

C++- Standard- Bibliothek

Heinz Dobler

Version 19, 2025



Permission to copy without any fee all or part of these slides is granted provided that copies are not made or distributed for commercial advantage, the copyright notice, the title of the presentation as well as its date appear, and notice is given that "copying is by permission of Dr. Heinz Dobler". To copy otherwise, or to republish, requires a fee and/or the specific permission.

- Motivation und Geschichte
- Grobstruktur und Übersicht
- Basiskonzepte (z. B. Namenraum *std*)
- Zeichenketten (*string*-Klassen)
- Weitere Komponenten (nicht näher behandelt)
- Ein-/Ausgabe
 - Vergleich von C++ mit C
 - Hierarchie der Klassen(schablonen) und ihre Zusammenhänge
 - Übersicht über Header-Dateien
 - Wichtige Klassen(schablonen) im Detail
- Exkurs: Manipulatoren
 - Idee
 - Realisierung (primitive und elegante)
 - Manipulatoren mit Parametern
- Formatierung

C-Bibliothek: Basis des Erfolgs von ANSI/ISO C

- Standard für die Sprache C
- Einheitliche "Basis" in Form der Standardbibliothek für C
- ⇒ Hohe Portabilität von C-Programmen und damit hohe Akzeptanz

Status bei C++ vor Standardisierung

- Mehrere, aber sehr ähnliche "Dialekte" von C++
- Gemeinsamkeiten bei den Klassenbibliotheken:
 - z. B. "ähnliche" Implementierungen der *iostream*-Klassen
- ⇒ Ziel: einheitliche Sprache und einheitl., umfangreichere Standardbibliothek

Geschichte

- 1989 Einrichtung eines Komitees zur Standardisierung von C++ (unter ANSI)
- 1991 Zusammenführung von ANSI X3J16, ISO WG21 und anderen Gremien
- 1994 Erweiterung der Bibliothek um die *Standard Template Library (STL)* aus Ada
- 1995/7 Fertigstellung der *Draft Version* und später der *Final Draft Version*
- 1998 Verabschiedung des ersten C++-Standards **C++ 1998**,
Umfang: fast 800 Seiten, 27 Kapitel (16 Sprache, 11 Bibliothek), 4 Anhänge
- 3 J. 2011 Verabschiedung des neuen C++-Standards **C++ 2011**: über 1300 Seiten
- 2014 C++ 14 mit kleineren Erweiterungen aber auch Abschwächungen von C++ 11
- 2017 Neue Version d. Standards (z. B. mit Bibliothek *filesystem: path, directory...*)
- 2020 Neue Version d. Standards (z. B. mit Drei-Wege-Vergleich <=> und Modulen)
- 2023 Neue Version d. Standards (z. B. auch Standardbibliothek in Form von Modulen)

- **Behälter, Iteratoren, Algorithmen** und ... (= STL)
 - **Ein/Ausgabe:** *iostream*-Klassen und deren Basis
 - **Zeichenketten:** insbesond. Klasse *string* zur Darstellung und Manipulation von Zeichenketten
 - **Mathematik:** insbesondere Schablonenklasse *complex* und Unterstützung von Berechnungen auf "Vektormaschinen" (Schablonenklasse *valarray*)
 - **Sprachunterstützung:** Implementierung spezieller Sprachmittel (z. B. dynamische Speicherplatzverwaltung)
 - **Allgemeine Dienste:** Hilfsdienste für alle anderen Komponenten in der Bibliothek und in Anwendungen
 - **Diagnose:** exception-Klassen, einheitliche Fehlermeldungen
 - **Lokalisierung:** Unterstützung landesüblicher Formate (z. B. Zahlen- und Datumsformate), Sortierreihenfolgen
- Standard Template Library (STL)**

Behälter	Iteratoren	Algorithmen
Ein/ Ausgabe	Zeichen- ketten	Mathematik
Diagnose		Lokalisierung
Sprachunterstützung		Allgemeine Dienste

Namenraum std

- Namenraumname `std` ist für Standardbibliothek reserviert
- Alle Komponenten der C++-Standardbibliothek (auch die aus der C-Standardbibliothek) sind im Namenraum `std` vereinbart, z. B.:

```
std::printf( "Hello, World\n" );           // <cstdio>
std::cout << "Hello, World" << std::endl; // <iostream>
```

Header-Dateien

- Header-Dateien d. C++-Standardbib. haben keine Endung mehr, z. B.:
`#include <iostream>`
- Header-Dateien der C-Standardbib. haben Präfix `c` zur Vermeidung von Mehrdeutigkeiten (notwendig aber nur wegen `string`), z. B.:
`#include <cstdlib>`
- Früher war die Endung ".h" bei Header-Dateien für die Kompatibilität zu alten C++-Dialekten noch erlaubt, z. B.:

```
#include <iostream.h>
```

stand für (und wird übersetzt in):

```
#include <iostream>
using namespace std;
```

```
#include <stdlib.h>
```

stand für (und wird übersetzt in):

```
#include <cstdlib>
using namespace std;
```

Zeichenketten und Darstellungen

- Zeichenkette ist Folge von "Werten" (nicht unbedingt vom Typ *char*)
- "Werte" können z. B. vom Typ *char*, *wchar_t* oder von einem (auch benutzerdefinierten) "passenden" Typ sein
- Zwei Möglichkeiten der Zeichenkettendarstellung:
 1. C-Zeichenketten (Typen *char[]* oder *char**) mit Unterstützung durch C-Bibliothek aus *<cstring>*
 2. C++-Zeichenketten durch spezielle *string*-Klassen aus *<string>*

Realisierung der C++-Zeichenketten

- Zwei generische Hilfsklassen (*template classes*):

```
template<class charT> struct char_traits; //comp.&assign  
template<class T> class allocator; //mem. alloc.
```
- Gen. Klasse *basic_string* als Basis für alle *string*-Klassen:

```
template <class charT, class traits=..., class alloc=...>  
class basic_string; // string w. elements of type charT
```
- Einige vordefinierte Ausprägungen (Klassen) dieser Schablone, z. B.:

```
typedef basic_string<char> string;  
typedef basic_string<wchar_t> wstring;
```

Exkurs: *char_traits*<...> für Eigenschaften von Zeichen

```
template<class charT>
struct char_traits { // no data comp., public static methods only
    typedef charT char_type; ...


---


    static void assign( char_type &c1, const char_type &c2); // c1 = c2 ;
    static bool eq(const char_type &c1, const char_type &c2); // c1 == c2 ?
    static bool lt(const char_type &c1, const char_type &c2); // c1 < c2 ?
    static int compare(const char_type *s1, const char_type *s2, size_t n);
    static size_t length(const char_type *s);
    static const char_type* find(const char_type *s, size_t n, const char_type &a);


---


    static char_type *move (char_type *s1, const char_type *s2, size_t n);
    static char_type *copy (char_type *s1, const char_type *s2, size_t n);
    static char_type *assign(char_type *s, size_t n, char_type a);


---


    static int_type not_eof (const int_type &c);
    static char_type to_char_type(const int_type &c);
    static int_type to_int_type (const char_type &c);
    static bool eq_int_type (const int_type &c1, const int_type &c2);
    static int_type eof();
}; // char_traits
```

Exkurs: *allocator*<...> für Speicherverwaltung

```
template <class T>
class allocator {      // no data components, so alloc. is stateless
public:
    typedef size_t      size_type;
    typedef ptrdiff_t   difference_type;
    typedef [const] T * [const_]pointer;
    typedef [const] T & [const_]reference;
    typedef T           value_type;

    template <class U> struct rebind {typedef allocator<U> other;};

    allocator() throw();
    allocator(const allocator &a) throw();
    template <class U> allocator(const allocator<U> &a) throw();
    ~allocator() throw();

    [const_]pointer address([const_]reference x) const;

    pointer   allocate(size_type n, allocator<void>::const_ptr hint = nullptr);
    void     deallocate(pointer p, size_type n);
    size_type max_size() const throw();

    void construct(pointer p, const value_type &val);
    void destroy (pointer p);

}; // allocator
```

Zeichenketten (2): *basic_string* (Teil 1)

```
template <class charT, class traits = char_traits<charT>,
          class alloc = allocator <charT> >
class basic_string {

private: // implement. specific, sizeof(string) typically == 24 (for 32 bit impl.)
...      // especially for gcc/clang and MS cl

public:

    static const size_type npos = -1; // not found position



---

explicit basic_string(const alloc &a = alloc());
basic_string(const charT *cs, [size_type n,] const alloc &a = ...);
basic_string(size_t n, charT c, const alloc &a = ...);
basic_string(const basic_string &bs
                [, size_t pos, size_t n = npos]);


---

~basic_string(); // not virtual, so do not ... or be careful;-)


---

void clear();
size_t length() const; // same result as size(), see below
const charT *c_str() const; // data() returns C str. without '\0'
[const] charT &operator[](size_t pos) [const]; // no runtime check
[const] charT &at(size_t pos) [const]; // may raise out_of_range
...
}
```

Zeichenketten (3): *basic_string* (Teil 2)

```
...
basic_string &operator= (      charT      c );
basic_string &operator= (const charT      *cs);
basic_string &operator= (const basic_string &bs);
basic_string &assign     (const charT      c );
basic_string &assign     (const charT      *cs);
basic_string &assign     (const basic_string &bs);
void          swap       (      basic_string &bs); // very efficient

basic_string &operator+=(const charT      *cs);
basic_string &operator+=(const basic_string &bs);
basic_string &append     (const charT      *cs);
basic_string &append     (const basic_string &bs);

int compare(const charT      *cs); // comp. operators ==, <, ...
int compare(const basic_string &bs); // as functions, see below

basic_string &insert (size_t pos,           const charT      *cs);
basic_string &insert (size_t pos,           const basic_string &bs);
basic_string &replace(size_t pos, size_t n, const charT      *cs);
basic_string &replace(size_t pos, size_t n, const basic_string &bs);
basic_string &erase  (size_t pos = 0, size_t n = npos);

basic_string substr(size_t pos = 0, size_t n = npos) const;
size_t [r]find(const basic_string &bs) const;
...
```

Zeichenketten (4): *basic_string* (Teil 3)

```
...
// basic_string as STL cont., vector semantics with iterators
typedef ... [const_][reverse_]iterator;


---

template<class InputIterator>
basic_string(InputIterator first, InputIterator last,
             const alloc& a = ...);


---

[const_][reverse_]iterator [c][r]begin() [const];
[const_][reverse_]iterator [c][r]end () [const]; } zwölf Methoden


---

bool empty() const; // empty() ⇔ length() == size() == 0
size_t size() const; // same result as length(), see above
size_t capacity() const;


---

void push_back(charT c);


---

void insert (iterator it, size_t n, charT c);
basic_string &replace(iterator it1, iterator it2,
                      const charT *cs, size_t n);
iterator erase (iterator it);
iterator erase (iterator first, iterator last);
}; // basic_string
```

Zeichenketten (5): *basic_string* (Teil 4)

Diverse generische Operatoren (als glob. Funkt.) für *basic_string*:

```
template <class charT, class traits<...>, class alloc<...> > ...
... bool
operator==(const basic_string<charT, traits, alloc> &l, // or charT*
             const basic_string<charT, traits, alloc> &r); // or charT*


---


... basic_string<charT, traits, alloc>
operator+(const basic_string<charT, traits, alloc> &l,
            const charT* r); // or basic_string
basic_string<charT, traits, alloc>
operator+(const charT* l,
            const basic_string<charT, traits, alloc> &r); // or charT*


---


... basic_ostream<charT, traits>& // insertion operator
operator<<(      basic_ostream<charT, traits>      &os,
                  const basic_string <charT, traits, alloc> &bs);
... basic_istream<charT, traits>& // extraction operator
operator>>(      basic_istream<charT, traits>      &is,
                  basic_string <charT, traits, alloc> &bs);
```

Vergleich: *string* und *vector<char>*

string-Semantik

```
#include <string>
using namespace std;

// construction

string s; // ≈ vect.<char>
string s2("..."); 
string s3(3, 'x');
string s4(s2);
```

// string operations

```
cout << s.length();
cout << s;
s = "";
s.swap(s2);
s.append('x');
```

```
s.insert(0, "xxx");
s.replace(0, 3, "yyy");

s.erase(0, 3);
```

// no sort method

vector<char>-Semantik

```
#include <string>
#include <algorithm> // see STL: copy, sort, ...
#include <iterator> // see STL: ostream_it.
#include <vector> // see STL: vector
using namespace std;

// construction

vector<char> v, v2, ... // ≈ string
... // fill v with chars
string s(v.begin(), v.end());
```

// vector operations on s, of course also possible on v

```
cout << s.size();
copy(s.begin(), s.end(), ostr._it.<char>(cout));

v.swap(v2); // vector has its own swap
s.push_back('x');
```

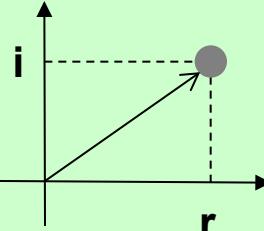
```
s.insert(s.begin(), v.begin(), v.end());
s.replace(s.begin(), s.begin() + 3,
          v.begin(), v.end());
s.erase(s.begin(), s.begin() + 3);
```

```
sort(s.begin(), s.end()); // alg. sort on string
```

Komplexe Zahlen

Komplexe Zahlen über numerischem Datentyp T , mit Ein-/Ausgabe

```
template<class T> //  
class complex { //  
    private: //  
        T r, i; //  
    public: //  
        complex(const T &r = T(), const T &i = T());  
        // operators: =, +, -, *, /, +=, -=, *=, /=  
        // methods for: sqrt, pow, exp, log, sin, cos, tan  
}; // complex
```



Bit-Mengen, eigentlich Bit-Vektoren für Mengen über *unsigned int*, effiziente Verwaltung von Bit-Mengen eigentlich Bit-Vektoren, log. Operationen, auch Ein-/Ausgabe (weniger flexibel als *vector<bool>*)

```
template<size_t n> bitset; // set of n bits: 0 .. n-1
```

Vektor-Operationen

Effiziente Operationen auf Vektoren über numerischem Datentyp T :

```
template<class T> valarray; // size via constructor
```

Gemeinsamkeiten

E/A ist in beiden Fällen nicht in die "eigentliche" Sprache integriert (Compiler weiß nichts von E/A-Operationen):

- in C durch Funktionsbibliothek `<stdio.h>`
- in C++ durch Klassenbibliothek `<iostream>`

E/A ist in C und C++ über *Folgen von Einzelzeichen* (= *Ströme*, engl. *streams*) definiert

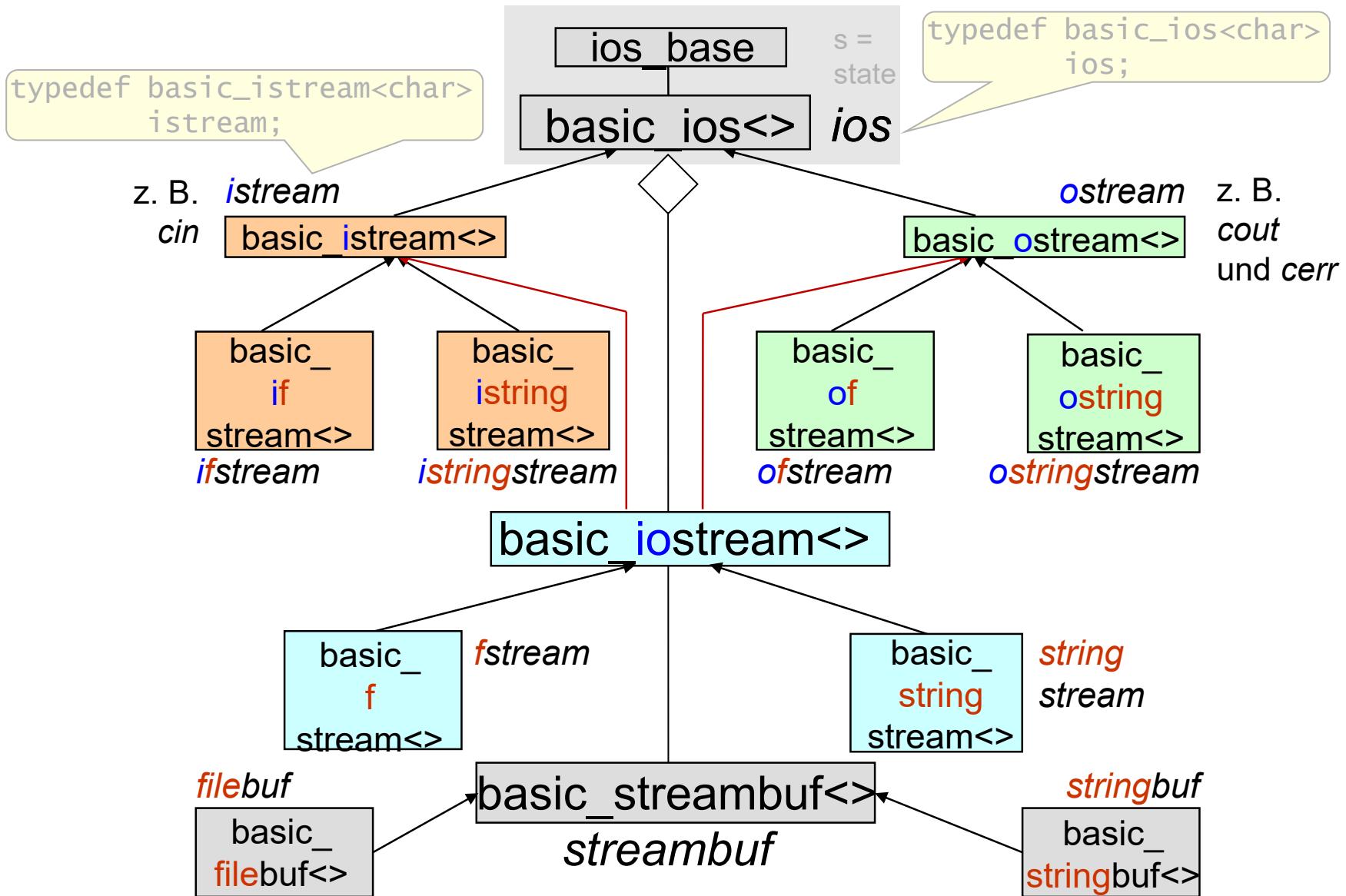
Grundidee in C++: Umwandlung beliebiger Werte in Zeichenfolgen und umgekehrt

Unterschiede

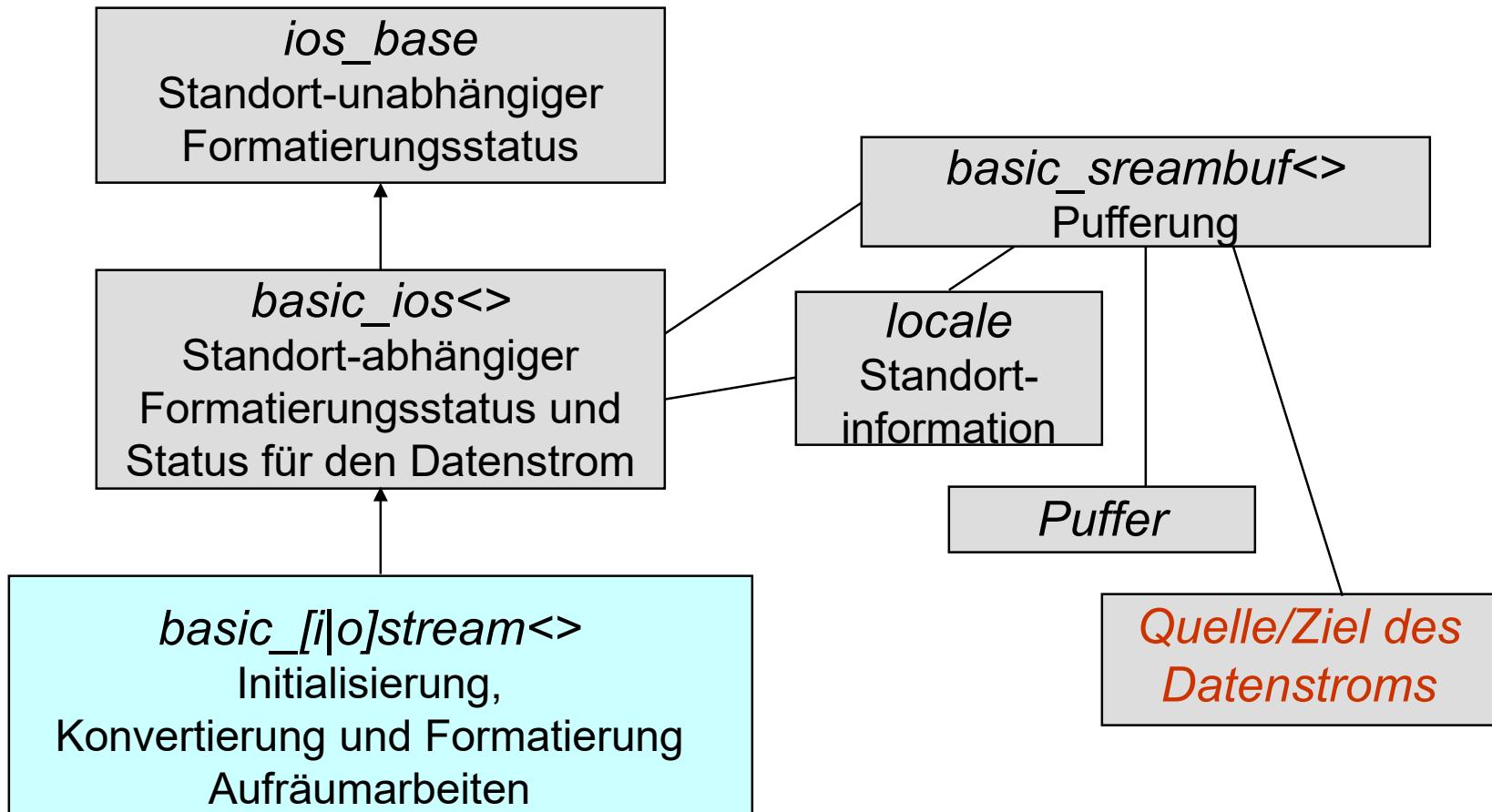
I/O-Stream-Klassen in C++ decken alle Mögl. von `stdio.h` in C ab und:

- Eingabe-Operator (`>>`) u. Ausgabe-Operator (`<<`) sind typsicher
- Für benutzerdefinierte Typen kann E/A selbst definiert werden, damit gleiche E/A-Funktionalität für eingebaute und benutzerdefinierte Typen
- I/O-Streams sind effizienter, denn `scanf` und `printf` basieren auf Formatzeichenketten, die zur Laufzeit interpretiert werden müssen
- I/O-Streams sind (nur) auf Zeichenebene Thread-sicher, können in Programmen verwendet werden, die mehrere Threads haben

Ein/Ausgabe: Hierarchie der Klassen(schablonen)



Ein/Ausgabe: Zusammenhänge



Header-Datei	Inhalt: Klassen, ...
<code><iostream></code>	Nur div. Vorwärts-Deklarationen (dadurch sehr klein), typische Verwendung in Header-Dateien ca. 1 KLOC
<code><streambuf></code>	<code>streambuf</code> -Hilfsklassen
<code><iostream></code>	Alle Deklarationen für <code>istream</code> -Klassen
<code><ostream></code>	Alle Deklarationen für <code>ostream</code> -Klassen
<code><iostream></code>	Alle Deklarationen für die Verwendung der <code>istream</code> -, <code>ostream</code> - und <code>iostream</code> -Klassen, der vordefinierten Objekte <code>cin</code> , <code>cout</code> , <code>cerr</code> und <code>clog</code> sowie wichtiger Manipulat. (s. u.) wie <code>flush</code> , <code>endl</code> , <code>ends</code> und <code>ws</code> ca. 15 KLOC
<code><fstream></code>	Deklarationen der <code>fstream</code> -Klassen für Datei-E/A
<code><sstream></code>	Deklarationen der Klassen für E/A über <code>string</code> -Objekte als Quelle/Ziel: <code>istringstream</code> , <code>ostringstream</code>
<code><istrstream></code>	Deklarationen der Klassen für E/A über <code>char*</code> -Werte als Quelle/Ziel: <code>istrstream</code> , <code>ostrstream</code>
<code><iomanip></code>	Spezielle Manipulatoren (s. u.)

Bearbeitung auf zwei Ebenen (logisch)

- Auf untere Ebene: Folgen von Zeichen
- Auf obere Ebene: Folgen von Werten eines best. Datentyps

Wichtige Basisklassen

- *basic_streambuf<>* und abgeleitete Klassen (*basic_stringbuf<>* und *basic_filebuf<>*) implementieren Puffer für Datenströme
- *ios_base* und *basic_io<>* verwalten Formatierungs-, Fehler- und Zustandsinformationen für die Streams
- Von *basic_io<>* abgeleitete Klassen (*basic_istream<>*, *basic_oiostream<>*, *basic_iostream<>*) realisieren Formatierung (Umwandlung v. Zeichenketten aus dem Puffer in Werte u. umgek.)

E/A für vor- und benutzerdefinierte Typen

- Für vordefinierte Typen sind E/A-Operatoren definiert, E/A von Werten unterschiedlichen Typs in einer Anweisung möglich, z. B.:

```
cout << anInt << ", " << aDouble << '\n' << aString;
```
- Für benutzerdefinierte Typen können E/A-Operatoren überladen werden, z. B.:

```
cin >> aPerson >> anAddress >> aTelNo;
```

Ein-Ausgabe: Klasse *ios_base*

```
class ios_base { // base class for all streams,  
public:    e.g., int // constructor is protected, so ...  
    typedef ... fmtflags; // bitmask, with bits:  
    static const fmtflags boolalpha; // bool values as text  
    static const fmtflags dec; // base 10, def. f. c(in|out)  
    static const fmtflags fixed; // floating point  
    static const fmtflags hex; // base 16  
    static const fmtflags left; // left aligned  
    static const fmtflags right; // right aligned  
...  
    typedef ... iostate; // bitmask, with bits:  
    static const iostate badbit; // stream is "in nirvana"  
    static const iostate eofbit; // end of file reached  
    static const iostate failbit; // last operation failed  
    static const iostate goodbit; // none of the above bits set  
...  
    typedef ... openmode; // bitmask, with bits:  
    static const openmode in; // input  
    static const openmode out; // output  
    static const openmode app; // append  
    static const openmode binary; // no text, binary data  
...  
    fmtflags flags() const; // get current flags  
    fmtflags setf(fmtflags f) // set new flags and get old ones  
...  
}; // ios_base
```

Ein-/Ausgabe: Klassenschablone `basic_ios`

```
template<class charT, class traits = char_traits<charT> >
class basic_ios: public ios_base {

public:
    // inherited from ios_base: typedef ... iostate; // bitmask
    basic_ostream<charT, traits> *tie() const;           } default:
    basic_ostream<charT, traits> *tie(                  }   cin.tie(&cout)
        basic_ostream<charT, traits> *ts) const;           ...
    ...
    charT fill() const; // get fill character
    charT fill(charT ch); // set new and get old fill character
    ...
    basic_ios &copyfmt(const basic_ios &src);
    ...
    iostate rdstate() const;
    void clear(iostate state = goodbit);
    void setstate(iostate state);
    ...
    bool good() const;
    bool eof() const;
    bool fail() const;
    bool bad() const;
    ...
    operator void*() const; // C++11: operator bool() const;
    bool operator!() const; // returns same result as fail()
};

// basic_ios<charT, traits>
```

Beispiele zur Verwendung:

```
iostate s = cin.rdstate();
if (s & ios::goodbit) ...
if (s.good()) ...
if (s) ... // same as good
if (cin.fail())...
if (!cin) ... // void* cast
```

Ein-/Ausgabe: Klassenschablone *basic_ostream*<>

Formatierte Ausgabe für Zeichentypen über Funktionsschablonen:

```
template<class charT, class traits>
basic_ostream<charT, traits> operator<<(
    basic_ostream<charT, traits> os, charT c);
```

Ein/Ausgabe: Ausgabe f. benutzerdef. Datentypen

Überladen des Ausgabeoperators (*insertion op.*) **operator<<** in Form einer Funktion für benutzerdefinierten Datentyp (i. d. R. eine Klasse)

Beispiel (benutzerdefinierte Klasse *complex*, nicht *std::complex<>*)

```
class complex { // better use std::complex<double>
    friend ostream &operator<<(ostream &os, const complex &c);
private:
    double re, im;
public:
    explicit complex(double re = 0.0, double im = 0.0);
    ...
}; // complex

ostream &operator<<(ostream &os, const complex &c) {
    return os << '(' << c.re() << ", " << c.im() << ')';
} // operator<<
```

Syntax

Verwendung

(re, im)

```
complex c(1.0, 2.0);
cout << "c = " << c << endl; // => c = (1.0, 2.0)
```

Ein-/Ausgabe: Klassenschablone *basic_istream*<>

```
template<class charT, class traits = char_traits<charT> >
class basic_istream: virtual public basic_ios<charT, traits> {
public:
    ...
    basic_istream &operator>>(bool &b);           // extraction ops.
    basic_istream &operator>>([unsigned] short &s);
    basic_istream &operator>>([unsigned] int &i);
    basic_istream &operator>>([unsigned] long &l);
    basic_istream &operator>>(float &f);
    basic_istream &operator>>([long] double &d);
    basic_istream &operator>>(void * &p);
    ...
    basic_istream &get(charT &c); // overload with add. delimiter
    basic_istream &getline(charT *s, int n); // overl. w. add. del.
    basic_istream &read(charT *s, int n);
    ...
    basic_istream &putback(charT);
};
// basic_istream<charT, traits>
```

zus. glob. Funktion
getline(istream&, string&)

Formatierte Eingabe für Zeichentypen über Funktionsschablonen:

```
template<class charT, class traits>
basic_ostream<charT, traits> operator>>(  
    basic_ostream<charT, traits> os, charT c); } analog für char,  
sowohl signed als  
auch unsigned
```

Überladen des Eingabeoperators (*extraction op.*) **operator>>** in Form einer Funktion für benutzerdefinierten Datentyp (i. d. R. eine Klasse)

Beispiel (für benutzerdefinierte Klasse *complex*, s.o.)

```
istream &operator>>(istream &is, complex &c) {
    double re = 0.0, im = 0.0; char ch;
    is >> ch;
    if (ch == '(') { // 1. or 2.
        is >> re >> ch;
        if (ch == ',') // 1.
            is >> im >> ch;
        if (ch != ')') // syntax error
            is.setstate(ios::failbit); // op.>> failed
    } else { // 3.
        is.putback(ch); // ch != '(', maybe a digit
        is >> re;
    } // else
    if (is.good()) { // no syntax error
        c.re = re; c.im = im;
    } // if
    return is;
} // operator>>
```

Syntax

1. (*re* , *im*)
2. (*re*)
3. *re*

Verwendung

```
complex c;
cin >> c;
if (cin.fail()) {
    cerr << ...;
    cin.clear();
} // if
// now cin.good()
```

Exkurs: Manipulatoren – Idee (1)

Viele Manipulationsmöglichkeiten auf *stream*-Objekten sind denkbar und z. T. über Zugriffsmethoden realisiert

Beispiel: Leeren des Ausgabepuffers

```
cout << "ERROR";  
cout.flush(); // write buffer to stdout  
  
cerr << "ERR..
```

Problem: Durch notwendige Aufteilung auf mehrere Anweisungen geht logischer Zusammenhang verloren

Wunsch: Operation *op* (z. B. *flush*) soll innerhalb einer Ein- bzw. Ausgabeanweisung möglich sein:

```
cout << ... << op << ...; // e.g. cout << flush;  
cin  >> ... >> op >> ...;
```

Lösung: Manipulatoren (anstelle von Methoden)

Exkurs: Manipulatoren – Realisierung (2)

1. Primitivlösung (z. B. für die Operation *flush*)

```
class flush_type { }; // type name only
ostream &operator<<(ostream &os, flush_type f) {
    os.flush(); return os;
} // operator<<
flush_type flush;
```

Verwendung:
cout << "ERROR" << flush;
// writes object flush

Nachteil: Für jede Operation ist Typ, Operator und Objekt notwendig

2. Elegante, weil flexible Lösung (für alle Operationen zu gebrauchen)

```
typedef ostream &(*omanip)(ostream &);
ostream &operator<<(ostream &os, omanip om) {
    return (*om)(os);
} // operator<<
```

Z. B. ist *flush* dann wie folgt deklariert:

```
ostream &flush(ostream &os) {
    os.flush(); return os;
} // flush
```

Verwendung:
cout << "ERROR" << flush;
// calls function flush

Vorteil: Eine Funktion mit passender Schnittstelle pro Operation reicht aus

Exkurs: Manipulatoren – mit Parametern (3)

Für Manipulatoren könnten Parameter nützlich/notwendig sein, z. B.:

```
cout << setprecision(4) << x; // same as: cout.precision(4); ...
```

Funktion, zum Setzen d. Parameters im *ostream*-Objekt muß aufgerufen und ein Objekt muß erzeugt werden, welches mit *operator<<* "ausgegeben" wird

```
class omanip_int { // in header <iomanip>
public:
    ostream &(*f)(ostream &, int); // function to call with ...
    int i; // ... parameter value
    omanip_int(ostream &(*f)(ostream &, int) f, int i) : f(f), i(i) {}
}; // omanip_int

ostream &operator<<(ostream &os, omanip_int &omi) {
    return (*omi.f)(os, omi.i);
} // operator<<

ostream &precision(ostream &os, int i) { os.precision(i); }

omanip_int setprecision(int i) {return omanip_int(precision, i);}


```

Aufruf von *setprecision* konstruiert anonymes *omanip_int*-Objekt, das mit *operator<<* ausgegeben wird, Funktion *precision* ruft und Genauigkeit auf 4 setzt

Weitere Verbesserung: Klassenschablonen für die versch. Typen von Parametern f. Manip., Resultat: *xMANIP<type>*-Klassen für x = O, I oder S

Exkurs: Manipulatoren – Beispiele aus <iomanip> (4)

1. Parameterloser Manipulatoren

Für Ein/Ausgabeströme:

```
ios &oct(ios &); // set to octal,      base 8
ios &dec(ios &); // set to decimal,    base 10
ios &hex(ios &); // set to hexadecimal, base 16
```

Für Ausgabeströme:

```
ostream &endl (ostream &); // write '\n' and flush
ostream &ends (ostream &); // write '\0' and flush
ostream &flush(ostream &); // flush buffer
```

Für Eingabeströme:

```
istream &ws(istream &); // consume white space
```

2. Parametrierbare Manipulatoren

Für Formatierungszwecke (s. u.):

```
SMANIP<int> setbase(int b);
SMANIP<int> setfill(int f);
SMANIP<int> setprecision(int p);
SMANIP<int> setw(int w);
```

Für Statusflag-Manipulation:

```
SMANIP<long> resetioflags(long b);
SMANIP<long> setioflags(long b);
```

Definition von Ausgabefeldern

```
int ios::width(int w);      // manipulator: setw
int ios::width() const;
char ios::fill (char f);    // manipulator: setfill
char ios::fill () const;
```

Steuerung der Fließkommagenaugigkeit

```
int ios::precision(int p); // manipulator: setprecision
int ios::precision() const;
```

Auswirkungen auf Formatstatus-Flags in *ios_base*

```
class ios_base {
    ...
    typedef ... fmtflags;
    static const fmtflags skipws;      // skip whitespace
    static const fmtflags dec;        // base, also allowed oct & hex
    static const fmtflags scientific; // notation: d.ddddddd Edd
    static const fmtflags fixed;      // notation: ddd.d
    ...
    fmtflags flags() const;          // get flags
    fmtflags setf(fmtflags f) // set flags
    ...
}; // ios_base
```

STL

--

Standard Template Library

Heinz Dobler

Version 19, 2025



Permission to copy without any fee all or part of these slides is granted provided that copies are not made or distributed for commercial advantage, the copyright notice, the title of the presentation as well as its date appear, and notice is given that "copying is by permission of Dr. Heinz Dobler". To copy otherwise, or to republish, requires a fee and/or the specific permission.

