

8. Schnittstellen in Java

- Problem
 - Bei der Erstellung von *großen Programmpaketen* sollten Klassen *wiederverwendet werden*.
 - Typischerweise werden dabei in einer Klasse Datenfelder vom Typ einer andere Klasse bereitgestellt.

```
public class Konto {  
    private String kontoNr;  
    private double kontoStand;  
    private Kunde k;           // k verweist auf ein Objekt der Klasse Kunde  
    ...  
}
```

- Dadurch wird die Klasse Konto von der Klasse Kunde abhängig!
 - Was passiert bei einer Änderung der Klasse Kunde?

8.1 Motivation

- Man stelle sich nun ein größeres Programm P vor, das aus sehr vielen, voneinander abhängigen Klassen besteht.
 - ➔ Dies ist keine gute Software, da kleine Änderungen immer wieder dazu führen, dass P neu übersetzt und getestet werden muss.
- Probleme bei zu vielen Abhängigkeiten im Programm
 - Schlechte Wiederverwendbarkeit einzelner Klassen
 - Klasse Konto kann nicht ohne Klasse Kunde verwendet werden.
 - Viele Fehler durch Änderungen
 - Änderungen in der Klasse Kunde kann dazu führen, dass Klasse Konto nicht mehr funktioniert.
 - Schlechte Erweiter- und Anpassbarkeit an neue Probleme
 - Schwierige Aufteilung eines Programms P in einzelne Teile, die unabhängig voneinander im Team entwickelt werden können.

Modulares Programmieren

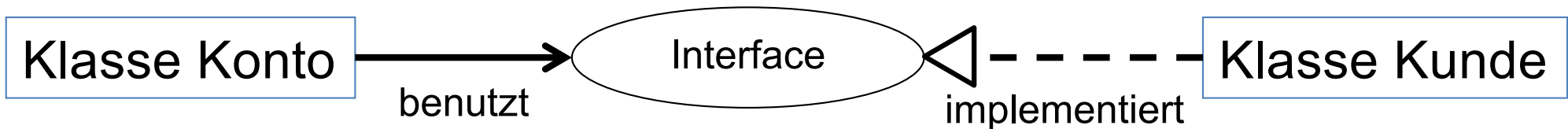
- Um diese Probleme zu beheben, wurde bereits in den 70er Jahren des letzten Jahrhunderts das modulare Programmieren postuliert.
 - Nikolaus Wirth hat dafür eine neue Programmiersprache Modula-2 entwickelt .
- Für **große Programmieraufgaben** ist es wünschenswert,
 - die Aufgabenstellung in sinnvolle Teile (**Module**) zu zerlegen,
 - diese Teile **unabhängig voneinander** zu programmieren, zu übersetzen und zu testen.
- Die **Vorteile** aus einer solchen Vorgehensweise sind :
 - Die Problemlösung wird **einfacher darstellbar** und **übersichtlicher**.
 - Mehrere Personen können **gleichzeitig** an einem Programm arbeiten.
 - Teile können leichter **getestet**, **verändert** und **gepflegt** werden.
 - Teile können in anderen Programmen **wiederverwendet** werden.



8.2 Schnittstellen in Java

- Modulares Programmieren wird in Java durch Interfaces realisiert.
 - Statt Klassen werden **Interfaces als Datentypen** verwendet.
 - ➔ Klassen sind damit unabhängig voneinander.
- Der deutsche Begriff für Interface ist **Schnittstelle**.

Schnittstellen als Vertrag



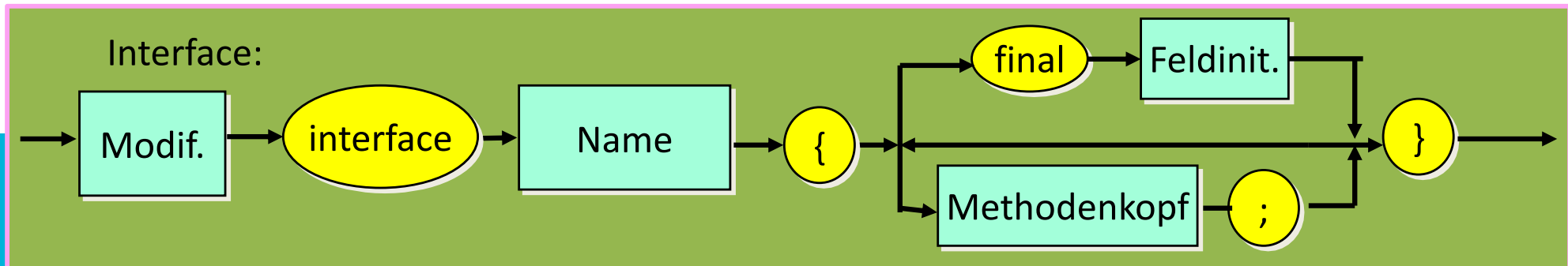
- Die Klasse Konto benutzt die Klasse Kunde, möchte aber unabhängig von der Klasse Kunde bleiben.
 - Es wird jetzt ein **Interface** vereinbart, das von der **Klasse Kunde implementiert** wird und von der **Klasse Konto verwendet** wird.
 - Ein Interface ist wie ein **Vertrag zwischen den beiden Klassen**, in dem die **Zusammenarbeit** geregelt wird.
 - Es wird dabei festgelegt, **was aber nicht wie** etwas geleistet werden soll.

