Einführung in die Informatik

Control Structures and Iterators

---

if, while, for und Iteratoren

Wolfram Burgard

Motivation

- Bisher bestanden die Rümpfe unserer Methoden aus einzelnen Statements, z.B. Wertzuweisungen oder Methodenaufrufen.
- Es gibt bisher keine Möglichkeit, Statements nur in Abhängigkeit bestimmter Umstände auszuführen.
- Durch **bedingte Anweisungen und Schleifen** können wir **flexiblere Methoden** schreiben und **deutlich mächtigere Modelle** entwickeln.

Das `if`-Statement

- Java stellt mit dem `if`-**Statement** eine Form der **bedingten Anweisung** zur Verfügung.
- Mit Hilfe des `if`-**Statements** können wir eine **Bedingung** testen und, je nach Ausgang des Tests, eine von zwei Anweisungen durchführen.

```
if (condition)
    statement1
else
    statement2
```

Das `if`-Statement

- Beispiel:

```
if (x == 2)
        result = 4;
else
        result = 5 * x;
```

- Zeile 1 enthält den Test, den wir ausführen.
- Zeile 2 enthält das Statement, das bei erfolgreichem Test ausgeführt wird.
- Zeile 3 enthält das Schlüsselwort `else` und läutet den Teil ein, der ausgeführt wird, wenn der Test fehlschlägt.
- Zeile 4 enthält das Statement, welches bei negativem Ausgang des Tests ausgeführt wird.

Mehrere Anweisungen in `if`-Statements

- In der Grundversion des `if`-Statements können nur einzelne Statements im `then`**-Teil** und `else`-Teil verwendet werden.
- Sollen **mehrere Statements** ausgeführt werden, muss man diese zu einem **Block zusammenfassen**, indem man sie in geschweifte Klammern (`{` und `}`) einschließt.

```
if (x>y)  {
    System.out.print(x);
    System.out.print(" is greater than ");
    System.out.println(y);
}
else      {
    System.out.print(x);
    System.out.print(" is not greater than ");
    System.out.println(y);
}
```

zusammen-gesetzte Statements

Multiple `if`-Statements

- Java erlaubt es auch das `else`-**Statement** wegzulassen, d.h. es wird kein Code ausgeführt, wenn die Bedingung falsch ist.
- Verschiedene `if`-**Statements** können auch **geschachtelt** werden

```
if (X > 2)
     if (X < 5)
               System.out.println("X ist größer als 2 und kleiner als 5");
     else
               System.out.println("X ist größer gleich 5");
```

- Des Weiteren sind **kaskardierte** if-**Statements** möglich

```
if (X > 2)
     System.out.println("X ist größer als 2");
else if (X < 0)
     System.out.println("X ist kleiner als 0");
else
     System.out.println("X ist größer gleich 0 und kleiner gleich 2");
```

Zu welchem `if` gehört ein `else`?

- **Ein `else` gehört immer zu dem letzten `if`, für das noch ein `else` fehlt**.
- Unser Beispiel entspricht daher:

```
if (X > 2) {
   if (X < 5)
      System.out.println("X ist größer als 2 und kleiner als 5");
   else
      System.out.println("X ist größer als 5");
}
```

- Hierbei sollte die **Einrückung der Statements** die **Zuordnung der Statements widerspiegeln**.

Bedingungen in `if`-Statements

- Die **Bedingung** eines `if`-Statements muss ein **Ausdruck** sein, der entweder wahr oder falsch ist.
- Im Moment schränken wir uns auf Vergleiche zwischen Zahlwerten ein.
- Java stellt folgende **Operatoren für den Vergleich von Zahlen** zur Verfügung:

Operator	Bedeutung
<	kleiner
>	größer
==	gleich
<=	kleiner gleich
>=	größer gleich
!=	ungleich

Der Typ `boolean`

- Für logischen Werte **wahr** und **falsch** gibt es in Java einen primitiven Datentyp `boolean`
- Die **möglichen Werte** von Variablen dieses Typs sind `true` und `false`.
- Wie Integer-Variablen kann man auch Variablen vom Typ `boolean` vereinbaren.
- Diesen Variablen können **Werte logischer Ausdrücke** zugewiesen werden.

Anwendung vom Typ `boolean`

Typische Situation:

```
boolean hasOvertime;
if (hours > 40)
       hasOvertime = true;
else
       hasOvertime = false;
...
if (hasOvertime)     // same as: if (hasOvertime == true)
       ...
```

Alternative:

```
boolean hasOvertime;
hasOvertime = (hours > 40);
...
if (hasOvertime)
       ...
```

Logische Operatoren und zusammengesetzte logische Ausdrücke

- Häufig besteht eine Bedingung aus **mehreren Teilbedingungen**, die gleichzeitig erfüllt sein müssen.
- Java erlaubt es, mehrere Tests mit Hilfe **logischer Operatoren** zu einem Test zusammenzusetzten:

```
hours > 40 && hours <= 60
```

- Der `&&`**-Operator** repräsentiert das logische **Und**.
- Der `||`**-Operator** realisiert das logische **Oder**.
- Der `!`**-Operator** realisiert die **Negation**.