- Autoren, Ziele und Philosophie
- Bestandteile: Übersicht u. Konzept d. Zusammenwirkens
- Beispiel in mehreren Versionen
- **Iteratoren** und Adapter für Iteratoren
- Sequentielle **Behälter**: *vector*, *list* und *deque*
und seit C++11 auch: *array* und *forward_list*
- Adapter f. seq. Behälter: *stack* und *[priority_]queue*
- Assoziative **Behälter**: *[multi]set* und *[multi]map*
und seit C++11 auch: *unordered_[multi](set|map)*
- Funktionsobjekte
- **Algorithmen** und Beispiele
- Basiskomponenten

- "Abklappern" eines Behälters
- *vector*, das bessere dynamische Feld
- Effizienzvergleich sequentieller Behälter
- Neue sequentielle Behälter in C++11
- *tree*, die Grundlage assoziativer Behälter
- Assoziative Behälter auf Basis von *Hash*-Tabellen
- Effizienzvergleich der *set*-Behälter
- Vergleichsoperatoren
- *Safe STL*

Ab 1987 erste Arbeiten an der STL in **Ada83**, Autoren waren

- Alexander Stepanov (damals bei Silicon Graphics Inc.)
- Meng Lee (damals bei den Hewlett-Packard-Laboratories)

Ab 1994 Integration in die **C++**-Bibliothek, 1998 dann im Standard

Ziele

1. Orthogonales Design ermöglicht es

- STL-Behälter in STL-Algorithmen zu verwenden (klar!), aber auch
- STL-Behälter in eigenen Algorithmen zu verwenden und
- STL-Algorithmen auf eigene Behälter anzuwenden

2. Effizienz: alle STL-Algorithmen weichen in der Laufzeit nur wenige Prozent von einer optimalen, handcodierten *speziellen* Version ab

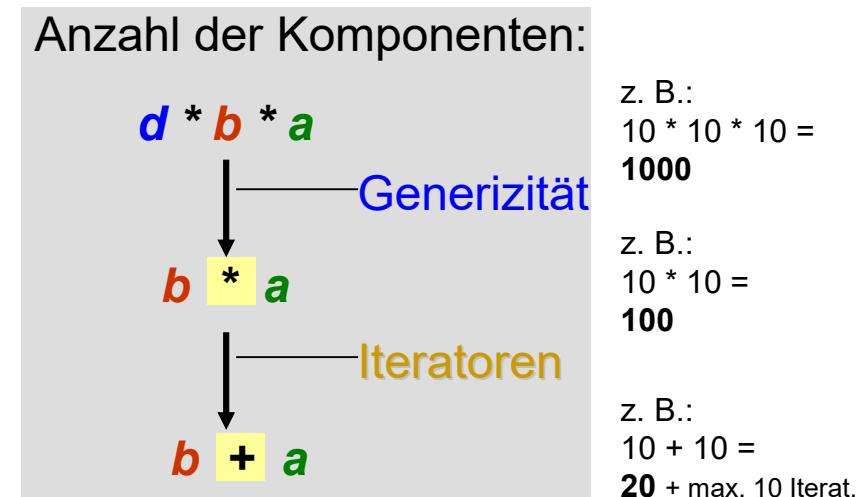
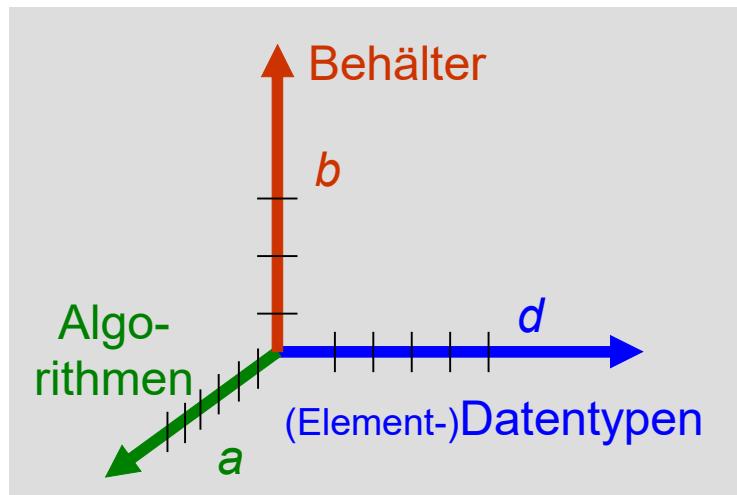
3. Fundierte theoretische Absicherung: trotz Einfachheit und Übersichtlichkeit im Design stabiles Theoriefundament und verlässliche Angaben zur asymptotischen Laufzeitkomplexität

Philosophie

- Generizität (Schablonen in C++) anstelle von objektorientierter Programmierung (mit Vererbung, Polymorphismus und dynamischer Bindung)
- Also: GP anstelle von OOP

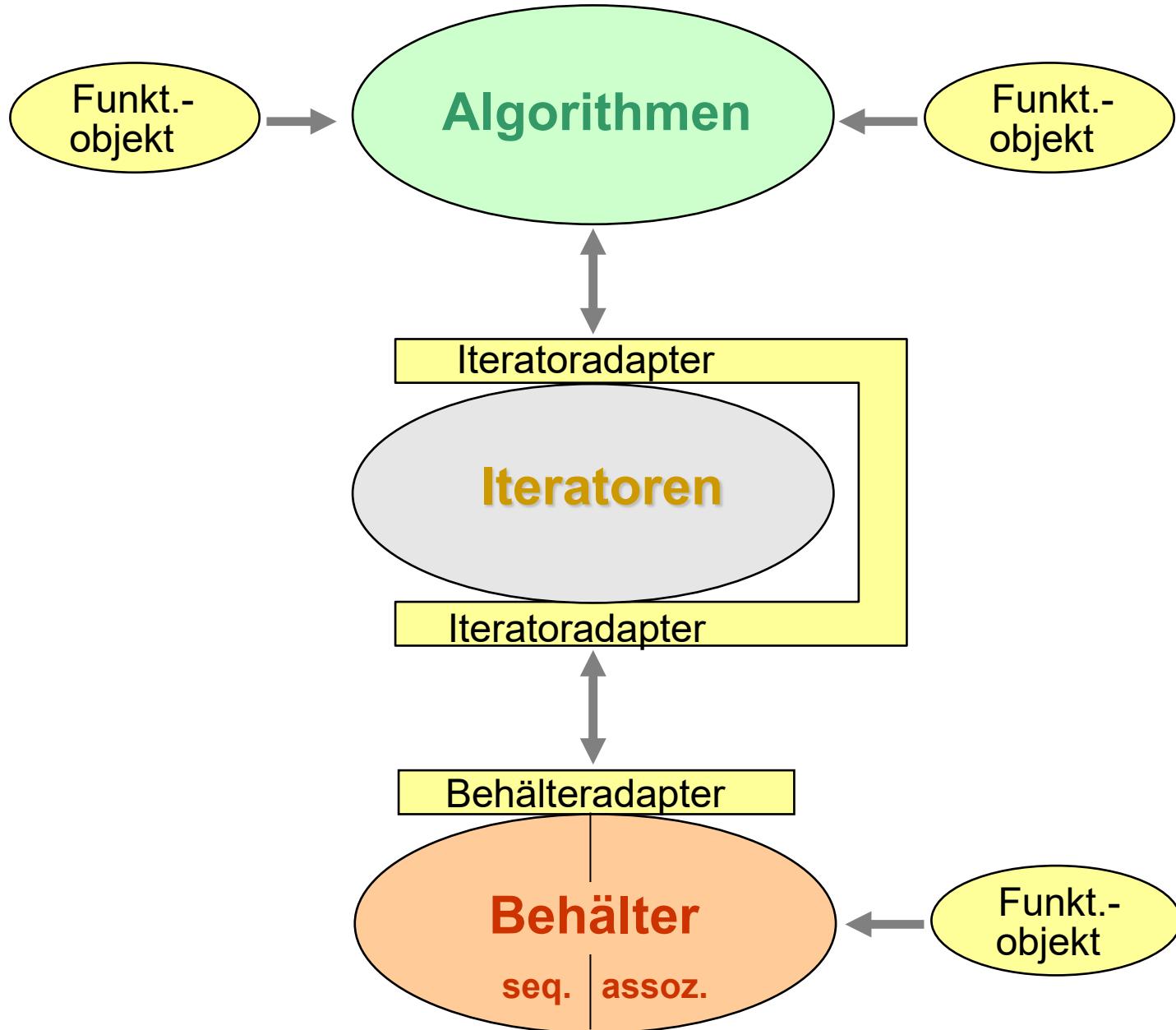
STL besteht im Wesentlichen aus fünf Komponenten:

1. **Behälter** (*container*): Objekte, die Werte (anderer) **Datentypen** enthalten, also "Verwalter von Speicherbereichen" sind
2. **Iteratoren**: Generalisierungen des Konzepts der Zeiger (*pointer*) in C und C++, Mittel zum Zugriff auf Elemente von Behältern
3. **Algorithmen**: Definitionen von algorithmischen Berechnungen



4. **Funktionsobjekte**: "Kapsel" für eine Funktion zur Verwendung in anderen Komponenten
5. **Adapter**: Möglichkeit der Anpassung einer Komponente (für Behälter, Iteratoren und Funktionen) an andere Schnittstellen

Konzept des Zusammenwirkens



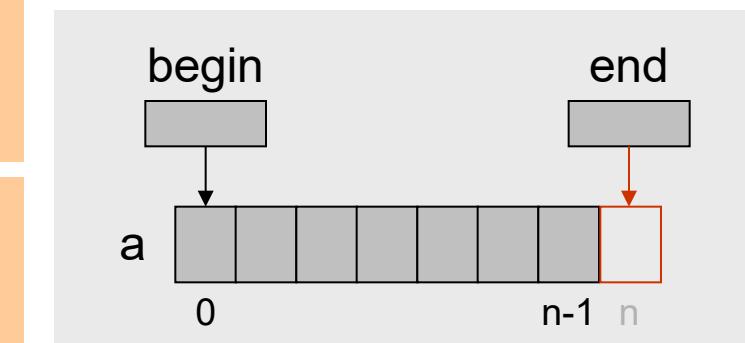
Beispiel: Version 1 mit Funktion *find*

Aufgabe: Sequentielles Suchen eines *int*-Werts in einem *int*-Feld

```
typedef int* Iterator;
```

```
Iterator find(Iterator begin, Iterator end, int val) {
    Iterator cur = begin;
    while ((cur != end) && (*cur != val))
        cur++;
    return cur;
} // find
```

```
int main(int argc, char *argv[]) {
    const int n = 100;
    int a[n], x;
    Iterator begin = &a[0]; // or a
    Iterator end = &a[n]; // or a + n, i.e. one past the end
    // fill a[0 .. n - 1] and get a value x to search for
    Iterator it = find(begin, end, x);
    if (it != end)
        cout << "position = " << it - begin << endl; ...
} // main
```



Erkenntnis: Suchfunktion *find* muß nichts über Behälter (hier nur normales Feld) wissen, Iterator (hier nur ein Zeiger) stellt Verbindung zur Datenstruktur her ... funktioniert aber nur für *int*-...

Beispiel: Version 2 mit generischer Funktion *find*

```
template<class Iterator, class T> // or typename T
Iterator find(Iterator begin, Iterator end, const T &val) {
    Iterator cur = begin;
    while ((cur != end) && (*cur != val))
        cur++;
    return cur;
} // find
```

```
typedef int* Iterator;
```

```
int main(int argc, char *argv[]) {
    const int n = 100;
    int a[n], x;
    Iterator begin = &a[0];
    Iterator end = &a[n]; // one past the end
    // fill a[0 .. n - 1] and get value for x to search for
    Iterator it = find(begin, end, x); // implicit instant., cf.
    if (it != end) // explicitly: find<Iter., int>(...)
        cout << "position = " << it - begin << endl; ...
} // main
```

Erkenntnis: Generische Funktion *find* ist anwendbar für

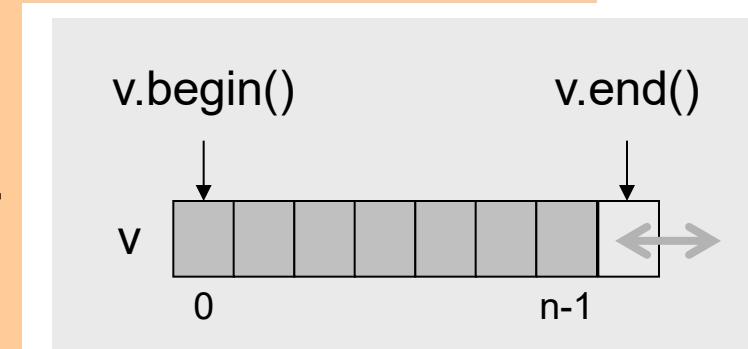
- **Iteratortypen** mit Initialisierung und den Operatoren `!=`, `*` und `++`
 - **Werttypen** mit dem Operator `!=`
- ... ist also ein "Algorithmus" (in STL-Terminologie)

Beispiel: Version 3 *find* auf STL-Behälter, z. B. *vector*

```
#include <vector> // provides generic class vector<T>
```

```
template<class Iterator, class T>
Iterator find(Iterator begin, Iterator end, const T &val) {
    Iterator cur = begin;
    while ( (cur != end) && (*cur != val) )
        cur++;
    return cur;
} // find
```

```
int main(int argc, char *argv[]) {
    int          x;
    vector<int> v;
    // fill v with values ...
    // ... and define val. x to search for
    vector<int>::iterator it =
        find(v.begin(), v.end(), x);
    if ( it != v.end() )
        cout << "position = " << it - v.begin() << endl; ...
} // main
```



Erkenntnis: Unveränderte gen. Funktion (Algorithmus) *find* kann auch mit STL-Behälter *vector* mit seinen Iteratoren arbeiten

Beispiel: Version 4 zus. mit STL-Algorithmus *find*

```
#include <vector>
#include <algorithm> // provides algorithm find<...>(...)
```

```
int main(int argc, char *argv[]) {
    const int n = 100;
    int x;
    vector<int> v;
    // fill v with values and get value x to search for
    cout << "x = "; cin >> x;
    vector<int>::iterator it = find(v.begin(), v.end(), x);
    if (it != v.end())
        cout << "position = " << it - v.begin() << endl;
} // main
```

Erkenntnisse

1. STL-Algorithmus *find* hat völlig identische Semantik (und Implementierung;-) wie bisherige Implementierung von *find*
2. Algorithmus *find* kann mit allen Iteratoren arbeiten, die folgende Operatoren bieten: Initialisierung, ***!=*** zum Vergleich,
* zur Dereferenzierung und ***++*** zum Weiterschalten
3. Minimaler Aufwand für Realisierung umfangreicher Aufgaben, weil Vieles (Behälter u. Algorithmen) vorhanden ist

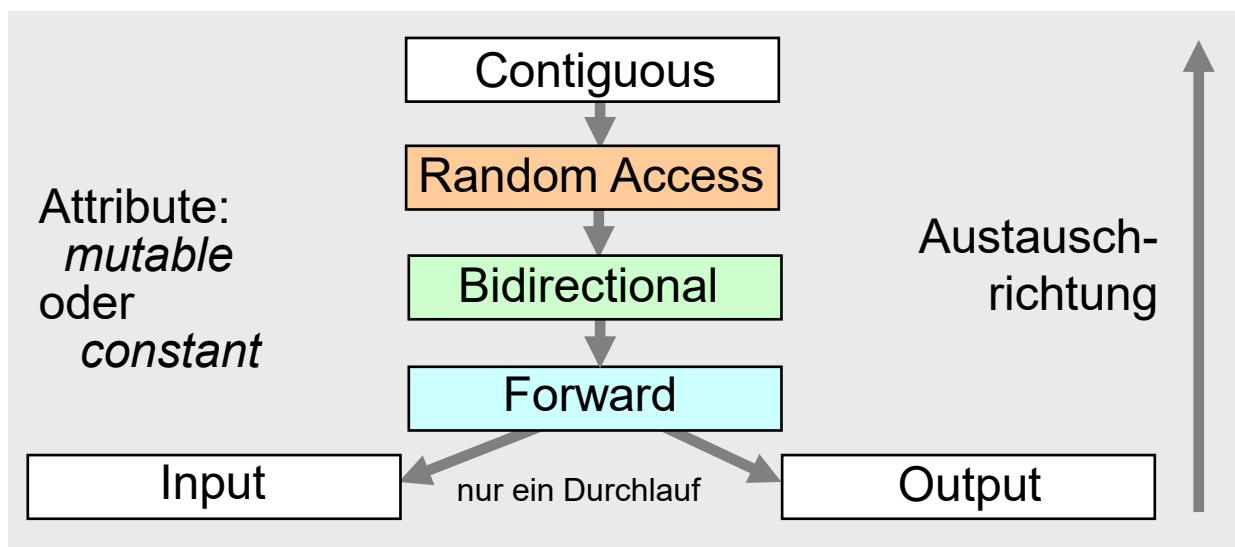
Konzept

- Verallgemeinertes Zeigerkonzept (Iteratoren u. Zeiger austauschbar)
- Erlauben Verarbeitung unterschiedl. Datenstrukturen in analoger Weise: abstrakte Schnittstelle zwischen Behälter und Algorithmus
- Formalisieren Schnittstelle u. Semantik, damit auch Laufzeitkomplex.

Iteratoren: beteiligte Datentypen

- *operator** liefert Wert des "Werttyps" *iterator::value_type*
- Wenn *operator<* definiert ist, dann gibt es auch den "Distanztyp" *iterator::difference_type* (ganzzahliger Typ, z. B. *ptrdiff_t*)

Arten von Iteratoren **Motto:** Was sich **verhält** wie ein Iterator, **ist** ein Iterator!



Iterator-Strategien:

- **Behälter** stellt **stärkst-möglichen** Iterator zur Verfügung
- **Algorithmus** verwendet **schwächst-möglichen** Iterator

1. Allgemeine Iteratoren

Vergleich (`operator==` und `operator!=`) sowie Dereferenzierung (`operator*`) für alle Arten v. Iteratoren, zusätzlich:

- **Vorwärtsiteratoren (*forward*)** erlauben Abarbeitung nur in einer Richtung (der "Vorwärtsrichtung") mit `operator++` (z. B. `it++`)
- **Bidirektionale Iteratoren (*bidirectional*)** erlauben zusätzlich Abarbeitung in beiden Richtungen mit `operator--` (z. B. `it--`)
- **Direktzugriffsiteratoren (*random access*)** erlauben zusätzlich Bewegungen um n Elemente nach vorne oder hinten:
 - Sprünge mit `operator+=` und `operator-=` für einen Iterator mit einer ganzen Zahl (z. B. `it += n`; oder `it -= n`;)
 - Addition `+` und Subtraktion `-` einer ganzen Zahl zu/von einem Iterat. (z. B. `it = it + n` oder `it = it - n`)
 - Subtraktion zweier Iteratoren, z. B. `n = it1 - it2`
 - Kl.-Vergleich zweier Iteratoren, z. B. `if (it1 < it2)`

Achtung: Ergebnis nur dann definiert, wenn beide Iteratoren über denselben Behälter ...

Exkurs: "Abklappern" eines Behälters

Gegeben sei irgendein STL-Behälter (*container*) mit Elementen vom Typ *T*:

```
container<T> c; // c is a container for elements of type T
```

1. "Abklappern" mittels **for**-Schleife

```
for (container<T>::iterator it = c.begin(); it != c.end(); it++)
    cout << *it;
```

2. "Abklappern" mittels **for(each)**-Schleife

```
#define foreach(ElemType, elem, cont) \
for (auto _it_ = (cont).begin(); \
     _it_ != (cont).end(); \
     _it_++) { \
    ElemType elem = *_it_;
```

```
foreach (T, e, c) // { \
    cout << e;
} // foreach
```

Seit C++11 viel besser mit:
range-based for loop

```
for ([const] T [&]e: c) {
    cout << e;
} // for
```

3. "Abklappern" mittels **for_each**-Algorithmus

```
for_each(c.begin(), c.end(), print<T>);
for_each(c.begin(), c.end(), [](int e){cout << e;});
```

```
template <typename T>
void print(const T &e) {
    cout << e;
} // print
```

2. Spezielle Iteratoren

Eingabe- bzw. Ausgabeiteratoren haben grundsätzlich gleiche Semantik wie Vorwärtsiteratoren, aber nicht alle Eigenschaften (hinsichtlich Zugriff) sind garantiert

- **Eingabeiteratoren (*input iterator*):**

$\dots = *iit$ ist definiert, aber

Zuweisung an $*iit$ wird nicht garantiert,

z. B. kann $*iit = \dots$ zu einem Fehler führen

- **Ausgabeiteratoren (*output iterator*):**

$*oit = \dots$ ist definiert, aber

Verwendung von $*oit$ zum Auslesen eines Werts wird nicht garantiert,

z. B. kann $\dots = *oit$ zu einem Fehler führen

i. d. R.
verwendet,
um über
Dateien zu
iterieren

Konzept für Dateien (*files*) in C++

- Dateien werden durch *stream*-Objekte (z. B. *cin* u. *cout*) repräsentiert
- *stream*-Objekte können wie Behälter Werte enthalten od. aufnehmen
- Somit sind *stream*-Objekte spezielle Behälter, also **Datei \approx Behälter**

Problem

- *stream*-Objekte haben zwar richtige Funktionalität (Behälterfunkt.) aber falsche Schnittstelle, nämlich *operator<<* und/oder *operator>>*
- *stream*-Objekte können nicht wie Behälter mit Iteratoren (z. B. mit *operator**) verwendet werden, passen also nicht zur STL

Lösung

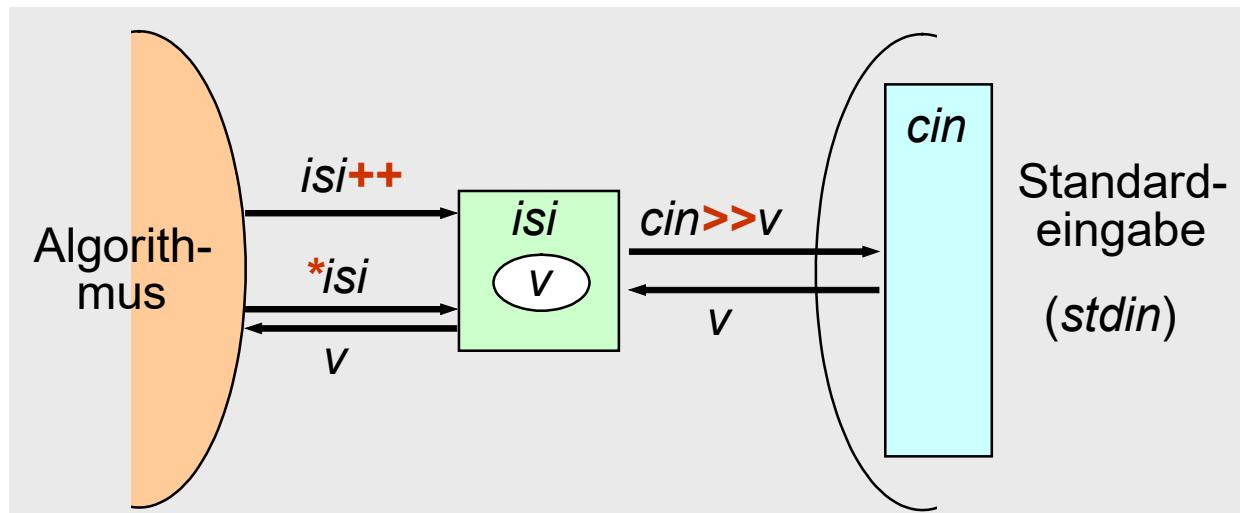
- *stream*-Iteratoren passen Schnittstelle von *stream*-Obj. an STL an
- Behandlung von *stream*-Objekten mittels Iteratoren wird möglich
- Alle STL-Algorithmen können somit auch auf Dateien arbeiten

Eingabeiterator *istream_iterator*<T>

istream_iterator<T> liest mittels *operator>>* Werte des Typs T vom angebundenen stream-Objekt ein und stellt diese über *operator** zur Verfügung

Beispiel

```
istream_iterator<int> isi(cin);
istream_iterator<int> eoisi; // end of istr.it.
while (isi != eoisi) {
    int i = *isi; // same as: cin >> i;
    cout << i << endl;
    isi++;
} // while
```

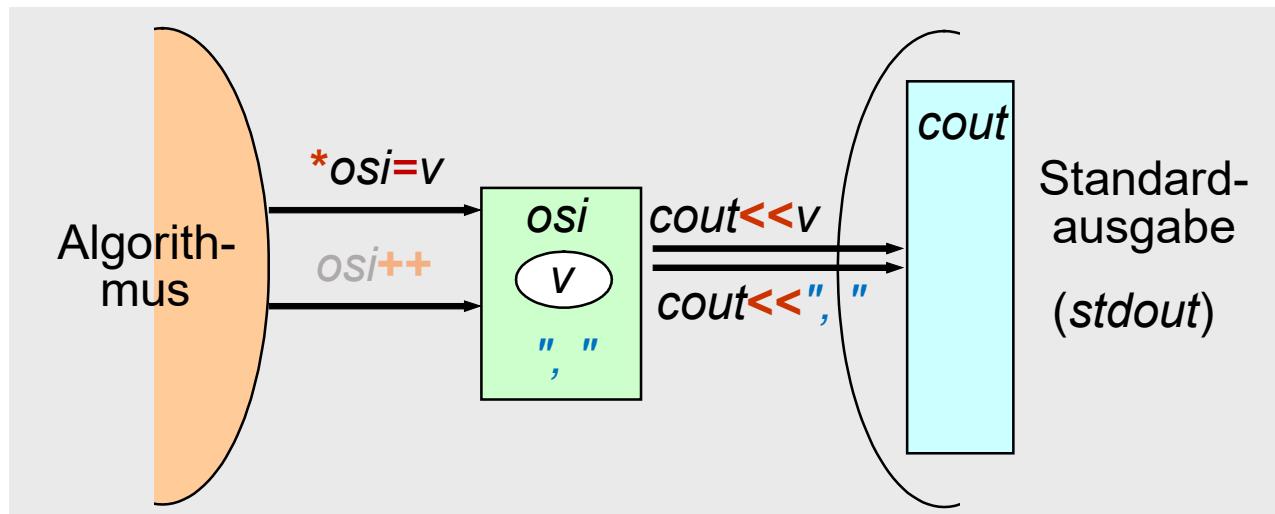


Ausgabeiterator *ostream_iterator*<T>

ostream_iterator<T> schreibt mittels *operator<<* Werte des Typs T auf das angebundene *stream*-Objekt aus,
im Konstruktor kann eine Trennkette angegeben werden

Beispiel

```
ostream_iterator<int> osi(cout, " ", ");  
cin >> i;  
while (!cin.eof()) {  
    *osi = i; // same as: cout << i << ", "  
    osi++; // superfluous: op.= generates output  
    cin >> i;  
} // while
```



Kopieren eines Eingabe-Datenstroms auf einen Ausgabe-Datenstrom
(hier von der Standard-Eingabe *cin* auf die Standardausgabe *cout*)

1. Explizit mit einer **Schleife**

```
istream_iterator<int> isi(cin);
istream_iterator<int> eo_isi; // end of istream_iterator
ostream_iterator<int> osi(cout, ", ");
while (isi != eo_isi) {
    *osi = *isi; // same as: cin >> i; cout << i << ", ";
    osi++;
    isi++;
} // while
```

2. Implizit mit **Algorithmus copy**

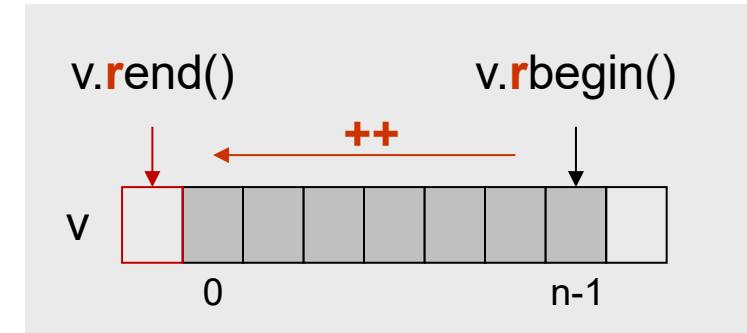
```
copy(istream_iterator<int>(cin),
      istream_iterator<int>(),
      ostream_iterator<int>(cout, ", "));
```

Adapter für Iteratoren: Umkehriterator

Für bidirektionale und Direktzugriffsiteratoren können "umgekehrte" Iteratoren erzeugt werden, die bei **++** oder **--** in d. andere Richtung laufen

Klassenschablone

```
template<class Iterator>
class reverse_iterator {  
    ...  
}; // reverse_iterator
```



liefert für einen bidirekionalen oder Direktzugriffsiterator-Typ *Iterator* einen Umkehriterator-Typ der gl. Art (s. Abb. als Beisp. f. *vector<T>*)

Beispiel (für einen *vector* und die sequentielle Suche)

```
vector<int> v{0, 1, 9, 3, 9, 5};  
vector<int>::iterator it;  
it = find(v.begin(), v.end(), 9); // find first occ.  
cout << it - v.begin(); // => 2 = index from the beginning  
reverse_iterator<vector<int>::iterator> rit;  
rit = find(v.rbegin(), v.rend(), 9); // find last occ.  
cout << rit - v.rbegin(); // => 1 = index from the end
```

Adapter für Iteratoren: Einfügeiterator

- Einfügeiteratoren erlauben Einfügen in Behälter auf einfache Art (indirekt über Methoden *push_back*, *push_front* bzw. *insert*)
- Drei Klassenschablonen liefern für passenden Behältertyp entsprechenden Typ für Einfügeiteratoren:

```
template<class Container>
class [back_|front_]insert_iterator : ... {
    public:                                     // for insert_iterators:
        [back_|front_]insert_iterator(Container c   [, Iterator it]      );
        ...
}; // ...insert_iterator
```

Beispiele

```
vector<int> v;
copy(istream_iterator<int>(cin), istream_iterator<int>(),
     back_insert_iterator<vector<int>>(v));
copy(v.begin(), v.end(), ostream_iterator<int>(cout));
```

Drei Funktionen zum einfachen Erzeugen entspr. Einfügeiteratoren direkt für einen Behälter: *back_inserter*, *front_inserter* und *inserter*, z. B.:

```
copy(istream_iterator<int>(cin), istream_iterator<int>(),
     back_inserter(v));
```

Manche Algorithmen benötigen Zugriff auf Eigenschaften der Iteratoren

```
template<class Iterator>
struct iterator_traits { ... };
```

1. Beteiligte Typen (*type tags*)

- Elementtyp mittels `iterator_traits<It>::value_type`
- Distanztyp mittels `iterator_traits<It>::distance_type`

2. Marken für Kategorien (*category tags*)

- Kategorie mittels `iterator_traits<It>::iterator_category`

```
template<class Iterator>
inline void f(Iterator i1, Iterator i2) {
    typename iterator_traits<Iterator>::iterator_category t;
    f(i1, i2, t); // uses overloading resolution
} // f
```

```
template<class BDI> // BidirectionalIterator
void f(BDI i1, BDI i2, bidirectional_iterator_tag t) {
    ... } // f
```

```
template<class RAI> // RandomAccessIterator
void f(RAI i1, RAI i2, random_access_iterator_tag t) {
    ... } // f
                                // no f for ForwardIterator
```

Funktionsschablonen für Iteratoren in <iterator>

- Nur mit Direktzugriffsiteratoren sind Sprünge über mehrere Elemente einfach und effizient, also in $O(1)$, möglich
- Funktionsschablone **advance** zur allgemeinen Formulierung v. Algorithmen

```
template<class InputIterator, class Dist>
void advance(InputIterator &it, Dist n) {...; advance(it, n, t); };
```

Je nach Iterator-Kategorie wird eine von drei Implementierungen verwendet:

```
template<class II, class Dist> // = forward_...
void advance(II &it, Dist n,           input_iterator_tag t) {
    while (n--) ++it; // works for n >= 0 only
} // advance

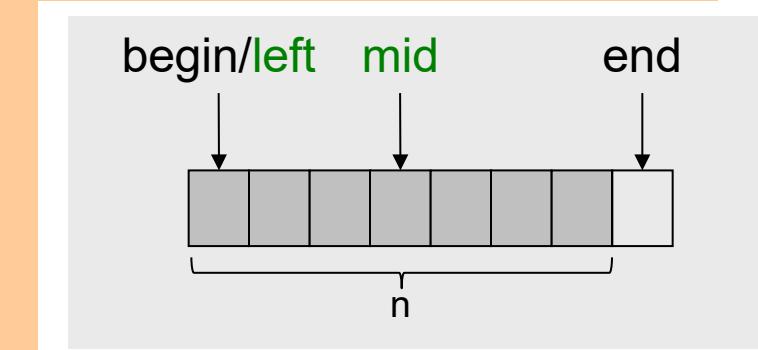
template<class BDI, class Dist>
void advance(BDI &it, Dist n, bidirectional_iterator_tag t) {
    if (n >= 0) while (n--) ++it;
    else         while (n++) --it;
} // advance

template<class RAI, class Dist>
void advance(RAI &it, Dist n, random_access_iterator_tag t) {
    it += n;
} // advance
```

