

- Referenzen auf Arrays können **im Rahmen ihrer Deklaration** auf mehrere Weisen **initialisiert** werden

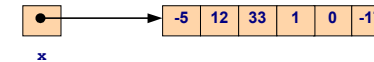
- zum einen, indem die Deklaration mit der Anwendung des **new**-Operators kombiniert wird

```
double[] messwert = new double[7];
```

- zum anderen mit Hilfe so genannter **Arrayinitialisierer**
 - in diesem Fall können auch andere als die Defaultwerte zur Initialisierung benutzt werden
 - wie in C/C++ sind die Initialwerte in geschweiften Klammern – durch Kommata separiert – der Reihe nach anzugeben

```
int[] x = {-5, 12, 33, 8 - 7, 0, -17}; // Arrayinitialisierer
```

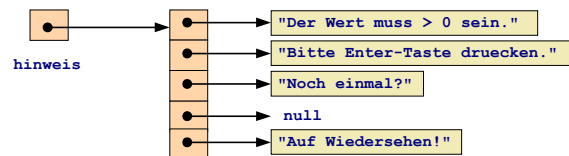
führt zu:



- bei der Initialisierung mit Hilfe eines Arrayinitialisierers erhält das Array genau so viele Komponenten, wie es Werte im Initialisierer gibt
- sie ist natürlich auch dann zulässig, wenn der Komponentendatentyp ein Referenzdatentyp ist

```
String[] hinweis = { "Der Wert muss > 0 sein.",
                    "Bitte Enter-Taste druecken.",
                    "Noch einmal?",
                    null,
                    "Auf Wiedersehen!"
                };
```

führt zu:



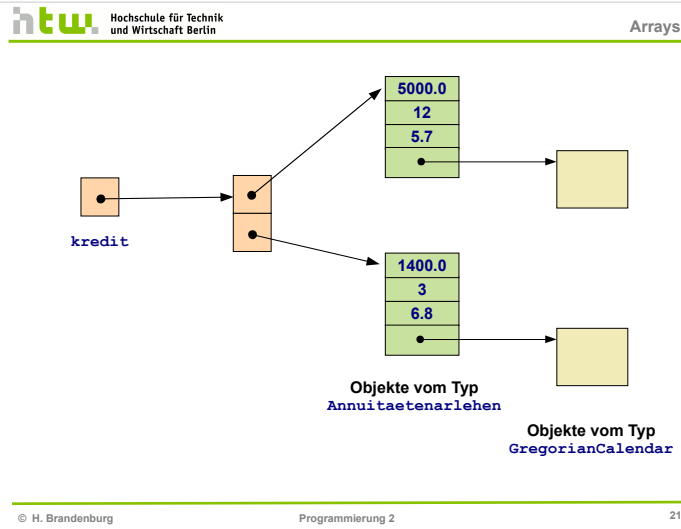
Objekte vom Typ String

- zur Initialisierung** können nicht nur Literale, sondern allgemeiner **auch Ausdrücke** benutzt werden

- im nächsten Beispiel enthält der Initialisierer zwei Ausdrücke, die jeweils aus der Anwendung des unären **new**-Operators bestehen

```
AnnuitaetenDarlehen[] kredit =
{ new AnnuitaetenDarlehen(5000.0, 12, 5.7),
  new AnnuitaetenDarlehen(1400.0,
    3,
    6.8,
    new GregorianCalendar(2011, 2, 15)) };
```

- dadurch entsteht folgende Situation:



htw Hochschule für Technik und Wirtschaft Berlin Arrays

- **Achtung:** das Erzeugen von Arrays mit Hilfe von Arrayinitialisierern ist **nur im Rahmen von Deklarationen** zulässig

```
int[] x = null;
// weitere Anweisungen
x = {-5, 122, 33, 8 - 7, 0, -17}; // nein!
```

führt zu folgender Fehlermeldung:

```
illegal start of expression
  x = {-5, 122, 33, 8 - 7, 0, -17};
    ^
```

© H. Brandenburg Programmierung 2 22

htw Hochschule für Technik und Wirtschaft Berlin Arrays

- es gibt aber noch **eine weitere Variante des new-Operators**, die das Erzeugen von Arrays mit der Initialisierung durch Arrayinitialisierer kombiniert
 - > sie **kann jederzeit eingesetzt werden**, d.h. sowohl bei späteren Wertzuweisungen als auch im Rahmen von Deklarationen
 - letzteres ist aber unüblich, da die ausschließliche Verwendung von Arrayinitialisierern syntaktisch einfacher ist

```
int[] x = null;
// weitere Anweisungen
x = new int[]{-5, 122, 33, 8 - 7, 0, -17}; // ok!
```

Achtung: keine Längenangabe!