Interface in Java - Eigenschaften

- Ein **Interface** ist **keine** Klasse - kann aber teilweise ähnliche Dienste anbieten.
 - Ein **Interface** kann in Java als **Typ von Variablen** verwendet werden.
 - Ein Interface hat jedoch **keine Konstruktoren** für die Objekterzeugung.
- Ein Interface ist primär eine Sammlung **abstrakter Methoden**
 - Eine abstrakte Methode besteht nur aus dem **Methodenkopf**, hat aber **keinen Rumpf**.
 - Zusätzlich können auch noch **Konstanten** (d.h. Felder mit den Schlüsselwörtern **final** und **static**) definiert werden.
 - Seit Java 8 können aber inzwischen auch implementierte Methoden Teil eines Interface sein. Wir werden diese sogenannten **Default-Methoden** später betrachten.
- Klassen können die abstrakten Methoden eines Interfaces implementieren.

Syntax eines Interface

- Ein vereinfachter Aufbau eines Interface hat die folgende Gestalt:



- Ähnlich zu Klassen kann einem Interface noch Schlüsselwörter vorangestellt werden, die den Zugriff auf die Schnittstelle definieren.

Beispiel

```
public interface RealFunc {  
    /** Eine Methode zur Auswertung von reellen Funktionen  
     * @param x Der Wert, an dem eine Funktion ausgewertet werden kann.  
     * @return Der Wert der Funktion an der Stelle x.  
     */  
    public double eval(double x);  
  
    /** Eine Methode zur Berechnung der Ableitung einer Funktion  
     * @return Liefert die Ableitung der Funktion.  
     */  
    public RealFunc derive();  
  
    /** Eine Methode zur Repräsentation einer Funktion als Zeichenkette.  
     * Diese Methode wird von System.out.print für die Ausgabe verwendet.  
     */  
    public String toString();  
}
```


Implementierung eines Interface

- Klassen können ein Interface implementieren.
 - Dazu muss hinter dem Klassennamen das Schlüsselwort **implements** und dann der Name der Interfaces folgen, die implementiert werden.

- Beispiel

```
public class Exp implements RealFunc {  
    ...  
}
```

- Diese Klassen müssen zu jeder abstrakten Methode des Interface auch eine Implementierung anbieten.
 - Sie können aber die im Interface definierten Konstanten nutzen.
- Es können gleichzeitig mehrere Interfaces (mit Komma getrennt) angegeben werden.
 - Dann müssen alle Interfaces auch implementiert werden (Beispiel später).

Die Klasse Exp

```
public class Exp implements RealFunc {  
    private double a;  
    private double b;  
  
    public Exp(double f, double g) {           // Konstruktor  
        a = f;  
        b = g;  
    }  
  
    public double eval(double xval) {          // Implementierung der eval-Methode  
        return a*Math.exp(b*xval);  
    }  
  
    public String toString() {                 // Implementierung der toString-Methode  
        return "" + a + "e^(" + b + "*x)";  
    }  
  
    public RealFunc derive() {                 // Implementierung der derive-Methode.  
        return new Exp(a*b, b);  
    }  
}
```

Klassen mit mehreren Interfaces

- Klassen können kein, ein oder mehrere Interfaces implementieren.
- Betrachten wir folgendes Szenario
 - Die Klasse Exp soll zwei Interfaces implementieren:
 - Das bisherige Interface RealFunc mit den Methoden eval, derive und toString.
 - Ein weiteres Interface Integrable mit der Methode antiDerive().
 - In der Klasse Exp müssen dann nach dem Schlüsselwort implements beide Interfaces angegeben werden.

```
public class Exp implements RealFunc, Integrable {  
    ...  
}
```

- Im Rumpf der Klasse müssen die vier Methoden implementiert sein.

Interface als Datentyp

- Wir können statt Klassen auch **Interfaces** nutzen, um Variablen und Datenfelder zu deklarieren.

```
RealFunc rf;
```

- Diese Variablen können auf Objekte der Klassen verweisen, die das Interface implementieren.

```
rf = new Exp(2.,3.);
```

- Es können nun über diese Variablen alle Methoden aufgerufen werden, die im Interface angegeben wurden.
 - Tatsächlich werden beim Aufruf der Methoden, die Methoden des Objekts benutzt, auf das die Variable verweist.
 - Man spricht dann von **dynamischen Binden**.

