

# Einführung in die Informatik

## Working with Multiple Objects and the `while`-Loop

Wolfram Burgard

7/1

# Motivation

- In der Praxis ist es 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.

7/2

# Verarbeiten mehrerer Objekte

Ein einfaches Problem ist die Situation,

- in der **alle Objekte unabhängig voneinander** verarbeitet werden können und
- in der **einmal verarbeitete Objekte nicht länger benötigt** werden.

Beispiel für eine solche Situation:

```
Employee e = Employee.readIn(br);
int hours = Integer.parseInt(br.readLine());
System.out.print("Employee " + e.getName() +
    " has earned " + e.calcPay(hours));
```

Diese Anweisungen sollen für alle Mitarbeiter jeweils einmal ausgeführt werden.

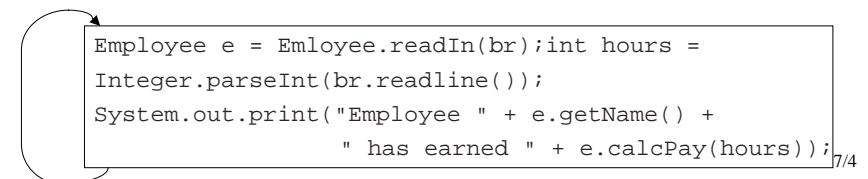
7/3

# Das Konstrukt einer Schleife

Gesucht ist eine Anweisung, die es uns erlaubt, eine gegebene Sequenz von Statements zu wiederholen. Eine entsprechende Konstruktion nennt man eine **Schleife** oder **Loop**.

**Bestandteile einer Schleife:**

- **Rumpf** der Schleife. Dieser enthält alle **Anweisungen, die wiederholt ausgeführt werden sollen**.
- Der **Kopf** der Schleife. Mit diesem wird festgelegt, **wie häufig die Schleife ausgeführt werden soll oder wann mit der Ausführung abgebrochen werden soll, d.h. wann sie terminieren soll**.



```
Employee e = Employee.readIn(br);int hours =
Integer.parseInt(br.readLine());
System.out.print("Employee " + e.getName() +
    " has earned " + e.calcPay(hours));
```

7/4

## Das while-Statement

---

- Das erste Wiederholungskonstrukt **while-Schleife**.
- Die allgemeine Form 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 ein logischer Ausdruck vom Typ `boolean`.
- Der **Rumpf** ist ein einfaches oder ein zusammengesetztes **Statement**.

7/5

## 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**.

7/6

## Typisches Schema für die Verwendung einer while-Schleife

---

- Vor der Schleife findet die **Initialisierung** für den ersten Durchlauf einer Schleife statt:
  1. **Bereitstellen der Objekte für den ersten Durchlauf** und
  2. **Variablen**, die **in der Abbruchbedingung** vorkommen, in Abhängigkeit von dem vorangegangenen Schritt **setzen**.
- **In jedem Schleifendurchlauf** dann:
  1. **Das aktuelle Objekt verarbeiten** und
  2. **Variablen für den nächsten Durchlauf setzen**, d.h. zum nächsten Objekt übergehen.
- Häufig werden dabei die einzelnen Schritte vermischt, d.h. mit dem Bereitstellen der Daten für den ersten Durchlauf wird auch gleich die Abbruchbedingung entsprechend gesetzt.

7/7

## Zurück zum Buchhaltungsbeispiel

---

```
Employee e = Employee.readIn(br);           //Read first object
while (e != null) {                          //Object read?
    int hours = Integer.parseInt(br.readLine()); //Process Object...
    System.out.println("Employee " + e.getName() +
                       " has earned " + e.calcPay(hours));
    e = Employee.readIn(br);                 //Read next object
}
```

Sofern das Einlesen des ersten Objektes gelingt, ist `e` ungleich `null`.

Dies ist ein **typisches Beispiel für die Verwendung einer while-Schleife**.