© H. Brandenburg Programmierung 2 23

htw Hochschule für Technik und Wirtschaft Berlin Arrays

- der **Zugriff auf die Komponenten** eines Arrays erfolgt nach folgendem allgemeinen Schema:
 - > wenn **x** eine Referenz auf das Array ist, bezeichnet **x[i]** die **i-te** Komponente des Arrays

- > mit ihr kann „ganz normal“ gearbeitet werden:

```
int[] x = {-5, 12, 33, 8 - 7, 0, 4, -17};
System.out.println(x[4]); // lesender Zugriff; Ausgabe: 0
int testWert = x[6]; // lesender Zugriff
x[0] -= testWert; // lesender und schreibender Zugriff
x[4] = x[0] / 2; // lesender und schreibender Zugriff
System.out.println(x[4]); // lesender Zugriff; Ausgabe: 6
```

© H. Brandenburg Programmierung 2 24

- **Achtung:** greift man in einem Java-Programm auf eine nicht existierende Arraykomponente zu, kommt es – anders als in C/C++ – **immer** zu einem **Fehler**

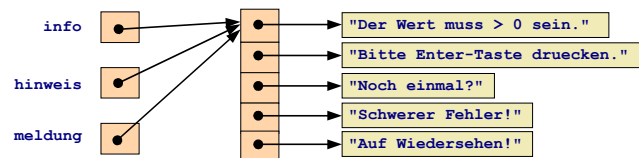
```
double[] temperatur = new double[3];
temperatur[1] = 3.7;
temperatur[2] = 12.1;
temperatur[3] = 23.4; // diese Komponente gibt es nicht!
```

- die Anweisungen werden zwar vom Compiler übersetzt, führen aber bei der Ausführung durch die JVM zu einer **ArrayIndexOutOfBoundsException**

- verweisen mehrere Referenzen auf ein und dasselbe Array, kann **über jede von ihnen** auf die Komponenten des Arrays zugegriffen werden

```
String[] info = { "Der Wert muss > 0 sein.",
                  "Bitte Enter-Taste druecken.",
                  "Noch einmal?",
                  "Schwerer Fehler!",
                  "Auf Wiedersehen!"
                };
String[] hinweis = null;
String[] meldung = null;
hinweis = info;
meldung = hinweis; // (*)
System.out.println(hinweis[0]);
System.out.println(info[3]);
System.out.println(meldung[1]);
System.out.println(info[4]);
```

- nach der durch **(*)** gekennzeichneten Anweisung ist folgende Situation entstanden:



daher wird auf dem Bildschirm ausgegeben:

```
Der Wert muss > 0 sein. // hinweis[0]
Schwerer Fehler! // info[3]
Bitte Enter-Taste druecken. // meldung[1]
Auf Wiedersehen! // info[4]
```

- aus interner Sicht sind **Arraydatentypen spezielle Klassen**, die alle ein Attribut namens **length** haben
 - beim Erzeugen von Arrays mit Hilfe von **new** wird in diesem Attribut automatisch die Länge der Arrays gespeichert
 - weil es **public** und **final** ist, kann auf **length** jederzeit zugegriffen werden
- mit anderen Worten: anders als in C/C++, „weiß“ in Java jedes Array, wie lang es ist

- dadurch entfällt in Java das aus C/C++ bekannte, umständliche Ermitteln der Länge von Arrays
- **for**-Anweisungen zum Bearbeiten der Komponenten eines Arrays „der Reihe nach“ können zum Beispiel einfach so gestaltet werden:

```
int[] x = {1, 2, 3, 5, 7, 11, 13, 17, 19};

for (int i = 0; i < x.length; i++)
    System.out.print(x[i] + " ");           // 1 2 3 5 7 11 13 17 19

for (int i = x.length - 1; i >= 0; i--)
    System.out.print(x[i] + " ");           // 19 17 13 11 7 5 3 2 1

for (int i = 0; i < x.length / 2; i++)
    System.out.print(x[i] + x[x.length - 1 - i] + " "); // 20 19 16 16 14
```

- seit der J2SE 5.0 gibt es in Java eine **neue Variante der for-Anweisung**, die es gestattet, bisher wie Idioms benutzte Konstrukte wie

```
for (int i = 0; i < x.length; i++)
    System.out.print(x[i] + " ");
```

noch kürzer und prägnanter zu formulieren:

```
for (int wert : x)
    System.out.print(wert + " ");
```

- gelesen wird das so:
„for each **wert** in **x** do“ ...
- man nennt die Variante daher **for-each-Anweisung**

- die **for-each**-Anweisung hat den **Vorteil, syntaktisch besonders einfach** zu sein, wie das allgemeine Schema für Arrays zeigt:

```
for (KomponentenDatentyp komponente : arrayReferenz)
    Anweisung
```

- **arrayReferenz** muss eine Referenz auf ein Array sein
- **KomponentenDatentyp** muss der Komponententyp dieses Arrays sein
- **komponente** muss ein frei wählbarer Bezeichner für den Inhalt der Komponenten des Arrays sein