Zusammengesetze `if`-Anweisungen und Operatoren

- `if`-Anweisungen mit Operatoren können auch zerlegt werden in einzelne `if`-Anweisungen

```
if (condition1)
    statement
else if (condition2)
    statement
```

```
if (condition1 || condition2)
    statement
```

```
if (condition1)
    if (condition2)
        statement
```

```
if (condition1 && condition2)
    statement
```

Präzedenzregeln für logische Operatoren

- Der `!`**-Operator** hat die höchste Präzedenz von den logischen Operatoren. Zweithöchste Präzedenz hat der `&&`**-Operator**. Schließlich folgt der `||`**-Operator**.
- Der Ausdruck

```
if (this.hours < hours ||
    this.hours == hours && this.minutes < minutes)
```

  hat daher die gleiche Bedeutung wie

```
if (this.hours < hours ||
    (this.hours == hours && this.minutes < minutes))
```

- **Durch Klammern werden (logische) Ausdrücke leichter lesbar!**

Wiederholungsanweisungen (Schleifen)

- Neben bedingten Anweisungen ist es in der Praxis häufig erforderlich, ein und dieselbe Anweisung oder Anweisungsfolge auf vielen Objekten zu wiederholen.
- Beispielsweise möchte man das Gehalt für mehrere tausend Mitarbeiter berechnen.
- In Java gibt es mit dem **`while`-Statement** eine weitere Möglichkeit die Programmausführung zu beeinflussen.
- Insbesondere lassen sich mit dem **`while`-Statement** Anweisungsfolgen beliebig oft wiederholen.

Das `while`-Statement

- Die allgemeine Form einer **`while`-Schleife** ist:

```
while(condition)
    body
```

- Dabei sind die **Bedingung** `(condition)` und der **Rumpf** `(body)` ebenso wie bei der `if`-Anweisung aufgebaut.
- Die **Bedingung** im **Schleifenkopf** ist eine logischer Ausdruck vom Typ `boolean`.
- Der **Rumpf** ist ein einfaches oder ein zusammengesetztes **Statement**.

Ausführung der `while`-Anweisung

1. Es wird **zunächst die Bedingung überprüft**.
2. Ist der **Wert des Ausdrucks `false`**, wird die **Schleife beendet** . Die Ausführung wird dann mit der nächsten Anweisung fortgesetzt, die unmittelbar auf den Rumpf folgt.
3. Wertet sich der **Ausdruck** hingegen zu **`true`** aus, so wird der **Rumpf der Schleife ausgeführt**.
4. Dieser Prozess wird **solange wiederholt, bis** in Schritt 2. der Fall eintritt, dass **sich der Ausdruck zu `false` auswertet**.

Nach Beendigung einer `while`-Schleife gilt somit immer die Negation ihrer Bedingung.

Beispiel: Einlesen aller Zeilen von `www.whitehouse.gov`

- Wir wollen ein Programm schreiben, das alle Zeilen einer Web-Seite einliest.
- Wie können wir feststellen, dass wir am **Ende der Datei** angekommen sind?
- Offensichtlich kann **am Ende einer Datei keine Zeile mehr eingelesen werden**.
- Um dies zu signalisieren liefert die **`readline`**-Methode einen speziellen Wert **`null`** zurück, der **repräsentiert**, dass eine **Referenz-Variable kein Objekt referenziert**.

Beispiel: Einlesen aller Zeilen von www.whitehouse.gov

```
import java.net.*;
import java.io.*;

class WHWWWLong {
    public static void main(String[] arg) throws Exception {
        URL u = new URL("http://www.whitehouse.gov/");
        BufferedReader whiteHouse = new BufferedReader(
            new InputStreamReader(u.openStream()));
        String line = whiteHouse.readLine();  // Read first object.
        while (line != null){                 // Something read?
            System.out.println(line);         // Process object.
            line = whiteHouse.readLine();     // Get next object.
        }
    }
}
```

Anwendung der `while`-Schleife zur Approximation

Viele Werte (Nullstellen, Extrema, ...) **lassen sich** (in Java) **nicht durch geschlossene Ausdrücke berechnen**, sondern müssen **durch geeignete Verfahren approximiert** werden.

Beispiel: Approximation von $\sqrt[3]{x}$

Ein beliebtes Verfahren ist die Folge $x_{n+1} = x_n - \frac{x_n^3 - x}{3x_n^2}$,

wobei $x_1 \neq 0$ ein beliebiger Startwert ist.

Mit $n \to \infty$ konvergiert[a] x_n gegen $\sqrt[3]{x}$, d.h. $\lim_{n\to\infty} x_n = \sqrt[3]{x}$

[a] Sofern kein $x_n = 0$

Muster einer Realisierung

- Zur näherungsweisen Berechnung verwenden wir eine `while`-Schleife.
- Dabei müssen wir **zwei Abbruchkriterien** berücksichtigen:
  1. Das **Ergebnis ist hinreichend genau**, d.h. x_{n+1} und x_n unterscheiden sich nur geringfügig.
  2. Um zu vermeiden, dass die Schleife nicht anhält, weil die gewünschte Genauigkeit nicht erreicht werden kann, muss man die **Anzahl von Schleifendurchläufen begrenzen**.
- Wir müssen also solange weiter rechnen wie folgendes gilt:

```
Math.abs((xnPlus1 - xn)) >= maxError && n < maxIterations
```

Das Programm zur Berechnung der Dritten Wurzel

```
import java.io.*;

class ProgramRoot {
    public static void main(String arg[]) throws Exception{
        BufferedReader br = new BufferedReader(
                            new InputStreamReader(System.in));
        int n = 1, maxIterations = 1000;
        double maxError = 1e-6, xnPlus1, xn = 1, x;

        x = Double.valueOf(br.readLine()).doubleValue();
        xnPlus1 = xn - ( xn * xn * xn - x) / (3 * xn * xn);
        while (Math.abs((xnPlus1 - xn)) >= maxError && n < maxIterations){
            xn = xnPlus1;
            xnPlus1 = xn - ( xn * xn * xn - x) / (3 * xn * xn);
            System.out.println("n = " + n + ": " + xnPlus1);
            n = n+1;
        }
    }
}
```

Anwendung des Programms

Eingabe: `-27`

```
n = 1: -5.685155555555555
n = 2: -4.068560488977107
n = 3: -3.256075689936079
n = 4: -3.019611247370567 4
n = 5: -3.000127091992528 7
n = 6: -3.000000053838210
n = 7: -3.0
Process ProgramRoot finished
```