Analoge Funktionsschablone **distance** ermittelt Abstand zweier Elemente

Beispiel: Binäre Suche nur mit Vorwärtsiterator

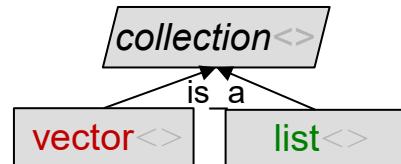
```
template <typename FI, typename T>
bool bin_search(FI begin, FI end, // cf. std::binary_search
                const T &val) {    // where T provides op.<
    FI left = begin;
    auto n = distance(left, end);
    while (n > 0) {
        FI mid = left;
        advance(mid, n / 2);
        if (*mid < val) {
            left = ++mid;
            n = n - (n / 2) - 1;
        } else // *mid >= val
            n =      n / 2;
    } // while
    return (left != end) && !(val < *left); // ... (val >= *left)
} // binary_search
```



Verwendung:

```
list<int> l; // doubly-linked list with bidirect. iterator
// fill the list so that elements are sorted
... = bin_search(l.begin(), l.end(), x); // but find (seq. search) is
... = std::find (l.begin(), l.end(), x); // 2 (c1) to 3 (gcc) x faster
```

Konzept bei oo Behältern (z. B. MiniLib, Java und .NET)



Klassenhierarchie mit gemeins. Basisklasse f. Polymorphismus und
program to an interface not to an implementation

```
collection *c; // pointer for dynamic objects
if (...) // decision at runtime
    c = new vector;
else
    c = new list;
c->add(...); // method call with dynamic binding
delete c;
```

Konzept bei STL-Behältern



Generische Klassen mit Methoden gleichen Namens und identischen Schnittstellen, vgl. *duck typing*

```
#if (...) // decision at compile time
    vector<...> c; // static object
#else
    list<...> c; // static object
#endif
c.push_back(...); // method call with static binding
```

Sequentielle Behälter: Übersicht

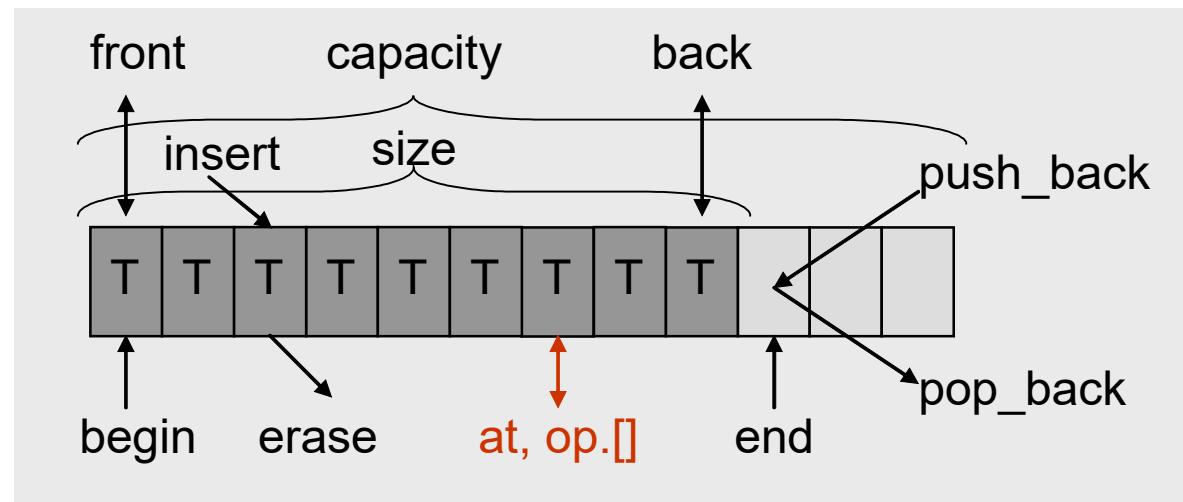
- Elemente werden sequentiell im Speicher angeordnet
- Zugriff über Index oder durch Reihenfolge

Behälter	Charakteristika
vector <T>	<ul style="list-style-type: none">• hinten offenes, dynamisches Feld• Direktzugriffsiterator und direkter Zugriff in $O(1)$• Einfügen und Entfernen am Ende in $O(1)$• ... am Anfang und in der Mitte in $O(n)$
list <T>	<ul style="list-style-type: none">• doppelt-verkettete Liste (mit Ankerelement)• bidirektonaler Iterator• Zugriff nur in $O(n)$• Einfügen und Entfernen überall in $O(1)$
deque <T>	<ul style="list-style-type: none">• beidseitig offenes, dynamisches Feld• Direktzugriffsiterator und direkter Zugriff in $O(1)$• Einfügen u. Entfernen am Anfang u. am Ende in $O(1)$• ... in der Mitte allerdings nur in $O(n)$

Sequentieller Behälter *vector*<>

- **Standard-Behälter** für sequentielle Speicherung
- Bietet Direktzugriffsiteratoren
- Einfügen und Entfernen ...
 - ... am Ende in konstanter Zeit
 - ... am Anfang und in der Mitte in linearer Zeit
- Automatische Speicherplatzverwaltung
(Wachsen und Schrumpfen, "Hinweise" verbessern Effizienz)

Konzeptuelle Sicht



Spezialfall: Anstelle von Vektoren über dem Datentyp *bool*

`class vector<bool> { ... }; // does not work correctly,
besser deque<bool> oder bitset<n> verwenden! // e.g., iterator`

Klassenschablone `vector<>`

```
template<class T, class Allocator = allocator<T> >
class vector { // sizeof(vecor<T>): 12 w. g++, 12 w. cl (32 bit)
public:
    explicit vector(const Allocator & = Allocator());
    vector(size_type, const T & = T(), const A. & = A.());
    vector(const vector<T, Allocator> &);
    vector(iterator first, iterator last, const A. & = A.());
    ~vector();
    vector<T , Allocator> &operator=(const vector<T, A.> &);
    [const_] [reverse_] iterator [c][r]begin() [const];
    [const_] [reverse_] iterator [c][r]end() [const]; } 8 Methoden!
    size_type (size|capacity)() const;
    bool empty() const;
    [const_] reference operator[](size_type) [const];
    [const_] reference at(size_type) [const]; // may throw out_of_range
    [const_] reference front() [const];
    [const_] reference back() [const]; } 4 Methoden!
    void push_back(const T &x);
    void pop_back();
    void swap(vector<T, A.> &x);
    iterator insert(iterator pos, const T &x);
    void insert(iterator pos, size_type n, const T &x);
    void insert(iterator pos, iterator first, iterator last);
    void erase(iterator pos);
    void erase(iterator first, iterator last);
    void clear(); // and: reserve, shrink_to_fit
}; // vector<T, Allocator>
```

Zusätzlich: alle relationalen
Operatoren in Form
globaler Funktionen

Exkurs: *vector*, das bessere dynamische Feld

- Dynamisches Feld in **C** (und wenn es sein muss, so auch in C++)

```
int *a = (int*)malloc(n * sizeof(int)); // hopefully 3 x int !!!  
if (a == NULL) { // assert(a) would be dangerous, because ...  
    fprintf(stderr, "ERROR ..."); exit(...); }  
/* use array a via a[i] or *(a + i) */  
free(a);
```

- Dynamisches Feld in **C++**

```
int *a = new int[n]; // may throw bad_alloc, only 2 x int ;-)  
// use array a via a[i] or *(a + i)  
delete[] a;
```

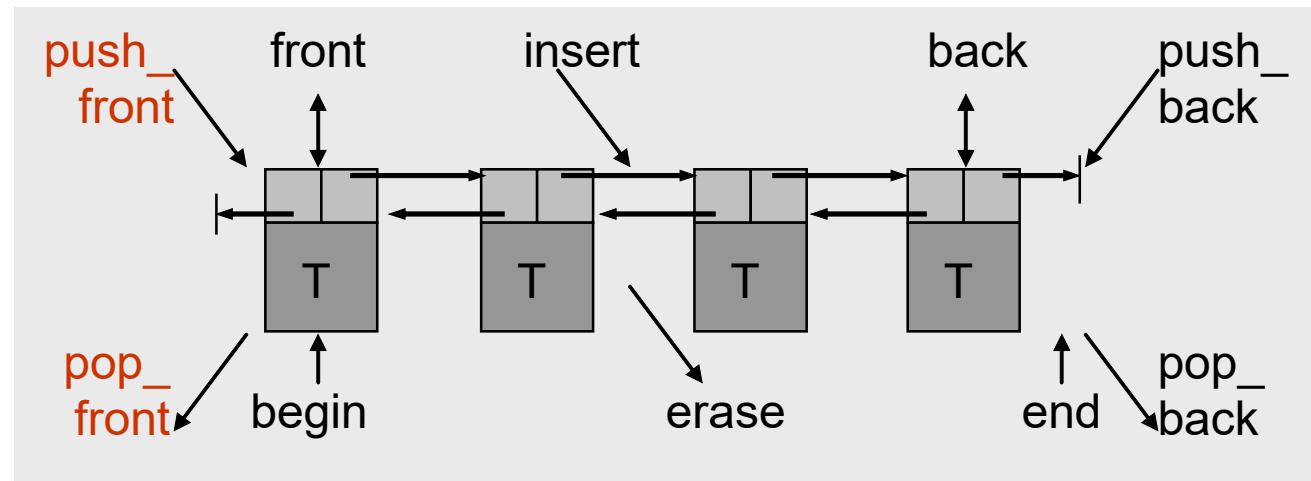
- Behälter *vector*<> der **STL**

```
vector<int> v(n); // vector with n elements, int only once  
// use vector v via v[i], v.at(i) or it. (vector<int>::iterator)  
// additional functionality for vectors only:  
v.resize(m); // possibly inserts "zero" elements at the end  
v.reserve(m); // increases capacity to ...
```

Sequentieller Behälter *list*<>

- Bietet nur bidirektionale Iteratoren
- Einfügen und Entfernen an allen Positionen in konstanter Zeit
- Automatische Speicherplatzverwaltung für die Knoten
- Unterschied zu *vector* und *deque*: kein direkter Zugriff in konstanter Zeit (jedoch in linearer Zeit mit Funktion *advance*, s.o.)
- Spezielle Operationen für Listen (*splice*, *remove*, *unique*, *merge*, *reverse* und *sort*) ohne Umweg über Iteratoren

Konzeptuelle Sicht



Tipp zur Verwendung: Bei häufigen Einfüge- und Löschoperationen in der Mitte der Sequenz

Klassenschablone *list*<>

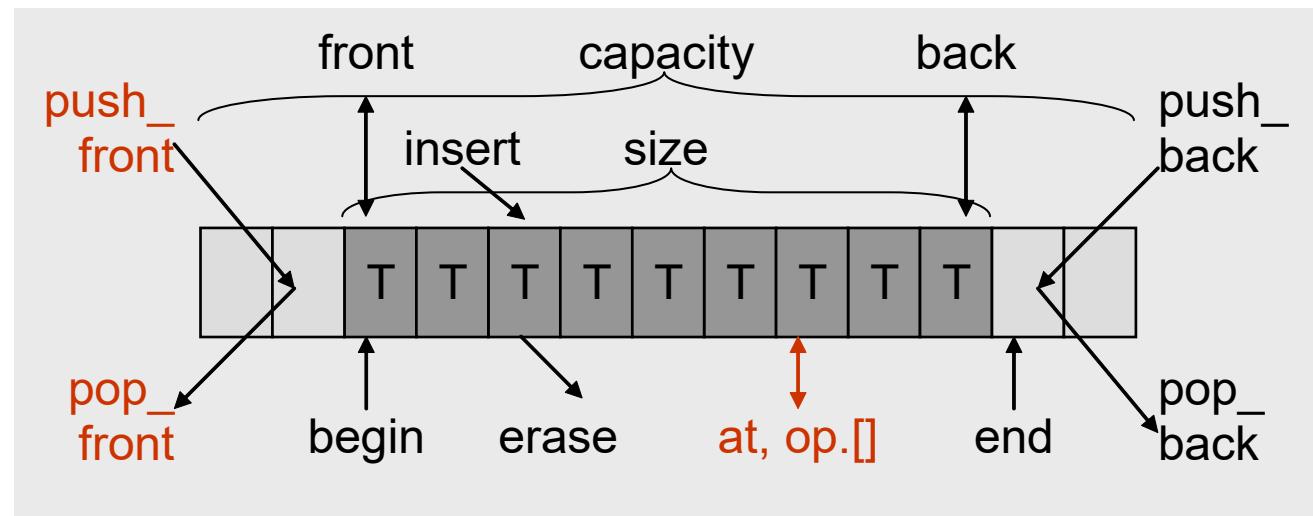
```
template<class T, class Allocator = allocator<T> >
class list { // sizeof(list<T>): 12 w. g++, 8 w. cl (32 bit)
public:
    explicit list(const Allocator & = Allocator());
    explicit list(size_type, const T & = T(), const A. & = A.());
    list(const list<T, Allocator> &);
    list(iterator first, iterator last, const A. & = A.());
    ~list();
    list<T, Allocator> &operator=(const list<T, A.> &);
    [const_] [reverse_] iterator [c] [r] (begin|end)() [const];
    size_type size() const;           kein capacity mehr
    bool empty() const;
    [const_] reference (front|back)() [const];
    void push_front(const T &);      } neu im Vergleich zu vector<>,
    void pop_front();                dafür kein operator[] und
    void push_back(const T &);       keine at-Methode mehr
    void pop_back();
    void swap(list<T, A.> &);
    iterator insert(iterator pos, const T &x);
    void erase(iterator pos);
    void erase(iterator first, iterator last);
    void splice(iterator pos, list<T, A.> &);
    void remove(const T &);
    void merge(list<T, A.> &); // for sorted lists
    void reverse();
    void sort([Compare cmp]);
}; // list<T, Allocator>
```

} nur für *list*<>
Zusätzlich: alle relationalen
Operatoren in Form
globaler Funktionen

Sequentieller Behälter *deque*<>

- Bietet Direktzugriffsiteratoren (wie *vector*)
- Einfügen und Entfernen ...
 - ... am Anfang und am Ende in konstanter Zeit
 - ... in der Mitte nur in linearer Zeit
- Automatische Speicherplatzverwaltung
(Wachsen und Schrumpfen)

Konzeptuelle Sicht



Tipp zur Verwendung: Bei häufigen Einfüge- und Löschoperationen am Anfang oder am Ende der Sequenz

Klassenschablone `deque<>`

```
template<class T, class Allocator = allocator<T> >
class deque {    // sizeof(deque<T>): 40 w. g++, 20 w. c1 (32 bit)
public:
    explicit deque(const Allocator & = Allocator());
    explicit deque(size_type, const T & = T(), const A. & = A.());
    deque(const deque<T, Allocator> &);
    deque(iterator first, iterator last, const A. & = A.());
    ~deque();
    deque<T, Allocator> &operator=(const deque<T, A.> &);
    [const_][reverse_]iterator [c][r](begin|end)() [const];
    size_type size() const;           kein capacity mehr
    bool empty() const;
    [const_]reference operator[](size_type) [const];
    [const_]reference at(size_type) [const]; // may throw out_of_r. } wie vector<>
    [const_]reference front() [const];
    [const_]reference back() [const];
    void push_front(const T& x);      } wie list<>
    void pop_front();
    void push_back(const T& x);
    void pop_back();
    void swap(deque<T, A.>& x);
    iterator insert(iterator pos,           const T &);
    void     insert(iterator pos, size_type, const T &);
    void     insert(iterator pos, iterator first, iterator last);
    void erase(iterator pos);
    void erase(iterator first, iterator last);
}; // deque<T, Allocator>
```

} wie list<>

Zusätzlich: alle rel. Operat.
in Form glob. Funktionen

Wichtigste Operationen auf sequentiellen Behältern

Tipps zur Verwendung:

- **vector** ist oft eine gute, weil auch effiziente Ausgangsbasis
- **deque** eignet sich besser, wenn häufige Veränderungen auch vorne notwendig sind
- **list** eignet sich am besten, wenn kein direkter Zugriff notwendig ist und häufige Veränderungen "in der Mitte" notwendig sind

Wichtigste Operationen	Behälter		
	vector	deque	list
<i>front back</i>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<i>(push pop)_back</i>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<i>(push pop)_front</i>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<i>at, operator[]</i>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Exkurs: Effizienzvergleich der sequentiellen Behälter

Messungen mit jeweils $n = 10$ Mio. Elementen in einem Behälter c

Compiler (32 bit) mit Bibliotheken: g++ V. 11.2 (unter msys2) bzw. Microsoft cl V. 19.29

Betriebssystem: Windows 10 mit 64 Bit, Prozessor: Intel Xeon E3-1271 mit 3,6 GHz

	vector<T>		deque<T>		list<T>	
	g++	cl	g++	cl	g++	cl
T = int , Zufallszahlen						
c.push_back(...)	0,19	0,33	0,14	0,48	3,81	1,97
for mit Iterator	0,19	0,30	0,27	1,08	0,16	0,37
range-based for	0,11	0,17	0,11	0,23	0,06	0,17
c[i]	0,03	0,03	1,87	0,95	-	-
c.at(i)	0,11	0,05	1,94	1,03	-	-
sort alg. or meth. for list	3,78	3,43	6,54	17,16	11,92	25,04
T = string , Zufallszahlen umgewandelt in string						
c.push_back(...)	13,99	6,49	7,44	3,21	10,80	4,21
for mit Iterator	0,34	1,64	0,41	2,36	0,28	1,64
range-based for	0,44	2,78	0,44	2,89	0,42	2,75
sort alg. or meth. for list	36,05	38,66	37,89	52,60	12,46	26,15

Exkurs: Neue sequentielle Behälter in C++11

- **array**: oo Schnittstelle für **statisches Feld** ohne "Overhead" (allokiert Speicher am Laufzeitkeller), also keine weiteren Datenkomponenten, aber mit Methoden analog zu *vector* (insbesondere für Iteratoren und *at*)

```
template<class T, size_t n> // fixed number of elements
class array {    // sizeof(array<T>): 4 w. g++, 4 w. c1 (32 bit)
public:
    [c][r]begin(), [c][r]end(),
    [max_]size(), empty(), // missing: capacity()
    operator[](...), at(...), front(), back(), ...
}; // array
```

- **forward_list**: einfach-verkettete Liste mit minimalem "Overhead", nicht einmal Elementzähler (für *size*) wird mitgeführt

```
template<class T, class Allocator = allocator<T> >
class forward_list { // sizeof(f.l.<T>): 4 w. gcc, 4 w. c1 (32 bit)
public:
    begin(), end(),      // missing: rbegin(), rend()
    max_size(), empty(), // missing: size(), cap.()
    push_front(), pop_front(), ... // missing: ..._back()
    sort(), reverse(), ...
}; // forward_list
```

Exkurs: Effizienzvergl. Feld- u. listenartiger Behälter

Betriebssystem: Windows 10 mit 64 Bit, Prozessor: Intel Xeon E3-1270 m. 3,6 GHz

Feldartige Behälter:

(Messungen mit jeweils $n = 1$ Mio. Elementen und 1000 Wiederholungen)

	int c[n]	array<int, n> c	vect.<int> c(n)			
	g++	cl	g++	cl	g++	cl
for mit Index c[i] =	1,95	1,85	(?)3,14	2,26	2,39	7,80
for mit Index c.at(i) =	-	-	3,65	2,42	6,45	20,93
for mit Iterator *it =	1,54	1,54	1,56	12,31	8,09	15,27
range-based for e =	(?)2,15	2,07	2,12	2,07	6,76	2,07

Listenartige Behälter:

(Messungen mit jeweils $n = 100$ Mio. Elementen für g++ und
nur $n = 10$ Mio. Elementen für cl, sonst *heap overflow*)

	forward_list<int> l;		list<int> l;	
	g++	cl	g++	cl
for mit ... l.push_front()	8,15	1,22	8,48	1,46
for mit Iterator ... = *it;	1,73	0,36	1,68	0,43
range-based for ... = e;	1,55	0,10	1,53	0,14

Anmerkung:
push_front weil
kein *push_back*
in *forward_list*

Adapter für sequentielle Behälter

- Behälteradapter stellen für Spezialanwend. "adaptierte Behälter" mit
 - eingeschränkter Schnittstelle und ... deshalb bieten Behälteradapter
 - spezieller Semantik z. B. keine Iteratoren!zur Verfügung
- Implementierung beruht auf vorhandenen sequentiellen Behältern
- Realisierung nicht durch (eingeschränkte) Vererbung sondern durch Schablonen mit Behältern als Argumente

Adapter	Anforderungen an Behälter	Mögl. Behälter
stack	<i>back,</i> <i>push_back,</i> <i>pop_back</i>	vector list deque
queue	<i>front, back,</i> <i>push_back,</i> <i>pop_front</i>	list deque
priority_queue	<i>random_access_iterator,</i> <i>front,</i> <i>push_back,</i> <i>pop_back</i>	vector deque

Behälteradapter *stack*<>

```
template<class T, Container = deque<T>>
class stack {
public:
    typedef Container::value_type value_type;
    typedef Container::size_type size_type;
protected:
    Container c;
public:
    explicit stack(const Container &c = Cont.())
        : c(c) {
    } // stack
    bool empty() const {
        return c.empty();
    } // empty
    size_type size() const {
        return c.size();
    } // size
    [const] value_type &top() [const] {
        return c.back();
    } // top
    void push(const value_type &x) {
        c.push_back(x);
    } // push
    void pop() {
        c.pop_back();
    } // pop
}; // stack<T, Container>
```

Beispiel zur Verwendung:

```
stack<int [, vector<int>]> is;
is.push(17); ...
while (!is.empty()) {
    cout << is.top();
    is.pop();
} // while
```

Zusätzlich: alle rel. Operatoren
in Form glob. Funktionen

Behälteradapter `queue`<>

```
template<class T, class Container = deque<T>>
class queue {
public:
    typedef Container::value_type value_type;
    typedef Container::size_type size_type;
protected:
    Container c;
public:
    explicit queue(const Cont. &c = c.())
        : c(c) {
    } // queue
    bool empty() const { ... }
    size_type size() const { ... }
    [const] value_type &front() [const] {
        return c.front();
    } // front
    [const] value_type &back() [const] {
        return c.back();
    } // back
    void push(const value_type &x) { // enqueue
        c.push_back(x);
    } // push
    void pop() { // dequeue
        c.pop_front();
    } // pop
}; // queue<T, Container>
```

Beispiel zur Verwendung:

```
queue<char [, list<char>]> cq;
cq.push('a'); ...
while (!cq.empty()) {
    cout << cq.front();
    cq.pop();
} // while
```

Zusätzlich: alle rel. Operatoren
in Form glob. Funktionen

Behälteradapter *priority_queue*<>

```
template<class T, class Container = vector<T>, class Compare = less<T>>
class priority_queue {
public:
    typedef Container::value_type value_type;
    typedef Container::size_type size_type;
protected:
    Container c;
    Compare comp;
public:
    priority_queue(const Compare &comp = C.(),
                   const Cont. &c = C.())
        : comp(comp), c(c) {
    } // priority_queue
    bool empty() const { ... }
    size_type size() const { ... }
    [const] value_type &top() [const] {
        return c.front();
    } // top
    void push(const value_type &x) {
        c.push_back(x);
        push_heap(c.begin(), c.end(), comp);
    } // push
    void pop() {
        pop_heap(c.begin(), c.end(), comp);
        c.pop_back();
    } // pop
}; // priority_queue<T, Container, Compare>
```

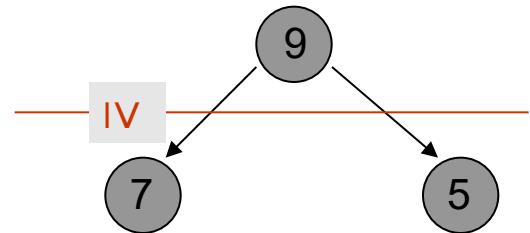
Zusätzlich: alle rel. Operatoren
in Form glob. Funktionen

Heap =

- dyn. Speicher oder
- spez. Datenstruktur

Heap-Datenstruktur

... entweder "dynamisch" :



... oder (besser) als Feld:

n	9	7	5	..
0	1	2	3	4

Assoziative Behälter

- Zugriff auf Elemente über spezielles Schlüsselattribut *key*
- Ordnungsrelation *Comp* muß totale Ordnung garantieren (z. B. op.<)

Behälter	Charakteristik
set <Key, Comp>	<ul style="list-style-type: none">• Menge von Elementen = Schlüssel• Eindeutige Schlüssel• Zugriff in $O(\log n)$
multiset <Key, Comp>	Wie <i>set</i> , erlaubt aber mehrere gleiche Elemente (= Schlüssel)
map <Key, T, Comp>	<ul style="list-style-type: none">• Abbildung von Schlüssel auf Datum• Eindeutige Schlüssel• Zugriff über Schlüssel in $O(\log n)$ auf Werte des Typs <i>T</i>
multimap <Key, T, Comp>	Wie <i>map</i> , erlaubt aber mehrere Einträge mit gleichem Schlüssel

Exkurs: *tree*, die Grundlage assoziativer Behälter (1)

- Notwendigkeit für alle assoziativen Behälter:
schnelle Suche aufgrund eines Schlüsselwerts
- Grundsätzlich zwei Realisierungsmöglichkeiten:
 - Hash-Tabellen oder
 - Balancierte binäre Suchbäume: AVL- → 2-3- → 2-3-4-Bäume → ...

Adelson-Velski
& Landis, 1962

... Rot-Schwarz-Bäume

- Modernste und effizienteste Form balancierter binärer Suchbäume
- Ausgleich mittels Färbung von Kanten, Farbe gesp. in den Knoten

Definition

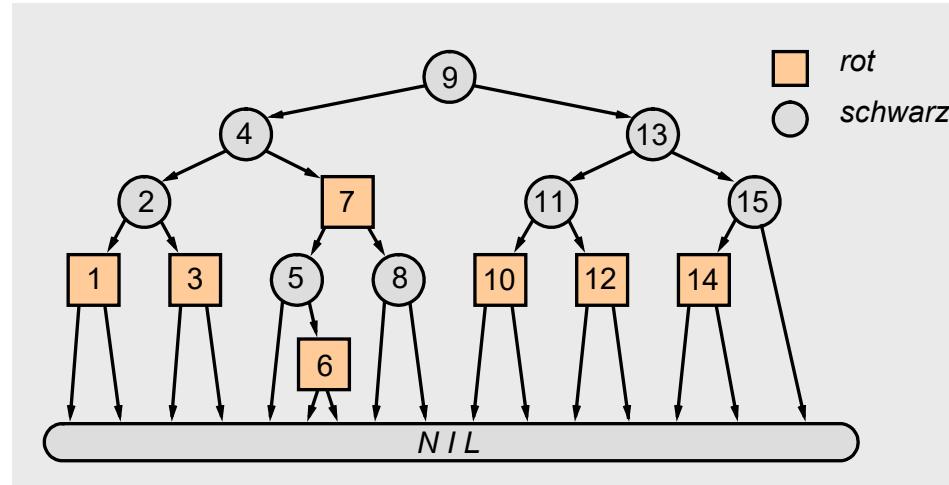
Ein gefärbter Binärbaum ist genau dann ein Rot-Schwarz-Baum, wenn folgende vier Bedingungen erfüllt sind:

1. Jeder Knoten ist entweder rot oder schwarz
2. Jedes Blatt ist schwarz
3. Wenn ein Knoten rot ist, dann sind seine beiden Söhne schwarz
4. Jeder Weg von einem Knoten zu einem seiner Blätter enthält die gleiche Zahl von schwarzen Knoten

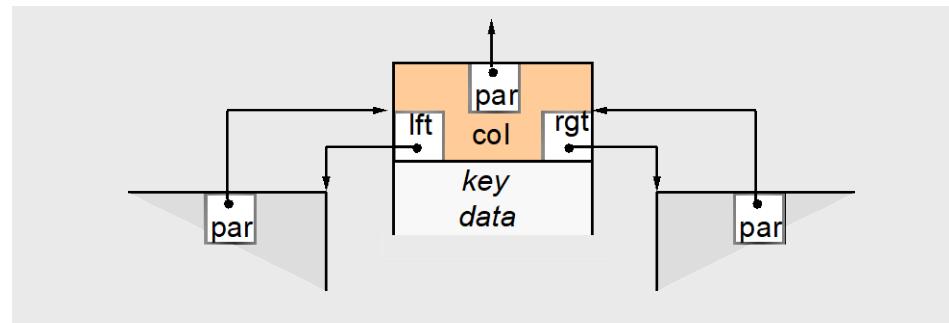
Exkurs: *tree*, die Grundlage assoziativer Behälter (2)

Beispiel

(jeder Weg hat
drei schwarze
Knoten)



Knoten- implementierung

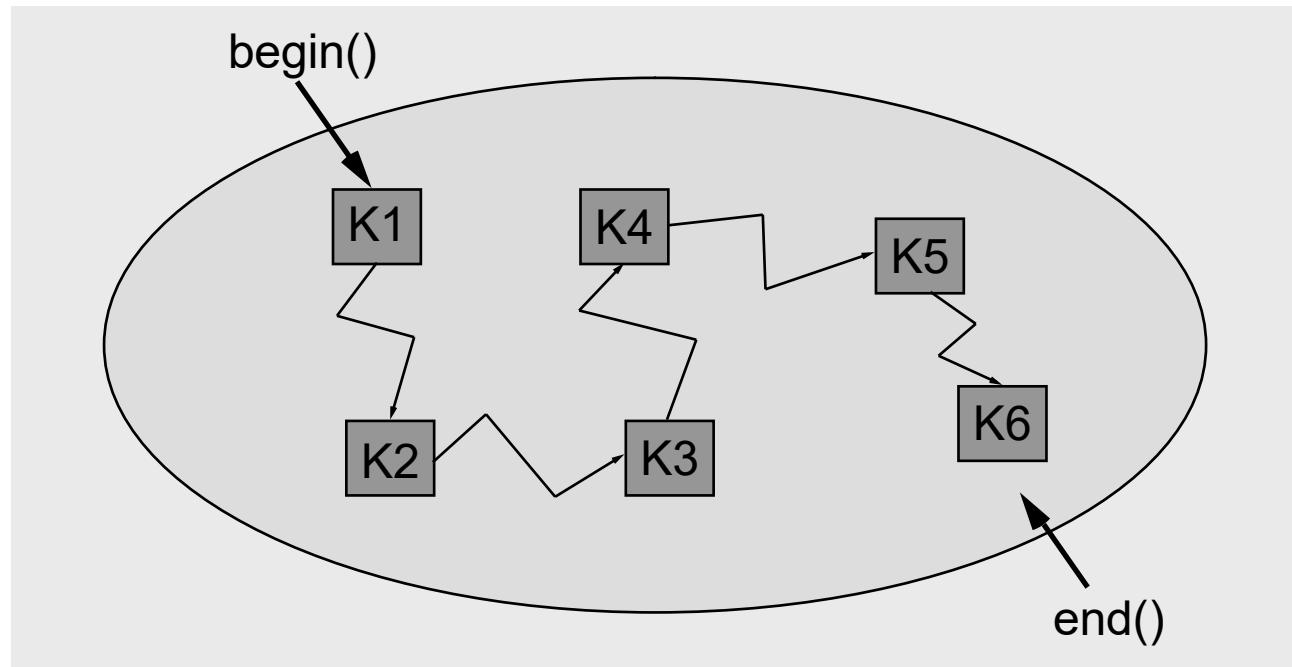


Zentrale Eigenschaften

- Bei n inneren Knoten maximale Höhe: $2 * \lceil d(n + 1) \rceil$
- Zeitkomplexität f. Suchen: $O(\lceil d(n) \rceil)$
- Zeitkomplexität f. Einfügen u. Entfernen: $O(\lceil d(n) \rceil)$

- Behälter für eindeutige Schlüsselwerte (entsprechend dem mathematischen Konzept einer Menge)
- Ermöglicht schnellen Zugriff auf diese Schlüsselwert
- Bidirekt. Iteratoren liefern Schlüsselwerte in sortierter Reihenfolge

Konzeptuelle Sicht



Klassenschablone `set<>`

```
template<class Key, class Compare = less<Key>, class Allocator = ... >
class set {    // sizeof(set<T>): 24 w. g++, 8 w. c1 (32 bit)
public:
    explicit set(const Compare &comp = Compare(), const A. &a = A.());
    set(const set<Key, Compare, Allocator> &s);
    set(iterator first, iterator last,
         const Compare &comp = Compare(), const A. &a = A.());
    ~set();
    set<Key, Compare, Allocator> &operator=(const set<K., C., A.> &s);
    [const_][reverse_]iterator [r]begin() [const];
    [const_][reverse_]iterator [r]end()   [const];
    bool empty() const;
    size_type size() const;
    size_type max_size() const;
    void swap(set<Key, Compare, Allocator> &s);
    pair<iterator, bool> insert(                           const value_type &x);
    iterator           insert(iterator pos,      const value_type &x);
    void               insert(iterator first, iterator last);
    void       erase(iterator pos);
    size_type erase(const key_type &x);
    void       erase(iterator first, iterator last);
    iterator   find (const key_type &x) const;
    size_type count(const key_type &x) const; // result == (0 or 1)
    iterator   lower_bound(const key_type &x) const; // additionally:
    iterator   upper_bound(const key_type &x) const; // equal_range returns
}; // set<Key, Compare, Allocator>                                // pair<it., it.>
```

Zusätzlich: alle relat.
Operatoren in Form
glob. Funktionen

1. Standard-Vergleichsoperation (`less`, also Operator `<`)

```
std::set< int /*, less<int> */ > is_std_less; // uses < for type int
```

2. Benutzerdefinierte Vergleichsoperation für `set`-Datentyp

```
struct int_greater { // function object see below, cf. greater<int>
    bool operator()(int i, int j) const {
        return i > j;
    } // operator()
}; // int_greater
```

```
std::set<int, int_greater > is_ud_greater; // u.d. type provides op.
```

Benutzerdefinierte Vergleichsfunktionen für jedes `set`-Objekt

```
inline bool int_less (int i, int j) { // function, no type
    return i < j;
} // int_less
```