Dynamisches Binden

- Es wird zur Laufzeit des Programms entschieden wird, welche konkrete Methode **beim Aufruf** ausgeführt wird.
 - Es wird die Methode ausgeführt, die in der Klasse des Objekts implementiert wurde.
- Beispiele

```
RealFunc rf = new Exp(2.,3.);  
double res;  
  
res = rf.eval(2.0); // Aufruf der Methode aus der Klasse Exp  
  
// Annahme die Klasse Polynom implementiert ebenfalls das Interface RealFunc  
rf = new Polynom(new double[]{1.,2.});  
  
res = rf.eval(2.0); // Aufruf der Methode aus der Klasse Polynom
```

Man kann nicht alles nutzen!

- Wird ein Objekt über eine Interface-Variable an, so stehen **ausschließlich nur die Methoden aus dem Interface** zur Verfügung.
- Die Klasse selbst kann über weitere Methoden verfügen.
 - Z. B. könnte in der Polynom-Klasse eine Methode `int getGrad()` den höchsten Exponenten des Polynoms liefern (2 im Fall einer Parabel).

```
RealFunc rf = new Polynom(new double[]{1.,2.});  
  
double res = rf.eval(2.0); // Aufruf der Methode aus der Klasse Polynom  
  
int grad = rf.getGrad();    // Funktioniert so nicht !
```

Typkonvertierung hilft!

- Das Problem kann behoben werden, indem man ein Cast auf den gewünschten Typ durchführt.

```
RealFunc rf = new Polynom(new double[]{1.,2.});  
  
double res = rf.eval(2.0); // Aufruf der Methode aus der Klasse Polynom  
  
Polynom p = (Polynom) rf;    // Cast: RealFunc → Polynom  
int grad = p.getGrad();      // Jetzt funktioniert alles wieder.
```

- Wenn der gewünschte Typ jedoch nicht passt, bekommen wir ein richtiges Problem!
 - Das Programm wirft dann eine Exception und wird möglicherweise beendet.

Was ist jetzt der Vorteil?

- Da das Interface RealFunc durch die beiden Klassen Polynom und Exp implementiert wird, vereinheitlicht sich unser Testprogramm.

```
RealFunc p = new Exp(new Double(args[0]), new Double(args[1]));
RealFunc d = p.derive();
System.out.println(p);
System.out.println(d);
for (double elem: new double[]{1.0, 2.0, 3.0, 4.0, 5.0}) {
    res = p.eval(elem);
    System.out.println("f( " + elem + " ) = " + res);
}
```

- Mit Ausnahme der Konstruktoraufrufe ist das Codefragment **komplett unabhängig** von den Klassen Exp und RealFunc.
 - Wir haben also fast die Unabhängigkeit sichergestellt.

Was ist jetzt der Vorteil?

- Da das Interface RealFunc durch die beiden Klassen Polynom und Exp implementiert wird, vereinheitlicht sich unser Testprogramm.

```
RealFunc p = new Exp(new Double(args[1]));
RealFunc d = p.derive();
System.out.println(p);
System.out.println(d);
for (double elem: new double[] { 1.0, 2.0, 3.0, 4.0, 5.0 }) {
    res = p.eval(elem);
    System.out.println("f( " + elem + " ) = " + res);
}
```

Gilt genauso für
andere Programmteile,
nicht nur Tests!

- Mit Ausnahme der Konstruktoraufrufe ist das Codefragment **komplett unabhängig** von den Klassen Exp und RealFunc.
 - Wir haben also fast die Unabhängigkeit sichergestellt.

Objekterzeugung in separaten Fabriken

- Soll dieser Makel der Abhängigkeit von Konstruktoren beseitigt werden, kann eine sogenannte **Factory** implementiert werden.

```
class RealFuncFactory {  
    public static RealFunc getRealFunc(String criteria, double[] args) {  
        if ( criteria.equals("exp") )  
            return new Exp(args[0], args[1]);  
        else  
            return new Polynom(args);  
    }  
}
```

- Diese Klasse lässt sich noch beliebig erweitern, wenn noch weitere Klassen hinzugefügt werden, welche die Schnittstelle RealFunc implementieren.

