*Fortführung des Beispiels
List<Angestellter> liste*



```
Iterator<Angestellter> it = liste.iterator();
```

```
while(it.hasNext()){  
    System.out.println(it.next().berechneJahreszahlung());  
}
```

liefert einen Iterator für die Liste





holt nächstes Objekt aus der Liste

- Mit Java 5.0 wurde eine Erweiterung der `for`-Schleife eingeführt, um Sammlungen einfacher zu durchlaufen:

```
List<Angestellter> liste = new LinkedList<Angestellter>();  
liste.add(new Angestellter("Mittelmeier", 5000.00));  
liste.add(new Vorstand("Obermeier", 100000.00, 10000.00));  
  
for (Angestellter a: liste){  
    System.out.println(a.berechneJahreszahlung());  
}
```

- Der zweite Parameter muss ein Ausdruck vom Typ `java.lang.iterable` oder ein Feld sein
- Der erste Parameter muss eine Objektreferenz vom passenden Typ sein
- Die Sammlung (das Feld) wird immer komplett durchlaufen

Sortieren von Sammlungen und Felder

- Die Klassen `Arrays` und `Collections` aus dem Paket `java.util` bieten jeweils eine Methode `sort`, um Felder und Sammlungen zu sortieren  *überladen*
- Nach welchen Kriterien soll die `sort`-Methode z.B. Objekte der Klasse `Angestellter` sortieren?  *die Methode `sort` kennt die Klasse `Angestellter` nicht*
- Damit das Sortierverfahren Informationen über die natürliche Ordnung der Objekte erhalten kann, muss die Klasse die Schnittstelle `Comparable` implementieren

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

`Comparable.java`

- Die Methode `int compareTo(T o)` muss nach Konvention die folgenden Werte liefern
 - `< 0`, falls das aktuelle Objekt kleiner als das Objekt `o` ist
 - `= 0`, falls beide Objekte gleich groß sind
 - `> 0`, falls das aktuelle Objekt größer als das Objekt `o` ist

- Dabei muss gewährleistet sein, dass die folgenden Beziehungen gelten

○ $\text{sgn}(x.\text{compareTo}(y)) == -\text{sgn}(y.\text{compareTo}(x))$

Symmetrie

○ $(x.\text{compareTo}(y) > 0) \ \&\& \ (y.\text{compareTo}(z) > 0) \Rightarrow x.\text{compareTo}(z) > 0$

Dreiecksungleichung

○ $x.\text{compareTo}(y) == 0 \Rightarrow (\text{sgn}(x.\text{compareTo}(z)) == \text{sgn}(y.\text{compareTo}(z)))$

Definitheit

Signumfunktion

- Es gilt die Empfehlung, dass die `compareTo`-Methode sich bezüglich der Gleichheit von Objekten wie die `equals`-Methode verhalten soll
 - `(x.compareTo(y)==0) == (x.equals(y))`
- Der Vergleich von einzelnen ganzzahligen Attributen ist unproblematisch
 - Allerdings sollte nicht einfach die Differenz der beiden zu vergleichenden Werte als Ergebnis der `compareTo`-Methode geliefert werden *Warum?*
- Zur Vereinfachung des Vergleichs bietet die Klasse `Integer` eine statische Methode `int compare(int x, int y)`
 - Diese Methode hält sich an der von `compareTo` erwarteten Konvention
- Für die übrigen ganzzahligen Datentypen bieten die Klassen `Long`, `Short` und `Byte` ebenfalls eine `compare`-Methode

- Beispiel für einen Sortieraufruf


```
// Achtung: Sortierung des Parameters liste wird verändert
public static void sortierteAusgabe(List<Angestellter> liste){
    Collections.sort(liste);
    for(Angestellter a : liste){
        a.druckeDaten();
    }
}
```

- Aufgabe 15: Angestellte sollen aufsteigend nach ihrem Monatsgehalt sortiert werden (mit `Collections.sort`). Ergänzen Sie die Implementierung der Klasse `Angestellter` entsprechend!

- Es gibt eine sort-Methode, die eine Liste (`List`) aus vergleichbaren Objekten (die Klasse implementiert `Comparable`) als Parameter erwartet
- Daneben gibt es eine überladene sort-Methode, die als zweiten Parameter ein Objekt vom Typ `Comparator` erwartet