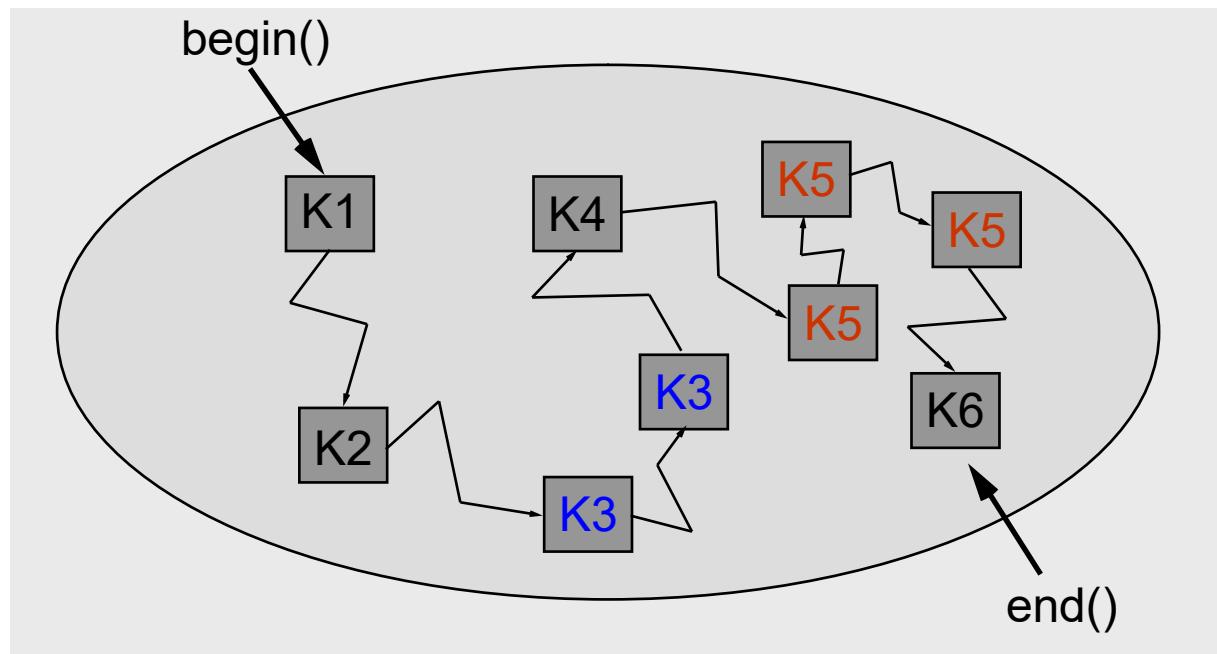
```
inline bool int_greater(int i, int j) { // function, no type
    return i > j;
} // int_greater
```

```
using int_set = std::set<int, bool (*)(int, int) >; // type only
int_set is0; // compiles but does not work at runtime
int_set is1(int_less ); // compares int elements with op.<
int_set is2(int_greater); // compares int elements with op.>
```

Assoziativer Behälter *multiset*<>

- Behälter für Schlüsselwerte, die auch mehrfach vorkommen dürfen (oft auch als *bag* bezeichnet)
- Ermöglicht schnellen Zugriff auf diese Schlüsselwerte
- Bidirekt. Iteratoren liefern Schlüsselwerte in sortierter Reihenfolge

Konzeptuelle Sicht



Klassenschablone *multiset*<>

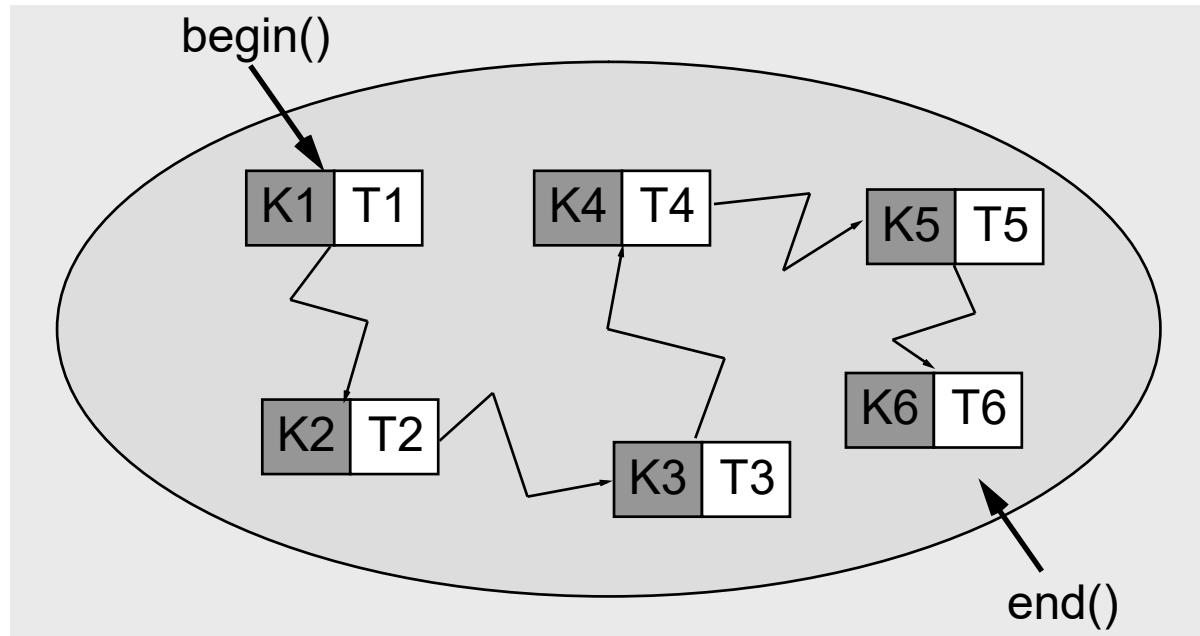
```
template<class Key, class Compare = less<Key>, Allocator = ... >
class multiset { // sizeof(multiset<T>): 24 w. g++, 8 w. c1 (32 bit)
public:
    explicit multiset(const Compare &comp = Compare(), const A. &a = A.());
    multiset(const multiset<Key, Compare, Allocator> &ms);
    multiset(iterator first, iterator last,
              const Compare& comp = Compare(), const A. &a = A.());
    ~multiset();
    multiset<Key, Compare, Allocator> &operator=(const multiset<K,C,A> &ms);
    [const_][reverse_]iterator [r]begin() [const];
    [const_][reverse_]iterator [r]end() [const];
    bool empty() const;
    size_type size() const;
    size_type max_size() const;
    void swap(multiset<Key, Compare, Allocator> &ms);
    iterator insert(           const value_type &x);
    iterator insert(iterator hintPos, const value_type &x);
    void     insert(iterator first, iterator last);
    void     erase(iterator pos);
    size_type erase(const key_type &x);
    void     erase(iterator first, iterator last);
    iterator find (const key_type &x) const;
    size_type count(const key_type &x) const; // result >= 0
    iterator lower_bound(const key_type &x) const; // additionally:
    iterator upper_bound(const key_type &x) const; // equal_range
}; // multiset<Key, Compare, Allocator>
```

Zusätzlich: relat.
Operatoren in Form
glob. Funktionen

Assoziativer Behälter *map*<>

- Behälter für eindeutige Paare aus (Schlüssel, Daten), entsprechend dem mathematischen Konzept einer Menge
- Ermöglicht schnellen Zugriff auf diese Paare über den Schlüssel
- Bidirekt. Iteratoren liefern Paare in nach Schlüsseln in sortierter Reihenfolge

Konzeptuelle Sicht



Klassenschablone *map*<>

```
template<class Key, class T, class Compare = less<Key>, class Alloc. = ... >
class map {    // sizeof(map<T>): 24 w. g++, 8 w. cl (32 bit)
public:
    typedef pair<Key, T> value_type;
    explicit map(const Compare &comp = Compare(), const Allocator &a = ...);
    map(iterator first, iterator last, const Compare &comp = ..., const A. ...);
    map(const map<Key, T, Compare, A.> &m);
    ~map();
    map<...> &operator=(const map<...> &m);
    [const_] [reverse_] iterator [r]begin() [con
    [const_] [reverse_] iterator [r]end()   [con
    size_type size() const;
    bool empty() const;
    size_type max_size() const;
    void swap(map<Key, T, Compare, A.> &m);
    T &operator[](const key_type &k); // no at
    pair<iterator, bool> insert(
        const value_type &x);
    iterator insert(iterator hintPos, const value_type &x);
    void insert(iterator first, iterator last);
    void erase(iterator pos);
    size_type erase(const key_type &x);
    void erase(iterator first, iterator last);
    iterator find (const key_type &x);
    size_type count(const key_type &x) const; // returns 0 or 1
    iterator lower_bound(const key_type &x); // additionally:
    iterator upper_bound(const key_type &x); // equal_range
}; // map<Key, T, Compare, Allocator>
```

Beispiel: assoziatives Feld

```
map<string, string> dict;
dict["one"] = "eins";
dict["two"] = "zwei";
...
cout << "eins = " <<
        dict["one"];
```

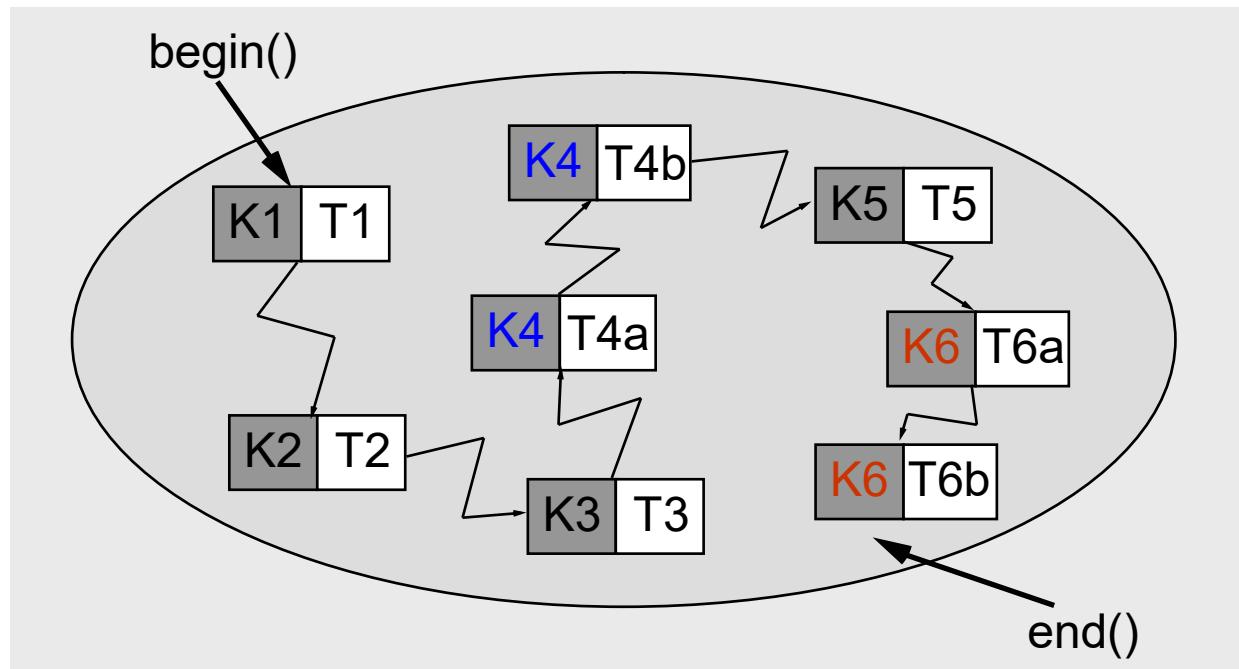
```
const value_type &x);
insert(iterator hintPos, const value_type &x);
insert(iterator first, iterator last);
```

Zusätzlich: relat. Op.
in Form glob. Funkt.

Assoziativer Behälter *multimap*<>

- Behälter für Paare aus (Schlüssel, Daten), wobei die Schlüssel auch mehrfach vorkommen dürfen
- Ermöglicht schnellen Zugriff auf diese Paare über den Schlüssel
- Bidirekt. Iteratoren liefern Paare in nach Schlüssen in sortierter Reihenfolge

Konzeptuelle Sicht



Klassenschablone *multimap*<>

```
template<class Key, class T, class Compare = less<Key>, class Allocator = ... >
class multimap {    // sizeof(multimap<T>): 24 w. g++, 8 w. cl (32 bit)
public:
    exp. multimap(const Compare &comp = Compare() , const Allocator &a = ...);
    multimap(iterator first, iterator last, const Compare &comp = ...);
    multimap(const multimap<Key, T, Compare, A.> &);
    ~multimap();
    multimap<K., T, C., A.> &operator=(const multimap<...> &mm);
    [const_] [reverse_] iterator [r]begin() [const];
    [const_] [reverse_] iterator [r]end() [const];
    bool empty() const;
    size_type size() const;
    size_type max_size() const;
    void swap(multimap<Key, T, Compare, A.> &mm);
    iterator insert(           const value_type &x);
    iterator insert(iterator pos, const value_type &x);
    void insert(iterator first, iterator last);
    void erase(iterator pos);
    size_type erase(const key_type &x);
    void erase(iterator first,
                iterator last);
    iterator find(const key_type &x);
    size_type count(const key_type &x) const;
    iterator lower_bound(const key_type &x);
    iterator upper_bound(const key_type &x);
}; // multimap<Key, T, Compare, Allocator>
```

Zusätzlich: relat.
Operatoren in Form
glob. Funktionen

Kein *operator[]* mehr,
nicht mehr als assoz.
Feld verwendbar!

Beispiel: 2 x Telefonbuch

```
multimap< string,
           list<int> > pb1;
multiset< pair<string,
           list<int>> > pb2;
```

Exkurs: Assoz. Behälter auf Basis von *Hash*-Tabellen

- Schon **früher** Ergänzungen der STL um assoziative Behälter (z.B. in Boost), die *Hash*-Tabellen anstelle balanz. binärer Suchbäume verwenden:
*hash*_[multi]set<Key, ... > und *hash*_[multi]map<Key, T, ... >
- Um Namenskonflikte zu vermeiden, ab **C++11** unter folgender Bezeichnung nun auch in der Standardbibliothek:
*unordered*_[multi]set<Key, ... > u. *unordered*_[multi]map<Key, T, ... >

Beispiel-Klassenschablone *unordered_map*<...>

```
template<class Key,
         class T,
         class Hash    = hash<Key>,           // hash function
         class Comp.  = equal_to<Key>,        // collision detect. & handl.
         class Alloc. = allocator< pair<const Key, T> > >
class unordered_map { // sizeof(uo.m.<T>): 28 w. g++, 32 w. c1 (32 bit)
```

Kollisions-
behandlung:
Verkettung

Es gibt vordef. *Hash*-Funktionen für alle Standard-Datentypen (inkl. *string*)

Beispiel: Telefonbuch zum Dritten

```
unordered_multimap< string, list<int> > pb3; // cf. pb2 before
```

Exkurs: Effizienzvergleich der set-Behälter

Messungen mit jeweils $n = 1$ Mio. Elementen

Betriebssystem: Windows 7 mit 64 Bit, Prozessor: Intel Xeon E3-1270 mit 3,4 GHz

	set<T>		unordered_set<T>	
	g++	cl	g++	cl
T = int , Zufallszahlen				
c.insert(...)	17,35	14,37	9,44	9,24
for mit Iterator	0,59	0,80	2,06	0,41
range-based for	0,53	0,66	0,23	0,27
c.find(...)	5,13	5,35	1,98	2,90
T = string , Zufallszahlen umgewandelt in string				
c.insert(...)	30,75	25,13	12,79	11,23
for mit Iterator	0,94	1,75	2,65	1,31
range-based for	0,97	2,42	0,87	1,89
c.find(...)	41,08	15,93	36,35	7,05

Anmerkung: map hat gleiches Laufzeitverhalten wie set,
weil `map<Key, T>` auf `set<pair<Key, T>>` abgebildet wird

Funktionsobjekte (1)

Funktionsobjekte sind Objekte einer Klasse mit öffentlichem **operator()**

```
class function_object {  
public:  
    T operator()(...);  
}; // function_object
```

Gleiche Verwendung wie Funktionszeiger

```
typedef T (*function_pointer)(T &);  
T f(T &x) { ... }  
function_pointer fp = &f;  
function_object fo;  
T t1, t2;  
t1 = fo(t2); // equals fo.op.()(t2)  
t1 = fp(t2); // equals (*fp)(t2)
```

Vorteile gegenüber Funktionszeigern:

- Indirektion bei Aufruf entfällt
- Code kann inline expandiert werden
- Unterschiedliche Objekte sind möglich
- Jedes Objekt kann eigenen Zustand verwalten (Parametrisierung)

Beispiel

```
class cmp {  
    int x;  
public:  
    cmp(int x): x(x) {}  
    bool op.()(int e) {  
        return e == x;  
    } // op.()  
}; // cmp
```

```
vector<int> v;  
find_if(v.begin(),  
        v.end(),  
        cmp(17));
```

alternativ:

```
[ ] (int e) {  
    return e == 17;
```

Funktionsobjekte (2)

1. Basisklassen: Zur einfachen Definition und Verwendung von Funktionsobjekten mit einem oder zwei Argumenten:

```
template<class Arg, class Result>
struct unary_function {
    typedef Arg      argument_type;
    typedef Result   result_type;
}; // unary_function<Arg, Result>

template<class Arg1, class Arg2, class Result>
struct binary_function {
    typedef Arg1     first_argument_type;
    typedef Arg2     second_argument_type;
    typedef Result   result_type;
}; // binary_function<Arg1, Arg2, Result>
```

2. Arithmetische Operationen: Für alle arithmetischen Operationen (+, -, *, /, %) gibt es entsprechende Funktionsobjekte (*plus*, *minus*, *times*, *divide*, *modulus* und *negate*), z. B.:

```
template<class T>
struct plus: public binary_function<T, T, T> {
    T operator()(const T &x, const T &y) {
        return x + y;
    } // operator()
}; // plus<T>
```

3. Vergleichsoperationen: Für alle Vergleichsoperationen (`==`, `!=`, `<`, `<=`, `>`, `>=`) gibt es entsprechende Funktionsobjekte (`equal_to`, `not_equal_to`, `less`, `less_equal`, `greater` und `greater_equal`), z. B.:

```
template<class T>
struct equal_to: public binary_function<T, T, bool> {
    bool operator()(const T &x, const T &y) {
        return x == y;
    } // operator()
}; // equal_to<T>
```

4. Logische Operationen: Für alle log. Operationen (`&&`, `||` und `!`) gibt es entsprechende Funktionsobjekte (`logical_and`, `logical_or` und `logical_not`), z. B.:

```
template<class T>
struct logical_and: public binary_function<T, T, bool> {
    bool operator()(const T &x, const T &y) {
        return x && y;
    } // operator()
}; // logical_and<T>
```

Konzept

- Algorithmen (ca. 80) sind unabhängig von der Implementierung der Datenstrukturen (Behälter)
- Parametrisierung erfolgt über Iteratortypen und ...

Verschiedene Ausprägungen

- Spezialisierung über Wert, z. B.:

```
template<class InputIterator, class T>
InputIterator find(..., const T &value);
```

- Spezialisierung über **Prädikat** (Ergebnis vom Typ *bool*), z. B.:

```
template<class InputIterator, class Predicate>
InputIterator find_if(..., Predicate pred);
```

- Oft zwei Versionen: *copy* und *in-place*, z. B.:

```
template <class InpIt1, class InpIt2,
          class OutpIt>
OutputIterator merge(InpIt1 f1, ... l1, InpIt2 f2, ... l2,
                     OutpIt result);
template <class BidirIt>
void inplace_merge(BidirIt first, middle, last);
```

Algorithmen: Übersicht (2)

1. Operationen auf Sequenzen ohne Änderung

*for_each, find(_if), adjacent_find,
count(_if), mismatch, equal, search*

2. Operationen auf Sequenzen mit Änderung

*copy(_backward), swap(_ranges), transform,
replace(_copy)(_if), fill(_n), generate(_n),
remove(_copy)(_if), unique(_copy), reverse(_copy), rotate(_copy),
[random_]shuffle, (stable_)partition*

3. Sortieren und verwandte Operationen

*(stable_)sort, partial_sort(_copy),
nth_element,
lower_bound, upper_bound, equal_range,
binary_search, (inplace_)merge,
includes, set_union, set_intersection, set_(symmetric_)difference,
push_heap, pop_heap, make_heap, sort_heap,
min(_element), max(_element),
lexicographical_compare,
next_permutation, prev_permutation*

4. Numerische Operationen

accumulate, inner_product, partial_sum, adjacent_difference

Algorithmen: Beispiele (1)

Algorithmus *for_each*

```
template<class InputIterator, class Function>
Function for_each(InputIterator first,
                  InputIterator last,
                  Function f) {
    while (first != last)
        f(*first++);
    return f;
} // for_each<...>
```

Algorithmus *find_if*

```
template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first,
                      InputIterator last,
                      Predicate pred) {
    while (first != last && !pred(*first))
        ++first;
    return first;
} // find_if<...>
```

Algorithmen: Beispiele (2)

Algorithmen `remove_copy_if`

```
template<class InputIterator, class OutputIterator,
         class Predicate>
OutputIterator remove_copy_if(InputIterator first, ...last,
                           OutputIterator result,
                           Predicate pred) {
    while (first != last) {
        if (!pred(*first))
            *result++ = *first;
        ++first;
    } // while
    return result;
} // remove_copy_if<...>
```

```
template<class ForwardIterator, class Predicate>
ForwardIterator remove_if(ForwardIterator first, ...last,
                        Predicate pred) {
    first = find_if(first, last, pred);
    ForwardIterator next = first;
    return first == last ?
        first :
        remove_copy_if(++next, last, first, pred);
} // remove_if<...>
```

1. Vergleichsoperatoren

Zur Vermeidung redundanter Definitionen von Vergleichsoperatoren basieren alle Vergleiche auf *operator==* und *operator<*, z. B.:

```
template<class T1, class T2>
inline bool operator!=(const T1 &x, const T2 &y) {
    return !(x == y);
} // operator!=
```

Und z. B.:

```
template<class T1, class T2>
inline bool operator<=(const T1 &x, const T2 &y) {
    return !(y < x);
} // operator<=
```

- Für alle **Standardtypen** sind diese Vergleichsoperatoren definiert
- Für **selbstdefinierte Datentypen** (Klassen) sollten *operator==* und *operator<* (wenn sinnvoll möglich) definiert werden:
 - als öffentliche Methoden oder
 - (besser) als globale Funktionen

Zur Not reicht aber *operator<* aus,
siehe Bsp. unten

Exkurs: Vergleichsoperatoren

```
#include <utility> // relational operators in namespace rel_ops
using namespace std::rel_ops; // provides access to !=, <=, >, >=
class person {
public:
    string fn, ln; // first name and last name
...
}; // person

bool operator<(const person &p1, const person &p2) {
    return (p1.ln < p2.ln) ||
           ((p1.ln == p2.ln) && (p1.fn < p2.fn));
} // operator<

// either specific for person only:
bool operator==(const person &p1, const person &p2) {
    return (p1.fn == p2.fn) && (p1.ln == p2.ln);
} // operator==

// or generic for any types T and based on operator<:
template <class T>
bool operator==(const T &x, const T &y) {
    return !(x < y) && !(y < x);
} // operator==

// all the others will be derived:           !=          from   ==
// (via gen. operators in rel_ops) and <=, >, >=  from   <
```

Wichtige Basiskomponenten (2)

2. Paare von heterogenen Werten werden wie folgt realisiert

```
template<class T1, class T2>
struct pair {
    T1 first;
    T2 second;
    pair(); // uses T1() and T2() for first and second
    pair(const T1 &first, const T2 &second);
}; // pair<...>
```

Vordefinierte Vergleichsoperatoren:

```
template<class T1, class T2>
inline bool operator==(const pair<T1, T2> &x, const pair<T1, T2> &y) {
    return (x.first == y.first) &&
           (x.second == y.second);
} // operator==
```

```
template<class T1, class T2>
inline bool operator< (const pair<T1, T2> &x, const pair<T1, T2> &y) {
    return (x.first < y.first) ||
           ( !(y.first < x.first) && (x.second < y.second) );
} // operator<
```

Generische Hilfsfunktion *make_pair* zur einfacheren Konstruktion:

```
pair<int, double>(5, 3.14); // long form
make_pair(5, 3.14); // short form
```

Header-Dateien

Datei	Inhalt
<i>Iteratoren</i>	
<iterator>	Iteratoren und div. Unterstützung für Iteratoren
<i>Sequentielle Behälter</i>	
<vector> <array>	Behälter <i>vector</i> [und <i>array</i>]
<[forward_]list>	Behälter (<i>forward_</i>) <i>list</i>
<deque>	Behälter <i>deque</i>
<i>Adapter für sequentielle Behälter</i>	
<stack>	Adapter für Behälter: <i>stack</i>
<queue>	Adapter für Behälter: [<i>priority_</i>] <i>queue</i>
<i>Assoziative Behälter</i>	
<[unordered_]map>	Behälter (<i>multi</i>) <i>map</i> mit bin. Suchbaum [oder Hashtabelle]
<[unordered_]set>	Behälter (<i>multi</i>) <i>set</i> mit bin. Suchbaum [oder Hashtabelle]
<bitset>	Spezielle Implementierung anstelle von <i>set<bool></i>
<i>Algorithmen</i>	
<algorithm>	alle Algorithmen
<i>Sonstiges</i>	
<utility>	Operatoren (f. Vergleich in <i>std::rel_ops</i>) und Paare (<i>pair</i>)
<functional>	Funktionsobjekte

Verschiedene Meinungen zu STL

- **Lob:** Effizienz, Eleganz und Mächtigkeit
- **Tadel:** z. T. schlechte Namen, keine Sicherheit (wie in C/C++ üblich)

Typische Fehler

1. Endlos-Iteration

```
list<int> a, b;  
list<int>::iterator p = find(a.begin(), b.end(), x);
```

Wenn `a.end()` erreicht wird, läuft `find` in Endlosschleife

2. "Schmutziger" Iterator

```
list<int> a, b;  
list<int>::iterator pa = a.begin();  
b.erase(pa); // ???
```

Resultat ist implementierungsabhängig: Im HP-Original wird

- Erstes Element von `a` gelöscht und
- Länge von `b` verringert

3. Bereichsüberschreitung

```
list<int>::iterator p = find(a.begin(), a.end(), x);  
cout << *p; // error if p is past the end
```

4. Nichtinitialisierter Iterator

```
list<int>::iterator p;  
cout << *p; // error
```

5. Iterator auf gelösches Element

```
list<int>::iterator p = a.begin();  
list<int>::iterator q = p;  
a.erase(q);  
cout << *p; // error
```

6. Iterator auf verschobenes Element

```
vector<int>::iterator p = b.begin();  
b.reserve(1000);  
cout << *p; // maybe an error
```

7. Iterator auf Element in gelösctem Behälter

```
list<int> *p1 = new list<int>;  
list<int>::iterator p = p1->begin();  
delete p1;  
cout << *p; // error
```

Verbesserungen von C. S. Horstmann

Zweck: kleine, aber nützliche Erweiterung der STL zur Verbesserung der Sicherheit bei gleicher Schnittstelle

Methode: Instrumentierung der Iteratoren mit vier Laufzeitprüfungen

1. Bei **Dereferenzierung** muß Iterator gültig sein: muß zu Behälter gehören und darf nicht hinter das Ende zeigen
2. Bei **Erhöhung** muß Iterator vorher gültig und nachher gültig sein oder hinter das Ende zeigen;
bei **Verminderung** muß Iterator vorher gültig sein oder hinter das Ende zeigen und nachher gültig sein
3. Bei **Differenzbildung** oder **Vergleich** ($<$, \leq , ...) zweier Iteratoren, müssen beide in den gleichen Behälter zeigen
4. Bei **Zugriff auf Behälter** über Iterator, muß Iterator in den jeweiligen Behälter zeigen, z. B. `c.erase(it)`

Außerdem: Änderung an Behältern (Zuweisung, Einfügen, Entfernen) ändert alle relevanten Iteratoren

Fehlerverhalten: Prüfung einer Assertion schlägt fehl
(Programmabbruch mit Meldung der Fehlerstelle, `assert.h`)

Nachteil: schlechteres Laufzeitverhalten, weil Prüfungen erst zur Laufzeiterfolgen können

Strategie

- Während Entwicklung Header-Dateien der Safe STL verwenden
- Für Produktionsversionen Standard-Header-Dat. verwenden

Anwendung

- Header-Dateien der Safe STL im Verzeichnis .../stl_safe
- Übersetzung mit

```
g++ -I.../stl_safe file.cpp
```

Header-Dateien der Safe STL:

Datei	Inhalt
<i>algobase.h</i>	Instrumentierte Basisfunktionalität für alle Algorithmen
<i>deque.h</i>	Instrumentierter Behälter <i>deque</i>
<i>list.h</i>	Instrumentierter Behälter <i>list</i>
<i>tree.h</i>	Instrumentierter Rot-Schwarz-Baum
<i>vector.h</i>	Instrumentierter Behälter <i>vector</i>

Rankings von Programmiersprachen

Heinz Dobler,
Version 14, 2025



Quelle: blog.vovici.com

Rankings nach \$, €, £, ¥, ... 2024



... from Sept. 2023
to Dec. **2024**,
DevJobsScanner
has analyzed more
than 10M dev job
offers from around
the world to help
us understand the
market and the
most trending and
paid languages....

Quelle:

[https://www.devjobsscanner.com/blog/
top-10-highest-
paid-programming-
languages/](https://www.devjobsscanner.com/blog/top-10-highest-paid-programming-languages/)

Übersicht: Rankings nach der Bedeutung



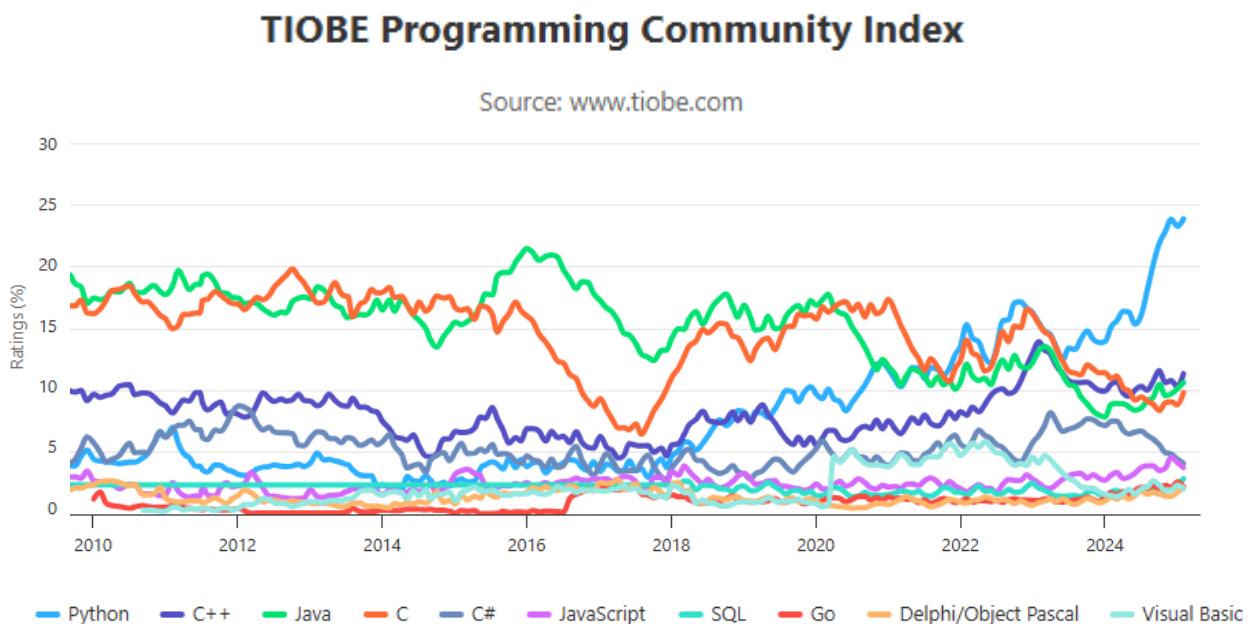
Es gibt viele Rankings von Programmiersprachen nach der Bedeutung auf Basis unterschiedlicher Kriterien

(siehe z. B. https://en.wikipedia.org/wiki/Measuring_programming_language_popularity)

Die drei wichtigsten Rankings:

- *Programming Community Index (PCI)*
- *PopularitY of Programming Language (PYPL) index*
- *RedMonk Programming Language Ranking*

Feb 25	Feb 24	Language
1	1	Python
2	3	▲ C++
3	4	▲ Java
4	2	▼ C
5	5	C#
6	6	JavaScript
7	7	SQL
8	8	Go
9	12	▲ Pascal
10	9	▼ Visual Basic
11	11	Fortran
12	15	▲ Scratch
13	18	▲ Rust
14	10	▼ PHP
15	21	▲ R
16	13	▼ MATLAB
17	14	▼ Assembly
18	19	▲ COBOL
19	20	▲ Ruby
20	24	▲ Prolog



"The Programming Community index is an **indicator of the popularity of programming languages**. The index is updated once a month. The ratings are based on the **number of skilled engineers world-wide, courses and third-party vendors**. The popular search engines Google, Bing, Yahoo, Wikipedia, Amazon, YouTube and Baidu are used to calculate the ratings. Observe that the TIOBE index is not about the *best* programming language or the language in which *most lines of code* have been written."

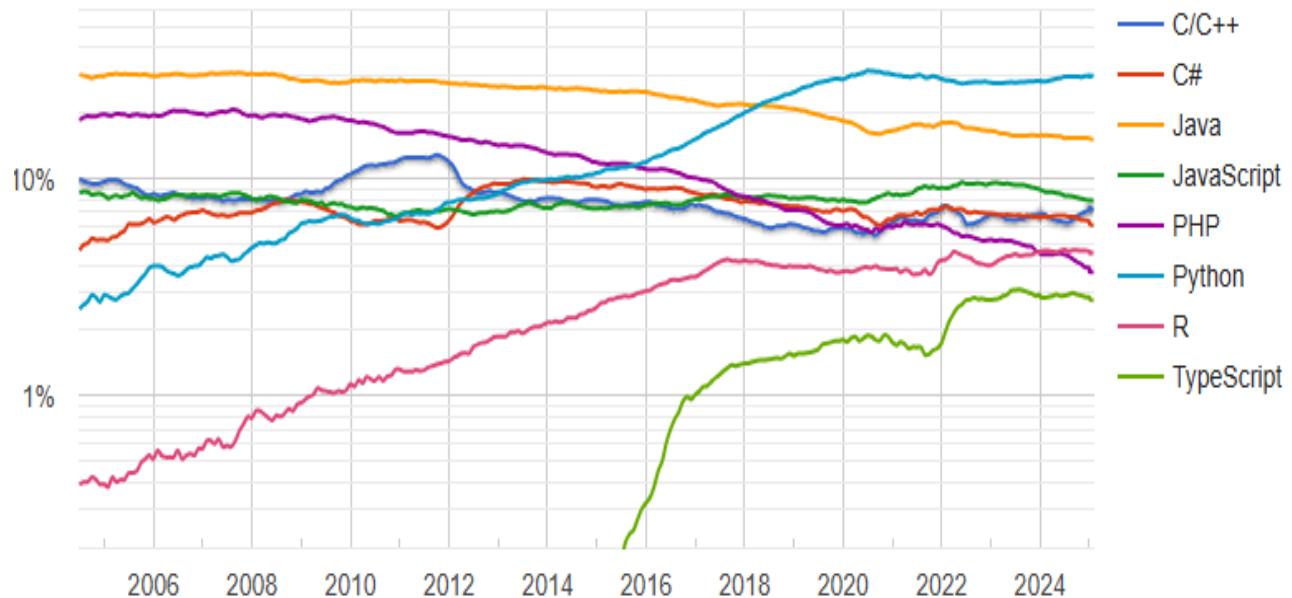
Quelle: <https://www.tiobe.com/tiobe-index/>
 "The Importance Of Being Earnest"

Popularity of Programming Language (PYPL) index

Worldwide, Feb 2025	
Rank	Language
1	Python
2	Java
3	JavaScript
4	C/C++
5	C#
6	R
7	PHP
8	Rust
9	Objective-C
10	TypeScript
11	Swift
12	Go
13	Kotlin
14	Matlab
15	Ada
16	Ruby
17	Dart
18	Powershell
19	VBA
20	Lua

Worldwide, Python is the most popular language, Rust grew the most in the last 5 years (2.3%) and Java lost the most (-2.6%)

PYPL Popularity of Programming Language



"The PYPL PopularitY of Programming Language index is **created by analyzing how often language tutorials are searched on Google**: the more a specific language tutorial is searched, the more popular the language is assumed to be. It is a leading indicator. The raw data comes from [Google Trends](#), so that anyone can verify it, or make the analysis for his own country".