Interfaces aus der Java-Bibliothek

- Das Java-System bietet bereits eine Vielzahl vordefinierter Klassen und Interfaces an.
- Für geordnete Daten verwendet man meist das **Interface Comparable**, das eine Methode für zum Vergleichen von Objekten vorgibt.
- Das Ergebnis von **compareTo** liefert ein Ergebnis vom Typ `int`. Es gilt folgende Konvention:
 - Wenn **a.compareTo(b)** **negativ** ist, interpretiert man dies als **a < b**.
 - Wenn **a.compareTo(b)** **0** ist, interpretiert man dies als **a = b**.
 - Wenn **a.compareTo(b)** **positiv** ist, interpretiert man dies als **a > b**.
- Wir werden später auf diese und andere Interfaces aus der Java-Bibliothek noch genauer eingehen.

8.3 Default-Methoden in Interfaces

- Interfaces können neben abstrakten Methoden auch **implementierte Methoden** mit Rumpf besitzen.
 - Diese Methoden werden als **Default-Methoden** bezeichnet.
 - Default-Methoden dürfen Methoden im Rumpf verwenden, die im Interface deklariert wurden
 - Die Methoden stehen dann auch in allen Klassen zur Verfügung, die dieses Interface implementieren.

Beispiel (1)

```
public interface RealFunc {
    public double eval(double x);
    public RealFunc derive();
    public String toString();

    default public double[] bulkEval(double[] arr) {
        double[] res = new double[arr.length];
        for (int i = 0; i < arr.length; i+=1)
            res[i] = eval(arr[i]);
        return res;
    }
}
```

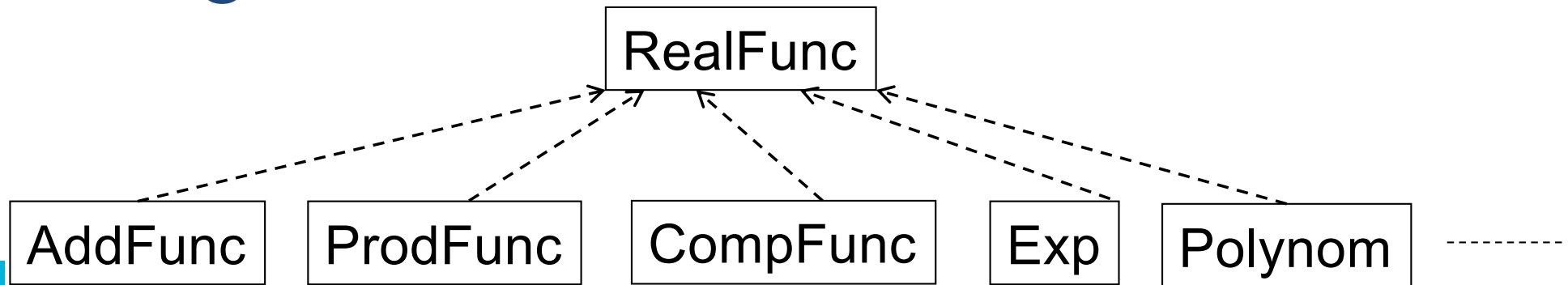
- In der Schnittstelle RealFunc soll eine Methode hinzugefügt werden, um die Funktion für jeden Wert eines double-Arrays auszuführen und die Ergebnisse als Array zu liefern.
- Diese Methode kann komplett mit Hilfe der anderen Methoden aus der Schnittstelle implementiert werden.

Beispiel (2)

- Damit kann man die Default-Methode bulkEval für Objekte der Klassen Exp nutzen.

```
public static void main(String[] args){  
    RealFunc q = new Exp(2.0, 3.0);  
  
    double[] rarr = q.bulkEval(new double[]{0, 1,2,3,4});  
    for (double y:rarr)  
        System.out.println("Wert " + y);  
}
```

Programm mit vielen Klassen

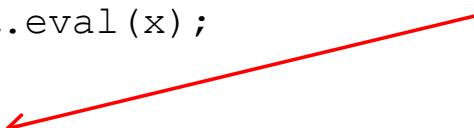


- Schnittstelle **RealFunc** kann von vielen Klassen implementiert werden.
 - **Exp** – Klasse der Exponentialfunktionen
 - **Polynom** – Klasse der Polynome $a_0 + a_1 \cdot x^1 + \dots + a_n \cdot x^n$
 - Die drei Klassen auf der linken Seite stellen für Funktionen $f(x)$ und $g(x)$ folgende Funktionen zur Verfügung.
 - **AddFunc** $f(x) + g(x)$
 - **ProdFunc** $f(x) \cdot g(x)$
 - **CompFunc** $f(g(x))$

Klasse AddFunc

```
public class AddFunc implements RealFunc {  
    private RealFunc left;  
    private RealFunc right;  
  
    public AddFunc(RealFunc f, RealFunc g) {  
        left = f;  
        right = g;  
    }  
  
    public double eval(double x) {  
        return left.eval(x) + right.eval(x);  
    }  
  
    public RealFunc derive() {  
        return new AddFunc(left.derive(), right.derive());  
    }  
  
    public String toString(){  
        return left.toString() + " + " + right.toString();  
    }  
}
```