- wie das folgende Beispiel zeigt, ist sie hervorragend geeignet, um den Inhalt aller Komponenten eines Arrays der Reihe nach zu „verarbeiten“:
- unsere **enum**-Klasse **PapierFormat** besitzt wie alle **enum**-Klassen die **static**-Methode **values**, die ein Array liefert, das alle **enum**-Konstanten enthält
- wir benutzen die **for-each**-Anweisung, um die darin enthaltenen Informationen auf dem Bildschirm auszugeben

```
public class PapierFormatMain
{
    public static void main(String[] args)
    {
        System.out.println("\nInformationen zu Papierformaten:\n");
        PapierFormat[] papierFormate = PapierFormat.values();
        for (PapierFormat format : papierFormate)
            schreibeFormatInfo(format);
    }

    private static void schreibeFormatInfo(PapierFormat format)
    {
        String ausgabe = String.format("%-10s", format.liefereFormatBezeichnung());
        ausgabe += String.format("%-16s", format.liefereBezeichnung());
        ausgabe += String.format("%5d mm \u00D7 ", format.liefereBreite());
        ausgabe += String.format("%4d mm =", format.liefereHoehe());
        ausgabe += String.format("%8d qmm", format.liefereFlaeche());
        System.out.println(ausgabe);
    }
}
```

auf dem Bildschirm wird ausgegeben:

Informationen zu Papierformaten:

Din A0	Doppelbogen	841 mm × 1189 mm	=	999949 qmm
Din A1	Bogen	594 mm × 841 mm	=	499554 qmm
Din A2	Halbbogen	420 mm × 594 mm	=	249480 qmm
Din A3	Viertelbogen	297 mm × 420 mm	=	124740 qmm
Din A4	Blatt	210 mm × 297 mm	=	62370 qmm
Din A5	Halbblatt	148 mm × 210 mm	=	31080 qmm
Din A6	Viertelblatt	105 mm × 148 mm	=	15540 qmm
Din A7	Achtelblatt	74 mm × 105 mm	=	7770 qmm
Din A8	Sechzehntelblatt	52 mm × 74 mm	=	3848 qmm
Din A9		37 mm × 52 mm	=	1924 qmm
Din A10		26 mm × 37 mm	=	962 qmm

- die **for-each**-Anweisung hat aber den **Nachteil**, dass **immer alle Komponenten** des Arrays bearbeitet werden, und zwar immer von links nach rechts
 - daher ist sie ungeeignet, wenn eine andere Reihenfolge sinnvoll ist oder wenn nur Teile eines Arrays betrachtet werden sollen
- nachteilig ist auch, dass **immer nur auf den Inhalt** der Komponenten zugegriffen werden kann, **nie auf deren Index**
 - weshalb sie bei weitem nicht so flexibel eingesetzt werden kann, wie die traditionelle **for**-Anweisung

- **Fazit:** in vielen Situationen ist die **for-each**-Anweisung eine **elegante Alternative** zur traditionellen **for**-Anweisung
 - aber: alles, was bei Arrays mit der **for-each**-Anweisung erreicht werden kann, kann auch mit der traditionellen **for**-Anweisung erreicht werden
 - umgekehrt ist das nicht immer möglich
- es ist daher jeweils genau zu überlegen, welche der beiden Varianten wann geeignet ist


- selbstverständlich können **Arrays** auch **als Input an Methoden** übergeben werden
 - anders als in C/C++ ist es dabei nicht nötig, die Länge separat zu übergeben
- weil bei der Übergabe nur Referenzen (= Zeiger) kopiert werden, sind Änderungen, die an einem als Input übergebenen Array innerhalb einer Methode vorgenommen werden, stets auch außerhalb der Methode sichtbar
- wie in C/C++ bietet es sich an, frühzeitig umfangreiche Sammlungen allgemeiner, wiederverwendbarer „Dienstleister“ für Arrays anzulegen

- Parametern von Methoden, deren Datentyp ein Arraydatentyp ist, können beim Aufruf **Arrays beliebiger Länge übergeben** werden

```
int[] x = new int[20000];
initialisiereArray(x, -5);
int[] y = {-34, 19};
long summe1 = liefereArraySumme(x);    // Länge des Inputs: 20000
long summe2 = liefereArraySumme(y);    // Länge des Inputs: 2
System.out.println(summe1);           // -100000
System.out.println(summe2);           // -15
```

- fasst man die übergebenen Werte nicht als Komponenten eines Arrays auf, sondern als Einzelwerte, entsteht der Eindruck, dass die Methode **liefereArraySumme** **variabel viele Inputparameter** vom Typ **int** hat

- diese Auffassung wird seit der J2SE 5.0 durch eine spezielle Syntax zur **Deklaration von Parameterlisten mit einer variablen Anzahl von Parametern** unterstützt



```
public long summiere(int ... summand)
{
    boolean inputOk = (summand != null) && (summand.length > 0);
    if (inputOk)
    {
        long summe = 0;
        for (int wert : summand)                // (*)
            summe += wert;
        return summe;
    }
    else
        throw new IllegalArgumentException();
}
```