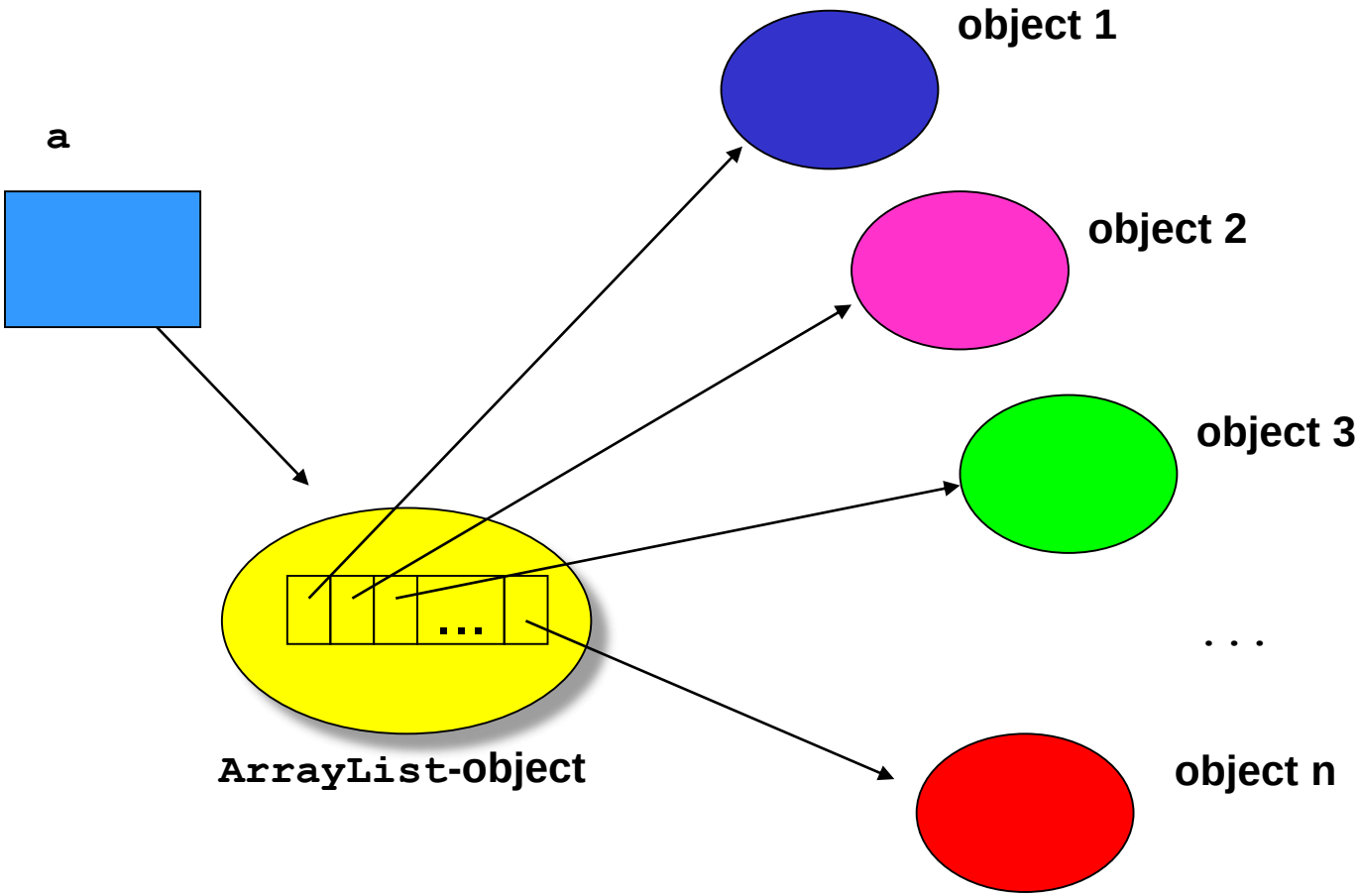
Eingabe: 10^{90}

```
n = 1: 2.2222222222222218E89
n = 2: 1.4814814814814810E89
n = 3: 9.8765432098765410E88
n = 4: 6.5843621399176940E88
...
n = 996: 9.999999999999999E29
n = 997: 1.0E30
n = 998: 9.999999999999999E29
n = 999: 1.0E30
Process ProgramRoot finished
```

Kollektionen mehrere Objekte: Die Klasse `ArrayList`

- Mit `ArrayList` stellt Java eine Klasse zur Verfügung, die eine **Zusammenfassung von unter Umständen auch verschiedenen Objekten in einer Sequenz** erlaubt.
- **Grundoperationen für Kollektionen** von Objekten sind:
  - das **Erzeugen** einer Kollektion (mit dem Konstruktor),
  - das **Hinzufügen** von Objekten in die Kollektion,
  - das **Löschen** von Objekten aus der Kollektion, und
  - das **Verarbeiten** von Objekten in der Kollektion.