Ableitungsregel:
 $(f+g)' = f' + g'$



Klasse ProdFunc

```
public class ProdFunc implements RealFunc {
    private RealFunc left;
    private RealFunc right;

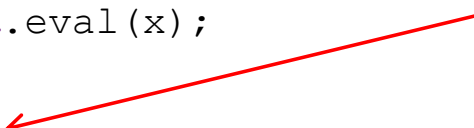
    public ProdFunc(RealFunc f, RealFunc g) {
        left = f;
        right = g;
    }

    public double eval(double x) {
        return left.eval(x) * right.eval(x);
    }

    public RealFunc derive() {
        return new AddFunc(new ProdFunc(left.derive(), right),
                           new ProdFunc(left, right.derive()));
    }

    public String toString(){
        return "(" + left.toString() + ") * (" + right.toString() + ")";
    }
}
```

Ableitungsregel:
 $(f \cdot g)' = f' \cdot g + f \cdot g'$



Zusammenfassung

- Interfaces sind in Java ein **wichtiges** Konzept, um Klassen unabhängig voneinander zu machen.
 - Interfaces als Datentypen bei der Variablendeklaration
- Interfaces können durch Klassen implementiert werden.
 - Eine Klasse kann mehrere Interfaces implementieren.
- Dynamisches Binden
 - Wird eine Methode über eine Interface-Variable aufgerufen, wird die konkrete Methode durch die Klasse des Objekts bestimmt.
- Factory-Klassen
 - Unabhängigkeit von den Konstruktoren der Klasse, in dem wir zu einem Interface noch eine solche Klasse bereitstellen.
- Default-Methoden

9. Die Klasse String

- Übersicht
 - Strings, String-Objekte, Literale
 - Stringerzeugung
 - Konkatination von Strings
 - Einige Methoden für Strings
 - Vergleiche von Strings
 - Strings und char-Arrays
 - StringBuilder,
 - Die Methode **format**

Motivation

- Die Informatik beschäftigt sich sehr oft mit der **Verarbeitung von Zeichenketten**.
 - Analyse von Emails
 - Twitter-Nachrichten
- Die **Klasse String** bietet als Datentyp **nicht-veränderbare Zeichenketten** mit folgenden Diensten an.
 - Deklaration von Variablen des Typs String
 - Diverse Konstruktoren zur Erzeugung
 - Viele Hilfsmethoden, um z. B.
 - die Länge eines Strings festzustellen: **length()**
 - zwei Strings auf Gleichheit zu testen: **equals()**
 - das i-te Zeichen in einem String zu lesen: **charAt()**

String-Erzeugung

- Man kann eine Variable/ ein Feld wie üblich definieren: `String bsp;`
- Meist nutzt man aber eine Definition mit Initialisierung:

```
String gruss1 = "Halli hallo";
```

- Dabei erhält **gruss1** eine Referenz auf das Objekt **"Halli hallo"**
- Man kann auch die üblichen Konstruktoren nutzen

```
String leer1 = new String();
```

- Dieser Konstruktor erzeugt eine Referenz auf ein leeres String-Objekt.
 - Die gleiche Wirkung hat: `String leer2 = "";`
- Es gibt auch einen Konstruktor, mit einem String-Parameter

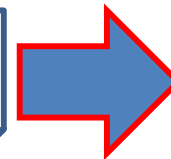
```
String gruss2 = new String("Halli hallo");
```

- Dieser erzeugt eine Referenz auf eine Kopie des angegebenen Strings.

Konkatenation von Strings

- Für die Konkatenation von Strings wird der „+“ – Operator genutzt.
 - `"Hallo" + " Welt"` ergibt: `"Hallo Welt"`
 - `"Bitte" + "nicht" + "stören"` ergibt: `"Bittenichtstören"`
- Bei der Konkatenation wird jeweils ein neues zusammengesetztes String-Objekt gebildet.
- Bei der Konkatenation wird, falls möglich, automatisch eine Umwandlung vorgenommen. Die Umwandlung erfolgt von links nach rechts:
 - `"Hallo" + 1` ergibt: `"Hallo1"`
 - `"Hallo" + 1 + 1` ergibt: `"Hallo11"`
 - Erst wird die erste 1 umgewandelt und konkateniert, dann die zweite 1 ...
 - `"Hallo" + (1 + 1)` ergibt: `"Hallo2"`
 - Die Klammerung bewirkt das erst addiert und dann umgewandelt wird

```
System.out.println("1 und 1 ist "+(1+1));  
System.out.println("1 und 1 ist "+1+1);
```

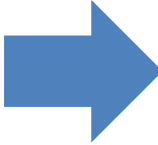


```
1 und 1 ist 2  
1 und 1 ist 11
```