- das Symbol **...** heißt **Ellipse** und wird wie folgt interpretiert:
 - der Methode **summiere** kann kein Wert, ein Wert oder mehrere Werte vom Datentyp **int** als Input übergeben werden
- die so deklarierte Methode kann sehr flexibel eingesetzt werden:

```
System.out.println(summiere(99));           // 99
System.out.println(summiere(1, 99));        // 100
System.out.println(summiere(1, 99, -12, -60, 22)); // 50
```

- es ist aber wichtig, zu verstehen, dass es sich bei Parameterlisten mit einer variablen Anzahl von Parametern **lediglich** um eine **syntaktische Neuerung** handelt
- wie die durch **(*)** markierte Anweisung von **summiere** ahnen lässt, ist ein mit einer Ellipse deklarierter Parameter nichts anderes als eine Referenz auf ein Array, dessen Komponententyp identisch ist mit dem vor der Ellipse angegebenen Typ

```
public long summiere(int ... summand)
```

ist intern tatsächlich dasselbe wie

```
public long summiere(int[] summand)
```

- so wurde der Parameter im Code von **summiere** auch behandelt !
- weil **summand** in Wirklichkeit eine Referenz auf ein **int**-Array ist, kann **summiere** sogar mit einem Array als Input aufgerufen werden

```
int[] x = new int[20000];
initialisiereArray(x, -5);
System.out.println(summiere(x));
```

- die einzige Neuerung durch Parameterlisten mit variabler Anzahl von Parametern ist also die Möglichkeit, **summiere** einfach so aufrufen zu können

```
long summe = summiere(1, 99, -12, -60, 22);
```

- ein derartiger Aufruf wird vom Compiler intern so behandelt:

```
int[] dummy = new int[] {1, 99, -12, -60, 22};
summand = dummy;
```

- **Achtung:** bei der Deklaration von Parametern mit Ellipse sind folgende **Einschränkungen** zu beachten
 - in jeder Parameterliste darf es nur **höchstens einen Parameter mit Ellipse** geben
 - und dieser muss **immer der letzte** sein

- wie in C/C++ können Java-Programmen beim Aufruf zusätzliche **Argumente** übergeben werden

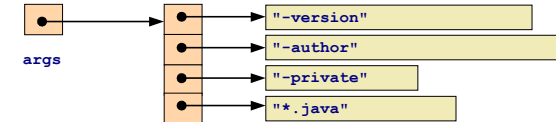
Beispiel:

```
javadoc -version -author -private *.java
```

Argument 0 Argument 1 Argument 2 Argument 3

- sie werden vom Betriebssystem der JVM übergeben, die sie als **String**-Array an den Parameter **args** von **main** weiterreicht

Achtung: anders als in C/C++ wird der Name des Programms nicht übergeben !



- das Programm kann diesen Input beliebig auswerten

- wenn es keine zusätzlichen Argumente gibt, erhält **args** ein leeres **String**-Array als Input

- es ist allgemein üblich, **main** so zu deklarieren:

```
public static void main(String[] args)
{
    // Anweisungen
}
```

- der Vollständigkeit halber erwähnen wir, dass es stattdessen auch möglich wäre,
 - einen anderen Bezeichner statt **args** zu wählen
 - statt eines **String**-Arrays einen **String**-Parameter mit Ellipse vorzusehen (seit J2SE 5.0)
- wir raten aber davon ab, von dieser Möglichkeit Gebrauch zu machen
- wie in C/C++ sind Programme, die zusätzliche Argumente erwarten, häufig nach folgendem Schema gestaltet:

```
public static void main(String[] args)
{
    if (parameterOk(args))
    {
        // führe das Programm aus
    }
    else
        schreibeAnleitung();
}
```

- hierbei ist **parameterOk** eine Hilfsmethode, die nur dann **true** liefert, wenn die übergebenen Argumente den Erwartungen entsprechen
- und **schreibeAnleitung** eine Hilfsmethode, die auf der Konsole eine Anleitung zur Bedienung des Programms ausgibt

- wir illustrieren das Prinzip anhand eines einfachen Programms zum Erstellen eines Abschreibungsplans bei linearer Abschreibung
- wie die **Ausgabe der Bedienungsanleitung** zeigt, erwartet das Programm drei zusätzliche Argumente

```
java LineareAbschreibungMain
```

```
Benutzung : java LineareAbschreibungMain startWert endWert anzJahre
startWert : Gleitkommazahl >= 1
endWert : 0 <= Gleitkommazahl <= startWert
anzJahre : ganze Zahl >= 1
```

- die Überprüfung des Inputs gestalten wir so:

```
private static boolean parameterOk(String[] args)
{
    boolean ok = false;
    if (args.length == 3)
    {
        try
        {
            double startWert = Double.parseDouble(args[0]);
            double endWert = Double.parseDouble(args[1]);
            int anzahlJahre = Integer.parseInt(args[2]);
            ok = (1 <= startWert) && (0 <= endWert) &&
                (endWert <= startWert) && (1 <= anzahlJahre);
        }
        catch (NumberFormatException nfe)
        {
        }
    }
    return ok;
}
```