Kollektion von (eventuell unterschiedlichen) Objekten mit der Klasse `ArrayList`



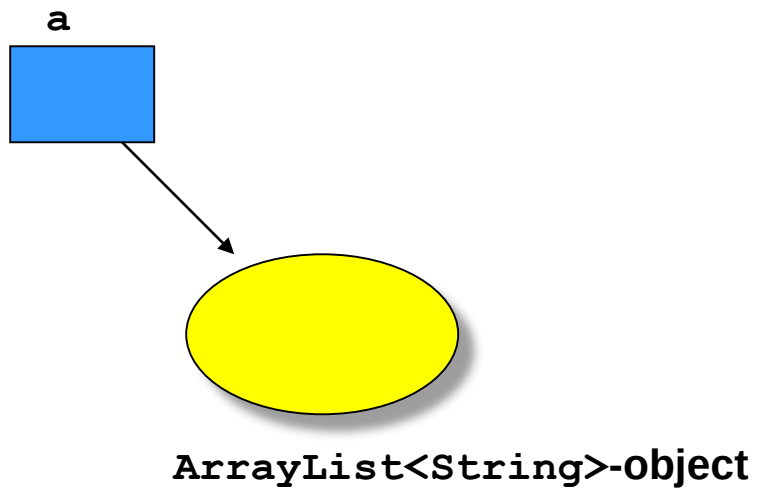
Erzeugen eines `ArrayList`-Objektes

- Wie auch andere Klassen werden `ArrayList`-Objekte mit dem Konstruktor der `ArrayList`-Klasse erzeugt.
- Der Konstruktor von `ArrayList` hat keine Argumente, allerdings sollte **zusätzlich angeben werden, welche Objekte die Kollektion speichern soll**. Dies wird mittels spitzer Klammern <> realisiert.
- Beispielsweise:

```
ArrayList<Integer> a1 = new ArrayList<Integer>();
ArrayList<String>  a2 = new ArrayList<String>();
```

Erzeugen eines `ArrayList`-Objektes

- Heute betrachten wir `ArrayList`-Objekte, die `String`-Objekte speichern
- `ArrayList<String> a = new ArrayList<String>();`
- Wirkung des Konstruktors:



Hinzufügen von Objekten zu einem `ArrayList`-Objekt

Um Objekte zu einem `ArrayList`-Objekt hinzuzufügen, verwenden wir die Methode `add`.

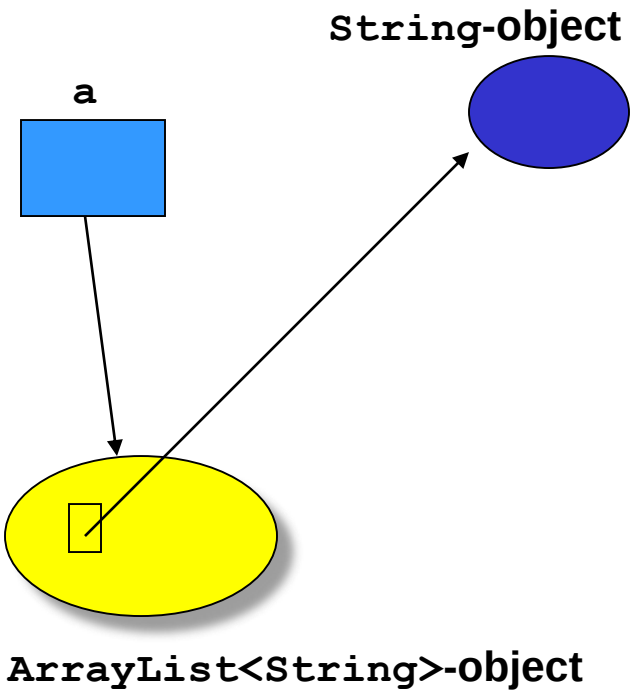
Dieser Methode geben wir als Argument das hinzuzufügende Objekt mit.

Das folgende Programm liest eine Sequenz von `String`-Objekten ein und fügt sie unserem `ArrayList`-Objekt hinzu:

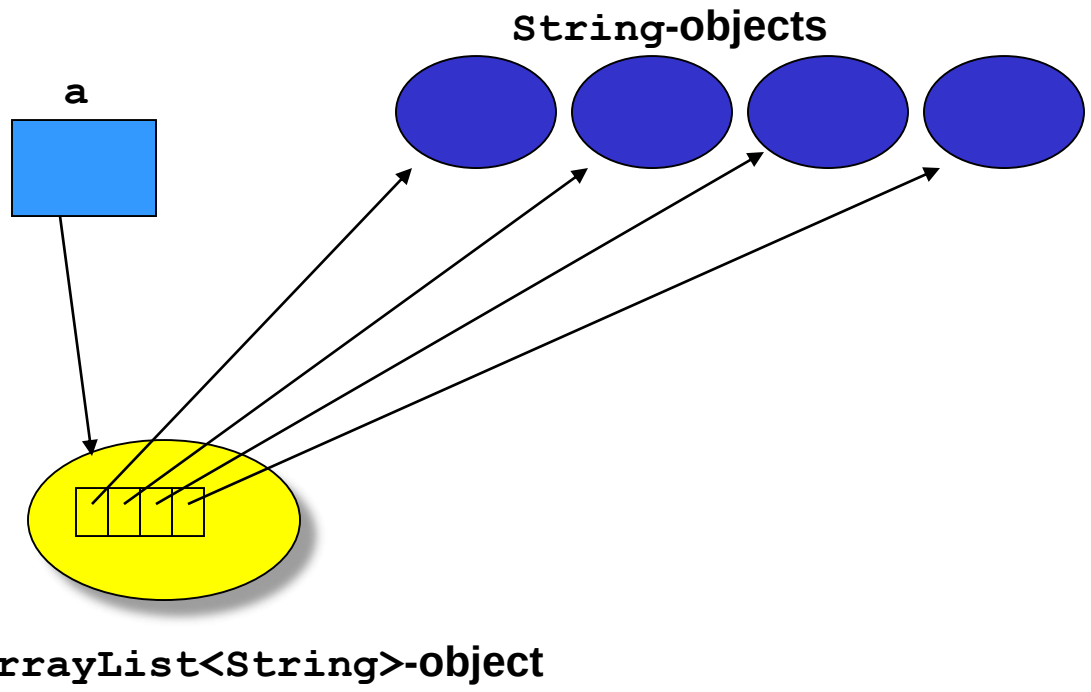
```
ArrayList<String> a = new ArrayList<String>();
String s = br.readLine(); // Read first String object
while (s != null){        // Something read?
   a.add(s);              // Processing adds s to list
   s = br.readLine();     // Read next String object
}
```

Anwendung dieses Programmstücks

1 Aufruf von `add`



4 Aufrufe von `add`



Unser `ArrayList<String>`-Objekt enthält lediglich Objekte der Klasse `String`.