Einige Methoden

- Die Länge eines Strings:

```
String gruss1 = "Halli hallo";  
System.out.println(gruss1.length());  
String gruss2 = new String("Halli hallo");  
System.out.println(gruss2.length());  
String leer1 = new String();  
System.out.println(leer1.length());  
String leer2 = "";  
System.out.println(leer2.length());  
System.out.println(" ".length());
```



11
11
0
0
1

- Zeichen an Position:

```
System.out.println(gruss1.charAt(4));
```



i

- Index von Zeichen:

```
System.out.println(gruss1.indexOf('l'));
```




2

- indexOf** gibt es auch in Varianten: Mit einem weiteren Parameter der die Position angibt, von der an gesucht werden soll.

- Teilstring

```
System.out.println(gruss1.substring(3,5) +  
gruss1.substring(9));
```



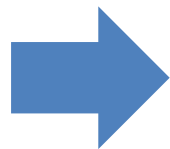
lilo

- Und viele andere mehr...

Vergleiche von Strings

- Wie für Objekte üblich werden beim Vergleich von zwei Objekten der Klasse String immer die **Referenzen** verglichen.
- Für den Vergleich der **Inhalte** von Strings sind in der Klasse String einige passende Methoden vordefiniert: **equals**, **equalsIgnoreCase**, **compareTo**, **compareToIgnoreCase**, **contains**.

```
String gruss1 = "Halli hallo";  
String gruss2 = new String("Halli Hallo");  
System.out.println(" gruss1 equals gruss2 ist " + gruss1.equals(gruss2));  
System.out.println(" gruss1 equalsIgnoreCase gruss2 ist " +  
                    gruss1.equalsIgnoreCase(gruss2));
```



```
gruss1 equals gruss2 ist false  
gruss1 equalsIgnoreCase gruss2 ist true
```


Vergleiche von Strings

- Die Methode **compareTo** vergleicht die Strings lexikografisch
 - Das Ergebnis ist 0, wenn sie gleich sind,
 - negativ, wenn der erste String lexikografisch kleiner ist als der zweite
 - positiv, wenn der erste String lexikografisch grösser ist als der zweite
- Die Methode compareTo stammt aus dem Interface Comparable, das von String implementiert wird
 - **ACHTUNG**: trotzdem darf der compareTo Methode für ein String-Objekt nur ein anderes String-Objekt übergeben werden

Strings sind nicht veränderbar

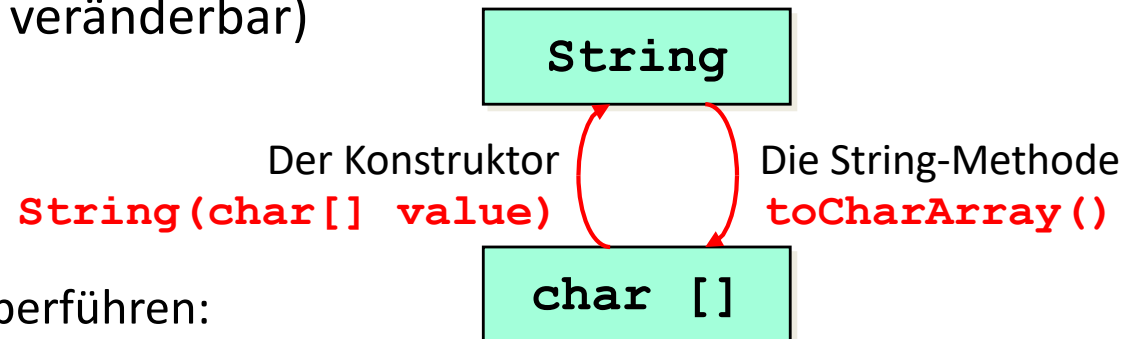
- Die Klasse String bietet nur Methoden, um den Inhalt auszulesen
- Wenn ein Text verändert werden soll, muss ein neues String-Objekt mit dem veränderten Wert erzeugt werden

```
class Reverse{  
  
    /** Methode zum Umdrehen von einer Zeichenkette.  
     * @param s Zeichkette, die umgedreht werden soll.  
     * @return Umgedrehte Zeichenkette  
     */  
    public static String reverseInefficient(String s) {  
        String newString = "";  
        for (int i = 0; i < s.length(); i++) {  
            newString = s.charAt(i) + newString;  
        }  
        return newString;  
    }  
    ...  
}
```

Konkatenation erzeugt einen String.
Im Beispiel: in jedem
Schleifendurchlauf.