- **main** ist nach dem allgemeinen Schema gestaltet

```
public static void main(String[] args)
{
    if (parameterOk(args))
    {
        double startWert = Double.parseDouble(args[0]);
        double endWert = Double.parseDouble(args[1]);
        int anzahlJahre = Integer.parseInt(args[2]);
        String ueberschrift = "\n\nAbschreibungsplan bei ";
        ueberschrift += "linearer Abschreibung\n\n";
        LinearAbschreiber abschreiber = new LinearAbschreiber(startWert,
                                                                endWert,
                                                                anzahlJahre);

        double[] dieDaten = abschreiber.lieferePlanDaten();
        double dieAbschreibung = abschreiber.liefereAbschreibung();
        schreibeAbschreibungsplan(ueberschrift,
                                   dieDaten,
                                   dieAbschreibung);
    }
    else
        schreibeAnleitung();
}
```

- bei korrekten Argumenten gibt das Programm zum Beispiel aus:

```
java LineareAbschreibungMain 123456.78 987.65 10
```

Abschreibungsplan bei linearer Abschreibung

Jahr	Anfang	Abschreibung	Ende
1	123456,78	12246,91	111209,87
2	111209,87	12246,91	98962,95
3	98962,95	12246,91	86716,04
4	86716,04	12246,91	74469,13
5	74469,13	12246,91	62222,22
6	62222,22	12246,91	49975,30
7	49975,30	12246,91	37728,39
8	37728,39	12246,91	25481,48
9	25481,48	12246,91	13234,56
10	13234,56	12246,91	987,65

- die Klasse **LinearAbschreiber** hat die Attribute **startWert**, **endWert** und **anzahlJahre**, die vom Konstruktor initialisiert werden

```
public LinearAbschreiber(double derStartWert,
                        double derEndWert,
                        int dieAnzahlJahre)
{
    boolean inputOk = (1 <= derStartWert) && (0 <= derEndWert) &&
                      (derEndWert <= derStartWert) &&
                      (1 <= dieAnzahlJahre);

    if (inputOk)
    {
        startWert = derStartWert;
        endWert = derEndWert;
        anzahlJahre = dieAnzahlJahre;
    }
    else
        throw new IllegalArgumentException();
}
```

- die Methoden **lieferePlanDaten** und **liefereAbschreibung** der Klasse **LinearAbschreiber** liefern die Daten für den Abschreibungsplan

```
public double[] lieferePlanDaten()
{
    double abschreibung = liefereAbschreibung();
    double[] planDaten = new double[anzahlJahre];
    for (int i = 0; i < planDaten.length; i++)
        planDaten[i] = (i == 0) ? startWert :
                        planDaten[i - 1] - abschreibung;
    return planDaten;
}

public double liefereAbschreibung()
{
    return (startWert - endWert) / anzahlJahre;
}
```

- geschrieben wird der Plan von der Methode **schreibeAbschreibungsplan** der Klasse **LineareAbschreibungMain**

```
private static void schreibeAbschreibungsplan(String info,
                                              double[] planDaten,
                                              double abschreibung)
{
    schreibeUeberschrift(info);
    for (int i = 0; i < planDaten.length; i++)
    {
        System.out.printf("%5d%14.2f", i + 1, planDaten[i]);
        System.out.printf("%14.2f", abschreibung);
        if (i < planDaten.length - 1)
            System.out.printf("%14.2f\n", planDaten[i + 1]);
        else
            System.out.printf("%14.2f\n", planDaten[i] - abschreibung);
    }
}
```

- die Methode **lieferePlanDaten** macht davon Gebrauch, dass der **Rückgabewert von Methoden** in Java – anders als in C/C++ – **ein Array** sein kann
 - daher erweist sich die Einschränkung, dass eine Methode nur maximal einen Wert als Output haben kann, als nicht wesentlich
 - bei Methoden, die mehr als einen Wert als Output liefern sollen, muss der **Output** lediglich **komplexer** sein!
- warum das in Java möglich ist, machen wir uns an Hand von **lieferePlanDaten** klar

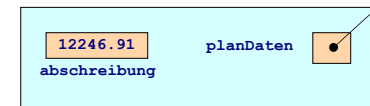
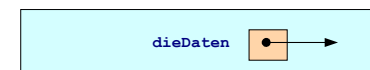
```
public double[] lieferePlanDaten()
{
    double abschreibung = liefereAbschreibung();
    double[] planDaten = new double[anzahlJahre];
    for (int i = 0; i < planDaten.length; i++) // (*)
        planDaten[i] = (i == 0) ? startWert :
                        planDaten[i - 1] - abschreibung;
    return planDaten;
}
```

- in **main** wird die Methode so aufgerufen:

```
double[] dieDaten = abschreiber.lieferePlanDaten();
```