Quelle: <http://pypl.github.io/PYPL.html>

RedMonk Programming Language Ranking

RedMonk ermittelt zweimal pro Jahr: **Anzahl d. Anfragen bei StackOverflow** und **Anzahl d. heruntergeladenen Projekte von GitHub**



Quelle: <https://redmonk.com/sogrady/2024/09/12/language-rankings-6-24/>

Java

Heinz Dobler
Version 21, 2025



Quelle: CIA World Factbook

Java-Foliensatz unterscheidet sich im Konzept und in der Struktur von den C- und C++-Foliensätzen

Inhaltliches Konzept

- Keine vollständige "Einführung in Java"
- Behandelt im Wesentlichen nur die Sprache Java, Klassenbibliothek im Überblick, Behälterklassen in eigener Präsentation
- Kenntnisse über C und C++ (sowie über objektorientierte Programmierung) werden vorausgesetzt

Mehrteilige Struktur

1. Allgemeines
2. Unterschiede zwischen Java und C
3. Objektorientierte Programmierung in Java

4. Neue Spracheigenschaften in Java...
1.1, 1.2, 1.3, 1.4, 5.0, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15 und 16 (mittlerweile gibt es schon 23)

} Basissprache:
Java 1.0
}
... und Erweiterungen

Übersicht

- Geschichte und Charakterisierung
- Eigenschaften
- Vom Quelltext zum Programm
- Programmstruktur
- Strukturierung des Namensraums
- Datentypen: Wert- und Referenz-DT.
- Zeichenketten
- Unterschiede zu C (und C++):
Operatoren, Anweisungen,
Ausnahmen und Behandlung
- Klassen und Objekte
- Klassenkomponenten
- Objektfreigabe und Finalisierung
- Abgeleitete Klassen und Vererbung
- Überdeckung in abgeleiteten Klassen
- Überschreiben von Methoden, Aufruf
- Sichtbarkeit: Klassen u. Komponent.

- Abstrakte Methoden und Klassen
- Schnittstellen
- Fehlende C++-Eigenschaften
- Überblick: Neuigkeiten in Java 1.1
- Überblick: Neuigkeiten in Java 1.2
- Überblick: Neuigkeiten in Java 1.3
- Überblick: Neuigkeiten in Java 1.4
- Überblick: Neuigkeiten in Java 5.0
- Java 5.0 im Detail:
 - Neue *for*-Schleife
 - Autom. Ein- u. Auspacken, *auto (un-)boxing*
 - Aufzählungstypen
 - Statische Importe u. variable Arg.Listen
 - Generizität
 - Annotationen
- Überblick: Java 6
- Überblick: Java 7 u. einige Details
- Überblick: Java 8 u. einige Details
- Überblick: Java 9 bis 16

Übersicht: Exkurse

- Benchmark-Ergebnisse für C(++) und Java
- Vom Quelltext zum (Bytecode in Textform)
- Übersicht über die Java-Plattform
- Auszug aus der Paket- u. Klassenhierarchie
- Einfaches Beispiel für Anwendung von *JavaDoc*
- Wurzelklasse *java.lang.Object*
- Ausgefranste Felder (*r/jagged arrays*)
- Felder ≈ Objekte
- Hilfsklasse *Arrays* für Felder
- Benchmark der Zeichenketten-Klassen
- Hierarchie der Ausnahmenklassen
- Alle *final*-en Möglichkeiten
- Geschachtelte Klassen und *class*-Dateien
- Listen-Beispiel zu geschachtelten Klassen
- Dem JIT-Compiler auf die Finger schau'n
- Vergleich polymorpher mit generischen Behältern

- 1990 Erste Arbeiten an **OAK** (*object application kernel*) für eingebettete Systeme bei Sun unter **James Gosling**
- 1996 Java **1.0**, JDK 1.0 212 Kl., 8 Pak.
- 1997 Java **1.1**, JDK + JRE 1.1 504 Kl., 23 Pak.
- 1998 Java **1.2**, J2SDK 1.2 1520 Kl., 59 Pak.
- JVM mit *JIT Comp.* und außerdem neuer Name: **Java-2 (J2)**
- 2000 Java **1.3**, Verbesserungen u. kleinere Änderungen keine wesentl. Erweiterungen 1842 Kl., 76 Pak.
- 2002 Java **1.4**, Spracherweiterung um *assert*, umfangr. Erw. der Bibliothek (z. B. XML, reg. Ausdr., *NewIO*, *Logging*, *WebStart*), drei Editionen: ME, SE, EE 2991 Kl., 135 Pak.
- 2004 Java **5.0**: Spracherweiterungen für Generizität, (*Un-)Boxing*, neu *for*-Schleife, ... 3562 Kl., 166 Pak.
- 2006 Java **6**: Erweiterungen der Bibliothek und bei Werkzeugen
- 2011 Java **7**: Viele kleinere Verbesserungen
- 2014 Java **8**: Lambda-Ausdrücke, Bibliothek: *Streams*, *Date*, *Time* ...
- 2017 Java 9: Modulsystem (auch Aufteilung der Standardbibliothek in Module)
- 2018 Java 10: kleinere Erweiterungen (*var* zum Typisieren lokaler Variablen)
Java **11 LTS**: min. Erweiterungen, i. d. Sprache nun *var* auch für Lambda-Param.
- 
- 
- 

Charakterisierung (Zitat von Sun Microsystems, 1996):

"Java is a simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded and dynamic language"

- **Objektorientierung:** Hauptaugenmerk gilt Objekten, klassenbasiert, Strukturierungsmöglichkeit durch Pakete
- **Interpretative Ausführung:** Java-Compiler erzeugt "Byte-Code" für abstrakte Java-Maschine (*Java virtual machine, JVM*) = Interpretierer
- **Architektur-neutral und portabel:** Java-Programme laufen auf allen Systemen, für die JVM existiert, egal auf welchem System übersetzt wurde
- **Dynamisch und verteilt:** kein Binden von Java-Programmen, Klassen werden vom Interpretierer geladen, Verteilung auf mehrere Rechner möglich
- **Einfach:** nur "unbedingt notwendige" Sprachmittel in bekanntem C/C++-Erscheinungsbild, (leider?) seit 5.0 und 8 nicht mehr ganz so einfach
- **Robust:** starke Typisierung, keine Zeigerarithmetik, Laufzeitprüfungen bei Feldern u. Zeichenketten, Ausnahmenbehandlung u. automat. Speicherber.
- **Sicher:** nicht vertrauenswürdiger (*untrusted*) Code wird beim Laden verifiziert und im "Sandkasten" ausgeführt (mit Einschränkungen z. B. kein Zugriff auf Dateisystem); digitale Signaturen möglich
- **Hohe Effizienz:** Java 1.0 ca. um Faktor 20, Java 1.1 nur mehr um Faktor 10 langsamer als C, weitere Verbesserungen durch *Just-in-Time*-Übersetzung (JIT-Compiler erzeugt für "heiße" Methoden Objektcode, z. B. *HotSpot*)
- **Leichtgewichtige Prozesse:** mehrere Ausführungsfäden (*threads*) in einem Java-Programm mit einfacher Synchronisation für Zugriff auf kritische Bereiche (*critical sections*)

Exkurs: Benchmark-Ergebnisse für C(++) und Java

Sortieren: Feld mit n Elementen

$n = 1.000.000$	Ohne Optim.	Mit Opt.: -O3 bzw. JITC
C <i>int</i> mit <i>qsort</i>	0,14	0,13
C++ <i>int</i> mit <i>sort</i>	0,17	0,06
C <i>char*</i> mit <i>qsort</i>	0,55	0,52
C++ <i>string*</i> mit <i>sort</i>	2,37	1,59
Java <i>int</i> mit <i>Arr.sort</i>	0,67 -Xint	0,09
Java <i>String</i> m. <i>A.sort</i>	4,18 -Xint	0,79

-Xint: interpretation only, no JIT compilation

Quick Sort

Merge Sort

Backtracking: n -Damen

$n = 12$ (14.200 Lös.)	Ohne Optim.	Mit Opt.: -O3 bzw. JITC
C/C++	0,39	0,16
Java	2,67	0,27
(Free-)Pascal INTEGER	+ 0,64 - 0,39	+ 0,60 - 0,29
(Free-)Pascal LONGINT	+ 0,54 - 0,41	+ 0,53 - 0,25

+/-: with or without run-time checks

Zeiten in Sekunden
 Prozessor: Intel Core 2 Duo, 3 GHz
 Betriebssystem: Windows 7, 64 Bit
 C/C++-Entwicklungswerkzeug: **gcc 4.8.1, 32 Bit**
 Java-Entwicklungswerkzeug: **Java 7, 32 Bit**
 Pascal-Entwicklungswerkzeug: **FreePascal 2.6.4**

Beispielprogramm: Sieve.java

```
import java.lang.*;  
  
public class Sieve extends Object {  
    public static final int SIZE = 1000;  
  
    public static void main(String[] args) {  
        boolean[] sieve = new boolean[SIZE + 1];  
        int col; // ... sieve is init. with false  
        for (int i = 3; i <= SIZE; i++)  
            sieve[i] = true;  
        System.out.println(2); // first prime  
        col = 0;  
        for (int i = 3; i <= SIZE; i = i + 2) {  
            if (sieve[i]) {  
                if (col == 8) {  
                    System.out.println();  
                    col = 0;  
                } // if  
                System.out.print(i + "\t");  
                col++;  
                for (int j = 2 * i; j <= SIZE; j += i)  
                    sieve[j] = false;  
            } // if  
        } // for ... no delete for sieve  
    } // main  
} // Sieve
```

Konventionen für Bezeichner:

- Pakete nur in Kleinbuchstaben
- Klassen beginnen mit Großbuchst.
- Methoden beginnen mit Kleinbuchst.
- Konstanten ganz in Großbuchst.
- Sonst CamelCasing

1. Übersetzung mit
javac Sieve.java

2. Ausführung mit ...
java Sieve

... liefert Ergebnis:

2
3 5 7 11 ...

Vom Quelltext zum laufenden Programm

Quelltext

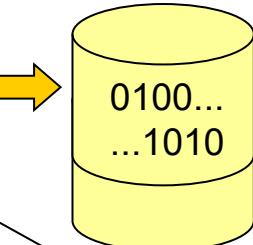
X.java

```
public  
class X {  
...  
} // X
```

Java-Compiler
javac

Bytecode
+ Metainformat.

X.class



Ein-
gaben

Ergeb-
nis

Virtuelle
Java-Maschine (JVM)
java
ev. mit
JITC

Nur
Typ-
infor-
ma-
tion

div.
Bib.
...

Typ-
infor-
ma-
tion
und Imple-
mentierun-
g

Java-Standardbibliothek



Object

String

System

z. B. Vector

Bis Java 8 in Datei **rt.jar**, Größe ca. 60 MB,
seit Java 9 (mit Modulen) im **Verzeichnis jmods** in jmod-Dateien, gesamt ca. 70 MB,
über 20.000 Klassen u./o. Schnittstellen insgesamt, davon
ca. 3.700 im Paket **java** und ca. 3.400 im Paket **javax**

Exkurs: Vom Quelltext zum Bytecode (in Textform)

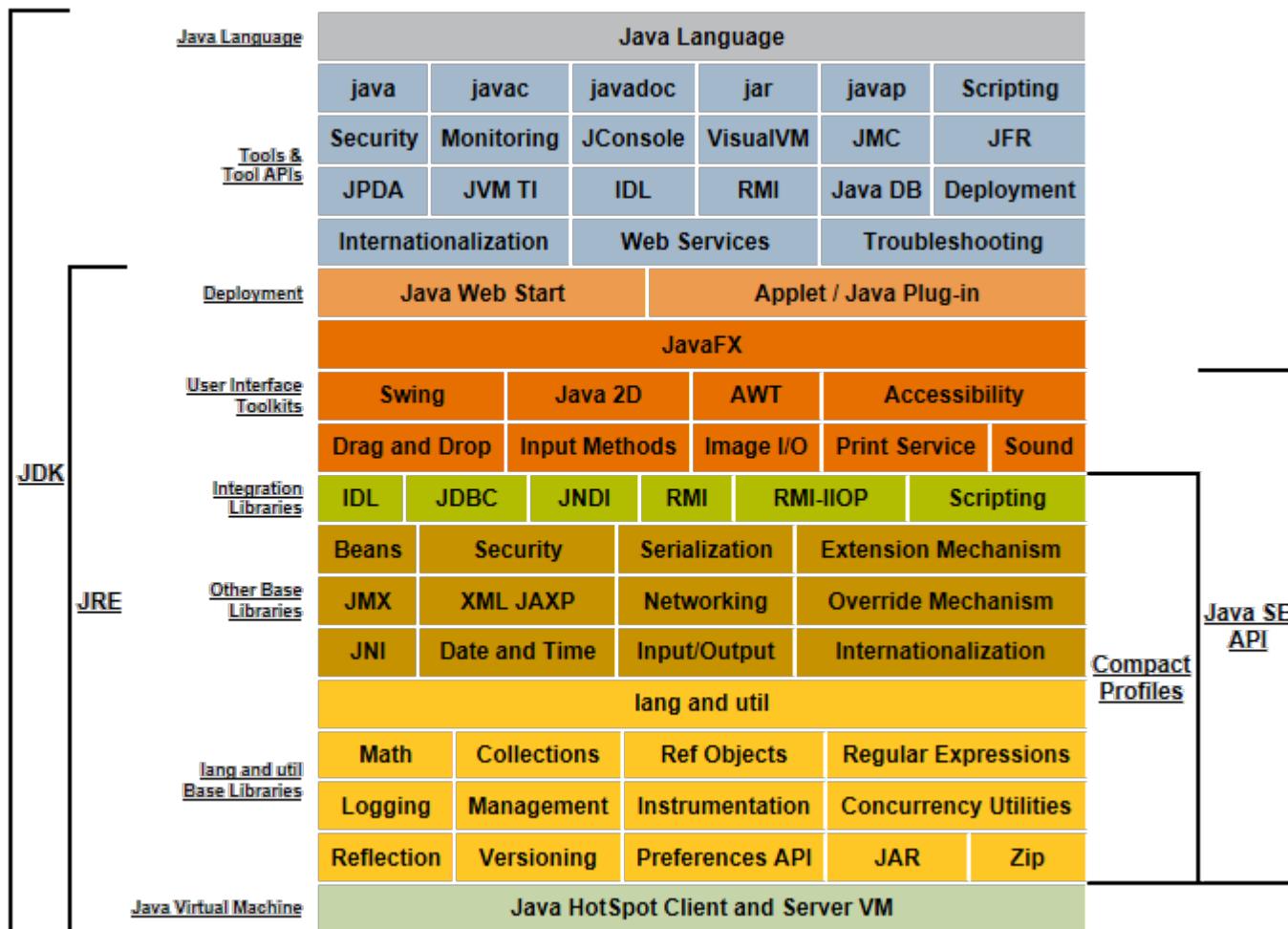
SVP.java:

```
public class SVP {  
  
    static int readInt() {  
        return Integer.parseInt(  
            System.console().readLine());  
    } // readInt  
  
    public static void main(  
        String[] args) {  
        // 1. 2. 3.  
        int a, b, cs;  
        a = readInt();  
        b = readInt();  
        cs = [a * a] + [b * b];  
        System.out.println(cs);  
    } // main  
  
} // SVP
```

javac SVP.java
javap -c SVP.class

```
public static void main(...);  
0: invokestatic #5 // push(readInt())  
3: istore_1 // a = pop()  
4: invokestatic #5 // push(readInt())  
7: istore_2 // b = pop()  
8: iload_1 // push(a)  
9: iload_1 // push(a)  
10: imul // push(pop() * pop())  
11: iload_2 // push(b)  
12: iload_2 // push(b)  
13: imul // push(pop() * pop())  
14: iadd // push(pop() + pop())  
15: istore_3 // cs = pop()  
16: getstatic #6 // push(System.out)  
19: iload_3 // push(cs)  
20: invokevirtual #7 // println(pop())  
23: return  
} // main
```

Exkurs: Übersicht über die Java-Plattform (für Java 8)



Quelle: <http://docs.oracle.com/javase/8/docs/index.html>

- Java-Programme können aus beliebig vielen Klassen bestehen
- Jede Klasse sollte in eigener Datei (mit Dateityp `.java`) gespeichert werden, pro Datei ist nur eine öffentliche (`public`) Klasse erlaubt
- Mind. eine Klasse muss eine öffentl. Klassenmethode `main` haben:

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

- Übersetzung: Alle Quelltexte (Dateityp `.java`) müssen mit Java-Compiler in Bytecode (Dateityp `.class`) übersetzt werden, z. B. `javac X.java`
- Ausführung: virt. Java-Maschine (JVM) wird mit Name der Klasse aufgerufen, deren `main`-Methode auszuführen ist, z. B. `java X`

Anmerkungen

- Kommandozeilen-Argumente werden in einem Feld `args` übergeben (Elemente sind `String`-Objekte), `args[0]` ist schon das erste Argument
- `main`-Methode liefert keinen Fehlercode, analog zu C/C++ aber möglich:

```
System.exit(errorCode); // e.c. has to be an int value
```

- Zugriff Einstellungen besser nicht über systemabh. Umgebungsvariablen (mit `System.getenv`) sondern über Eigenschaften (`properties`), z. B.:

```
String homeDir = System.getProperty("user.home");
```

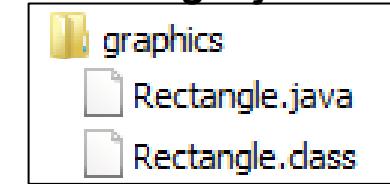
Strukturierung des Namensraums

- Nur Klassen, Schnittst. u. Pakete haben glob. Namen (keine glob. Var./Funkt.)
- Datenkomponenten und Methoden sind Elemente von Objekten/Klassen und müssen über ein Objekt **o** oder mit dem Klassennamen **C** angesprochen werden, z. B. **o.data** (ohne Dereferenzierung) oder **C.method(...)**
- Lokale Variablen werden innerhalb von Methoden definiert und verwendet

Pakete

- Sammlungen von logisch zusammengehörigen Klassen u./o. Schnittstellen
- Zuordnung von Klassen/Schnittst. zu Paketen durch *package*-Anweisung (erste Anw. in Datei), z. B. für *graphics.Rectangle* in Datei *Rectangle.java*:

```
package graphics;  
public class Rectangle {...}
```



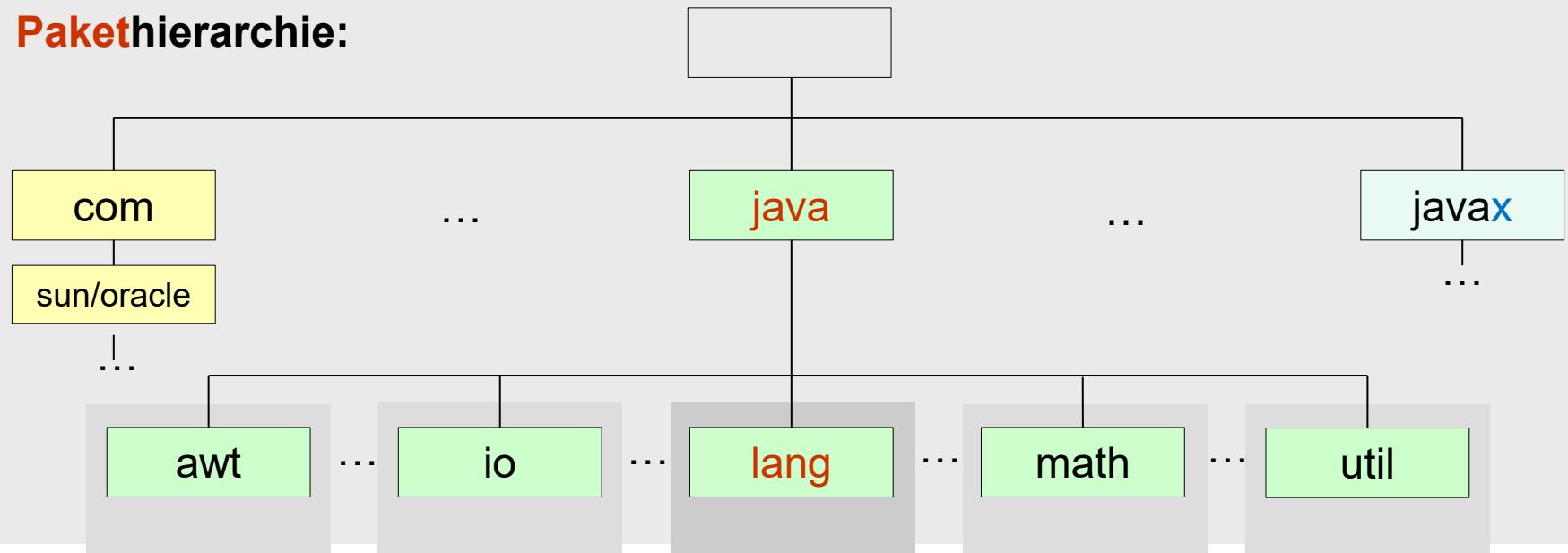
- Unbenanntes Standard-Paket für alle anderen Klassen
- Pakete können hierarchisch strukturiert werden, z. B. *java.lang*
- Hierarchische Paketnamen werden 1:1 auf hierarchische Verzeichnisse abgebildet, z. B. *java.lang* ⇒ ...\\java\\lang\\... bzw. .../java/lang/... (je nach B.S.)

Klassenpfade

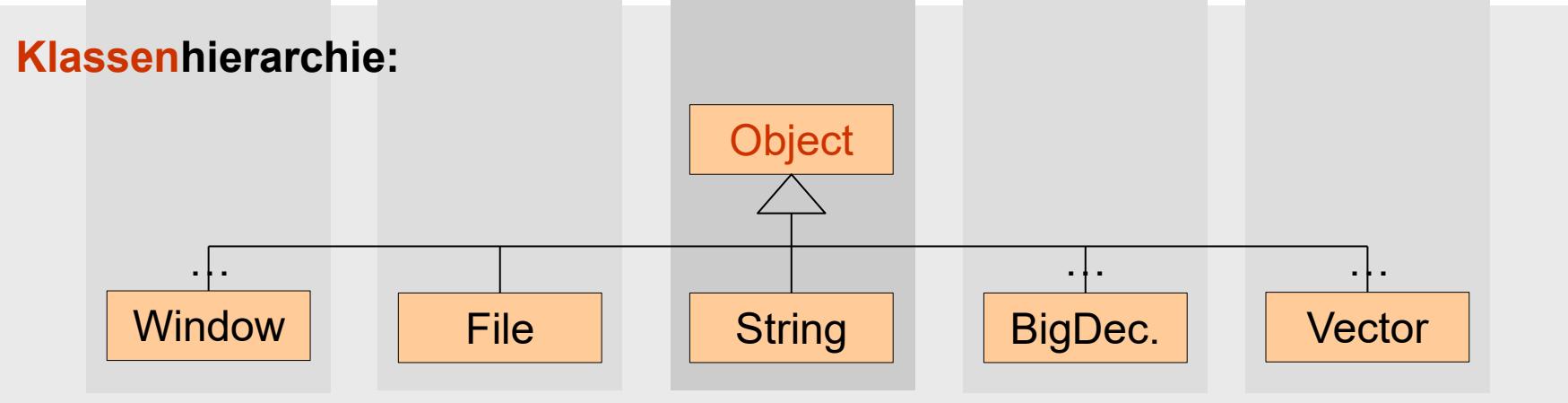
- Pakete u. Klassen der Java-Standardbibliothek werden vom Compiler und der virtuellen Maschine (JVM) automatisch gefunden (über *property "java.home"*)
- Für andere (auch eigene) Bibliotheken müssen Pfade angegeben werden:
 - mittels Umgebungsvariable *CLASSPATH* oder
 - über Kommandozeilenparameter *-classpath* oder kürzer *-cp*

Exkurs: Auszug aus der Paket- u. Klassenhierarchie

Pakethierarchie:



Klassenhierarchie:



Siehe Online-Dokumentation der Standardbibliothek: <https://docs.oracle.com/javase/8/docs/api/index.html>

<https://docs.oracle.com/en/java/javase/19/docs/api/index.html>

Schlüsselwörter (keywords)

abstract	default	goto n.u.	package	synchronized
assert 1.4	do	if	private	this
boolean	double	implements	protected	throw
break	else	import	public	throws
byte	enum 5.0	instanceof	return	transient
case	exports 9	int	requires 9	try
catch	extends	interface	short	var aus J. 10 ist "nur" kontextsensitiv
char	final	long	static	
class	finally	module 9	strictfp 1.2	
const n.u.	Float	native	super	
continue	for	new	switch	void

Anzahl: 53 (zum Vergleich: 37 in C, 51 in Pascal und 84 in C++17)

Bemerkungen

- *const* und *goto* sind reserviert, haben aber keine Bedeutung (*not used, n.u.*)
- Es gibt *new* aber kein *delete* (dafür autom. Speicherber., *garabge collection*)
- *true*, *false* und *null* sind syntaktisch Literale (aber auch reservierte Namen)
- Drei Schlüsselwörter sind erst später dazugekommen (in J. 1.x bzw. 5.0)
- Viel neue Funktionalität wird seit J. 5.0 mittels Annotationen in Form von Metainformation realisiert (z. B. *@Override* oder *@SuppressWarnings*)
- In J. 10 erstes kontextsensitives Schlüsselwort: *var* (vgl. C# oder *auto* in C++)

Java hat/benötigt keinen Präprozessor: alle Aufgaben, die bei C/C++ der Präprozessor erledigt, übernimmt der Java-Compiler selbst

Beispiele

- **Inkludieren von Dateien:** mit der *import*-Anweisung können ganze Pakete (mit all deren Typen) oder einzelne Klassen/Schn. "importiert" werden, z. B.:

```
// import java.lang.*;    is implicit  
import java.io.*;        // "imports" all types in java.io  
import mypack.MyClass;   // "imports" class MyClass only
```

In Java gibt es nur mehr Definitionen,
Header-Dateien mit Deklarationen sind unnötig, dafür Metainformation in *class*-Dateien

- **Konstanten:** Jede *final*-"Variable" ist eine "Konstante", z. B.:

```
public static final double PI = 3.14; // better: Math.PI
```

Konstanten haben pro Klasse eindeutige Namen und sind typisiert

- **Makros:** Optimierende Java-Compiler machen *inline*-Expansion von kurzen (statisch bindbaren) Java-Methoden (Aufruf-Overhead entfällt)

- **Bedingte Übersetzung:** Durch Plattformunabhängigkeit theoretisch überflüssig, praktisch ist Nachbildung mit Konstanten und *if*-Anw. möglich

```
public static final boolean DEBUG = false;  
if (DEBUG) ...
```

Drei Arten von Kommentaren

- C-Kommentare von /* bis */ (nicht geschachtelt)
- C++-Kommentare von // bis zum Zeilenende
- Spezielle doc-Kommentare von /** bis */ (ebenfalls nicht geschachtelt), aus denen mit einem Werkzeug (*javadoc*) automatisch Online-Dokumentation erzeugt werden kann

Beispiel
auf
nächster
Seite

Unicode und spezielle Zeichendarstellungen

- Zeichen, Zeichenketten und Namen bestehen aus 16-Bit Unicode-Zeichen ⇒ bessere Möglichkeiten für Internationalisierung
- 16-Bit Unicode ist kompatibel mit ASCII und in den ersten 256 Zeichen identisch mit ISO 8859-1 (*Latin-1*)
- Java-API und *String*-Klasse sorgen für transparente Zeichenbehandlung; werden nur Latin-1-Zeichen benutzt, ist Unterscheidung zwischen 8-Bit- und 16-Bit-Zeichen nicht möglich
- Unterstützung der C-Zeichendarstellungen (z. B. '\n', '\t' und '\xxx', wobei xxx für eine Folge von Oktalziffern steht)
- Unicode-Zeichen über spezielle Darstellung '\uxxxx', wobei xxxx eine Folge von Hexadezimalziffern ist (z. B. *int* \u00F6; anstelle von *int* ö;)

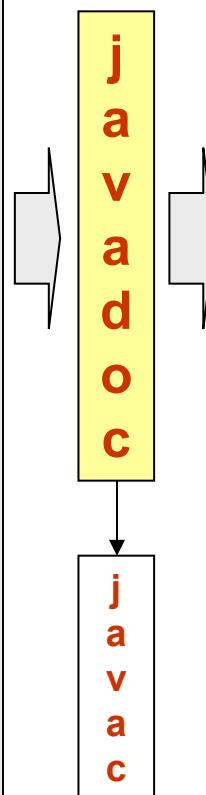
Exkurs: Einfaches Beispiel für JavaDoc

Person.java

```
/*
 * This is a very simple ...
 * @author Heinz Dobler
 * @version 1.0
 */
public class Person {
    /**
     * Minimal set of data components...
     */
    String firstName, lastName;
    java.util.Date dateOfBirth;

    /**
     * The one and only constructor ...
     * @param fn first name
     * @param ln last name
     * @param dob date ...
     */
    public Person(String fn, String ln,
                  java.util.Date dob) {
        firstName = fn;
        lastName = ln;
        dateOfBirth = dob;
    } // Person

    /**
     * Prepends a greeting to the name...
     * @return string representation
     */
    public String toString() {
        return "Hello, my name is " +
               firstName + " " + lastName;
    } // toString
} // Person
```



Person.html

Class Person

java.lang.Object
Person

public class Person
extends java.lang.Object

This is a very simple Person class, built only to demonstrate basic features of javadoc.

Constructor Summary

Constructors

Constructor and Description

Person(java.lang.String fn, java.lang.String ln,
java.util.Date dob)

The one and only constructor to fully initialize a Person object.

Method Summary

All Methods Instance Methods Concrete Methods

Modifier and Type Method and Description

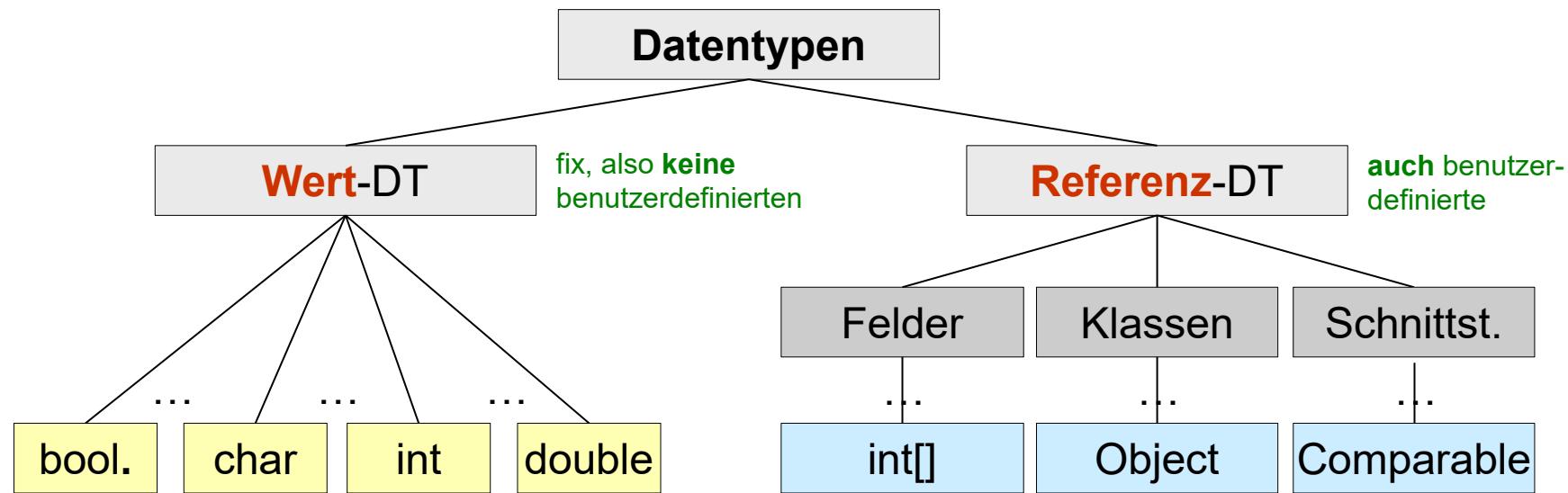
java.lang.String **toString()**

Prepends a greeting to the name, as Persons should
be polite;-)

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify,
notifyAll, wait, wait, wait

Typsyste



Beispiele:

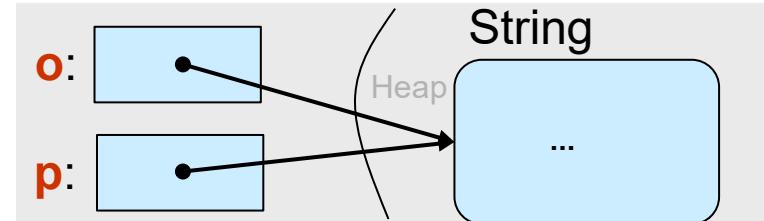
Variablen im Quelltext ...

```
int i, j; ...
i = 17;
j = i; // copies value
```

```
Object o, p; // need no init.  
o = new String(...);  
p = o; // copies reference
```

... und im Speicher

j: 17



Wert- oder skalare Datentypen (1)

Übersicht

→ autom. Typumwandlung

Name	Wert	Std.Wert	Länge (in Bit)	Adapter- klasse
<i>boolean</i>	<i>false</i> oder <i>true</i>	<i>false</i>	1 ; -)	<i>Boolean</i>
<i>char</i>	Unicode-Zeichen auch ganzzahlig aber ohne VZ	\u0000	16	<i>Character</i>
<i>byte</i>	Ganzzahl mit VZ	0	8	<i>Byte</i>
<i>short</i>	Ganzzahl mit VZ	0	16	<i>Short</i>
<i>int</i>	Ganzzahl mit VZ <i>unsigned</i>	0	32	<i>Integer</i>
<i>long</i>	Ganzzahl mit VZ	0	64	<i>Long</i>
<i>float</i>	Fließkommazahl (IEEE 754)	0.0	32	<i>Float</i>
<i>double</i>	Fließkommazahl (IEEE 754)	0.0	64	<i>Double</i>

Datentyp *boolean*

- *true* u. *false* sind keine Zahlen: keine Arithmetik möglich
- Keine automatische Umwandlung in und von anderen Typen
- Umwandlung nur explizit, auch nicht mit *cast*, sondern z. B. für *int i* und *boolean b*:

```
b = (i != 0);  
i = (b) ? 1 : 0;
```

Beispiel zu Adapterkl.:

```
int i = 17;  
Integer io =  
    new Integer(17);  
// or Integer.valueof(17)  
...  
i = (int)io; // or:  
i = io.intvalue();
```

Datentyp **char**

- Werte repräsentieren 16-Bit-Unicode-Zeichen, sind keine Zahlen, haben somit auch kein Vorzeichen
- Zeichen können nach *int* automatisch und mit *cast* nach *byte* oder *short* umgewandelt werden und haben dann ein Vorzeichen (+)

Ganzzahl-Datentypen **byte**, **short**, **int**, **long**

- Werte repräsentieren ganze Zahlen (alle mit Vorzeichen)
- Literale wie in C notiert (*long*-Konstanten mit *l* oder *L*)
- Nur Division (/) durch 0 oder Restbildung (%) mit 0 führt zu einer Ausnahmesituation (*ArithmeticException*), alle anderen arithmetischen Operationen werden *nicht* geprüft (!)