Durchlauf durch eine ArrayList

- Der Prozess des **Verarbeitens aller Objekte einer Kollektion** wird auch **Durchlauf** genannt.
- Ziel ist es, eine (von der Anwendung abhängige) Operation auf allen Objekten der Kollektion auszuführen.
- Dazu verwenden wir eine **`while`-Schleife** der Form:

```
while (es gibt noch Objekte, die zu besuchen sind)
   besuche das nächste Objekt
```

- Die **zentralen Aufgaben**, die wir dabei durchführen müssen, sind:
  - auf die **Objekte einer Kollektion zugreifen**,
  - **zum nächsten Element** einer Kollektion **übergehen** und
  - **testen, ob es noch weitere Objekte gibt**, die besucht werden müssen.

Wie kann man Durchläufe realisieren?

- Offensichtlich müssen diese **Funktionen von jeder Kollektionsklasse realisiert werden**.
- Daher sollten die entsprechenden Methoden möglichst so sein, dass sie **nicht von der verwendeten Kollektionsklasse** abhängen.
- Weiter ist es wünschenswert, dass **sich jede Kollektionsklasse an einen Standard** bei diesen Methoden **hält**.
- Auf diese Weise kann man sehr **leicht zu anderen Kollektionsklassen übergehen, ohne dass man das Programm ändern muss**, welches die Kollektionsklasse verwendet.

Iteratoren

Java bietet das **Interface `Iterator`** zur Realisierung von **Durchläufen durch `ArrayList`-Objekte** und andere Kollektionsklassen an.

Jede Kollektionsklasse stellt eine **Methode zur Erzeugung eines `Iterator`-Objektes** zur Verfügung.

Die Klasse **`ArrayList`** stellt eine Methode **`iterator()`** zur Verfügung. Diese liefert eine Referenz auf ein **`Iterator`**-Objekt. Ihr Prototyp ist:

```
Iterator<Object> iterator()    // Liefert einen Iterator für ein
                               // ArrayList<Object>
```

Die entsprechende **`Interator`** Klasse wiederum bietet die folgenden Methoden

```
boolean hasNext()      // True, falls es weitere Elemente gibt
Object next()          // Liefert das nächste Objekt
void remove()          // Entfernt das zuletzt betrachtete Element
```

Der Return-Type von `next`

- Im Prinzip muss die Methode **`next()`** **Referenzen auf Objekte beliebiger Klassen** liefern – je nach dem, welche Klasse im **`ArrayList`**-Objekt gespeichert wird.
- Um eine breite Anwendbarkeit realisieren zu können, müssen Klassen wie **`ArrayList`** oder **`Iterator`** diese **Flexibilität** haben.

Der Return-Type von `next()`

- Bei ArrayListen gibt man die zu speichernden Elemente an, z.B. `ArrayList<Integer>` oder `ArrayList<String>`
- Die gleiche Technik wird auch bei Iteratoren verwendet, d.h.: `Iterator<Integer>` oder `Iterator<String>`
- Somit weiß eine Methode wie **`next()`**, welchen Typ sie zurückgeben muss.
- **Castings** entfallen somit.

Durchlauf durch ein `ArrayList`-Objekt

Um einen Durchlauf durch unser `ArrayList<String>`-Objekt `list` zu realisieren, gehen wir nun wie folgt vor:

```
while (es gibt weitere Elemente) {
   x = hole das nächste Element
   verarbeite x
}
```

Dies wird nun überführt zu

```
Iterator<String> e = a.iterator();
while (e.hasNext()) {
   String s = e.next();
   System.out.print(s);
}
```

Anwendung von `ArrayList` zur Modellierung von Mengen

- Auf der Basis solcher Kollektionsklassen wie **`ArrayList`** lassen sich nun andere Kollektionsklassen definieren.
- Im folgenden modellieren wir Mengen mit Hilfe der **`ArrayList`**-Klasse.
- Ziel ist die Implementierung einer eigenen Klasse **`Set`** einschließlich typischer Mengen-Operationen.

Festlegen des Verhaltens der `Set`-Klasse

In unserem Beispiel wollen wir die folgenden Mengenoperationen bzw. Methoden zur Verfügung stellen:

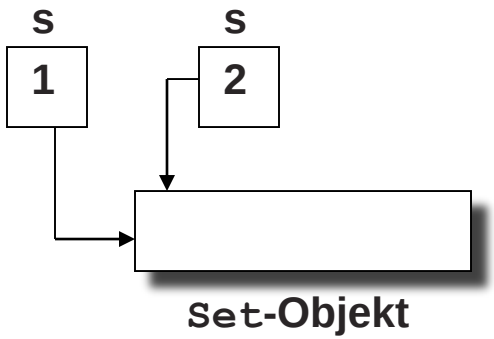
- Den `Set`-Konstruktor
- `contains` (Elementtest)
- `isEmpty` (Test auf die leere Menge)
- `add` (hinzufügen eines Elements)
- `copy` (Kopie einer Menge erzeugen)
- `size` (Anzahl der Elemente)
- `iterator` (Durchlauf durch eine Menge)
- `union` (Vereinigung)
- `intersection` (Durchschnitt) Alle Elemente ausgeben
- `toString` (Ausgabe der Elemente)

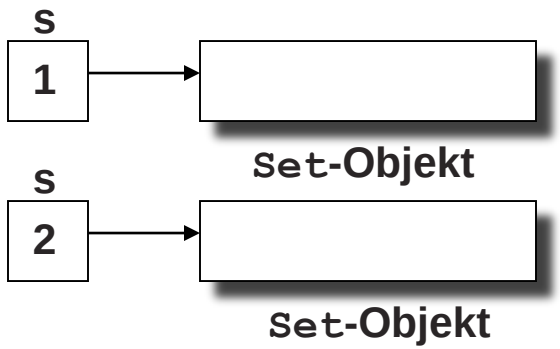
Notwendigkeit der `copy`-Operation

Der Effekt der Anweisung `s2 = s1 = new Set()` ist, dass es zwei Referenzen auf ein- und dasselbe `Set`-Objekt gibt:

Da Methoden wie `add` ein `Set`-Objekt verändern, benötigen wir eine Kopier-Operation um eine Menge zu speichern.