- daher liegt nach Ausführung der durch **(*)** markierten Anweisung folgende Situation vor:

Teil der lokalen Daten von **main** im Stacksegment



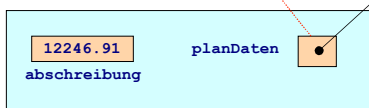
lokale Daten von **lieferePlanDaten** im Stacksegment

123456.78
111209.87
98962.95
86716.04
74469.13
62222.22
49975.30
37728.39
25481.48
13234.56

auf dem Heap

- bei der Rückgabe des Outputs wird der Wert von **planDaten** nach **dieDaten** **kopiert**, wodurch diese Situation entsteht:

Teil der lokalen Daten von **main** im Stacksegment



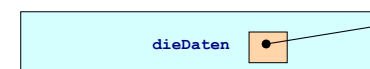
lokale Daten von **lieferePlanDaten** im Stacksegment

123456.78
111209.87
98962.95
86716.04
74469.13
62222.22
49975.30
37728.39
25481.48
13234.56

auf dem Heap

- danach werden die lokalen Daten von **lieferePlanDaten** vom Stack entfernt

Teil der lokalen Daten von **main** im Stacksegment



das Array verbleibt aber auf dem Heap und kann über die Referenz **dieDaten** weiter benutzt werden!

123456.78
111209.87
98962.95
86716.04
74469.13
62222.22
49975.30
37728.39
25481.48
13234.56

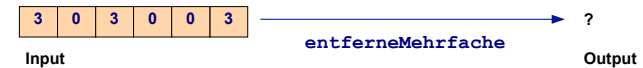
auf dem Heap

- bei der Deklaration von Methoden, deren **Output ein Array** ist, kommt es gelegentlich zu einem Problem, das wir an einem einfachen Beispiel erläutern

- es ist eine Methode **entferneMehrfache** zu schreiben, die Folgendes leistet:
 - Input ist ein **int**-Array
 - Output ist das **int**-Array, das aus dem Input dadurch entsteht, dass alle Werte, die mehrfach in ihm vorkommen, entfernt werden



- welchen Output soll **entferneMehrfache** liefern, wenn **alle** Werte des Inputs mehrfach vorkommen?



- für derartige Situationen ist es äußerst nützlich, dass es in Java zulässig ist, **Arrays der Länge 0** zu bilden
 - gäbe es diese Möglichkeit (wie zum Beispiel in C) nicht, müsste **entferneMehrfache** in diesem Fall **null** als Rückgabewert liefern
 - dann dürfte das Ergebnis aber nicht Input von Methoden sein, die ein Array erwarten

- unsere (nicht allzu effiziente) Implementierung von **entferneMehrfache** liefert in der oben beschriebenen Situation tatsächlich ein Array der Länge 0

```
public int[] entferneMehrfache(int[] array)
{
    if (array != null)
    {
        int[] puffer = new int[array.length];
        int nichtBelegt = 0;
        for (int wert : array)
        {
            if (wieOftEnthalten(wert, array) == 1)
                puffer[nichtBelegt++] = wert;
        }
        int[] ergebnis = new int[nichtBelegt];
        for (int i = 0; i < nichtBelegt; i++)
            ergebnis[i] = puffer[i];
        return ergebnis;
    }
    else
        throw new IllegalArgumentException();
}
```

- sie kann daher auch als Input einer Methode **schreibeArray** dienen

```
int[] x = {9, -2, 0, 1, 7, 0, -1};
schreibeArray(entferneMehrfache(x), " "); // 9 -2 1 7 -1
System.out.println();
int[] y = {9, -2, 0, 1, 7, 0, -1, 1, 9, -2, -1, 7};
schreibeArray(entferneMehrfache(y), " "); // " "
```

- **entferneMehrfache** benutzt eine Variante der Methode **wieOftEnthalten**, die zählt, wie oft ein **int**-Wert in einem **int**-Array vorkommt

```
public int wieOftEnthalten(int wert, int[] array)
{
    return wieOftEnthalten(wert, 0, array.length - 1, array);
}
```

- › sie basiert auf dieser Variante von **wieOftEnthalten**

```
public int wieOftEnthalten(int wert, int von, int bis, int[] array)
{
    boolean inputOk = (array != null) &&
        (0 <= von) &&
        (von <= bis) &&
        (bis < array.length);

    if (inputOk)
    {
        int wieOft = 0;
        for (int i = von; i <= bis; i++)
            wieOft += (wert == array[i] ? 1 : 0);
        return wieOft;
    }
    else
        throw new IllegalArgumentException();
}
```

- **Hinweis:** wiederverwendbare Dienstleister sollten stets **möglichst allgemein** gehalten werden
 - › daher sollten Sie (zur Übung) die Methode **entferneMehrfache** unter folgenden Aspekten verallgemeinern
 - im Output-Array sind genau die Werte enthalten, die im Input-Array genau n-mal vorkommen (bei uns ist n = 1)
 - es sollte möglich sein, nur Teile von Arrays zu bearbeiten (von ... bis)