7/8

## 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.
        }
    }
}
```

7/9

## Eine kompaktere Version

Das Einlesen kann mit einer Anweisung durchgeführt werden, wenn eine Wertzuweisung im Test verwendet wird:

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

class WHWWWAll {
    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;
        while ((line = whiteHouse.readLine()) != null)
            System.out.println(line);
    }
}
```

7/10

## 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 
$$x_{n+1} = x_n - \frac{x_n^3 - x}{3 * x_n^2},$$

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

Mit  $n \rightarrow \infty$  konvergiert<sup>a</sup>  $x_n$  gegen  $\sqrt[3]{x}$ , d.h.  $\lim_{n \rightarrow \infty} x_n = \sqrt[3]{x}$

<sup>a</sup> Sofern kein  $x_n = 0$

7/11

## 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
```

7/12

## 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;
        }
    }
}
```

3

## Anwendung des Programms

Eingabe: -27

Eingabe: 10<sup>90</sup>

n = 1: -5.6851555555555555  
 n = 2: -4.068560488977107  
 n = 3: -3.256075689936079  
 n = 4: -3.0196112473705674  
 n = 5: -3.0001270919925287  
 n = 6: -3.000000005383821  
 n = 7: -3.0

n = 1: 2.2222222222222218E89  
 n = 2: 1.481481481481481E89  
 n = 3: 9.876543209876541E88  
 n = 4: 6.584362139917694E88  
 ...

Process ProgramRoot finished

n = 996: 9.999999999999999E29  
 n = 997: 1.0E30  
 n = 998: 9.999999999999999E29  
 n = 999: 1.0E30

Process ProgramRoot finished

7/14

## Anwendung der while-Schleife: Berechnung des ggT

- Das Verfahren zur **Berechnung des größten gemeinsamen Teilers** (ggT) von zwei Zahlen a und b geht zurück auf **Euklid** (etwa 300 v. Chr.)
- Idee des Verfahrens:

$$ggT(a,b) = \begin{cases} b & , \text{falls } a \bmod b = 0 \\ ggT(b, a \bmod b) & , \text{sonst} \end{cases}$$

- Auswertungsbeispiele:

$$ggT(15, 6) = ggT(6, 3) = 3$$

$$ggT(24, 16) = ggT(16, 8) = 8$$

7/15

## Anwendung des Verfahrens

$$ggT(a,b) = \begin{cases} b & , \text{falls } a \bmod b = 0 \\ ggT(b, a \bmod b) & , \text{sonst} \end{cases}$$

Beispiel: a = 27, b = 48

a                      b                      a&b

7/16

## Das komplette Programm

```
import java.io.*;

class ProgramggT {
    public static void main(String arg[]) throws java.io.IOException{
        BufferedReader br = new BufferedReader(
            new InputStreamReader(System.in));

        int a = Integer.parseInt(br.readLine());
        int b = Integer.parseInt(br.readLine());

        while (a % b !=0){    //Done?
            int r = a % b;    // Process data and prepare next round
            a = b;
            b = r;
        }
        System.out.println("Der ggT ist " + b);
    }
}
```

7/17

## Anwendungsbeispiel

Betrachten wir die Eingaben 81 für  $a$  und 15 für  $b$ .

Die folgende Tabelle enthält die Werte der Ausdrücke bzw. Variablen zum Zeitpunkt des  $i$ -ten Tests der Bedingung  $a \% b == 0$  oder  $while$ -Schleife:

Test Nr.	a	b	a%b
1	81	15	6
2	15	6	3
3	6	3	0

Ausgabe daher: 3

7/18

## Geschachtelte while-Schleifen

Ebenso wie  $if$ -Anweisungen, können auch **while-Schleifen geschachtelt** werden. Eine typische Struktur ist:

```
while (condition1) {
    ...
    while (condition2) {
        ...
    }
    ...
}
```

Dabei wird für jeden Durchlauf der äußeren Schleife ( $condition1$ ) die innere Schleife ( $condition2$ ) ausgeführt.

7/19

## Beispiel: Berechnung aller Primzahlen unter 10000

- $n$  ist durch  $i$  teilbar, wenn  $n \% i == 0$ .
- Eine Zahl ist Primzahl, wenn sie durch keine andere Zahl teilbar ist.
- D.h. für jede Zahl  $i$ , die kleiner als  $n$  ist, müssen wir prüfen, ob  $i$  Teiler von  $n$  ist.
- Dies müssen wir für jedes  $n < 10000$  durchführen. Dabei genügt es, bei 2 zu beginnen.