Strings und char Arrays (1)

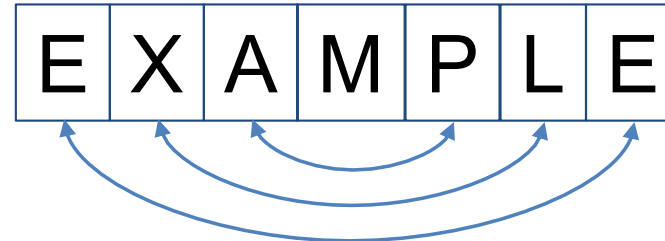
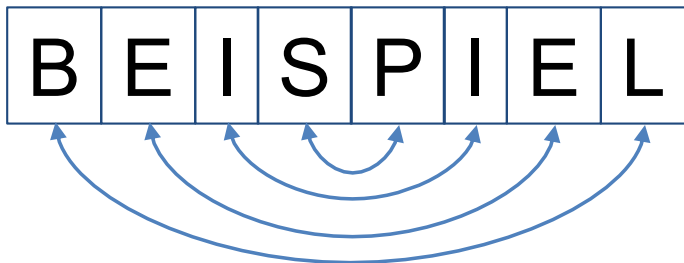
- Umweg über char-Array (das ist veränderbar)
- Strings und char-Arrays haben zwar viele Ähnlichkeiten, sind aber verschiedene Typen.
 - Man kann sie aber ineinander überführen:



```
...  
/** Methode zum Umdrehen von einer Zeichenkette  
 * @param s Zeichkette, die umgedreht werden soll.  
 * @return Umgedrehte Zeichenkette  
 */  
static String reverse(String s){  
    char[] car = s.toCharArray(); // String → char[]  
    int lastIndex = car.length - 1;  
    for (int i = 0; i < (lastIndex + 1)/2; i++)  
        swap(car, i, lastIndex - i);  
    return new String(car); // char[] → String  
}
```

Strings und char Arrays (2)

```
...  
/** Methode zum Umdrehen von einer Zeichenkette  
 * @param s Zeichkette, die umgedreht werden soll.  
 * @return Umgedrehte Zeichenkette  
 */  
static String reverse(String s){  
    char[] car = s.toCharArray(); // String → char[]  
    int lastIndex = car.length - 1;  
    for (int i = 0; i < (lastIndex + 1)/2; i++)  
        swap(car, i, lastIndex - i);  
    return new String(car);        // char[] → String  
}
```



- *Noch besser wäre gewesen, wenn wir zuvor geschaut hätten, ob eine solche Funktion bereits in einer anderen Klasse implementiert wurde.*

Strings und char Arrays (2)

```
...  
/** Die Methode vertauscht in einem char-Array zwei Zeichen.  
 * @param a das char-Array  
 * @param i erste Position im Array a  
 * @param k zweite Position im Array a  
 */  
static void swap(char[] a, int i, int k){  
    char t = a[i];  
    a[i]=a[k];  
    a[k]=t;  
}  
}
```

- *Noch besser wäre gewesen, wenn wir zuvor geschaut hätten, ob eine solche Funktion bereits in einer anderen Klasse implementiert wurde.*

Die Klasse StringBuilder

- Die Klasse **StringBuilder** erlaubt Zeichenketten zu verändern, ohne dabei immer wieder neue Objekte zu erzeugen. Die wesentlichen Operationen sind dabei:
 - **append** zum Anhängen am Ende,
 - **insert** zum Einfügen an einer beliebigen Stelle,
 - **delete** zum Entfernen eines beliebigen Teilstrings.
- Objekte der Klasse **StringBuilder** haben eine variable **Kapazität**, um Zeichen bis zu der Größe aufzunehmen.
 - Mit **trimToSize** kann die Kapazität explizit verkleinert werden, mit **ensureCapacity** kann sie explizit vergrößert werden.
 - Wenn die Kapazität nicht mehr ausreicht, wird automatisch zusätzlicher Speicher angefordert – in größeren Inkrementen.
- Die Klasse **StringBuilder** bietet auch eine Methode **reverse**!
 - **Wir nutzen diese Methode** und vermeiden somit eine Neuentwicklung.

Ausgabe von Zeichenketten

- Problem
 - Direkte Ausgabe einer Zeichenkette oder Zahl soll schöner werden.

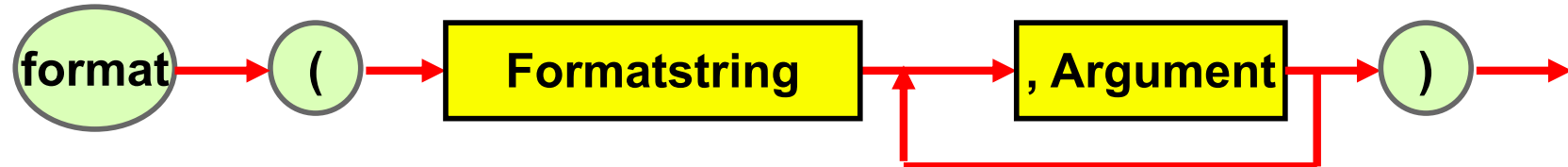
```
System.out.println("Diese Zahl ist zu lang: " + 42.12345678901234);
```

- Es gibt folgende zwei Arten das Format der Ausgabe zu ändern.
 - In der Klasse `PrintStream` gibt es die Methode `printf`.
 - In der Klasse `String` gibt es die static-Methode `format`.
- Beide Methoden beruhen auf dem gleichen Prinzip, das man eine sogenannte **Formatzeichenkette** als Parameter verwendet.