Nach der Anweisung `s2 = s1.copy()` gibt es zwei Referenzen auf zwei unterschiedliche Objekte mit gleichem Inhalt.






```
s2 = s1 = new Set();
```

```
s2 = s1.copy();
```

Festlegen der Schnittstellen

Prototypen der einzelnen Methoden:

```
public Set()
public boolean isEmpty()
public int size()
public boolean contains(Object o)
public void add(Object o)
public Set copy()
public Set union(Set s)
public Set intersection(Set s)
public Iterator<Object> iterator()
public String toString()
```

Ein typisches Beispielprogramm

```
class UseSet {
   public static void main(String [] args) {
        Set s1 = new Set();
        s1.add("A");
        s1.add("B");
        s1.add("C");
        s1.add("A");
        System.out.println(s1);
        Set s2 = new Set();
        s2.add("B");
        s2.add("C");
        s2.add("D");
        s2.add("D");
        System.out.println(s2);
        System.out.println(s1.union(s2));
        System.out.println(s1.intersection(s2));
   }
}
```

Das Skelett der `Set`-Klasse

```
class Set {
   public Set() {... };
   public boolean isEmpty() {... };
   public int size() {... };
   public boolean contains(Object o) {... };
   public void add(Object o) {... };
   public Set copy() {... };
   public Set union(Set s) {... };
   public Set intersection(Set s) {... };
   public Iterator<Object> iterator() {... };
   public String toString() {... };
   ...

   private ArrayList<Object> theElements;
}
```

Implementierung der Methoden (1)

- Der **Konstruktor** ruft lediglich die entsprechende Methode der `ArrayList`-Klasse auf:

```
public Set() {
    this.theElements = new ArrayList<Object>();
}
```

- Die **Methoden** `size` **und** `empty` nutzen ebenfalls vordefinierte Methoden der Klasse `ArrayList`:

```
public boolean isEmpty() {
    return this.theElements.isEmpty();
}

public int size() {
    return this.theElements.size();
}
```

Implementierung der Methoden (2)

- Um alle **Elemente der Menge aufzuzählen**, müssen wir eine Methode `iterator` realisieren:

```
Iterator<Object> iterator() {
    return this.theElements.iterator();
}
```

- Die `copy`-**Methode** muss alle Elemente des `ArrayList`-Objektes durchlaufen und sie einem neuen `Set`-Objekt hinzufügen:

```
public Set copy() {
    Set destSet = new Set();
    Iterator<Object> e = this.iterator();
    while (e.hasNext())
        destSet.add(e.next());
    return destSet;
}
```

Implementierung der Methoden (3)

- Da Mengen jeden Wert höchstens einmal enthalten, müssen wir vor dem **Einfügen** prüfen, ob der entsprechende Wert bereits enthalten ist:

```
public void add(Object o) {
   if (!this.contains(o))
        this.theElements.add(o);
}
```

Implementierung der Methoden (4)

- Um die **Vereinigung von zwei Mengen** zu berechnen, kopieren wir die erste Menge und fügen der Kopie alle noch nicht enthaltenen Elemente aus der zweiten Menge hinzu.

```
public Set union(Set s) {
    Set unionSet = s.copy();
    Iterator<Object> e = this.iterator();
    while (e.hasNext())
        unionSet.add(e.next());
    return unionSet;
}
```

Implementierung der Methoden (5)

- Um den **Durchschnitt** von zwei Mengen zu berechnen, starten wir mit der leeren Menge. Dann durchlaufen wir das Empfänger-`Set` und fügen alle Elemente zu der neuen Menge hinzu, sofern sie auch in dem zweiten `Set`-Objekt vorkommen.

```
public Set intersection(Set s) {
    Set interSet = new Set();
    Iterator<Object> e = this.iterator();
    while (e.hasNext()) {
        Object elem = e.next();
        if (s.contains(elem))
            interSet.add(elem);
    }
    return interSet;
}
```

Implementierung der Methoden (6)

Um zu **testen, ob ein Objekt in einer Menge enthalten ist**, müssen wir einen Durchlauf realisieren. Dabei testen wir in jedem Schritt, ob das gegebene Objekt mit dem aktuellen Objekt in der Menge übereinstimmt:

- Hierbei ist zu beachten, dass der Gleichheitstest `==` lediglich testet, ob der Wert von zwei Variablen gleich ist, d.h. bei Referenzvariablen, ob sie **dasselbe** Objekt referenzieren (im Gegensatz zu „das gleiche“).
- Um beliebige Objekte einer Klasse miteinander vergleichen zu können, stellt die Klasse `Object` eine Methode `equals` zur Verfügung.
- Spezielle Klassen wie z.B. `Integer` oder `String` aber auch programmierte Klassen können ihre eigene `equals`-Methode bereitstellen.
- Im Folgenden gehen wir davon aus, dass eine solche Methode stets existiert.

Implementierung der Methoden (6)

- Daraus resultiert die folgende Implementierung der Methode `contains`:

```
public boolean contains(Object o) {
        Iterator<Object> e = this.iterator();
        while (e.hasNext()) {
           Object elem = e.next();
           if (elem.equals(o))
               return true;
        }
        return false;
    }
```

Implementierung der Methoden (7)

Um die **Elemente auszugeben**, verwenden wir ebenfalls wieder einen Durchlauf. Dabei gehen wir erneut davon aus, dass die Klasse des referenzierten Objektes (wie die `Object`-Klasse) eine Methode `toString` bereitstellt.