7/20

## Das komplette Programm

```
class ProgramPrimes1 {
    public static void main(String arg[]) {
        int m = 10000;
        int n = 2;
        while (n < m){
            int i = 2;
            boolean prime = true; // Assume n is a prime
            while (i < n && prime){
                prime = ((n % i) != 0); // Is n dividable by i?
                i++;
            }
            if (prime)
                System.out.println("Prime number : " + n);
            n++;
        }
    }
}
```

7/21

## Entwicklung einer effizienteren Version

- In dieser Version wird die „innere Schleife“ jeweils  $n$ -mal „durchlaufen“.
- Tatsächlich genügt es jedoch, nur die Zahlen  $i \leq \sqrt{n}$  zu betrachten
  1. Ist  $i$  Teiler von  $n$ , so existiert ein  $j$  mit  $i*j=n$
  2. Offensichtlich genügt es dann, alle  $i$  mit  $i \leq j$  zu betrachten.
  3. Aus  $i*j \leq n$  und  $i \leq j$  folgt aber, dass  $i \leq \sqrt{n}$  ist.

7/22

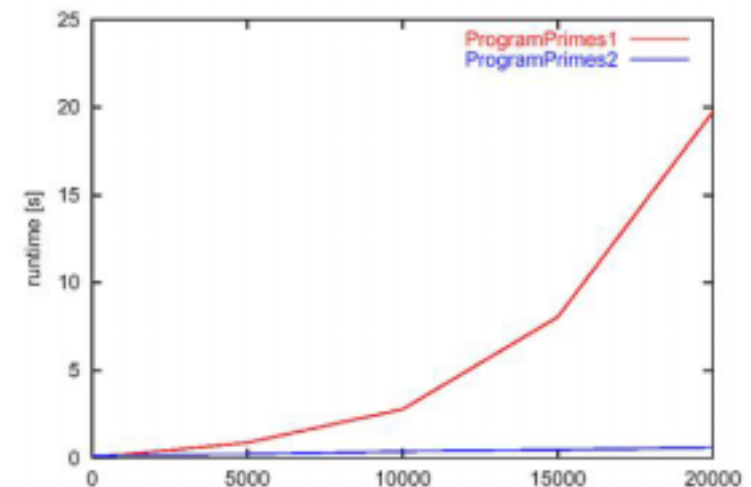
## Eine effizientere Version

Es genügt, alle  $i$  mit  $i*i \leq n$  zu betrachten!

```
class ProgramPrimes2 {
    public static void main(String arg[]) {
        int m = 10000;
        int n = 2;
        while (n < m){
            int i = 2;
            boolean prime = true; // Assume n is a prime
            while (i*i <= n && prime){
                prime = ((n % i) != 0); // Is n dividable by i?
                i++;
            }
            if (prime)
                System.out.println("Prime number : " + n);
            n++;
        }
    }
}
```

7/23

## Vergleich der Laufzeiten



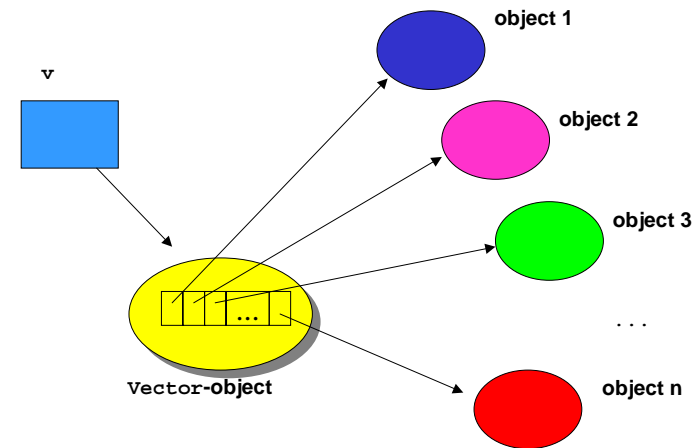
7/24

## Kollektionen mehrere Objekte: Die Klasse `Vector`

- Mit `Vector` stellt Java eine Klasse zur Verfügung, die eine **Zusammenfassung von unter Umständen auch verschiedenen Objekten in einer Liste** oder Reihe erlaubt.
- Die Klasse `Vector` ist **nicht** gedacht **für Vektoren im mathematischen Sinn und deren Operationen**.
- **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.