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
    boolean equals(Object obj);  
}
```

Comparator.java




*Vergleich von Comparatoren
(wird in der Vorlesung nicht betrachtet)*

- Die Rückgabewerte der Methode `compare` werden analog der Methode `compareTo` berechnet


– Einsatzgebiet von *Comparatoren*

- 1) Die natürliche Ordnung soll durch eine spezielle Ordnung ersetzt werden
- 2) Es sollen Objekte sortiert werden, die nicht das Interface `Comparable` implementieren



Über das Interface `Comparator<T>` ist auch ein Vergleich mit `null`-Referenzen möglich. `Comparable<T>` fordert in diesem Fall per Konvention eine `NullPointerException`

– Aufgabe 16: Programmieren Sie einen *Comparator*, der Personen aufsteigend nach dem Namen sortiert.



Auch die Klasse `String` implementiert die Schnittstelle `Comparable`

Wrapper

- Die Collections der Java-API können nur Objekte verwalten
- Für die primitiven Datentypen (`int`, `double`, ...) können die Sammlungen nicht genutzt werden
- Java stellt daher **Wrapper**-Klassen zur Verfügung, um primitive Datentypen in Objekte zu verpacken (`java.lang`)

primitiver Datentyp	Wrapper-Klasse
<code>byte</code>	<code>Byte</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>
<code>boolean</code>	<code>Boolean</code>
<code>char</code>	<code>Character</code>
<code>void</code>	<code>Void</code>

- Jede Wrapper-Klasse enthält ein Attribut vom entsprechenden primitiven Datentyp
- Der primitive Wert kann dem Konstruktor übergeben werden

```
Integer io = new Integer(10);
```

- Falls der Wert in Form einer Zeichenkette übergeben wird, findet eine entsprechende Konvertierung statt

```
Integer io = new Integer("10");
```

- Die Wrapper-Klassen besitzen Methoden, um die gespeicherten Werte in verschiedene primitive Typen zurückzuliefern, z.B.


```
int i = io.intValue();  
String is = io.toString();
```

- Die Wrapper-Klassen enthalten eine Reihe statischer Methoden für die Typ-Konvertierung
- Die Klasse `Integer` besitzt u.a. die folgenden Methoden:

Statische Methode	Bedeutung
<code>int parseInt(String s)</code>	Wandelt die Zeichenkette <code>s</code> in eine Zahl vom Typ <code>int</code>
<code>String toString(int i)</code>	Wandelt die ganze Zahl <code>i</code> in einen <code>String</code>
<code>Integer valueOf(int i)</code>	Liefert für <code>i</code> eine Instanz vom Typ <code>Integer</code>
<code>Integer valueOf(String s)</code>	Wandelt <code>s</code> in eine Zahl vom Typ <code>int</code> und liefert für diese Zahl eine Instanz vom Typ <code>Integer</code>


- Bis zur Java Version 5.0 musste die Typumwandlung manuell vorgenommen werden

```
List zahlen = new ArrayList();  
zahlen.add(new Integer(1));
```

 *int-Wert wird in ein Objekt verpackt*

hier wird aus dem Objekt wieder ein Integer gemacht

```
Integer io = (Integer) zahlen.get(0);  
int i = io.intValue();
```

 *hier wird aus dem Integer-Objekt der int-Wert geholt*

- Das ist umständlich und fehleranfällig!

- Ab Java Version 5.0 gibt es das *Autoboxing*
- Die erforderlichen Typumwandlungen zwischen primitiven Typen und Wrapper –Klassen werden automatisch vom Compiler vorgenommen
- Der Unterschied zwischen primitiven Datentypen und Wrapper-Klassen verschwimmt

```
List<Integer> zahlen = new ArrayList<Integer>();  
zahlen.add(1); 1 wird automatisch in ein Integer-Objekt verpackt  
  
int i = zahlen.get(0);
```


Formatter

- Für eine formatierte Textausgabe kann die Klasse `java.util.Formatter` verwendet werden (ab J2SE 5.0)
- Für eine einfache Ausgabe auf die Konsole ist die Nutzung der Klasse `Formatter` etwas umständlich

```
Formatter f = new Formatter(System.out);  
f.format("Programmierkurs 1\n");  
f.close();
```

Standardausgabestrom

*Datenströme werden
noch behandelt*

- Die Klassen `String` und `PrintStream` bieten eine statische Methode `format`
- Das statische Attribut `out` der Klasse `System` stellt uns direkt den Standardausgabestrom (`PrintStream`) zur Verfügung

```
System.out.format("Programmierkurs 1\n");
```

- Die Methode `format` ist überladen und kann mit mehreren Parametern aufgerufen werden
- Für uns ist die folgende Form am wichtigsten

```
static String format(String f, Object... args)
```

- Bei `f` handelt es sich um einen `String`, der einen Teil der Ausgabe und optionale Formatangaben (*format specifiers*) enthält
- Formatangaben haben den folgenden Aufbau