Fließkomma-Datentypen **float**, **double** (gem. IEEE 754)

- Werte repräsentieren reelle Zahlen (mit Vorzeichen)
- Literale wie in C notiert (*float*-Konstanten werden mit *f* oder *F*, *double*-Konstanten können mit *d* oder *D* gekennzeichnet werden)
- Spezielle Werte als Ergebnis arithm. Operationen möglich (z. B. *POSITIVE_INFINITY*, *NEGATIVE_INFINITY*, *NaN*), entsprechende Konstanten in *java.lang.Float/Double*

Charakterisierung

- Referenz-Datentypen (DT) sind Felder, Klassen oder Schnittstellen
- Instanzen von Referenz-DT (Felder und Objekte) sind nur indirekt über ihre Referenzen zugänglich
- Alle benutzerdefinierten DT sind Referenz-DT

Referenzen

- Referenzen sind besondere Art von "Zeigern"
- Dereferenzierung von Referenzen erfolgt automatisch (kein `*p` oder `p->` wie in C/C++ notwendig, auch nicht möglich)
- Keine Referenzarithmetik (z. B. kein `r + i`, sehr wohl aber `a[i]`)
- Keine Möglichkeit zur Ermittlung der Adressen (kein Adressoperator)
- Keine Möglichkeit zur Ermittlung der Größe von Daten (kein `sizeof`)
- Keine Umwandlung (`cast`) von Referenzen in Zahlen oder umgekehrt

Sun: "Java has no pointers"

Referenzliteral `null`

- Vordefinierter Wert (Literal), der "ungültige Referenz" bedeutet
- Standardwert für alle Referenzdatentypen
- `null` kann jeder Variablen eines Referenzdatentyps zugewiesen werden (Ausnahme von der starken Typisierung)
- `null` kann/braucht nicht umgewandelt werden

- Java kennt nur dynamische Objekte (im dyn. Speicherbereich, *heap*)
- Objekte können nur über Referenzen angesprochen werden

Objektreferenzdefinition und Erzeugung

Hinweis: Java hat keine Standard-Klasse *Complex*

- Definition einer (Referenz-)Variablen legt noch kein Objekt an
`Complex c; // no init. necessary but value of c depends on ...`
- Objekterzeugung explizit über den Operator *new*
`c = new Complex(1.0, 2.0); // new may throw OutOfMemoryError`

Zugriff auf Objektkomponenten

- Zugriff auf Datenkomponenten (nur!) mit der Punktnotation
`c.im = 2.0; // operator . may throw NullPointerException ;-)`
- Aufruf von Methoden ebenfalls mit der Punktnotation
`double m = c.magnitude(); // operator . may throw ...`

Automatische Speicherbereinigung (*garbage collection, gc*)

- Es gibt kein Pendant zum Operator *new*, also kein *delete*
- Objekte, die nicht mehr erreichbar sind, können (bei Bedarf oder wenn nichts zu tun ist) automatisch freigegeben werden

`c = null; // helpful because now the Complex object may be gc'd`

Exkurs: Wurzelklasse *java.lang.Object*

Details z. B. auf: <https://docs.oracle.com/javase/8/docs/api/index.html>

```
package java.lang;  
public class Object { // concrete class  
    // no data components!  
    public Object();  
    public String toString();  
    public boolean equals(Object o);  
    public int hashCode(); !
```

```
prot. Object clone() throws CloneNotSupportedException;  
// for meta information (reflection) purposes  
public final Class<?> getClass(); // generic <?> since Java 5.0  
  
// for threading purposes  
public final void notify(); // ... one of the waiting threads  
public final void notifyAll(); // ... all waiting threads  
public final void wait() throws InterruptedException;  
public final void wait(long timeout) throws InterruptedException;  
public final void wait(long timeout, int ns) throws InterruptedException;  
  
// called before deallocation by gc (at most once)  
prot. void finalize() throws Throwable; // ... deprecated since J. 9  
} // Object
```

Beispiel:

```
Object o = new Object();  
...println(o.toString());  
...println(o.equals(new Object()));  
...println(o.hashCode());
```

Ergebnis:

```
java.lang.Object@1cde100 // toString  
false // equals  
30269696 // hashCode
```

Referenz-Datentypen – Felder (1)

Für Felder gilt wie für Objekte:

- sie müssen mit Operator *new* angelegt werden
- sie können nur über Referenzen angesprochen werden
- sie können autom. freigegeben werden (wenn nicht mehr referenzierbar)

Definition von Feld.-Ref. und Erzeugung von eindim. Feldern

Zwei Möglichkeiten: $T a[] \Leftrightarrow T[] a$

1. Angabe der Feldgröße im Operator *new* (Elemente werden initialisiert)

```
int heap[] = new int[1024]; // arr. of ints
```

```
Button[] b = new Button[8]; // arr. of Button ref.s
```

2. Angabe der Größe und der Werte durch Initialisierung

```
int[] twoPower = new int[]{1, 2, 4, 8, 16, 32, 64, 128};
```

```
oder: int[] twoPower = {1, 2, 4, 8, 16, 32, 64, 128};
```

Mehrdimensionale Felder

- Mehrdimensionale Felder sind tatsächlich Felder von Feldern

```
int[][] chessBoard = new int[8][8];
```

- Größe muss beim Allok. zumindest f. d. erste Dimension angeg. werden

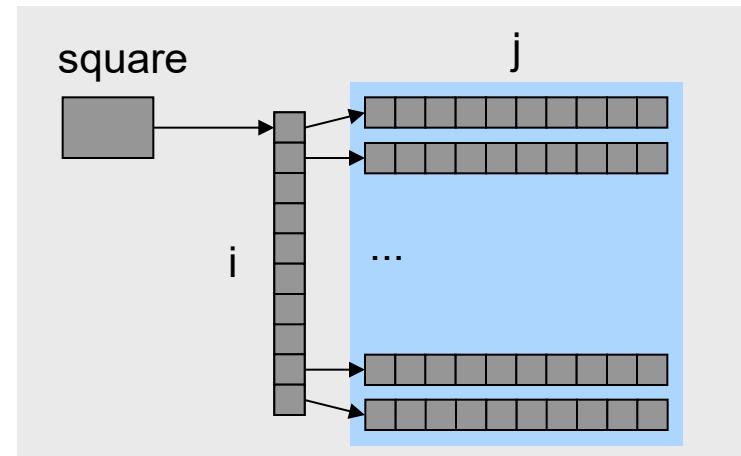
```
double[][][] cube = new double[10][][];
```

Exkurs: Ausgefranste Felder (*r*/jagged arrays)

Zwei typische Formen von mehrdim. Feldern (hier z. B. für zwei Dim.):

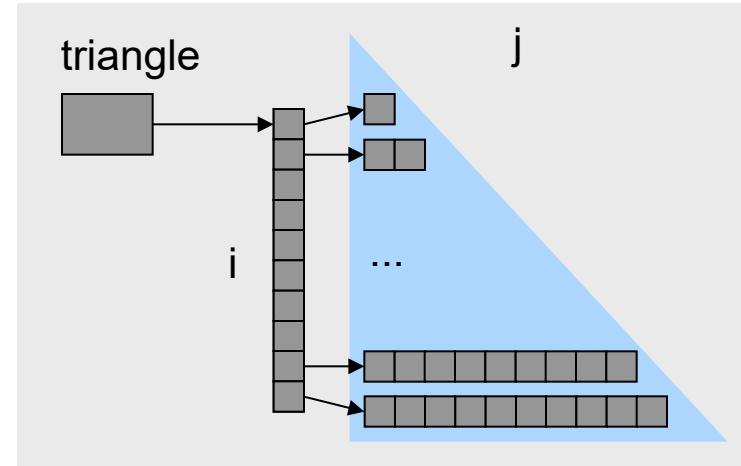
1. "Rechteckig" (*rectangular*), z. B.

```
double[][] square =  
    new double[10][10];  
square[i][j] = ...;
```



2. "Ausgefranst" (*ragged/jagged*), z. B.

```
double[][] triangle =  
    new double[10][];  
// triangle.length == 10, see next slide  
for (int i = 0; i <= 9; i++) {  
    triangle[i] =  
        new double[i + 1];  
    // triangle[i].length == i + 1  
} // for  
triangle[i][j] = ...;
```



Zugriff auf Feldelemente

- Elementzugriff nur mittels Indizierung [...] und ganzzahligem Ausdruck, immer (!) mit Laufzeitprüfung, z. B.:

```
int[] a = new int[8]; ...
a[i] = 0; // may throw ArrayIndexOutOfBoundsException
```

- Zugriff auf Länge eines Felds über konst. (!) Datenkomp. `length`, z. B.:

```
a[0] = 1; // == 2^0
for (int i = 1; i < a.length; i++)
    a[i] = a[i - 1] * 2; // now a[i] == 2^i
```

Datentyp eines Felds

- Datentyp eines Felds umfasst nur
 - Anzahl der Dimension(en) und
 - Elementdatentyp
- Länge des Felds (Anzahl der Elemente) ist nicht Bestandteil des Datentyps, z. B.:

```
char[] ca1, ca2; // variables of same type
ca1 = new char[1024];
ca2 = ca1;          // only reference is assigned
```

Felder sind spezielle Objekte

Ähnlichkeiten mit "echten" Objekten:

- Felder sind über Referenzen anzusprechen
- Zugriff auf Länge des Felds über konst. Datenkomponente *length*
- Methoden der Klasse *Object* auch für Felder anwendbar
- Zuweisung von Feld an *Object*-Variable möglich (umgek. mit *cast*)

Methoden *toString*, *hashCode* u. *equals* liefern aber unbefriedigende Ergebnisse, weil sie -- wie bei *Object* -- nur Adressen der Felder verwenden, nicht deren Inhalt

Beispiel:

```
int[] a1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
...println("a1.length      = " + a1.length      );  
...println("a1.toString() = " + a1.toString());  
...println("a1.hashCode() = " + a1.hashCode());  
  
int[] a2 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
...println("a1 == a2 : " + (a1 == a2));  
...println("a1.equals(a2) : " + (a1.equals(a2)));  
  
Object o = a1; // via polymorphism, see next slide  
a2 = (int[])o; // explicit cast necessary
```

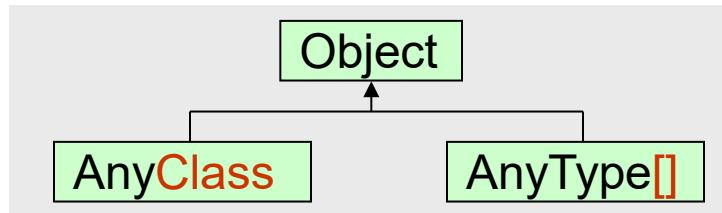
Ausgabe:

```
a1.length      = 10  
a1.toString() = [I@4517d9a3  
a1.hashCode() = 1159190947
```

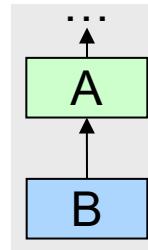
a1 == a2 : false
a1.equals(a2) : false

Exkurs 2: Mögliche Typprobleme mit Feldern

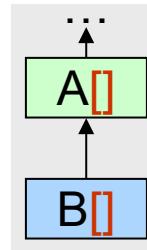
(Klassen|Typ)-Hierarchien



... und aus:

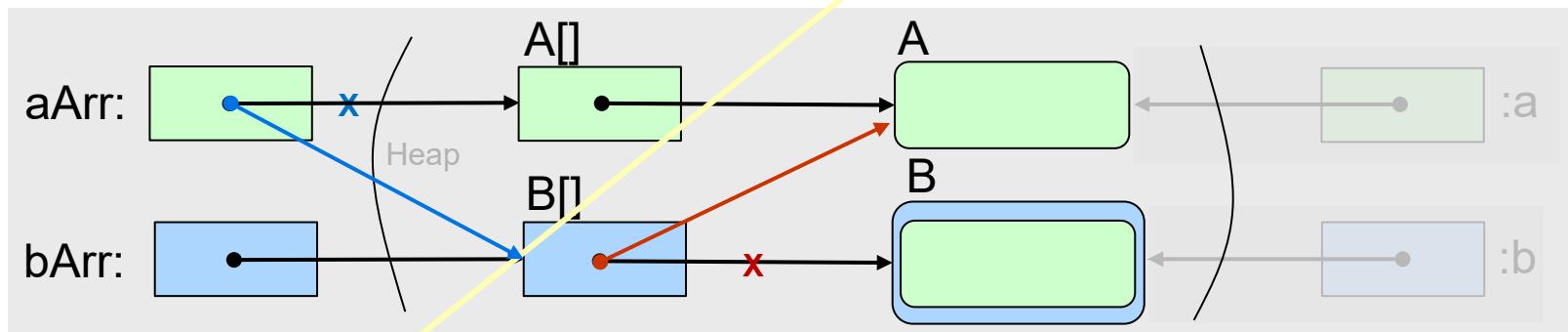


folgt:



Mögliche Problem dadurch, z. B.

```
A[] aArr = new A[1]; A a = new A(); aArr[0] = a; // ok
B[] bArr = new B[1]; B b = new B(); bArr[0] = b; // ok
aArr = bArr; // ok, aArr[i] is at least of type A
```



```
aArr[0] = a; // throws ArrayStoreException, as
// bArr[0] would reference an A obj.
```

Exkurs 2: Hilfsklasse Arrays für Felder

Hilfsklasse `java.util.Arrays` stellt seit Java 1.2 umfangreiche Funktionalität f. Felder in Form von stark überladenen Klassenmethoden zur Verfügung, meist für Felder von Standarddatentypen (unten nur für `int` gezeigt) und für `Object`-Felder

```
public class Arrays { // no public constructor => no Arrays objects
// working replacements for methods from class Object (using all elements)

    pub. stat. String toString(int [] a); // result is ...
    pub. stat. String toString(Obj. [] a); // ... "[e1, e2, ... en]"

    pub. stat. bool. equals (int [] a1, int [] a2); // use all elements,
    pub. stat. bool. equals (Obj. [] a1, Obj. [] a2); // additionally compare

    pub. stat. int hashCode(int [] a); // hash code computed ...
    pub. stat. int hashCode(Obj. [] a); // ... from all the elements

// new methods (some of the most important ones)

    pub. stat. void fill(int [] a, int val);
    pub. stat. void fill(Obj. [] a, Obj. val);

    pub. stat. int [] copyOf(int [] a, int newLength);
    pub. stat. Obj. [] copyOf(Obj. [] a, int newLength);

    pub. stat. void [parallel]sort(int [] a); // quick sort ] elems. must be
    pub. stat. void [parallel]sort(Obj. [] a); // merge sort ] comparable

    pub. stat. int binarySearch(int [] a, [int from, int to,] int key);
    pub. stat. int binarySearch(Obj. [] a, [int from, int to,] Obj. key);

} // Arrays
```

Beispiel:

```
int[] a = new int[10];
Arrays.fill(a, 17);
System.out.println(
    Arrays.toString(a));
```

Zeichenketten (Klasse *String*)

- Zeichenketten sind keine Felder von Einzelzeichen mehr, die durch Null-Zeichen ('\0') terminiert werden (wie in C)
- Alle Zeichenketten sind Objekte der Klasse *java.lang.String*
- *String*-Objekte sind **unveränderlich (*immutable*)**
es gibt keine Methoden zur Änderung der von einem *String*-Objekt repräsentierten Zeichenkette
- Auch für Zeichenkettenliterale ("...") werden vom Compiler automatisch *String*-Objekte erzeugt, z. B.:

```
String s1 = "..."; // it's an assignment!
String s2 = new String("...."); // no good idea,
                                cf. string pooling
```
- Wichtige Methoden: *equals* (aus *Object*), **length**, *charAt*, *compareTo*, *indexOf*, *lastIndexOf* und *concat*, aber z. B. kein *append*
- (Nur!) Operatoren **+** und **+=** sind für *String*-Objekte überladen, z. B.:

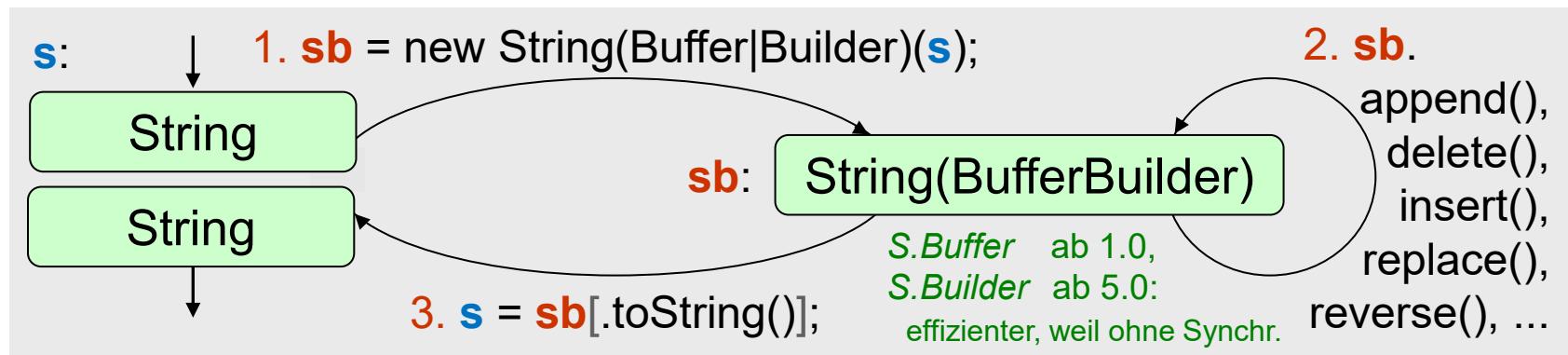
```
s = s.concat("..."); ⇔ s = s + "..."; ⇔ s += "...";
```
- Alle Methoden auch auf Zeichenkettenliterale anwendbar, z. B.:

```
System.out.println("Hello, world!".length()); // -> 13
```

Veränderbare Zeichenketten, Kl. *String(Buffer|Builder)*

Klassen *java.lang.String(Buffer|Builder)* bieten veränderbare (*mutable*) Zeichenketten, generelles Vorgehen:

1. Aus *String*-Objekt neues *String(Buffer|Builder)*-Objekt erzeugen
 2. Dieses *String(Buffer|Builder)*-Objekt beliebig (oft) manipulieren
 3. Aus *String(Buffer|Builder)*-Objekt neues *String*-Objekt erzeugen
- (Alle *String...*-Klassen implement. die Schnittstelle *java.lang.CharSequence*)



Beispiel

```
String s = ...;
Str.(Buffer|Builder) sb = new Str.(Buffer|Builder)(s);
sb.append("..."); sb.delete(i, j); sb.insert(k, "...");
s = sb[.toString](); // the "old" string is garbage now
```

Beispiel: Erzeugung einer Zeichenkette mit n Leerzeichen:

1. Klasse *String*

```
String s = new String();
for (int i = 0; i < n; i++)
    s = s.concat(" ");
// or: s = s + " " or: s += " "
// now: s.length() == n
```

2.+3. Klassen *String(Buffer|Builder)*

```
String s = null;
StringBuffer|Builder sb =
    new Str.(Buffer|Builder)();
for (int i = 0; i < n; i++)
    sb.append(" ");
s = sb.toString(); sb = null;
// now: s.length() == n
```

Laufzeiten in Millisekunden [ms]:

$n = 1.000 *$	mit ... Synchronisation ... ohne		
	1. <i>String</i>	2. <i>StringBuffer</i>	3. <i>StringBuilder</i>
1	4	0	0
10	35	2	0
100	1.802	5	3
1.000	293.527	29	15

= ca. FÜNF Minuten!

Ähnlichkeiten mit C

- Java verfügt über fast alle C-Operatoren (fehlende s. u.)
- Operatoren haben gleichen Vorrang u. gleiche Assoziativität wie in C

Unterschiede zu C

- Kein Komma-Operator (nicht mehr erlaubt z. B. `i = expr1, expr2, ...;`)
- Keine Dereferenzierungsoperatoren (notwendig), weder `*` noch `->`
- Kein Adress- (**&**) und kein Größenoperator (**sizeof**)
- Indizierung (z. B. `a[i]`) u. Selektion (z. B. `o.data`) sind syntaktisch keine Operatoren mehr

Neue Operatoren

- **+** neben Addition auch für Zeichenketten-Konkatenation
- **instanceof** Liefert `true`, wenn linker Operand (*Objekt*) vom Typ des rechten Operanden (*Klasse*) ist oder die Klasse des Objekts diese *Schnittstelle* implementiert
- **>>>** *shift-right* (ohne VZ), links werden 0-Bits eingefügt
- **& und |** Für *ganzzahlige* Werte *bitweises* und/oder
Für *boolesche* Werte *logisches* und/oder mit vollst. Auswertung
&& und **||** sind Kurzschlussoperatoren nur auf *boolean*

Ähnlichkeiten mit C

- *if-, else- , while- und do-while-Anweisungen* sind identisch mit C, allerdings müssen die Bedingungen vom Typ *boolean* sein
- *switch-Anweisung* kann mit Werten der Typen *byte, char, short, int* oder *long* als *case*-Marken arbeiten, auch *default*-Zweig ist möglich
- (Nur) *for-Schleife* erlaubt mehrere durch Komma getrennte Ausdrücke im Initialisierungs- und Veränderungsteil; wie in C++ ist Definition einer lokalen Schleifenvariable möglich

Unterschiede

- Keine *goto-Anweisung* mehr
- Aber Schlüsselwort *goto* ist weiterhin reserviert ;-)

dafür

Neue Anweisungen

- *break- und continue-Anweisungen* können mit Marken versehen werden (um bestimmte, markierte Anweisungen oder Schleifen zu verlassen oder zu wiederholen)
- *synchronized-Anweisung* regelt Zugriff mehrerer Threads auf gem. genutzte Ressourcen ("kritischer Bereich", *critical section*)
- *package- und import-Anweisungen* geben Zugehörigkeit zu einem Paket an bzw. die zu "importierende(n)" Klassen an

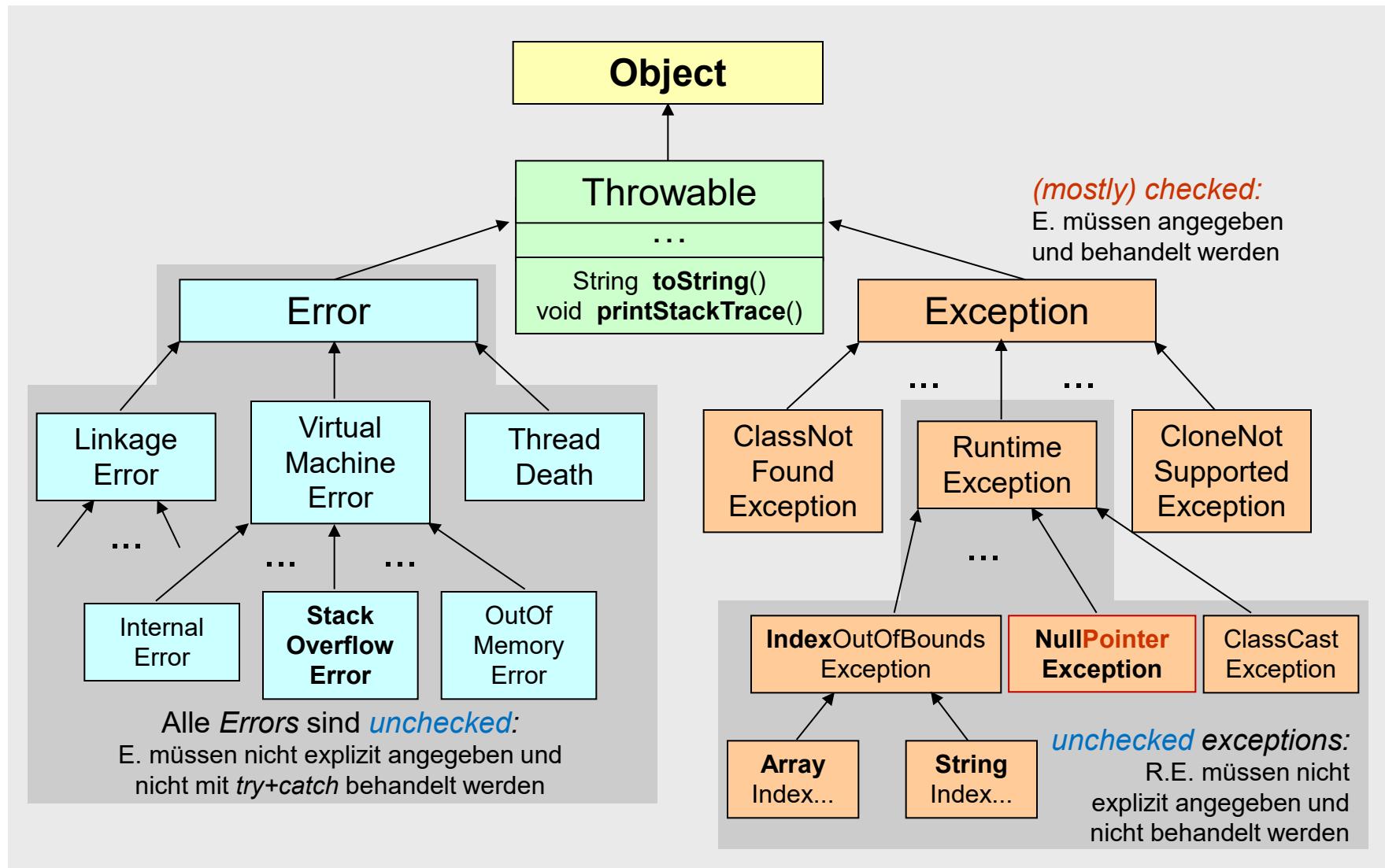
Ausnahmen und deren Behandlung

- Ausnahmen sind Objekte der Klasse `java.lang.Throwable` oder einer davon abgeleiteten Klasse (z. B. `Error` oder `Exception`, s.u.)
- `try/catch/finally`-Anweisung stellt den Rahmen für das Auslösen und die Behandlung von Ausnahmen dar:

```
try {  
    // code which might produce or throw exceptions  
} catch (SomeException e) {  
    // handle an exception of type SomeException  
} finally {  
    // code which is always executed, i.e. when  
    // 1) try block threw no exceptions  
    // 2) SomeException has been handled in catch  
    // 3) another ex. occurred in try or catch block  
} // finally
```

- Methoden müssen alle "normale" Ausnahmen (*checked exceptions*), die sie auslösen könnten, explizit angeben
 - ...**method(...)** **throws** SomeException { ... }
- Ausnahmesituationen können mit `throw`-Anweisung ausgelöst werden
 - throw** new SomeException(...);

Exkurs: Hierarchie der Ausnahmen-Klassen



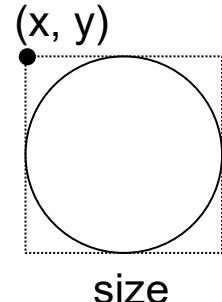
Weitere Unterschiede zu C (und C++)

- Lokale Variablen können (wie in C++) an beliebigen Stellen innerhalb eines Anweisungsblocks definiert werden (keine autom. Initialisierung)
- Vorwärtsreferenzen sind weder in C noch in C++ erlaubt, daher Unterscheidung zwischen Deklaration und Definition; in Java sind Vorwärtsreferenzen erlaubt (Verwendung von Datenkomp. und Aufruf von Methoden vor ihrer Definition)
- Überladen von Methoden ist (wie in C++) erlaubt, Überladen von Operatoren ist nicht möglich (nur + und += sind ...)
- `void` wird in Methodenschnittstellen beim Ergebnisdatentyp wie in C++ verwendet, es gibt aber keinen Datentyp `void*`
- Es gibt weder Strukturen (*struct*) -- Ersatz dafür sind Klassen (*class*), noch Vereinigungen (*union*) -- Ersatz dafür sind Klassen mit Vererbung
- Es gibt (bis 5.0) keine Aufzählungstypen (*enum*)
- Es gibt keine Funktions- oder Methodenzeiger (dafür Schnittstellen und seit V. 8 Lambdas, also anonyme Methoden)
- Es gibt keine Bit-Felder (aber die Klasse `java.util.BitSet`)
- Es gibt keine Typdefinitionen (*typedef*)
- Es gibt (bis V. 5.0) keine variablen Argumentlisten (mit ...)