7/25

## Kollektion eventuell unterschiedlicher Objekte mit der Klasse `Vector`

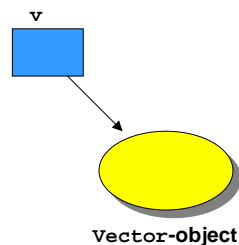


7/26

## Erzeugen eines `Vector`-Objektes

- Wie bei anderen Klassen auch werden Objekte `Vector` mit dem Konstruktor erzeugt.
- Der Konstruktor von `Vector` hat keine Argumente:  

```
Vector v = new Vector();
```
- Wirkung des Konstruktors:



7/27

## Hinzufügen von Objekten zu einem `Vector`-Objekt

Um Objekte zu einem `Vector`-Objekt hinzuzufügen, verwenden wir die Methode `addElement`.

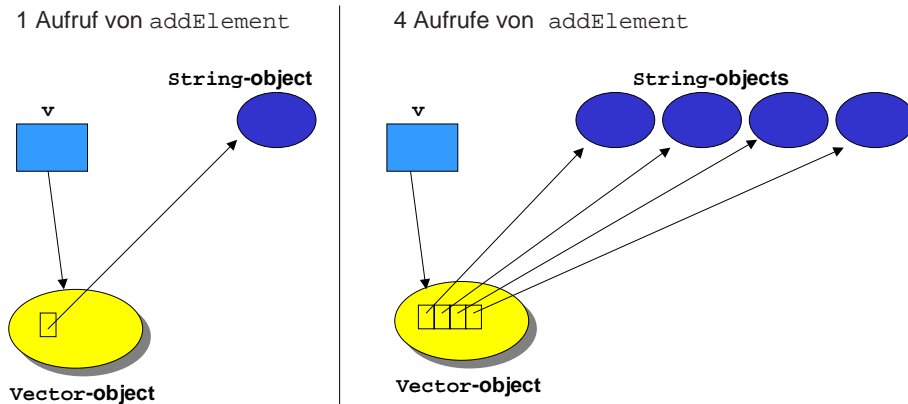
Dieser Methoden geben wir als Argument das hinzuzufügende Objekt mit.

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

```
String s = br.readLine() // Read first string
while (s != null){       // Something read?
    v.addElement(s);     // Processing adds s to v
    s = br.readLine();   // Read next string
}
```

7/28

## Anwendung dieses Programmstücks



Hier referenziert unser Vector-Objekt lediglich Objekte der Klasse String. 7/29

## Durchlauf durch einen Vektor

- 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 **übergangen** und
  - **testen, ob es noch weitere Objekte gibt**, die besucht werden müssen.

7/30

## Wie kann man Durchläufe realisieren?

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

7/31

## Enumerations

Java bietet eine **abstrakte Klasse Enumeration** zur Realisierung von **Durchläufen durch Vector-Objekte** und andere Kollektionsklassen.

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

Die Klasse **Vector** enthält eine Methode **elements**, die eine Referenz auf ein **Enumeration-Objekt** liefert. Ihr Prototyp ist:

```
Enumeration elements() // Liefert eine Enumeration für einen Vector
```

Die Klasse **Enumeration** wiederum bietet die folgenden Methoden

```
boolean hasMoreElements() // True, falls es weitere Elemente gibt  
Object nextElement() // Liefert das nächste Objekt
```

7/32



## Der Return-Type von nextElement

- Im Prinzip muss die Methode `nextElement` **Referenzen auf Objekte beliebiger Klassen** liefern.
- Um eine breite Anwendbarkeit realisieren zu können, müssen Klassen wie `Vector` oder `Enumeration` diese **Flexibilität** haben.
- Aus diesem Grund liefern solche Methoden eine **Referenz auf ein Object-Objekt**.
- **Object ist eine Klasse** und in Java ist **jedes Objekt** auch ein **Object-Objekt**.
- Somit kann also eine Methode wie `nextElement` mit ihrem **Object-Rückgabewert beliebige Objekte zurückgeben**.
- Allerdings muss man in Java mittels **Casting** stets mitteilen, was für ein Objekt durch einen Aufruf von `nextElement` zurückgegeben wird.