Die Methode format

- Zweck
 - Formatierte Ausgabe von Daten.

- Syntax



- Die statische Methode liefert als Ergebnis die formatierte Zeichenkette zurück und kann beliebig viele Parameter besitzen.

Aufbau eines Formatstrings

- Der Formatstring setzt sich aus **Text und Formatanweisungen** zusammen.
- Eine Formatanweisung besteht aus einem **Prozentzeichen und einem weiteren Zeichen für den Datentyp**.
 - Die Formatanweisung wird durch das nächste noch nicht benutzte Argument der format-Methode ersetzt.
- Formatanweisungen für spezifische Datentypen
 - ganze Zahlen
 - Gleitpunktzahlen
 - Datum

Formatstring für Zahlen

- Ganze Zahlen (Beispiele)
 - %d Ausgabe als Dezimalzahl
 - %o Ausgabe als Oktalzahl ohne Vorzeichen
 - %x Ausgabe als Hexadezimalzahl
- Gleitpunktzahlen (Beispiele)
 - %f Ausgabe von float und double im Format [-]m.d (# Nachpunktstellen = 6).
 - %e Ausgabe von float und double im Format [-]m.dex (#Nachpunktstellen= 6)
 - %g Ausgabe von float und double im Format %e oder %f, je nach Größe des Exponenten

Formatstring für Zahlen

- Ganze Zahlen (Beispiele)

- %d Ausgabe als Dezimalzahl
- %o Ausgabe als Oktalzahl ohne Vorzeichen
- %x Ausgabe als Hexadezimalzahl

- Gleitpunktzahlen (Beispiele)

- %f Ausgabe von float und double im Format [-]m.d (#Nachpunktstellen= 6)
- %e Ausgabe von float und double im Format [-]m.dex (#Nachpunktstellen= 6)
- %g Ausgabe von float und double im Format %e oder %f, je nach Größe des Exponenten

m: Mantisse (Vor-Komma-Stellen)

d: Dezimalstellen

x: Exponent

Beispiele

```
public static void main(String[] args) {  
    int vi = 42;  
    String svi = String.format(  
        "Der Wert von vi ist dezimal %3d, oktal %3o und hexadezimal %3x " ,  
        vi, vi, vi);  
    System.out.println(svi);  
  
    String test = String.format(  
        "Einige Zufallszahlen: %10.4f %10.4f %10.4f %10.4f %10.4f " ,  
        Math.random(), Math.random(), Math.random(), Math.random(),  
        Math.random());  
    System.out.println(test);  
}
```

```
Der Wert von vi ist dezimal  42, oktal  52 und hexadezimal  2a  
Einige Zufallszahlen:      0,0861      0,1296      0,0395      0,4423      0,3328
```

Beispiele

```

public class Main {
    public static void main(String[] args) {
        int vi = 12;
        String svi = String.format(
            "Der Wert von vi ist dezimal %3d,
            vi, vi, vi);
        System.out.println(svi);

        String test = String.format(
            "Einige Zufallszahlen: %10.4f %10.4f %10.4f %10.4f %10.4f " ,
            Math.random(), Math.random(), Math.random(), Math.random(),
            Math.random());
        System.out.println(test);
    }
}

```

Mindestens 3 Zeichen, evtl. mit vorangestellten Leerzeichen.

Mindestens 10 Zeichen (evtl. Leerzeichen vorangestellt), davon 4 für Nachkommastellen.

Verwendet lokale Einstellungen für Formate. Daher wird z.B. hier das Komma als Dezimaltrennzeichen ausgegeben.

```

Der Wert von vi ist dezimal 12, oktal 52 und hexadezimal 2a
Einige Zufallszahlen:      0,0861      0,1296      0,0395      0,4423      0,3328

```

Zusammenfassung

- Die Klasse String kurz vorgestellt.
 - Objekte repräsentieren unveränderbare Zeichenketten.
- Funktionalität der Klasse
 - Konstruktoren
 - Diverse Methoden
 - Formatieren von Zeichenketten
 - **Suche nach Zeichenketten in einer Zeichenkette (siehe Übungen)**
- Assoziierte Klassen
 - StringBuilder