Klassen und Objekte

- **Klassen:** Beschreibungen von Objekten (mit Datenkomponenten und Methoden), Klassen sind (Referenz-)Datentypen, z. B.:

```
[public] class Circle {  
    // data components:  
    public double x, y; // top/left of bound. box  
    public double size; // 2 * radius  
    // methods:  
    public double area() {  
        return 3.1415 * (size/2) * (size/2);  
    } // area  
} // Circle
```



- **Objekte** ("Instanzen von Klassen"): nur mit *new* angelegte, dynamische Objekte, die über Referenzen ansprechbar sind
- Zugriff auf Daten und Methoden von Objekten mit Punktnotation (keine explizite Dereferenzierung), z. B.:

```
circle c = new Circle();  
c.x = 3.0; c.y = 5.0; c.size = 10.0;  
double a = c.area();
```

- Referenz auf Empfängerobjekt ist wie in C++ impliziter Parameter *this*, z. B.:

```
public double area( /*final Circle this*/ ) {  
    return 3.14159 * (this.size/2) * (size/2);  
} // area
```

- Für Klassen ohne Konstruktor erzeugt der Java-Compiler einen öffentlichen, parameterlosen Standardkonstruktor zur Initialisierung
- Objekterzeugung mit *new X()* ruft nach Allok. Standardkonstruktor auf
- Wie Methoden können auch Konstruktoren überladen und bei Objekterzeugung mit *new X(any arguments)* selekt. aufger. werden

Beispiel

```
[public] class Circle {  
    // data components, now private  
    private double x, y, size; // init.ed with 0.0  
  
    // constructors  
    public Circle() {  
        x = 0.0; y = 0.0; size = 0.0; }  
    public Circle(double x, double y, double s) {  
        this.x = x; this.y = y; this.size = s; }  
    public Circle(double s) {  
        this(0.0, 0.0, s); }  
    public Circle(Circle c) { // copy constr.  
        this(c.x, c.y, c.size); }  
  
    // methods  
    public double area() {  
        return 3.1415 * (size/2) * (size/2); }  
} // Circle
```



Keine
Initialisierungslisten
wie in C++

Exkurs: Initialisierer für Objektdaten

Datenkomponenten werden meist in **Konstruktoren** initialisiert

```
[public] class A {  
    private int data; // default value 0 w.o. constructor  
    public A(...) {  
        data = 17; // overrides 0 with 17  
    } // A  
...}
```

Alternativen oder zusätzlich (weil **vor** jedem Konstruktor ausgeführt):

- **Initialisierer** für Datenkomponenten in der Vereinbarung (*initializer*)

```
[public] class A {  
    private int data = 17; // is an initializer  
...}
```

- **Objekt-Initialisierer** sind beliebige Code-Blöcke (*instance initializer*)

```
[public] class A {  
    private int data; // default value 0  
    { // this block is an instance initializer, too  
        data = 17; // and any other code, e.g. loops  
    }  
...}
```

Klassenkomponenten und "Konstanten"

- **Klassen-Datenkomponenten** und **-Methoden** sind "Eigenschaften der Klasse", nicht der jeweiligen Objekte
- Vereinbarung mittels Schlüsselwort `static`, z. B.:

```
public class Circle {  
    // class components: data and method  
    private static int noc = 0; // num. of circles  
    public static int numberof() {return noc;}  
    // object components: constructor  
    public Circle() { noc++; ... }  
} // circle
```

- Zugriff auf Klassenkomponenten durch Qualifikation mit Klassennamen, z. B. `Circle.noc` oder über Objekt der Klasse z. B. `c.noc`

```
System.out.println(Circle.numberof());
```

- **"Konstanten"** sind Klassendatenkomponenten, die mit dem Schlüsselwort `final` vereinbart also unveränderlich sind, z. B.:

```
public class Circle {  
    public static final double PI = 3.14159;  
} // circle
```

Exkurs: Initialisierer für Klassendaten

Klassen-Datenkomponenten werden meist mit Zuweisung initialisiert:

```
[public] class A {  
    private static int data = 17; // default value would be 0  
    public A(...) {  
        data = 99; // reinitialization allowed, but ...  
    } // A  
    ...
```

Werden z. B. Schleifen für die Initialisierung von Klassen-Datenkomponenten benötigt, muss dafür ein Klasseninitialisierer (*static initializer*) verwendet werden:

```
[public] class A {  
    private static int[] a; // default value would be null, but...  
    static { // ... this block is a static initializer  
        a = new int[10];  
        for (int i = 0; i < a.length; i++)  
            a[i] = ...;  
    } // static init.  
    ...
```

Objektfreigabe und "Finalisierung"

- Objekte können nach Erzeugung mit `new` nicht explizit freigegeben werden
- Automatische Speicherbereinigung (*garbage collection, GC*) sorgt für Freigabe von "nicht mehr benötigten" (nicht mehr referenzierbaren) Objekten
- Objekte, können aber müssen nicht vom GC freigegeben werden
- GC ist leichtgewichtiger Prozess (*thread*) mit niedriger Priorität
- GC arbeitet in *idle time*, unterbricht andere Threads nur bei Speicherknappheit

```
circle c = new Circle();
...
c = null; // now Circle object is "garbage"
```

1. expl. Aufruf:
`System.gc();`

- Java kennt keine Destruktoren, aber "Finalisierungsmethoden"

```
class Object {
    protected void finalize() throws Throwable;
} // Object
```

- Parameterlose *finalize*-Methode (*finalizer*) kann zur Bereinigung implementiert werden, wird vom GC aber höchstens einmal aufgerufen, wenn überhaupt

Beispiel: *finalize*-Methode der Klasse *FileOutputStream*

```
protected void finalize() throws IOException {
    if (fd != null) // file descriptor
        close();
    super.finalize(); // "finalizer chaining"
} // finalize
```

2. expl. Aufruf:
`System.runFinalization();`

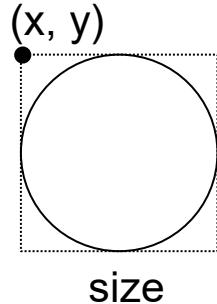
Abgeleitete Klassen und Vererbung

- Von genau einer Basisklasse kann mit dem Schlüsselwort `extends` eine neue Klasse abgeleitet werden
- Basisklasse vererbt alle Eigenschaften an abgeleitete Klasse
- Klasse `Object` ist Basisklasse aller Klassen ohne explizites `extends`
- Alle Java-Klassen zusammen bilden eine Baumhierarchie, Wurzel ist `Object`

Beispiel

```
public class Graphiccircle extends circle {  
    private boolean fill;  
    public void draw(java.awt.Graphics g) {  
        if (fill)  
            g.filloval(x, y, size, size);  
        else  
            g.drawoval(x, y, size, size);  
    } // draw  
} // Graphiccircle
```

Circle:



- Klassen, die mit dem Schlüsselwort `final` definiert werden, können nicht durch Vererbung erweitert werden (ermöglicht Optimierungen bei der Übersetzung), diese Klassen bilden die Blätter des Vererbungsbaums

Verkettung von Konstruktoren und Finalisierern

- Abgeleitete Klassen können eigene Konstruktoren definieren, z. B.:

```
public GraphicCircle(double x, double y, double size,  
                     boolean fill) {  
    this.x = x; this.y = y; this.size = size; // bad!  
    this.fill = fill;  
} // GraphicCircle
```

- Konstruktoren abgeleiteter Klassen können u. sollen mit Schlüsselwort `super` Konstruktoren der Basisklasse aufrufen, die Argumenten benötigen, z. B.:

```
public GraphicCircle(double x, double y, double size,  
                     boolean fill) {  
    super(x, y, size); // much better!  
    this.fill = fill;  
} // GraphicCircle
```

- Konstruktoren werden automatisch verkettet: fehlt `super`-Aufruf wird zu Beginn automatisch Standardkonstruktor der Basisklasse aufgerufen, Schema:

```
public class DerivedClass extends BaseClass {  
    public DerivedClass() {  
        // super(); automatic call of BaseClass' constr.  
        ... // rest of constructor for DerivedClass  
    } // DerivedClass  
} // DerivedClass
```

- **Achtung:** `finalize`-Methoden werden nicht automatisch verkettet, expliziter Aufruf mit `super.finalize()` notwendig

Überdeckung in abgeleiteten Klassen

- **Datenkomponenten** in abgeleiteten Klassen können gleichnamige Datenkomponenten in Basisklassen (auch anderen Typs) überdecken

```
class Circle /*extends Object*/ {  
    double r;      // radius = size/2  
} // Circle
```

```
class GraphicCircle extends Circle {  
    int   r;      // resolution in dots per inch  
} // Graphiccircle
```

- Zugriff auf überdeckte Namen mit dem Schlüsselwort *super* oder mit *this* und explizitem *type cast*

```
class GraphicCircle extends Circle {  
    ... anyMethod(...) {  
        this.r  = ...           // resolution  
        super.r = ...           // radius  
        ((Circle) this).r = ... // radius  
    } // method  
} // Graphiccircle
```

- **Methoden** von Basisklassen werden in abgeleiteten Klassen nicht überdeckt sondern bei gleicher Schnittstelle überschrieben, sonst überladen

Überschreiben von Methoden

- Abgeleitete Klassen können Methoden von Basisklassen überschreiben (wenn diese nicht *final* definiert sind)
- Überschreibende Methode muss **gleichen Namen, gleichen Ergebnistyp und gleiche Argumenttypen** wie Methode der Basisklasse haben (sonst Überladen, Überdecken gibt es nicht)
ab J. 5.0 sind wie in C++ auch kovariante (spezialis./eingeschr.) Ergebnistypen möglich

Beispiel

```
class A {  
    public char c = 'A';  
    public char m() {  
        return c;  
    } // m  
} // A
```

```
class B extends A {  
    public char c = 'B';  
    public char m() {  
        return c;  
    } // m  
} // B
```

```
public class Test {  
  
    public static void main(...) {  
  
        B b = new B();  
        System.out.println(b.c); // B  
        System.out.println(b.m()); // B  
  
        A a = b;  
        System.out.println(a.c); // A  
        System.out.println(a.m()); // B  
  
    } // main  
  
} // Test
```

Aufruf überschriebener Methoden

- Überschreiben ist nicht Überdecken
- Überdeckte Datenkomponenten sind z. B. mittels *cast* anzusprechen
- Überschriebene Methoden können mittels *super* oder *cast* aufgerufen werden

Beispiel 1

```
public class A {  
    public int i = 1;  
    public int m() { return i; }  
} // A  
  
public class B extends A {  
    public int i = 2;  
    public int m() { i = ((A)this).i + 1; // i = super.i + 1;  
                    return super.m() + i; }  
} // B
```

Beispiel 2

```
public class A /*extends Object*/ {  
    protected void finalize() { ...; super.finalize(); }  
} // A  
  
public class B extends A {  
    protected void finalize() { ...; super.finalize(); }  
} // B
```

Pakete (packages): Sammlungen von logisch zusammengehörigen Datentypen (Klassen und Schnittstellen); Pakete können hierarchisch strukturiert werden

Sichtbarkeit von Klassen

- Klassen, die **public** definiert sind, können auch außerhalb des Pakets benutzt werden
- Klassen, die **nicht public** definiert sind, sind nur innerhalb ihres Pakets sichtbar

Sichtbarkeit von Komponenten

- Sichtbarkeit von Komponenten (Daten und Methoden) wird durch eines der drei Schlüsselwörter **public**, **protected** oder **private** definiert
- Ohne Angabe eines Sichtbarkeitsattributs wird der Sichtbarkeitsbereich Paket (**package visibility**) definiert, das Schlüsselwort **package** darf dafür allerdings nicht verwendet werden

Komponenten sind zugreifbar in wenn für Komponente gilt:			
	<i>public</i>	<i>protected</i>	(<i>package</i>)	<i>private</i>
... selber Klasse	ja	ja	ja	ja
... anderer Klasse in gleichem Paket	ja	ja	ja	nein
... abgeleit. Klasse in anderem Paket	ja	ja	nein	nein
... anderer Klasse in anderem Paket	ja	nein	nein	nein

Anwendung (*application*), Applet (im Web Browser) oder Servlet (im Web Server)

Paket

(Basis-)Klasse

`private`

`protected`

`(package)`

`public`

Andere Klasse im gl. P.

Anderes

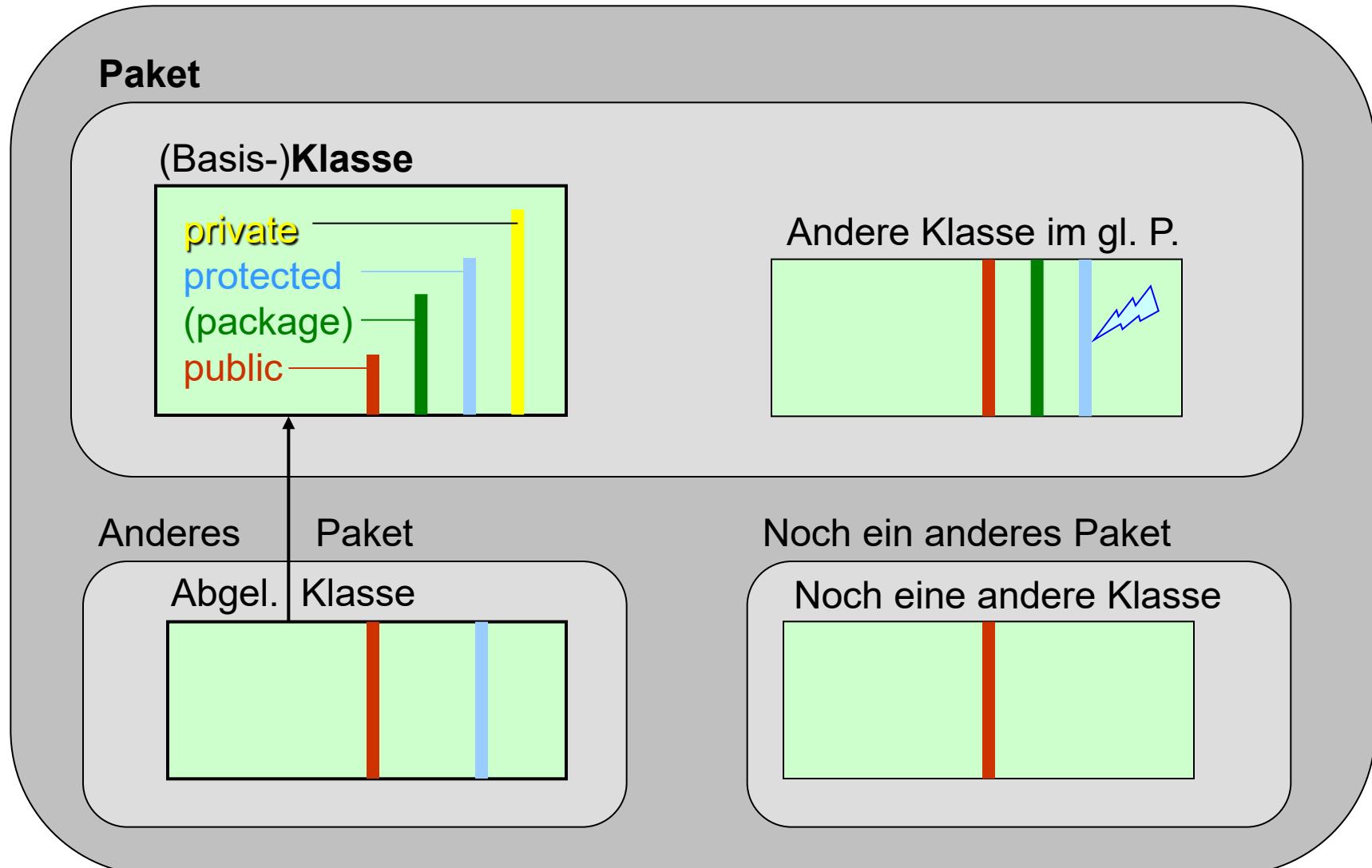
Paket

Abgel.

Klasse

Noch ein anderes Paket

Noch eine andere Klasse



- **Abstrakte Methoden** (mit Schlüsselwort *abstract* versehen) definieren nur Methoden-Schnittstellen, haben keine Implementierung, z. B.:

```
public class Shape { // implicitly abstract
    public abstract void draw(Graphics g);
} // Shape
```

- Klassen, die abstrakte Methoden haben, sind **abstrakte Klassen** (es können keine Objekte davon angelegt werden)
- Auch Klassen können mit dem Schlüsselwort *abstract* explizit als abstrakte Klassen markiert werden

```
public abstract class Shape { // explicit
    public abstract void draw(Graphics g);
} // Shape
```

- Sogar Klassen, die keine abstrakten Methoden enthalten, können als abstrakt definiert werden (um zu verhindern, dass Objekte davon erzeugt werden), z. B.:

```
public abstract class Shape {
    public void draw(Graphics g) { ... }
} // Shape
```

Schnittstellen (*interfaces*)

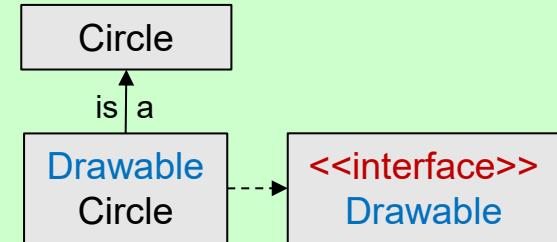
<https://docs.oracle.com/javase/8/docs/api/index.html>

- Schnittstellen sind vergleichbar mit Klassen, die nur abstrakte Methoden und/oder Konstanten definieren, z. B.:

```
interface Drawable {           // like an abstr. class
    public void draw(Graphics g); // like an abstr. meth.
} // Drawable
```

- Klassen erben von genau einer Basisklasse und können beliebig viele Schnittstellen implementieren, z. B.:

```
class DrawableCircle
    extends Circle
    implements Drawable {
    double x, y, size;
    public void draw(Graphics g) {...}
} // Drawablecircle
```



- Schnittstellen sind (Referenz-)Datentypen, z. B.:

```
Drawable d = new DrawableCircle(); // new needs a class
d.draw(g); // no other methods available
```

- Schnittstellen können auch mehrere (!) Basisschnittstellen haben, Eigenschaften der Basis werden vererbt, neue Methoden und Konstanten können in der abgeleiteten Schnittstelle hinzugefügt
- Leere Schnittstellen (*marker interfaces*) können zur Markierung von Klassen verwendet werden (z. B. `java.io.Serializable`)

Generell fehlen in Java jene Eigenschaften von C++, welche die Sprache signifikant komplizierter machen würden

Beispiele

- Keine statischen Objekte
(Java kennt nur dynamische, über Referenzen manipulierbare Objekte)
- Keine "Wertsemantik" bei der Objektmanipulation, z. B. Zuweisung
(Java kennt nur "Referenzsemantik" für Objekte, Objekte müssen bei Bedarf explizit mit *clone* kopiert werden)
- Keine mehrfache Vererbung bei Klassen
(Java-Klassen können **nur von einer Basisklasse erben** aber sie können **beliebig viele Schnittstellen implementieren**)
- Keine Überladung von Operatoren
- Keine Konvertierungsmethoden, die Objekte automatisch in andere Datentypen konvertieren
- Keine "Befreundung" von Methoden und Klassen
(Java kennt dafür Pakete, die "befreundete" Klassen enthalten)
- **Bis Java 5.0:** Keine Generizität
(bis dahin nur Polymorphismus)

Allgemeines

- Java 1.1 war "großes Release"
- Wichtige Spracherweiterungen, vor allem geschachtelte also "innere" Klassen (*inner classes*), s. u.
- Erhebliche Effizienzsteigerungen bei der virtuellen Maschine
- Viele Änderungen u. Erweiterungen in der Klassenbibliothek (führen z.T. zu *deprecated features*, z. B. Methode *System.getenv* mit der vorgeschlagenen Alternative *System.getProperty* und der Möglichkeit, dafür mit *java -Dname=value ...* weitere Eigenschaften beim Programmstart zu definieren):

	Java 1.0	Java 1.1
Anzahl der Pakete	8	23
Anzahl der Klassen	212	504

- Neue Werkzeuge im JDK, z. B. *jar* und *javakey*

Neue Standardpakete

- JavaBeans: *java.beans.**
- Java Database Connectivity (JDBC): *java.sql.**
- Remote Methode Invocation (RMI): *java.rmi.**
- Security: *java.security.**

Geschachtelte (innere) Klassen

Vereinbarung von Klassen *innerhalb von Klassen* (also als Komponenten) und sogar *innerhalb von Anweisungsblöcken* möglich

Vier Arten

1. Geschachtelte Klasse auf oberster Ebene oder Schnittstelle (*top-level nested class or interface*)

- Klasse ist **static**-Komponente einer äußeren Klasse (oberster Ebene)
- Geschachtelte Schnittstelle ist automatisch **static**
- Verhalten wie "normale" Klasse oder Schnittstelle

!

2. Innere Klasse (*inner class*)

- Klasse ist (**non-static**-)Komponente einer anderen (äußeren) Klasse
- Objekte einer solchen Komponenten-Klasse haben automatisch eine Referenz auf Objekt der umgebenden Klasse
- Methoden der inneren Klasse können Komponenten des referenzierten Objekts der umgebenden Klassen verwenden

3. Lokale Klasse (*local class*)

- Klasse, die innerhalb einer Anweisungsfolge (**block**) vereinbart ist
- Hauptverwendung als Adapterklassen in AWT 1.1, besser aber:

4. Anonyme Klasse (*anonymous class*)

- Klasse, die sogar innerhalb eines Ausdrucks (**expression**) vereinbart ist
- Gleichzeitig Vereinbarung der Klasse und Erzeugung d. einzigen Objekts

!

Exkurs: Geschachtelte Klassen und *class*-Dateien

Quelltext *OuterClass.java*:

```
public class OuterClass {  
    // 1: top-level nested class or interface  
    static class TopLevelNestedClass {  
    } // TopLevelNestedClass  
  
    /*[static]*/  
    interface TopLevelNestedInterface {  
    } // TopLevelNestedInterface  
  
    // 2: non-static inner class  
    class NonStaticInnerClass {  
    } // NonStaticInnerClass  
  
    public static void main(String[] argv) {  
        // 3: local class  
        class LocalClass {  
        } // LocalClass  
  
        // 4: anonymous class  
        Obj. o = new BaseClassOrInterface() {  
            }; // anonymous class  
    } // main  
} // OuterClass
```

javac OuterClass.java erzeugt:

⇒ OuterClass.class

⇒ OuterClass\$TopLevelNestedClass.class

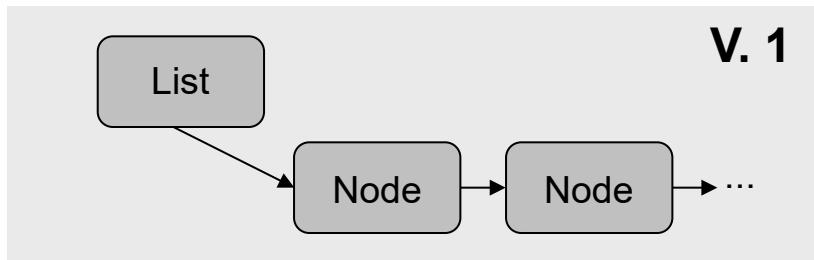
⇒ OuterClass\$TopLevelNestedInterface.class

⇒ OuterClass\$NonStaticInnerClass.class

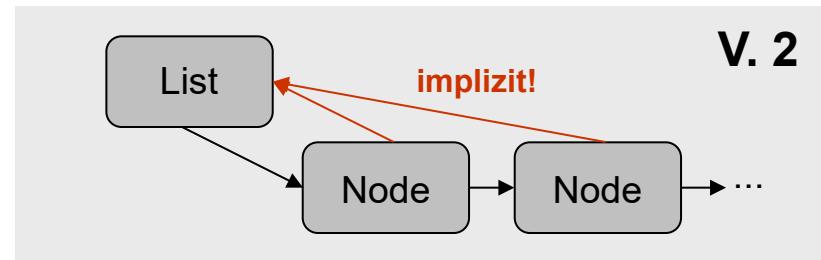
⇒ OuterClass\$1LocalClass.class

⇒ OuterClass\$1.class

Exkurs: Listen-Beispiel zu geschachtelten Klassen



```
public class List {  
    private static class Node {  
        Node next = null;  
        Object val;  
        Node(Object val) {  
            this.val = val;  
        } // Node  
    } Node  
    priv. Node head = null;  
    pub. v. prepend(Object o) {  
        Node n = new Node(o);  
        n.next = head;  
        head = n;  
    } // prepend  
} // List
```



```
public class List {  
    private /**/ class Node {  
        Node next = null;  
        Object val;  
        Node(Object val) {  
            this.val = val;  
            next = head; // of List  
            head = this; // Node  
        } // Node  
    } Node  
    priv. Node head = null;  
    pub. v. prepend(Object o) {  
        Node n = new Node(o);  
        // new node n has a  
        // reference to this  
    } // prepend  
} // List
```

Sonstige Spracherweiterungen

- Lokale Variablen, Parameter von Methoden und *catch*-Anweisungen dürfen *final* definiert werden: nachdem sie einen Wert bekommen haben, können sie nicht mehr geändert werden
- *Blank finals*: *final*-Größen (Datenkomponenten und Variablen) müssen nicht mehr bei Definition initialisiert werden, jedoch kann nur einmal ein Wert zugewiesen werden (i.d.R. im Konstruktor)
- Anonyme Felder: *new type[] {...}* erzeugt anonymes Feld, z. B.:

```
int[] a = {1, 2, 3, 4, 5}; // named array
int total = sum(a); // one and only usage of a
int total = sum(new int[] {1, 2, 3, 4, 5});
System.out.println(new char[] {'o', 'k'});
```

- Class-Objekte auch für alle Standarddatentypen (z. B. *int*, inkl. *void*)
- Vereinfachte Schreibweise zur Ermittlung des Class-Objekts (auch für Standarddatentypen) mit *AnyType.class*, z. B.:

```
Class c = Boolean.TYPE; // since 1.1 also: c = boolean.class;
Class c = Class.forName("java.util.Vector");
Class c = void.TYPE; // valid although void is "abnormal" type
Class c = Vector.class;
Object o = c.newInstance(); // needs def. constructor
```

Exkurs: Alle *final*-en Möglichkeiten

```
/*public*/ final class Finalclass { // no subclasses allowed

    /*public*/ static final int /*GLOBAL_*/CONSTANT = 100;

    [static] void m1() {
        final int LOCAL_CONSTANT = 10;
        final Object BLANK_FINAL; // no initialization yet
        BLANK_FINAL = new AnyClass(); // the one and only assign.
        // no further assignments to BLANK_FINAL allowed ...
    } // m1      ... but the referenced object may be modified

    [static] void m2(final int inParam) {
        int v;
        v = inParam;
        // no assignment to inParam allowed
    } // m2

    final void finalMethod() { // no overriding ...
        ...
        // ... in derived class allowed
    }; // finalMethod

} // Finalclass
```

Allgemeines

- Java 1.2 war wieder "großes Release" (auch "Java 2 Platform" kurz J2)
- Keine Sprachänderungen oder -erweiterungen (nur Beseitigung von Fehlern im Compiler)
- Viele Änderungen (führen auch zu weiteren *deprecated features*) und vor allem sehr viele Erweiterungen in der Klassenbibliothek:

	Java 1.1	Java 1.2
Anzahl der Pakete	23	59
Anzahl der Klassen	504	1520

- Neue Werkzeuge im JDK von Sun, z. B. *extcheck*, *jarsigner*, *keytool* und *policytool*

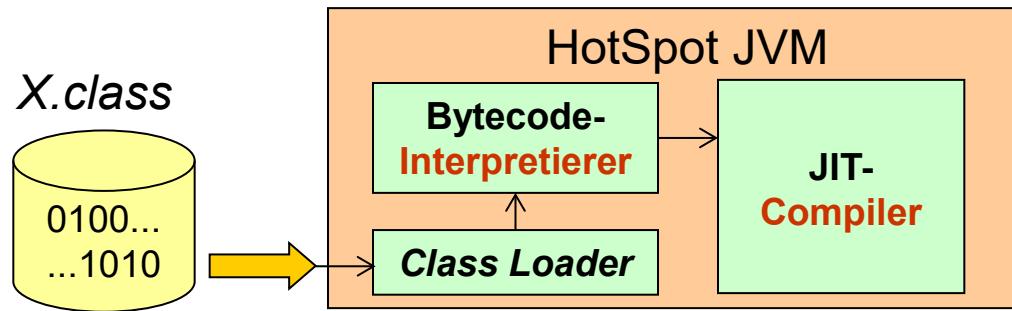
Wichtige neue Klassen und Pakete

- Klasse *Arrays* und **Java Collection Framework** (JCF), massive Erweiterungen in *java.util* mit gutem oo Design
- *Java Foundation Classes*: *Swing* und *Java 2D*
- *Java IDL* und CORBA mit *IOP* für technologisch heterogene Systeme

Mehr dazu in
eigenem JCF-
Foliensatz

Überblick: Neuigkeiten in Java 1.3 (2000)

- Keine Spracherweiterungen
- Kleinere Erweiterungen der Bibliothek
- Optimierung in der neuen virtuellen Java-Maschine (JVM) *HotSpot* durch **Just-in-time-Compiler (JIT-Comp.)**: häufig ausführter Bytecode wird zur Laufzeit in optimierten Maschinencode übersetzt (Optimierungen: *Inlining*, *dead code elimination*, *loop unrolling*, ...)



Bei *HotSpot* dienen zwei Schwellenwerte als Kriterium zur Aktivierung des JIT-Compilers (auf Methodenebene):

1. Anzahl der Aufrufe einer Methode (z. B. 1500 bei Client VM)
2. Anzahl der Schleifendurchläufe in einer Methode,
zur Erkennung selten aufgerufener aber wegen Schleife(n) lang laufender Methoden

Exkurs: Dem JIT-Compiler auf die Finger schau'n

```
pub. class NQueens { // s = nr. of solutions  
pub. stat. int n = 14, s = 0;  
pub. stat. int[] col = new int[n + 1];  
  
pub. stat. bool qFits(int i, int j) {  
    for (int k = 1; k < i; k++)  
        if ((col[k] == j) ||  
            (k + col[k] == i + j) ||  
            (k - col[k] == i - j))  
            return false;  
    return true;  
} // qFits  
  
pub. stat. void placeQ(int i) {  
    for (int j = 1; j <= n; j++)  
        if (qFits(i, j)) {  
            col[i] = j;  
            if (i == n)  
                s++; // impl. details see below  
            else  
                placeQ(i + 1);  
            col[i] = 0;  
        } // if  
} // placeQ  
  
pub. stat. void main(String[] args) {  
    placeQ(1);  
    System.out.println(s);  
} // main  
  
} // NQueens
```

```
javac NQueens.java  
java -XX:+PrintCompilation NQueens
```

```
... j.l.String::hashCode (55 bytes)  
... j.l.String::equals (81 bytes)  
... j.l.String::indexOf (70 bytes)  
... j.l.Object::<init> (1 bytes)  
... j.l.String::charAt (29 bytes)  
... some other methods ...  
... j.l.String::toLowerCase (439 bytes)  
... j.l.String::lastIndexOf (52 bytes)  
... j.l.Char::toLowerCase (9 bytes)  
... j.l.Char.DataLatin1::toLowerCase (39 bytes)  
... j.l.Abstr.Str.Builder::ens.Cap.Int. (27 b.)  
... j.l.Abstr.Str.Builder::append (29 bytes)  
... NQueens::qFits (52 bytes)  
... NQueens::placeQ (60 bytes)
```

Vom JIT-Compiler gen. **Maschinencode** kann **disassembliert** werden, notwendig dafür unter Windows für **Java 8** mit 32 Bit:

hsdis-i386.dll in C:\Program Files (x86)\Java\jre1.8...\bin\client
bzw. für **Java 1x** mit 64 Bit:

hsdis-amd64.dll in C:\(Open)Jdk-1x\bin\server

```
java -XX:+UnlockDiagnosticVMOptions  
      -XX:+PrintAssembly -xx:PrintAssemblyOptions=intel  
NQueens
```

Für Erläuterungen und *hsdis-xxx.dll* siehe z. B.

<https://blogs.oracle.com/javamagazine/post/java-hotspot-hsdis-disassembler>

Exkurs: Vom Compiler und JIT-Compiler gen. Code

Quelltext

```
public class NQueens {  
    public static void placeQ(int i) {  
        ...  
        s++;  
        ...  
    } // placeQ  
} // NQueens
```

```
javac NQueens.java  
→ javap -c NQueens
```

```
pub. stat. v. placeQ (int);  
...  
30: getstatic #5 // Field s:I  
33: iconst_1      // iinc for  
34: iadd          // locals only  
35: putstatic #5 // Field s:I  
...
```

```
java -XX:+UnlockDiagnosticVMOptions  
-XX:+PrintAssembly NQueens
```

Vom JIT-Compiler generierter Assembler-Code (f. Intel)

```
...  
# {method} {0x1503030c} 'placeQ' '(I)V' in 'NQueens'  
# parm0:    ecx      = int  
#           [sp+0x30]  (sp of caller)  
  
0x02cc7b9e: mov    edi, 4e630e8h          ; addr. of class  
0x02cc7ba3: mov    ebx, dword ptr [edi+60h] ; *getstatic s  
0x02cc7ba6: inc    ebx                   ; s++  
0x02cc7ba7: mov    dword ptr [edi+60h], ebx ; *putstatic s  
...
```

Spracherweiterung: zwei Formen der **assert**-Anweisung

- Einfache Form: **assert booleanExpr;**
Semantik: Wenn *booleanExpr == false* wird Ausnahme vom Typ *AssertionError* ausgelöst (ohne nähere Erläuterung)
- Erweiterte Form: **assert booleanExpr: msgStr;**
Semantik: Wenn *booleanExpr == false* wird Ausnahme vom Typ *AssertionError* ausgelöst **und** *msgStr* liefert Erläuterung dazu

Ursprünglich war das Zulassen von *assert*-Anweisungen noch zur Übersetzungszeit notwendig:

```
javac -source 1.4 x.java
```

Ein-/Ausschalten der Prüfung v. Assertionen aber zur Ausführungszeit:

```
java -enableassertions x od. java -ea x  
java -disableassertions x od. java -da x
```

Standardmäßig (ohne Option) werden Assertionen nicht geprüft (!)

Erweiterung der Bibliothek um Möglichkeiten zur Bildung und Verwendung regulärer Ausdrücke: Paket *java.util.regex* mit den beiden Klassen *Pattern* und *Matcher*

Überblick: Neuigkeiten in Java 5.0 (2005)



In Anlehnung an Microsoft .NET und C# **viele Erweiterungen** in der Programmiersprache und der Bibliothek

Überblick: Spracherweiterungen

- Neue *for*-Schleife
- Automatisches ver- u. entpacken v. Werten (*auto boxing/unboxing*)
- Aufzählungstypen
- Einfacher Import von Klassenkomponenten, "statische Importe"
- Variable Argumentlisten
- Generizität (für Klassen, Schnittstellen und Methoden)
- Anmerkungen oder Annotationen (*annotations*)

Überblick: Erweiterungen der Bibliothek

- Erweiterung des *Java Collection Frameworks* um neue Algorithmen
- Neue Klassen, z. B. *StringBuilder*, *Formatter* und *Scanner*
- Erweiterung der Metainformation für Generizität

Kuriosität am Rande:

"The *System.getenv* method is *no longer deprecated*."