7/33

## Durchlauf durch ein Vector-Objekt

Um einen Durchlauf durch unser `Vector`-Objekt `v` wir nun wie folgt vor:

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

Dies wird nun überführt zu

```
Enumeration en = v.elements();  
while (en.hasMoreElements()) {  
    String s = (String) en.nextElement();  
    System.out.print(s);  
}
```

7/34

## Primitive Datentypen und Vector-Objekte

- Kollektionsklassen wie `Vector` können nur Objekte aufnehmen.
- Primitive Datentypen wie `int`, `float` und `boolean` stellen keine `Objects` dar und können daher nicht als Parameter von `addElement` verwendet werden.
- Um dieses Problem zu lösen bietet Java so genannte **Wrapper**-Klassen für primitive Datentypen dar, z.B.:

Primitiver Typ	Wrapper-Typ
<code>int</code>	<code>Integer</code>
<code>boolean</code>	<code>Boolean</code>
<code>float</code>	<code>Float</code>

- Darüber hinaus enthalten die `Wrapper`-Klassen auch **static-Methoden** für den jeweiligen primitiven Datentyp.

7/35

## Einfügen von ints in ein Vector-Objekt

```
Vector vi = new Vector();  
int i;  
i =1;  
vi.addElement(new Integer(i));  
i =2;  
vi.addElement(new Integer(i));  
i =3;  
vi.addElement(new Integer(i));  
Enumeration e = vi.elements();  
while (e.hasMoreElements()) {  
    Integer il = (Integer) e.nextElement();  
    System.out.println(il.intValue());  
}
```

Für `double`, `boolean`, `float` etc. ist das Verfahren analog.

7/36

## Anwendung von `Vector` zur Modellierung von Mengen

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

7/37

## 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)
- `addElement` (hinzufügen eines Elements)
- `copy` (Kopie einer Menge erzeugen)
- `size` (Anzahl der Elemente)
- `elements` (Durchlauf durch eine Menge)
- `union` (Vereinigung)
- `intersection` (Durchschnitt) Alle Elemente ausgeben

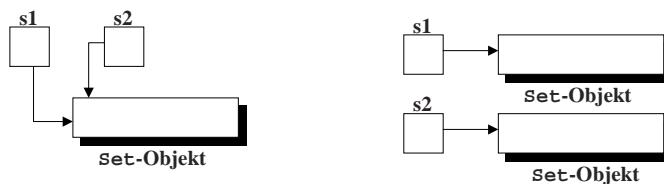
7/38

## 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 `addElement` 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();`

7/39

## Festlegen der Schnittstellen

Prototypen der einzelnen Methoden:

```
public Set()
public boolean isEmpty()
public int size()
public boolean contains(Object o)
public void addElement(Object o)
public Set copy()
public Set union(Set s)
public Set intersection(Set s)
public Enumeration elements()
```

7/40

## Ein typisches Beispielprogramm

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

7/41

## Das Skelett der Set-Klasse

```
class Set {
    public Set() {... };
    public boolean isEmpty() {... };
    public int size() {... };
    public boolean contains(Object o) {... };
    public void addElement(Object o) {... };
    public Set copy() {... };
    public Set union(Set s) {... };
    public Set intersection(Set s) {... };
    public Enumeration elements() {... };
    ...
    private Vector theElements;
}
```

7/42

## Implementierung der Methoden (1)

1. Der **Konstruktor** ruft lediglich die entsprechende Methode der Vector-Klasse auf:

```
public Set() {
    this.theElements = new Vector();
}
```

2. Die **Methoden size und empty** nutzen ebenfalls vordefinierte Methoden der Klasse Vector:

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

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

7/43

## Implementierung der Methoden (2)

3. Um alle **Elemente der Menge aufzuzählen**, müssen wir eine Methode elements realisieren:

```
Enumeration elements() {
    return this.theElements.elements();
}
```

4. Die **copy-Methode** muss alle Elemente des Vector-Objektes durchlaufen und sie einem neuen Set-Objekt hinzufügen:

```
public Set copy() {
    Set destSet = new Set();
    Enumeration e = this.elements();
    while (e.hasMoreElements())
        destSet.addElement(e.nextElement());
    return destSet;
}
```