- Referenzen auf Arrays können selbstverständlich auch als **final** deklariert werden (Attribute, Parameter, Variablen)
 - › dadurch wird festgelegt, dass **der Wert der Referenz nicht verändert** werden kann

```
final String[] info = {"Das", "ist", "ok!"};
info = new String[]{"Wegen final", "ist", "das", "nicht", "ok!"};
```

führt daher zu folgender **Fehlermeldung**

```
cannot assign a value to final variable info
info = new String[]{"Wegen final", "ist", "das", "nicht", "ok!"};
^
1 error
```

- **Achtung:** **final** wirkt sich aber **nicht auf den Inhalt** von Arrays aus!

```
final String[] info = {"Das", "ist", "ok!"};
for (String text : info)
    System.out.print(text + " ");           // Das ist ok!
System.out.println();
info[0] = "Aendern des Inhalts";             // ok!
info[2] = "aber trotz final moeglich!";      // ok!
for (String text : info)
    System.out.print(text + " ");
// Aendern des Inhalts ist aber trotz final moeglich!
```

- **Hinweis:** es gibt keine Möglichkeit, den Inhalt von Arrays vor Änderungen zu schützen!

- von besonderer Wichtigkeit ist diese Erkenntnis für **private Attribute** von Klassen, **die Referenzen auf Arrays sind**
 - wir illustrieren das am Beispiel einer (bewusst einfach gehaltenen) Klasse **TagesTemperatur** zum Erfassen und Speichern von Temperaturen
 - sie hat zwei private Attribute:

```
public class TagesTemperatur
{
    private MeinDatum tag = null;
    private double[] temperatur = null;
}
```

tag dient zum Speichern eines Datums, wobei **MeinDatum** eine selbst geschriebene Klasse mit den Attributen **tag**, **monat** und **jahr** ist

- temperatur** ist eine Referenz auf ein **double**-Array, in dessen **i**-ter Komponente die zur Stunde **i** gemessene Temperatur gespeichert werden soll
- die Temperaturen werden (in der Realität zum Beispiel durch Sensoren) erfasst und dem Konstruktor als Input übergeben

```
public TagesTemperatur(MeinDatum derTag, double[] dieTemperatur)
{
    tag = derTag;
    temperatur = dieTemperatur;
}
```

- Objekte der Klasse **TagesTemperatur** sollen zum Beispiel eingesetzt werden, um Informationen folgender Art zu erzeugen:

Werte fuer den 12. 4. 2008 :

0.00 Uhr :	3,0	12.00 Uhr :	8,4
1.00 Uhr :	2,3	13.00 Uhr :	9,5
2.00 Uhr :	1,7	14.00 Uhr :	10,8
3.00 Uhr :	1,2	15.00 Uhr :	12,3
4.00 Uhr :	0,2	16.00 Uhr :	14,1
5.00 Uhr :	-0,3	17.00 Uhr :	13,2
6.00 Uhr :	1,0	18.00 Uhr :	12,0
7.00 Uhr :	2,4	19.00 Uhr :	11,2
8.00 Uhr :	3,2	20.00 Uhr :	10,1
9.00 Uhr :	4,4	21.00 Uhr :	8,6
10.00 Uhr :	5,9	22.00 Uhr :	7,0
11.00 Uhr :	6,9	23.00 Uhr :	5,7

Durchschnittstemperatur: 6,5

- dazu muss es möglich sein, auf den Inhalt der privaten Attribute **tag** und **temperatur** zuzugreifen
 - das ermöglichen zum Beispiel folgende Methoden:

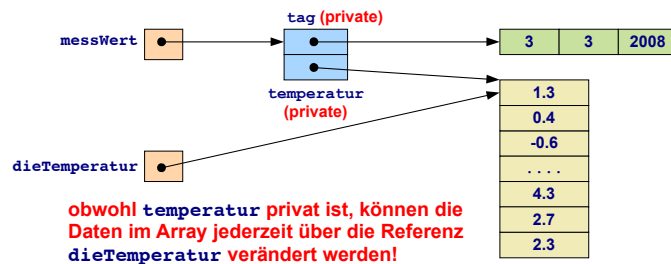
```
public MeinDatum liefereDenTag()
{
    return tag;
}
```

```
public double[] liefereDieTemperatur() // gefährlich !
{
    return temperatur;
}
```

- obwohl sie harmlos aussieht, hat die so implementierte Methode **liefereDieTemperatur** einen **erheblichen Mangel**, den wir am folgenden Beispiel verdeutlichen

```
TagesTemperatur messWert =
    new TagesTemperatur(new MeinDatum(3, 3, 2008),
        erzeugeTemperaturen());
double[] dieTemperatur = messWert.liefereDieTemperatur();
```

> die Anweisungen führen zu dieser Situation:



- durch die Methode **liefereDieTemperatur** kann die für die Objektorientierung charakteristische Datenkapselung umgangen werden!

```
TagesTemperatur messWert =
    new TagesTemperatur(new MeinDatum(3, 3, 2008),
        erzeugeTemperaturen());
schreibeTemperaturen(messWert, 5); // vorher
double[] temperatur = messWert.liefereDieTemperatur(); // (*)
temperatur[3] = 123.4; // (*)
schreibeTemperaturen(messWert, 5); // nachher
```

vorher

0.00 Uhr :	1,3
1.00 Uhr :	0,4
2.00 Uhr :	-0,6
3.00 Uhr :	-1,8
4.00 Uhr :	-2,9

die Änderung (*) des Arrays **temperatur** ändert auch das Objekt **messWert** !

nachher

0.00 Uhr :	1,3
1.00 Uhr :	0,4
2.00 Uhr :	-0,6
3.00 Uhr :	123,4
4.00 Uhr :	-2,9

- Fazit:** Methoden, die **public** sind, sollten als Output **nie eine Referenz auf ein als private deklariertes Array liefern!**

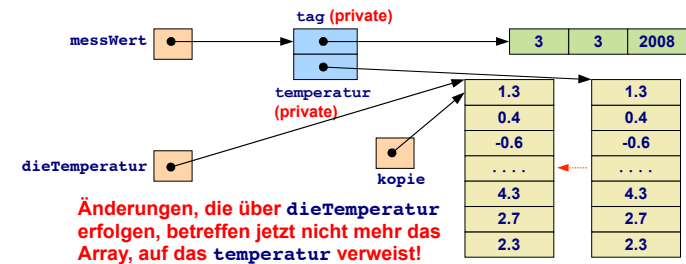
> in unserem Fall kann aber gefahrlos eine Referenz auf eine **Kopie** des privaten Arrays geliefert werden

```
public double[] liefereDieTemperatur()
{
    double[] kopie = new double[temperatur.length];
    for (int i = 0; i < temperatur.length; i++)
        kopie[i] = temperatur[i];
    return kopie;
}
```

> wir zeigen, warum es mit der so deklarierten Methode keine Probleme gibt

```
TagesTemperatur messWert =
    new TagesTemperatur(new MeinDatum(3, 3, 2008),
        erzeugeTemperaturen());
double[] dieTemperatur = messWert.liefereDieTemperatur();
```

die Anweisungen führen jetzt zu dieser Situation:



- › tatsächlich ist bei der so implementierten Methode **liefereDieTemperatur** das Problem verschwunden

```
TagesTemperatur messWert =
    new TagesTemperatur(new MeinDatum(3, 3, 2008),
        erzeugeTemperaturen());
schreibeTemperaturen(messWert, 5); // vorher
double[] temperatur = messWert.liefereDieTemperatur();
temperatur[3] = 123.4; // (*)
schreibeTemperaturen(messWert, 5); // nachher
```

vorher

0.00 Uhr :	1,3
1.00 Uhr :	0,4
2.00 Uhr :	-0,6
3.00 Uhr :	-1,8
4.00 Uhr :	-2,9

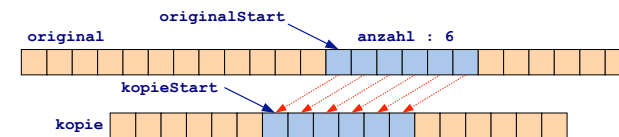
die Änderung (*) des
Arrays **temperatur**
hat keinen Einfluss
auf das Objekt
messWert !

nachher

0.00 Uhr :	1,3
1.00 Uhr :	0,4
2.00 Uhr :	-0,6
3.00 Uhr :	-1,8
4.00 Uhr :	-2,9

- weil Arrays häufig kopiert werden müssen, gibt es in der Klasse **java.lang.System** der Java-Klassenbibliothek eine vordefinierte Methode **arraycopy**
 - › mit ihr kann ein Teil eines Originalarrays in ein Zielarray kopiert werden:

```
public static void arraycopy(Object original, int originalStart,
    Object kopie, int kopieStart,
    int anzahl)
```



- **Achtung:** damit **arraycopy** korrekt arbeitet, müssen **einige Voraussetzungen** erfüllt sein, die in deren Dokumentation beschrieben sind
- **arraycopy** ist einer selbst geschriebenen Lösung stets vorzuziehen, weil sie
 - › schneller ist
 - die Methode ist nicht in Java geschrieben, sondern **native**
 - › sogar funktioniert, wenn **original** und **kopie** identisch sind und sich der zu kopierende Bereich und der Zielbereich überlappen

- › unsere Methode **liefereDieTemperatur** implementieren wir daher besser so:

```
public double[] liefereDieTemperatur()
{
    double[] kopie = new double[temperatur.length];
    System.arraycopy(temperatur, 0, kopie, 0, temperatur.length);
    return kopie;
}
```

- **Achtung:** unser „Trick“, den Mangel der ersten Version von **liefereDieTemperatur** einfach durch Kopieren des Arrays zu beheben, **funktioniert leider nicht immer**