```
%[Argument-Index$][Flags][Width][.Precision]Conversion
```

- Der Formatstring `f` wird von links nach rechts ausgewertet
 - Für eine Formatangabe muss es einen passenden Parameter in den nachfolgenden Argumenten (`args`) geben
 - Die Formatangabe in `f` wird dann durch das entsprechend formatierte Argument ersetzt

- Mit `Conversion` wird der Datentyp vorgegeben

<i>Conversion</i>	Datentyp (Formatierung)
b	boolean (ausgeschrieben)
c	char
d	Ganzzahl (Dezimaldarstellung)
x	Ganzzahl (Hexadezimaldarstellung)
f	Fließkommazahl
e	Fließkommazahl (mit Exponent)
t	Datums-/Zeitangabe
s	String

Ausschnitt

- Mit *Flags* können abhängig vom Datentyp weitere Ausgabeoptionen eingestellt werden

<i>Flag</i>	Ausgabeoption
–	linksbündig
+	Vorzeichen immer ausgeben
0	Zahlen mit Nullen auffüllen
,	Zahlen mit Tausenderpunkten ausgeben

Ausschnitt

- Mit *width* wird die Breite der Ausgabe in Zeichen vorgegeben. Wenn weniger Zeichen benötigt werden, dann wird mit Leerzeichen aufgefüllt
- Mit *.Precision* kann für Fließkommazahlen die Anzahl der Nachkommastellen vorgegeben werden

- Die Formatangabe %n erzeugt einen Zeilenumbruch und erfordert kein Argument
- Zusätzlich kann der Formatstring auch Escape-Sequenzen enthalten

Sequenz	Bedeutung
\n	Zeilenumbruch
\"	Anführungszeichen "
\\	\

Ausschnitt

– Beispiel:

```
for(int i=0; i < 8; i++){  
    System.out.format("2 hoch %d ist %f%n", i, Math.pow(2,i));  
}
```

Handwritten note in blue: 4. Stelle (nach Komma) ist die Nachkommastelle

```
2 hoch 0 ist 1,000000  
2 hoch 1 ist 2,000000  
2 hoch 2 ist 4,000000  
2 hoch 3 ist 8,000000  
2 hoch 4 ist 16,000000  
2 hoch 5 ist 32,000000  
2 hoch 6 ist 64,000000  
2 hoch 7 ist 128,000000
```

- Aufgabe 17: Modifizieren Sie die Formatangaben im obigen Beispiel, so dass keine Nachkommastellen ausgegeben werden und die Ausgabe der Zweierpotenzen rechtsbündig auf einer Breite von 4 Zeichen erfolgt.

- Mit `%i$` kann gezielt das *i*-te Argument ausgewählt werden. Der Formatstring hat den Index 0
- Aufgabe 18: Welche Ausgabe erzeugt die folgende Anweisung?

```
System.out.format("%2$s\n%1$s\n%3$s", "Beta", "Alpha", "Gamma");
```

alpha
Beta
Gamma

- Die Arbeitsweise der Methode `format` orientiert sich sehr stark an der `printf`-Funktion der Sprache C
- Daher wird die Methode `format` alternativ auch unter der Bezeichnung `printf` angeboten

```
System.out.printf("%2$s\n%1$s\n%3$s", "Beta", "Alpha", "Gamma");
```

Alpha
Beta
Gamma

Scanner (Tastatureingaben)

- Eingaben von der Tastatur werden über den Eingabestrom `System.in` geliefert
- Die Folge von eingegebenen Zeichen muss nun geeignet zusammengefasst werden
 - als ganze Zahl (`byte`, `short`, `int`, `long`)
 - als Fließkommazahl (`float`, `double`)
 - als Zeichenkette (`String`)
- Diese Aufgabe übernimmt die Klasse `Scanner`
 - Ein Text-Scanner, der primitive Datentypen und Strings parsen kann
 - Verwendung von regulären Ausdrücken ist möglich

– Beispiel:

```
System.out.printf("Taschenrechner%n");  
Scanner sc = new Scanner(System.in);
```

```
System.out.println("Zahl 1:");  
int zahl1 = sc.nextInt();
```

← Exceptions sind möglich

```
System.out.println("Zahl 2:");  
int zahl2 = sc.nextInt();
```

*← für weitere Typen existieren
analoge Methoden -> siehe API*

```
System.out.printf("%d * %d = %d%n", zahl1, zahl2,  
zahl1*zahl2);
```

- Alternativ gibt es eine einfache Möglichkeit, über die statische Methode

Exkurs

```
String showInputDialog(Component parent, Object message)
```

der Klasse `javax.swing.JOptionPane` Tastatureingaben über ein Fenster entgegenzunehmen

- da wir noch keine übergeordneten Fenster berücksichtigen müssen, können wir `parent = null` setzen
- für `message` übergeben wir einen `String`, der eine Eingabeaufforderung enthält
- Die Methode `showInputDialog` liefert die Tastatureingabe immer als `String`, daher kann eine Typkonvertierung erforderlich sein

– Beispiel:

```
String eingabe = JOptionPane.showInputDialog(null, "Operand");  
int x = Integer.parseInt(eingabe);  
  
Formatter formatter = new Formatter();  
formatter.format("%d * %d = %d", x, x , x*x);  
  
JOptionPane.showMessageDialog(null, formatter.toString());  
formatter.close();
```

Exceptions sind möglich

Anzeigefenster, analog showInputDialog