- Prinzipiell gibt es hierfür verschiedene Alternativen.
- Eine offensichtliche Möglichkeit besteht darin, eine Methode `print(PrintStream ps)` zu implementieren.
- In Java gibt es aber eine elegantere Variante: Es genügt eine Methode `toString()` zu realisieren.
- Diese wird immer dann aufgerufen, wenn ein `Set`-Objekt als Empfänger-Objekt einer `print`-Methode ist.

Die Methode `toString()`

```
public String toString(){
    String s = "[";
    Iterator<Object> e = this.iterator();
    if (e.hasNext())
       s += e.next().toString();
    while (e.hasNext())
       s += ", " + e.next().toString();
    return s + "]";
}
```

Die komplette Klasse `Set`

```
import java.io.*;
import java.util.*;
class Set {
public Set() {
          this.theElements = new ArrayList<Object>();
    }
    public boolean isEmpty() {
             return this.theElements.isEmpty();
    }
    public int size() {
             return this.theElements.size();
    }
    Iterator<Object> iterator() {
             return this.theElements.iterator();
    }
    public boolean contains(Object o) {
             Iterator<Object> e = this.iterator();
             while (e.hasNext()) {
                 Object elem = e.next();
                 if (elem.equals(o))
                     return true;
             }
             return false;
    }
    public void add(Object o) {
             if (!this.contains(o))
             this.theElements.add(o);
    }
    public Set copy() {
             Set destSet = new Set();
             Iterator<Object> e = this.iterator();
             while (e.hasNext())
                destSet.add(e.nextt());
             return destSet;
       }
       public Set union(Set s) {
             Set unionSet = s.copy();
             Iterator<Object> e = this.iterator();
             while (e.hasNext())
                unionSet.add(e.next());
             return unionSet;
       }
       public Set intersection(Set s) {
             Set interSet = new Set();
             Iterator<Object> e = this.iterator();
             while (e.hasNext()) {
                Object elem = e.next();
                if (s.contains(elem))
                    interSet.add(elem);
             }
             return interSet;
       }
       void removeAllElements() {
             this.theElements.removeAllElements();
       }
       public String toString(){
             String s = "[";
             Iterator<Object> e = this.iterator();
             if (e.hasNext())
                s += e.next().toString();
             while (e.hasNext())
                s += ", " + e.next().toString();
             return s + "]";
       }
       private ArrayList<Object> theElements;
}
```

Unser Beispielprogramm (erneut)

```
class UseSet {
    public static void main(String [] args) {
        Set s1 = new Set();
        s1.add("A");
        s1.add("B");
        s1.add("C");
        s1.add("A");
        System.out.println(s1);
        Set s2 = new Set();
        s2.add("B");
        s2.add("C");
        s2.add("D");
        s2.add("D");
        System.out.println(s2);
        System.out.println(s1.union(s2));
        System.out.println(s1.intersection(s2));
    }
}
```

Ausgabe des Beispielprogramms

```
java useSet

[A, B, C]
[B, C, D]
[B, C, D, A]
[B, C]

Process useSet finished
```

Eine generische Klasse GenericSet

```
import java.util.*;
class GenericSet <E> {
    public GenericSet() {
            this.theElements = new ArrayList<E>();
    }
    public boolean isEmpty() {
            return this.theElements.isEmpty();
    }
    public int size() {
            return this.theElements.size();
    }
    Iterator <E> iterator() {
            return this.theElements.iterator();
    }
    public boolean contains(E o) {
            Iterator<E> it = this.iterator();
            while (it.hasNext()) {
                Object elem = it.next();
                if (elem.equals(o))
                        return true;
            }
            return false;
    }
    public void add(E o) {
            if (!this.contains(o))
                this.theElements.add(o);
    }
    public GenericSet <E> copy() {
            GenericSet <E> destSet = new GenericSet <E>
();
            Iterator<E> it = this.iterator();
            while (it.hasNext())
                destSet.add(it.next());
            return destSet;
    }
```

```
 public GenericSet <E> union(GenericSet <E> s) {
            GenericSet <E> unionSet = s.copy();
            Iterator<E> it = this.iterator();
            while (it.hasNext())
                unionSet.add(it.next());
            return unionSet;
    }
    public GenericSet <E> intersection(GenericSet <E> s) {
            GenericSet <E> interSet = new GenericSet <E>
();
            Iterator<E> it = this.iterator();
            while (it.hasNext()) {
                E elem = it.next();
                if (s.contains(elem))
                        interSet.add(elem);
            }
            return interSet;
    }
    void removeAllElements() {
            this.theElements.removeAllElements();
    }
    public String toString(){
            String s = "[";
            Iterator<E> it = this.iterator();
            if (it.hasNext())
                s += it.next().toString();
            while (it.hasNext())
                s += ", " + it.next().toString();
            return s + "]";
    }
    private ArrayList<E> theElements;
}
```

Beispielprogramm für generische Sets

```
class UseGenericSet {
    public static void main(String [] args) {
        GenericSet <String> s1 = new  GenericSet <String> ();
        s1.add("A");
        s1.add("B");
        s1.add("C");
        s1.add("A");
        System.out.println(s1);
        GenericSet  <String> s2 = new GenericSet <String> ();
        s2.add("B");
        s2.add("C");
        s2.add("D");
        s2.add("D");
        System.out.println(s2);
        System.out.println(s1.union(s2));
        System.out.println(s1.intersection(s2));
    }
}
```

Vorteile von generischen Klassen

- Mit Hilfe von generischen Klassen wird erreicht, dass Klassen auf einer Vielzahl von anderen Klassen operieren können, wobei aber gleichzeitig eine größere Sicherheit bereits zur Übersetzungszeit erreicht wird.
- Damit erhält man Flexibilität und kann gleichzeitig Programme sicherer machen, weil viele Überprüfungen bereits bei der Übersetzung gemacht werden können.
- Beispielsweise wird durch

  ```
  GenericSet <String> s1 = new  GenericSet <String> ();
  ```

  sichergestellt, dass `s1` lediglich String-Objekte referenzieren kann.
- Wertzuweisungen können so zur Übersetzungszeit geprüft werden. In früheren Versionen von Java ging das lediglich zur Laufzeit durch die Casting-Operation.