7/44

## Implementierung der Methoden (3)

---

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

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

7/45

## Implementierung der Methoden (4)

---

6. 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();
    Enumeration e = this.elements();
    while (e.hasMoreElements())
        unionSet.addElement(e.nextElement());
    return unionSet;
}
```

7/46

## Implementierung der Methoden (5)

---

7. 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();
    Enumeration e = this.elements();
    while (e.hasMoreElements()) {
        Object elem = e.nextElement();
        if (s.contains(elem))
            interSet.addElement(elem);
    }
    return interSet;
}
```

7/47

## Implementierung der Methoden (6)

---

8. 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:
- Dies wirft das Problem auf, dass wir Objekte vergleichen müssen, ohne dass wir wissen, zu welcher Klasse sie gehören.
  - 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 Objekt 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.

7/48

## Implementierung der Methoden (6)

9. Daraus resultiert die folgende Implementierung der Methode `contains`:

```
public boolean contains(Object o) {
    Enumeration e = this.elements();
    while (e.hasMoreElements()) {
        Object elem = e.nextElement();
        if (elem.equals(o))
            return true;
    }
    return false;
}
```

7/49

## Implementierung der Methoden (7)

10. 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 Argument einer `print`-Methode verwendet wird.

7/50

## Die Methode `toString()`

```
public String toString(){
    String s = "[";
    Enumeration e = this.elements();
    if (e.hasMoreElements())
        s += e.nextElement().toString();
    while (e.hasMoreElements())
        s += ", " + e.nextElement().toString();
    return s + "]";
}
```

7/51

## Die komplette Klasse `set`

```
import java.io.*;
import java.util.*;
class Set {
    public Set() {
        this.theElements = new Vector();
    }
    public boolean isEmpty() {
        return this.theElements.isEmpty();
    }
    public int size() {
        return this.theElements.size();
    }
    Enumeration elements() {
        return this.theElements.elements();
    }
    public boolean contains(Object o) {
        Enumeration e = this.elements();
        while (e.hasMoreElements()) {
            Object elem = e.nextElement();
            if (elem.equals(o))
                return true;
        }
        return false;
    }
    public void addElement(Object o) {
        if (!this.contains(o))
            this.theElements.addElement(o);
    }
    public Set copy() {
        Set destSet = new Set();
        Enumeration e = this.elements();
        while (e.hasMoreElements())
```

```
        destSet.addElement(e.nextElement());
        return destSet;
    }
    public Set union(Set s) {
        Set unionSet = s.copy();
        Enumeration e = this.elements();
        while (e.hasMoreElements())
            unionSet.addElement(e.nextElement());
        return unionSet;
    }
    public Set intersection(Set s) {
        Set interSet = new Set();
        Enumeration e = this.elements();
        while (e.hasMoreElements()) {
            Object elem = e.nextElement();
            if (s.contains(elem))
                interSet.addElement(elem);
        }
        return interSet;
    }
    void removeAllElements() {
        this.theElements.removeAllElements();
    }
    public String toString(){
        String s = "[";
        Enumeration e = this.elements();
        if (e.hasMoreElements())
            s += e.nextElement().toString();
        while (e.hasMoreElements())
            s += ", " + e.nextElement().toString();
        return s + "]";
    }
    private Vector theElements;
}
```

7/52

## Unser Beispielprogramm (erneut)

---

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

7/53

## Ausgabe des Beispielprogramms

---

```
java useSet

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

Process useSet finished
```

7/54

## Zusammenfassung (1)

---

- 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.
- **Kollektionen** sind Objekte, die es erlauben, **Objekte zusammenzufassen**.
- **Vector** ist eine solche **Kollektionsklasse**, mit der **Objekte beliebiger Klassen** zusammengefasst werden können.

7/55

## Zusammenfassung (2)

---

- Die **einzelnen Objekte** eines `Vector`-Objektes können mit **Durchläufen** unter Verwendung eines Objektes der Klasse **Enumeration** **prozessiert** werden.
- Mit Hilfe der Klasse `Vector` können wir dann **andere Kollektionsklassen definieren** (wie z.B. eine `Set`-Klasse).
- Für **primitive Datentypen** verwenden wir **Wrapper-Klassen**, um sie in Kollektionen einzufügen.

7/56