Spezialisierung generischer Klassen

- Die Spezialisierung unserer generischen Klasse `GenericSet` kann beispielsweise mit Hilfe von Textueller Ersetzung durchgeführt werden.
- Ersetze an allen Stellen "` <E>`" durch "`-E`".
- Ersetze danach an allen Stellen das E, welches für eine beliebige Klasse steht durch die konkrete Klasse, beispielsweise `String`.
- Dadurch entsteht eine spezialisierte Klasse `GenericSet-String`, welche nicht generisch ist und keine Konstrukte generischer Klassen enthält.
- Diese könnte man prinzipiell mit einem alten Compiler übersetzen.
- Dazu muss allerdings auch der Iterator spezialisiert werden.

Die for-Schleife

- Speziell für Situationen, in denen die Anzahl der Durchläufe von Beginn an feststeht, stellt Java mit der `for`-Schleife eine Alternative zur `while`-Schleife zur Verfügung.
- Die allgemeine Form der `for`-Schleife ist:

```
for (Initialisierungsanweisung; Bedingung; Inkrementierung)
    Rumpf
```

- Sie ist äquivalent zu

```
Initialisierungsanweisung
while (Bedingung){
    Rumpf
    Inkrementierung
}
```

Anwendung: Potenzieren mit der `for`-Schleife

- Zur Formulierung des Verfahrens betrachten wir zunächst, wie wir die Berechnung von x^y per Hand durchführen würden:

$$x^y = \begin{cases} 1 & \text{falls } y = 0 \\ \underbrace{x * \ldots * x}_{y \text{ mal}} & \text{sonst} \end{cases} = 1 * \underbrace{x * \ldots * x}_{y \text{ mal}}$$

- Daraus ergibt sich ein informelles Verfahren:
  1. starte mit `1`
  2. multipliziere sie mit `x`
  3. multipliziere das Ergebnis mit `x`
  4. führe Schritt 3) solange aus, bis `y` Multiplikationen durchgeführt wurden.

Potenzierung mit der for-Anweisung

- Bei der Potenzierung mussten wir genau `y` Multiplikationen durchführen.
- Die Anzahl durchgeführter Multiplikationen wird einfach in einer Variablen `count` gespeichert.

```
static int power(int x, int y){
    int count, result = 1;

    for (count = 0; count < y; count++)
        result *= x;

    return result;
}
```

Komplexere for-Anweisungen

- Die Initialisierungs- und die Inkrementierungsanweisung können aus mehreren, durch Kommata getrennten Anweisungen bestehen.
- Betrachten wir die analoge `while`-Schleife, so werden die Initialisierungsanweisungen vor dem ersten Schleifendurchlauf ausgeführt.
- Auf der anderen Seite werden die Inkrementierungsanweisungen am Ende jedes Durchlaufs ausgeführt.
- Damit können wir auch folgende `for`-Anweisung zur Berechnung von x^y verwenden:

```
for (count = 0, result = 1; count < y; result*=x,
    count++);
```

- Solche **kompakten Formen der for-Anweisung** sind **üblicherweise schwerer verständlich** und daher **für die Praxis nicht zu empfehlen.**

Zusammenfassung (1)

- **Bedingte Anweisungen** erlauben es, in Abhängigkeit von der Auswertung einer Bedingung im Programm **verschiedene Anweisungen durchzuführen**.
- Dadurch kann der Programmierer den **Kontrollfluss steuern** und in seinem Programm entsprechend **verzweigen**.
- Mit einem `if`**-Statement** kann man **zwei Fälle** unterscheiden.
- Durch Kaskardierung kann man **mehr als zwei Fälle** unterscheiden.

Zusammenfassung (2)

- Bedingungen sind **Boolesche Ausdrücke**, die zu `true` oder `false` ausgewertet werden..
- In Java gibt es dafür den primitiven Datentyp `boolean` mit den beiden Werten `true` und `false`.
- Einfache **Boolesche Ausdrücke** können mit den **Vergleichsoperatoren** `<`, `>`, `<=`, `>=`, `==`, und `!=`, die auf Zahltypen operieren, definiert werden.
- **Komplexere Boolesche Ausdrücke** werden mit den **logischen Operatoren** `&&`, `||` und `!` zusammengesetzt.

Zusammenfassung (3)

- Die **Wiederholung von Anweisungssequenzen** durch **Schleifen** oder **Loops** ist eines der **mächtigsten Programmierkonstrukte**.
- Mit Hilfe von Schleifen wie der **`while`-Schleife** können Sequenzen von **Anweisungen beliebig häufig wiederholt** werden.
- Die **`for`**-Schleife ist ein äquivalentes Konstrukt zur **`while`**-Schleife. Die **`for`**-Schleife eignet sich besonders, wenn die Anzahl der Iterationen im Vorhinein bekannt ist.

Zusammenfassung (4)

- **Kollektionen** sind Objekte, die es erlauben, **Objekte zusammenzufassen**.
- **`ArrayList`** ist eine solche **Kollektionsklasse**, mit der **Objekte beliebiger Klassen** zusammengefasst werden können.
- Die **einzelnen Objekte** eines `ArrayList`-Objektes können mit **Durchläufen** unter Verwendung eines Objektes der Klasse **`Iterator`** **prozessiert** werden.
- Mit Hilfe der Klasse `ArrayList` können wir dann **andere Kollektionsklassen definieren** (wie z.B. eine `Set`-Klasse).