Java 5.0: Neue *for*-Schleife(n)

Umsetzung der *foreach*-Schleife aus C# für Java (vgl. r.-b. *for* ab C++11):

1. Schleife über alle Elemente eines **Felds**

int[] a; // any array	
// up to now: for (int i = 0; i < a.length; i++) { // do anything with a[i] } // for	// since 5.0: for (int e: a) { // use e, it's a copy } // for

2. Schleife über alle Elemente eines **Behälters** mittels *Iterator* (Behälterklasse muss Schnittstelle *Iterable* implementieren)

Iterable c = new ... ; // any collection, e.g., Vector	
// up to now: Iterator it = c.iterator() while (it.hasNext()) { Object o = it.next(); // do anything with the object // referenced by o } // while	// since 5.0: for (Object o: c) { // do anything with // the object // referenced by o } // for

Java 5.0: Auto Boxing/Unboxing

- Wrapper-Klassen für element. Typen (z. B. *Character* für *char*, *Integer* für *int*)
- Aufgabe: *Werte* in *Objekte* ver- (*boxing*) und wieder auspacken (*unboxing*)

Explizites Ver- und Auspacken (von Werten in Objekte) seit Java 1.0, z. B.:

```
Integer io = new Integer(17); // meanwhile deprecated ...
int      iv = io.intValue(); // ... use Integer.valueOf instead
```

Java 5.0 bietet **automatisches** Ver- und Entpacken, z. B.:

```
Integer io = 17; // boxing via Integer.valueOf using preconstr. objs.
int      iv = io; // unboxing via io.intValue()
```

Beispiel: *boxing* und *unboxing* in Kombination mit neuer *for*-Schleife

```
Vector v = new Vector(); // container for any type of objects
v.add(17); // auto boxing
for (Object o: v) {
    int i = (Integer)o; // or: (int)o, but cast is necessary
    // ... now do anything with i
} // for
```

Achtung: Vergleiche immer mit Methoden (z. B. *equals*), weil hier kein ...

```
Object o1 = ..., o2 = ...;
if (o1 == o2) ... // compares references, no unboxing!!!
if (o1.equals(o2)) ... // compares data within objects, if ...
```

checks for identity not equality

Java 5.0: Aufzählungstypen (1)

Bisher: Einfache, aber unsichere Nachbildung mittels `int`-Konstanten, z. B.:

```
class TrafficLight {  
    public static final int RED      = 0;  
    public static final int YELLOW   = 1;  
    public static final int GREEN    = 2;  
} // TrafficLight
```

```
int t1;  
t1 = TrafficLight.RED;  
t1 = 17; // color???
```

Verbesserung mittels speziellem Datentyp (genannt *enum pattern*), z. B.:

```
class TrafficLight {  
    public static final TrafficLight RED = new TrafficLight(0);  
    ... // same for YELLOW and GREEN  
    public final int value; // blank final, set in constructor  
    private TrafficLight(int value) {  
        this.value = value; // from now on this.value is fixed  
    } // TrafficLight  
    public String toString() {  
        return "" + value; // or switch (value) ...  
    } // toString  
} // TrafficLight
```

```
TrafficLight t1;  
t1 = TrafficLight.RED;
```

Probleme: Viel Schreibaufwand für die Klasse und trotzdem z. B. keine Verwendung in `switch`-Anweisung möglich

Java 5.0: Aufzählungstypen (2)

Java 5.0 bietet "echte" Aufzählungsdatentypen, z. B.:

```
enum TrafficLight {  
    RED, YELLOW, GREEN;  
} // TrafficLight
```

} Compiler erzeugt daraus aber Bytecode wie für:
`class TrafficLight extends Enum<TrafficLight>
 implements Comparable<..> {...}`

Aufzählungskonstanten müssen mit Klassennamen qualifiziert werden, z. B.

```
TrafficLight t1 = TrafficLight.GREEN;
```

Verwendung in *switch*-Anweisung möglich, z. B.:

```
switch (t1) {  
    case GREEN: ... // no qualific. necessary in this context
```

Methode *toString* liefert Name der Konstante, z. B.:

```
System.out.println(t1.toString()); // => e.g., GREEN
```

Methode *values* liefert Feld mit allen Werten, z. B.:

```
TrafficLight[] tls = TrafficLight.values();  
for (TrafficLight t1: tls) ...
```

Generierte *enum*-Klasse implementiert *Comparable*

```
if (TrafficLight.RED.compareTo(TrafficLight.GREEN) == ...
```

Weitere Möglichkeiten: Synonyme (als Ergebnis von *toString*) und weitere Methoden möglich, die aber selbst zu implementieren sind

Java 5.0: Statische Importe, variable Argumentlisten

Statische Importe

Bisher umständliche Qualifikation bei Klassenkomponenten notwendig, z. B.:

```
// import java.lang.*; implicitly, includes System & Math  
System.out.println(Math.PI);
```

Für Klassenkomponenten kann Qualifikation vermieden werden (auch mit *):

```
import static java.lang.Math.PI;  
import static java.lang.System.*; // e.g., for out  
out.println(PI);
```

Variable Argumentlisten

Ermöglicht beliebig viele Parameter für Methoden, vgl. `<stdarg.h>` aus C, schon in der neuen Version der Standardbibliothek verwendet, z. B.:

```
System.out.printf("%3$d = %1$d + %2$d\n", 17, 4, 21);  
// 1 2 3
```

Auch für eigene Methoden nutzbar, z. B.:

```
static int sum(int... a) { // same as int[] a  
    int s = 0;  
    for (int i = 0; i < a.length; i++) // for (int e: a)  
        s += a[i]; // s += e;  
    return s;  
} // sum  
int result = sum(1, 2, 3); // same as ...new int[]{1, 2, 3})
```

Java 5.0: Generizität (1)

Generische Klassen (und Schnittstellen) Compiler: $T \rightarrow \text{Object}$
= type erasure

Definition einer **generischen Klasse**, z. B.:

```
class Wrapper<T> { // T is a ref. type var.  
    private T data;  
    Wrapper(T data) { this.data = data; }  
    void set(T data) { this.data = data; }  
    T get() { return data; }  
} // Wrapper
```

Roher Datentyp (raw type):

```
class Wrapper/*<Object>*/ {  
    private Object data;  
    Wrapper(Object data)...  
    void set(Object data)...  
    Object get()...  
} // Wrapper
```

Instantiierung (nur mit Referenzdatentypen, also auch mit Schnittstellen und Feldern), z. B.:

```
wrapper<String> w = new wrapper<String>(); // short: new wrapper<>();
```

Weitere Möglichkeiten

wird übernommen

- Mehrere Typvariablen, z. B.:

```
class C<T1, T2, ... > { ... }
```

- Einschränkungen (sodass nur bestimmte Typen bei der Instantiierung erlaubt sind), z. B.:

```
class C<T extends BaseClassOrInterface> { ... // but no ...<T implements Int.>
```

- Mehrfache Einschränkungen (mit Typen und Schnittstellen), z. B.:

```
class C<T extends BaseClass & Int1 & Int2 & ... > { ... }
```

- Platzhalter (? , unbound wildcard) für beliebige Typen bei der Instantiierung, z. B.:

```
Arrayof<?> any; // not allowed here:  
any = new Arrayof<String>();  
any = new Arrayof<Integer>(); // new Arrayof<?>()
```

Exkurs: Vgl. polymorpher mit generischen Behältern

Polymorphe Behälter

```
@SuppressWarnings("unchecked") // @ see below
Collection<Object> c =
    new Vector<Object>();
c.add(new String(...));
c.add(new Integer(...));
c.add(new Person(...));
```

```
// either with Iterator:
Iterator it =
    c.iterator();
while (it.hasNext()) {
    Object o = it.next();
    System.out.println(o);
} // while
```

```
// or with foreach:
for (Object o: c) {
    System.out.println(o);
} // for
```

Generische Behälter (ab J. 5.0)

```
Collection<String> sc =
    new Vector<String>();
sc.add(new String(...));
sc.add(new String(...));
sc.add(new String(...));
```

```
// either with Iterator:
Iterator<String> it =
    sc.iterator();
while (it.hasNext()) {
    String s = it.next();
    System.out.println(s);
} // while
```

```
// or with foreach:
for (String s: sc) {
    System.out.println(s);
} // for
```

Generische Methoden

Definition einer generischen Methode , z. B.:

```
class Any {  
    public static <T> void print(T t) {  
        System.out.println(t.toString());  
    } // // print  
} // Any
```

Verwendung (nur implizite Instantiierung und nur mit Objekten/Feldern):

```
Any.print("..."); // T = String  
Any.print(17); // => Any.print(new Integer(17));
```

Weitere Möglichkeiten (wie bei generischen Klassen)

- Mehrere Typvariablen
- Einschränkungen (nur bestimmte Klassen bei der Verwendung möglich)
- Mehrfache Einschränkungen
- Platzhalter (*wildcards*) für Typen für die Verwendung

Änderungen an der Metainformation (aber nicht am Bytecode)

- Klasse `java.lang.Class` ist nun generisch: `Class<T>`
- Neue Schnittstellen-Typen, wie z. B. `Type` und `TypeVariable`

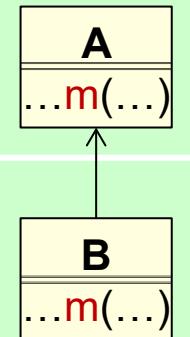
Java 5.0: Annotationen (1)

- "Umsetzung" des Konzepts der Attribute aus .NET für Java
- Vordefinierte und benutzerdefinierte (!) Annotationen (neue spez. Datentypen)

Beispiele (vordefinierte Annotationen aus dem Paket *java.lang*)

1. "Sicherheitsnetz" beim Überschreiben von Methoden

```
class A {  
    ... m(...) { ... }  
} // A  
class B extends A [implements ...] {  
    @Override // yields a compilation error, when ...  
    ... m(...) { ... }  
} // B
```



2. Unterdrückung von Warnungen

```
@SuppressWarnings("unchecked") // or "deprecation" or both  
class MySimpleExperimentalClass { ... // or "unused" or ...}
```

3. Markierung von nicht mehr zu verwendenden Methoden

```
@Deprecated // better use MyFineNewClass instead of ...  
class MyGoodOldClass { ...}
```

Java 5.0: Annotationen (2)

Beispiel: Definition benutzerdefinierter Annotation, z. B. *Test* für Testzwecke

```
import java.lang.annotation.*; // for Retention and Target:  
@Retention(RetentionPolicy.RUNTIME) // predefined annotations ...  
@Target(ElementType.METHOD)      // ... with parameters  
public @interface Test { // user defined annotation  
} // Test
```

... und deren Verwendung, z. B.:

```
public class Any {  
    public static void m1() {...}  
    @Test public static void m2() {...} // @Test indicates: method ...  
    public static void m3() {...} // ... needs further tests  
} // Any  
Class c = Class.forName("Any");  
for (Method m: c.getMethods())  
    if (m.isAnnotationPresent(Test.class))  
        try { m.invoke(null); // call method which needs further tests  
    } catch (Exception e) { System.err.printf("Test %s failed ", m);  
    } // catch
```

Beispiel: Personenobjekte in einer relationalen Datenbank verwalten

```
public class Person {  
    @NotNull                         // user defined  
    @DBColumn(name = "ID", sqlType = "INTEGER") // user defined  
    public Integer id;
```

Überblick: Neuigkeiten in Java 6 (2007)

- Keine Änderungen/Erweiterungen an der Sprache
- Erweiterungen im Paket `java.util` beim *Java Collection Framework*: z. B. Schnittstellen `Deque` und `NavigableSet/Map` sowie neue Implementierungen davon, z. B. `ArrayDeque` und Erweiterungen vorhandener Implementierungen, z. B. implementiert `LinkedList` nun auch `Deque`
- Erweiterungen im Paket `java.io`: neue Klasse `Console`, die einfache zeichenbasierte Ein/Ausgabe bietet, z. B. Methoden `readLine` u. `printf`

```
Console c = System.console(); // Console has no publ. construct.  
String l = c.readLine(); c.printf("line just read: %s", l);
```

und neue Funktionalität in der Klasse `File`
- Völlig neue Möglichkeiten bietet das *Java Scripting API* im Paket `javax.script`, z. B.:

```
ScriptEngine e =  
    (new ScriptEngineManager()).getEngineByName("JavaScript");  
e.eval("for (var i = 0; i < 10; i++) print(i);");
```

- Neue Möglichkeiten zur Instrumentierung des Bytecodes im Paket `java.lang.instrumentation`
- Verbesserungen in den Bereichen *Reflection*, *RMI*, ...

Überblick: Neuigkeiten in Java 7 (2011)

- **Aufteilung** der *Standard Edition* (SE) in mehrere kleine Teile, die dynamisch herunter- und nachgeladen werden können (kürzere Ladezeiten, schnelleres Starten und geringerer Speicherbedarf), aber noch nicht das "Modulkonzept" von Java 9
- Bessere Unterstützung weiterer **Programmiersprachen** (vor allem Skriptsprachen wie z. B. Ruby und Python) durch neuen (!) Bytecode-Befehl *InvokeDynamic* (ist analog zu *InvokeVirtual* aber weniger restriktiv bei der Bytecode-Prüfung und auch schneller)
- Verbesserung der **Produktivität** in der Entwicklung:
 - Kleinere Sprachänderungen und -erweiterungen (z. B. Zeichenketten in der *switch*-Anweisung, s. u.)
 - Weitere Verbesserungen bei den Behälterklassen
 - Weitere Möglichkeiten zur Annotation von Datentypen (z. B.: `List<@NotNull Object> l = ...;`)
- Verbesserung der **Performanz**, z. B. durch neuen *Garbage Collector* (*G1* als Ersatz für *Concurrent Mark&Sweep*) oder komprimierte 64-Bit-Objektreferenzen für 64-Bit-Betriebssysteme)

Java 7: Einige Spracherweiterungen im Detail

- Binäre ganzzahlige Literale (neben oktal, dezimalen und hexadez.), z. B.:

```
byte b = 0b01010101;
```

- Unterstriche zur Strukturierung der Ziffernfolge von num. Literalen, z. B.:

```
int max1 = 32_767;  
int max2 = 0b01111111_11111111; // invalid: 0b_011...
```

- Mehrfachverzweigung (*switch*) auf Basis von Zeichenketten, z. B.:

```
String day; ...  
switch (day) { // primarily uses method hashCode  
    case "monday": ... // but also equals for collision detect.  
} // switch
```

- Automatisches Schließen von Ressourcen, die in *try*-Blöcken verwendet werden (*try with resources*), *finally*-Block kann damit entfallen, z. B.:

```
try (FileWriter fw = // class F.W. implements AutoCloseable  
     new FileWriter("ReadMe.txt")) { // "open" fw  
    ... fw.write(...); ...  
} // fw.close() called by default in all cases ...
```

Gibt es für den *try*-Block ein oder mehrere *catch* und/oder *finally*-Blöcke, werden die Ressourcen schon vor deren Ausführung geschlossen

- Behandlung mehrerer Ausnahme-Datentypen in einem *catch*-Block, z. B.:

```
try { ... } catch (IOException | SQLException e) { ... }
```

Als Ergänzung zu `java.lang.Object`: seit Java 7 neue Hilfsklasse `java.util.Objects`, für einfacheren Umgang mit `null`-Referenzen (vgl. `java.util.Arrays` für Felder)

Motivation: z. B. gilt für zwei `String`-Referenzen `s1` und `s2`: `s1 == s2` \Leftrightarrow `s2 == s1`, aber `s1.equals(s2)` kann anderes Ergebnis liefern als `s2.equals(s1)`
Bei `Objects.equals` spielen Reihenfolge der Parameter und ev. `null(s)` keine Rolle

```
package java.util;  
public final class Objects { // „static“ class, no objects  
    public static String toString(Object o [, String nullDefstr]);  
    public static boolean equals(Object o1, Object o2);  
    public static boolean deepEquals(Object o1, Object o2);  
    public static int compare(Object o1, Object o2,  
        /*cf. Comparable with*/ compareTo*/ Comparator comp);  
    public static int hashCode(Object o); // returns 0 for null  
    public static int hash(Object... os); // hashes all objs.  
    public static boolean isNull(Object o); // o == null ?  
    public static boolean nonNull(Object o); // o != null ?  
    public static void requireNonNull(Object o [, String message]);  
} // Objects
```

- **Lambda-Ausdrücke**: Codestücke als Parameter (vgl. C++11), s. u.
- **Standard-Methoden** für Schnittstellen, s. u.
- Paralleles Sortieren: Klasse `java.util.Arrays` hat nun jeweils zwei Sortiermethoden für Felder elementarer Datentypen, z. B. für `int[]`:

```
public static void sort(int a[] [, int from, int to]);  
public static void parallelSort(int a[] [, int from, int to]);
```

- Einteilung der SE-Klassenbibliothek in drei Profile: `compact` 1 (mit 14 MB) bis 3 (21 MB), wobei Profil $i+1$ das Profil i vollständig enthält
- Neue Datums- und Zeitklassen im Paket `java.time` (Ablöse für `java.util.Date` aus 1.0 und `java.util.Calendar` aus 1.1) mit Dauer
- **Stream-Klassen** für die parallele, einmalige (!) Abarbeitung von Daten (mit Filtern, `map`- u. `reduce`-Operationen), s.u. ... haben nichts mit `Input`- od. `Output`-
`Stream` zu tun!
- JavaFX 8: neue Version des Frameworks für die Programmierung flexibler u. effizienter GUIs (Ablöse für AWT und Swing) mit FXML
- Kleinere virtuelle Maschine(n) mit Größe von ca. 3 MB (bisher 6 MB für Client und 9 MB für Server) möglich, bessere Unterstützung kleiner Systeme
- Verbesserungen bei der autom. Speicherbereinigung (*garbage collection*): Beseitigung der permanenten Generation der Halde

Java 8: Lambda-Ausdrücke etwas detaillierter

Bestandteile von Lambda-Ausdrücken

1. Formalparameterliste
 2. Codestück (entweder ein Ausdruck oder eine Blockanweisung)
 3. Werte für die freien Variablen, die im Codestück verwendet werden (implizit)

Aber kein Zugriff auf Umgebung möglich (wie z. B. in C++11)

Syntax von Lambda-Ausdrücken

- $(form.\ param.\ list) \rightarrow expression$ oder
 - $(form.\ param.\ list) \rightarrow \{ statement\ sequence\ }$

Beispiele

```
Set<Person> s1 = new TreeSet<Person>( // Comparator:  
    (Person p1, Person p2) -> p1.ssnr - p2.ssnr );  
Set<Person> s2 = new TreeSet<Person>( // Comparator:  
    /*Pers.*/ p1, /*Pers.*/ p2) -> p1.name.compareTo(p2.name);
```

```
class Person {  
    int    ssnr;  
    String name;  
    ... // equals only  
} // Person
```

Methoden- und Konstruktor-Referenzen

Anstelle von Lambda-Ausdrücken können auch Referenzen auf Methoden (`C::m` oder `o::m`) und Konstruktoren (mit `C::new`) übergeben werden

Beispiel

```
Set<Person> s3 = new TreeSet<Person>(Person::ageCmp);
```

```
class Person { ...
    Date dateOfBirth;
    static int ageCmp(
        Person p1, Person p2) {...}
} // Person
```

Java 8: Standard-Methoden für Schnittstellen

Problem: Wenn eine Schnittstelle um eine Methode erweitert werden soll, müssen alle betroffenen Klassen, die neue Methode zusätzlich implementieren, z. B. sollte in Java 8 *Iterator* um Methode *forEachRemaining* erweitert werden

Lösung: Methoden können ab Java 8 auch eine Standard-Implementierung für (neue) Methoden haben, sodass diese in implementierenden Klassen übernommen werden können und dann keine Änderungen notwendig sind

```
interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    default forEachRemaining(Consumer<? extends E> action) {  
        while (hasNext())  
            action.accept(next());  
    } // forEachRemaining  
} // Iterator
```

```
interface Consumer<T> {  
    void accept(T t);  
    default Consumer<T> andThen(Consumer<? super T> after) { ... }  
} // Consumer
```

Schnittstellen können auch *Schnittstellen*-Methoden (mit *static* und Implementierung) haben: werden mit Schnittstellen- und Methodennamen aufgerufen werden, z. B. auch von Standard-Methoden aus, Schnittstellen-Methoden gelten in den implementierenden Klassen automatisch als definiert

Java 8: Stream-Funktionalität etwas detaillierter (1)

- Neuer Mechanismus für einmalige (!) Bearbeitung von Sequenzen von Daten, vor allem im Paket `java.util.stream`, Basisschnittstellen (*Base*)`Stream`
- Für jeden Behälter `c` kann ein `Stream` erzeugt werden, z. B. mit `c.stream()`
- Reihenfolge d. Abarbeitung (ev. parallel) wird Implementierung überlassen

```
public interface Stream<T> extends BaseStream<T, Stream<T>> {  
    long count();  
    Stream<T> filter(Predicate<? super T> pred); // long str -> short str  
    Stream<R> map(Func.<? sup. T, ? ext. R> m); // str<T> -> str<R>  
    Stream mapToX(ToXFunction<? super T> mapper); // X = Int, Double  
    Stream<T> (distinct|sorted)();  
    Stream<T> (limit|skip)(long n);  
    void forEach(Consumer<? super T> action);  
    Object[] toArray(); // stream -> array ←  
    T reduce([T identity,] BinaryOperator<T> accumulator);  
    Optional<T> (min|max)(Comparator<? super T> comparator);  
    boolean (any|all|none)Match(Predicate<? super T> predicate);  
    Optional<T> find(First|Any)();  
    public static<T> Stream<T> of(T... values); // array -> stream ←  
    public static<T> Stream<T> generate(Supplier<T> s);  
    public static<T> Stream<T> iterate(T seed, UnaryOperator<T> f);  
    public static<T> Stream<T> concat(Stream<? extends T> a, ... b);  
} // Stream
```

Beispiele: Bearbeitung von Behältern über Streams

```
Collection<String> c = Arrays.asList("...", "...", ...);

int longwords = 0, x = ...;           // v1: via foreach
for (String w: c)
    if (w.length() > x)
        longwords++;
System.out.println(longwords);

Stream<String> s1 = c.stream(); // v2: via stream
Stream<String> s2 = s1.filter(w -> w.length() > x);
longwords = s2.count();

longwords = c.stream().filter(w -> w.length() > x).count();
longwords = c.parallelStream().filter(w -> w.length() > x).count();

Stream<String> s = Stream.of("...", "...", ...);
```

Beispiele: Direkte Erzeugung von (potentiell unendlich langen) Streams

```
Stream<Double> randoms = Stream.generate(Math::random);

Stream<Integer> powersof2 = Stream.iterate(1, i -> i = 2 * i);
powersof2.limit(10).forEach(System.out::println);
```

Java 9

Wesentliche Erweiterung ist das **Modulsystem (Jigsaw)**: Java-Module ...

- ... sind benannter Behälter, die aus Paketen bestehen,
- ... können notwendige Module benennen und enthaltene Pakete exportieren
- ... müssen in einer Datei *module-info.java* beschrieben werden, z. B.:

```
module myModule {  
    requires anotherModule; // optional: static and transitive  
    exports aPackage; // optional: to ... for specific packages  
} // module
```

Java 10

Datentyp lokaler Variablen kann mit kontextspezifischem Schlüsselwort *var* aus Initialisierungsausdruck gewonnen werden (vgl. C# und *auto* in C++), z. B.:

```
int var; // int variable named var, may be used later on  
var c = new ArrayList<Integer>(); // c's type is ArrayList<Int.>  
for (var e: c) e = var;
```

Java 11 (*with long term support, LTS*)

Auch Formalparameter von Lambdas können mit *var* implizit typisiert werden

Java 12 und 13

Keine Erweiterungen in der Sprache, nur Kleinigkeiten in der Bibliothek

Java 14

Vor allem syntaktische Erweiterungen für *switch* mit *case*, z. B.:

```
static String colorof(int i) {
    // version 4 with
    // switch as expression
    return switch (i) {
        case 1      -> "green";
        case 2, 3, 4 -> "yellow";
        default       -> "red";
    }; // switch
} // colorof
```

Zusätzlich zu Erweiterungen in der Bibliothek auch wieder kleinere Spracherweiterungen:

- Klassen, die mit dem neuen Schlüsselwort `sealed` markiert sind, können mittels `permits` festlegen, wer von ihnen ableiten darf, sind also weniger einschränkend als `final`-Klassen, die ein Ableiten verhindern, z. B.:

```
[public] sealed class Shape permits Rectangle, Circle { ... }
```

- Mehrzeilige Zeichenkettenliterale (*text blocks*) können nun auch ohne Operator `+` gebildet werden, indem sie mit `"""` geklammert werden und ihre Konstruktion erfolgt schon zur Übersetzungszeit, z. B.

```
String oldPangram = "The quick brown fox " +
                    "jumps over a lazy dog";  
  
String newPangram = """
                    The quick brown fox \
                    jumps over a lazy dog\
                    """;  
// now oldPangram.equals(newPangram) == true
```

Zusätzlich zu Erweiterungen in der Bibliothek auch eine interessante Spracherweiterungen: Klassen f. unveränderliche Objekte (am Heap) mittels *record*

Beispiel:

```
[public] record Person(int ssnr, String name) {  
    // additional methods allowed  
} // Person
```

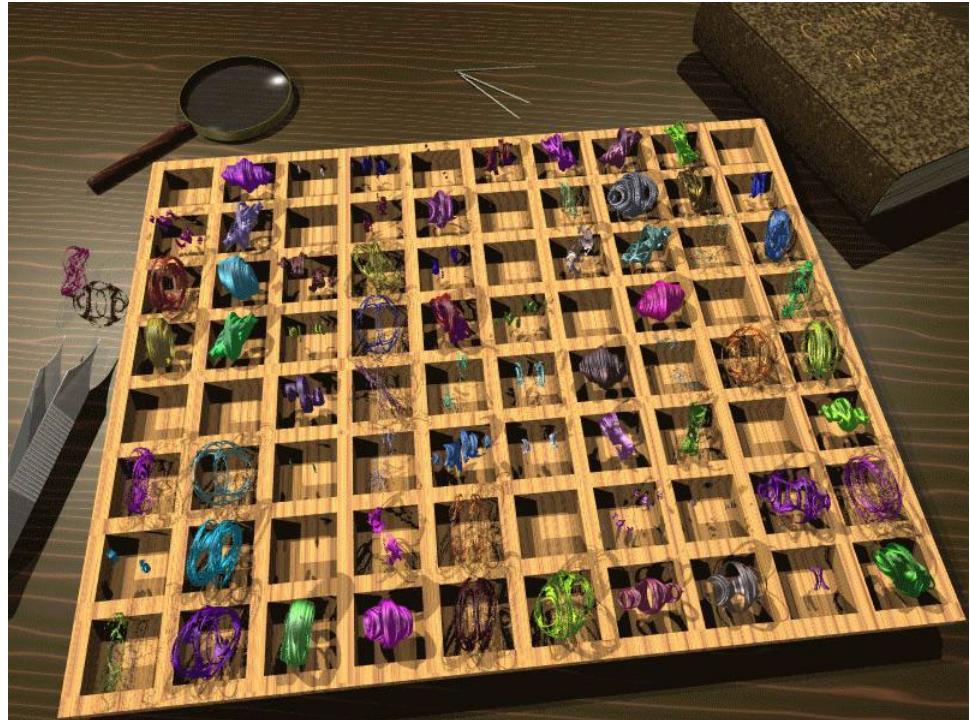
Ist gleichbedeutend mit:

```
[public] class Person extends Object {  
    private final int ssnr; // blank final  
    private final String name; // blank final  
    public Person(int ssnr, String name) { /* set ssnr and name */ }  
    public int ssnr() { return ssnr; } // no prefix get, no setter  
    public String name() { return name; } // no prefix get, no setter  
    // overridden methods from Object  
    public String toString() { /* returns ssnr and name in a string */ }  
    public boolean equals(Object o) { /* compares ssnr and name */ }  
    public int hashCode() { return Objects.hash(ssnr, name); }  
    // additional methods, copied from record definition  
} // Person
```

J C F

**Java
Collections
Framework**

Heinz Dobler
Version 21, 2025



© Anders Sandberg, www.nada.kth.se/~asa/ray.html, 1997

Permission to copy without any fee all or part of these slides is granted provided that copies are not made or distributed for commercial advantage, the copyright notice, the title of the presentation as well as its date appear, and notice is given that "copying is by permission of Dr. Heinz Dobler". To copy otherwise, or to republish, requires a fee and/or the specific permission.

- **Java 1.0** (1995): Schon Behälter, aber ...
- **Java 1.2** (1999): ... jetzt erst richtige: *Java Collections Framework (JCF)*
 - **Behälterarchitektur mit "drei Schichten":**
 1. Schnittstellen,
 2. abstrakte Klassen und
 3. konkrete Klassen
 - **Algorithmen** auf Behältern und Feldern
- **Java 5.0** (2005): Neue Möglichkeiten für Behälter durch Generizität
- **Java 6** (2007): Neue Schnittstellen und ...
- **Java 8** (2014): Kleinere Erweiterungen
- Weitere (nicht stand.) Behälterklassen-Bibliotheken in und für Java

Java 1.0 (1995): Schon Behälter, aber ...

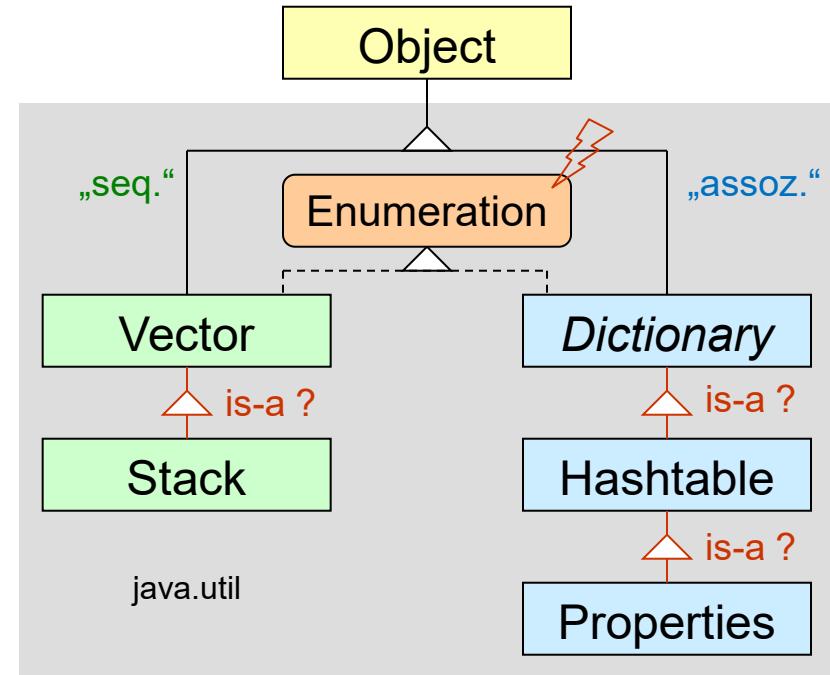
Schon in Java 1.0 gab es einige Behälterklassen in der Java-Standardbibliothek:

- Zwei "sequentielle" und zwei (konkrete) "assoziative" Behälterkl. im Paket *java.util*
- Unübliches Iterator-Konzept über Schnittstelle *Enumeration*
- Schlechtes oo Design: *is-a* ?

Beispiel zu "seq." Behälter:

```
vector v = new Vector();
v.add(...);

Enumeration e = v.elements()
while (e.hasMoreElements()) {
    Object o = e.nextElement();
    ... // do anything with o
} // while
```



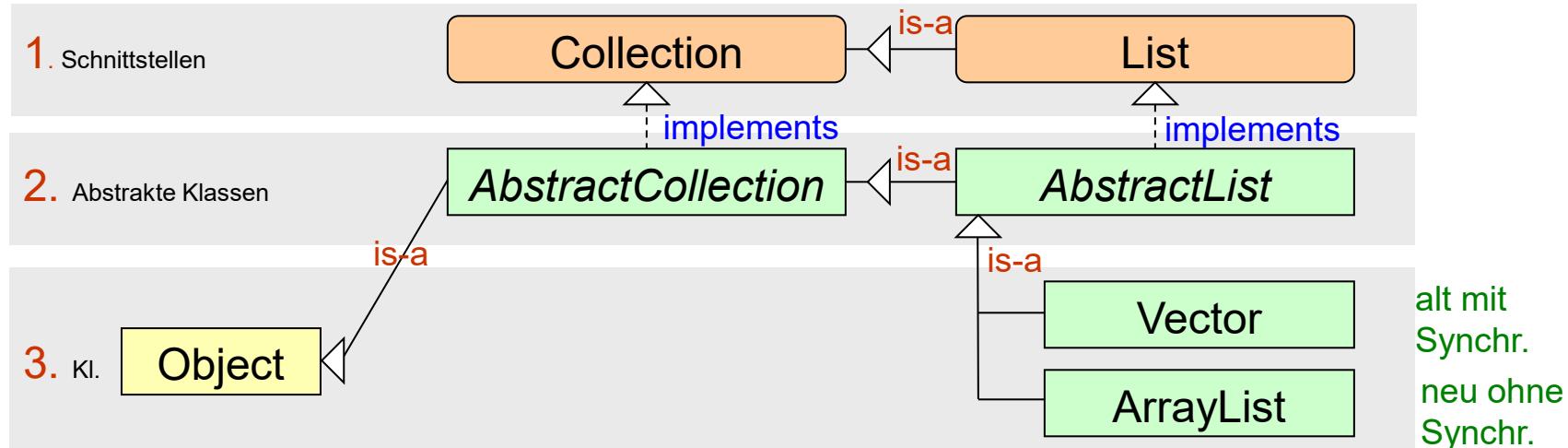
Beispiel zu "assoz." Behälter:

```
Dictionary d =
    new Hashtable();
d.put("one", "eins");
...print (d.get("one")); // eins

Enum. ek = d.keys();
Enum. ev = d.values();
```

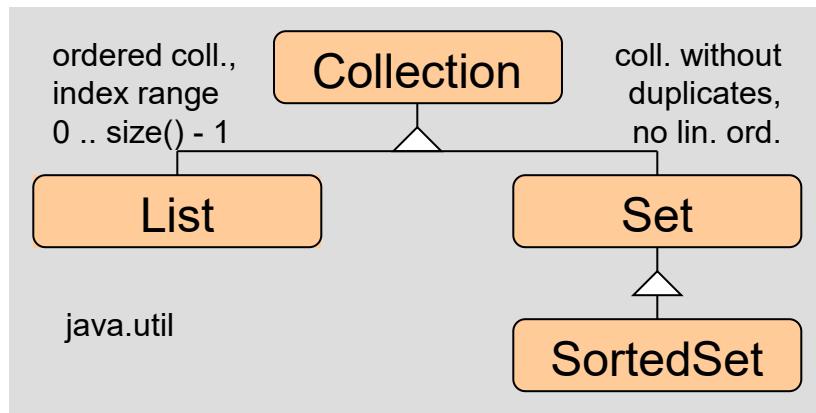
- Seit Java 1.2 ("Java 2") umfangreiche Bibliothek mit zehn Schnittstellen und 13 Klassen für Behälter, Iteratoren (zus. *Enumerator*) und Algorithmen ⇒ *Java Collections Framework (JCF)* im Paket `java.util`
- Dreistufiger Aufbau (von sehr abstrakt bis konkret):
 1. Hierarchie von **Schnittstellen**
 2. Hierarchie **abstrakter Klassen**, die Schnittst. teilw. implement.
 3. Hierarchie **konkreter Klassen**, von den abstrakten K. abgeleitet

Beispiele zum dreistufigen Aufbau, Ausschnitte aus dem *JCF*:



Schnittstellen für Behälterklassen (1)

```
public interface Collection {  
    int      size();  
    boolean isEmpty(); // size() == 0 ?  
    boolean add      (Obj. e);  
    boolean remove   (Obj. e);  
    boolean contains(Obj. e); // uses ...  
    Iterator iterator(); // ... equals()  
} // Collection
```



```
public interface List  
    extends Collection {  
    int      indexOf(Obj. e);  
    Object   get(int index);  
    Object   set(int index, Obj. e);  
    ListIterator listIterator();  
} // List
```

Alle hier aufgeführten Schnittstellen zeigen nur die wichtigsten Methoden!

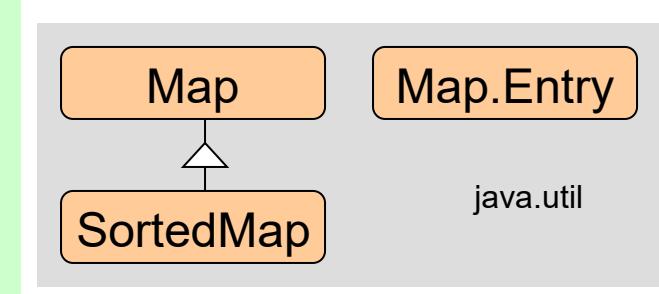
```
public interface Set  
    extends Collection {  
} // Set      //  
public interface SortedSet  
    extends Set {  
    Obj.     first();  
    SortedSet headSet(Obj. to);  
    Obj.     last();  
    SortedSet tailSet(Obj. from);  
    SortedSet subset (Obj. from,  
                      Obj. to);  
} // SortedSet
```

Schnittstellen für Behälterklassen (2)

```
public interface Map { // not iterable!
    int size();
    boolean isEmpty();
    boolean put(Object key, Object value);
    boolean remove(Object key);
    Object get(Object key);
    boolean containsKey (Object key);
    boolean containsValue(Object value);
    Set entrySet(); // -> iterable set of Entries
}

public interface Entry {
    Object getKey(); // no set method for key!
    Object getValue();
    Object setValue(Object newValue);
} // Entry

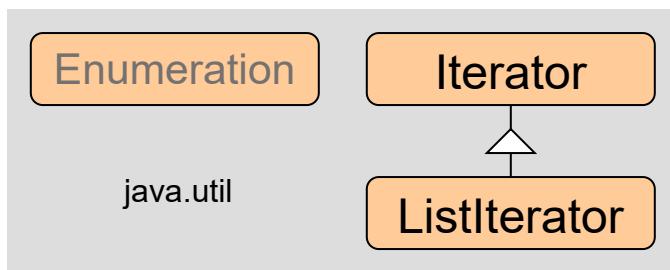
} // Map
```



Alle hier aufgeführten Schnittstellen zeigen nur die wichtigsten Methoden!

```
public interface SortedMap extends Map {
    Object firstKey();
    SortedMap headMap(Object toKey);
    Object lastKey();
    SortedMap tailMap(Object fromKey);
    SortedMap subMap (Object fromKey,
                      Object toKey);
} // SortedMap
```

Schnittstellen für Behälterklassen (3)



```
public interface Enumeration {
    boolean hasMoreElements();
    Object nextElement ();
} // Enumeration
```

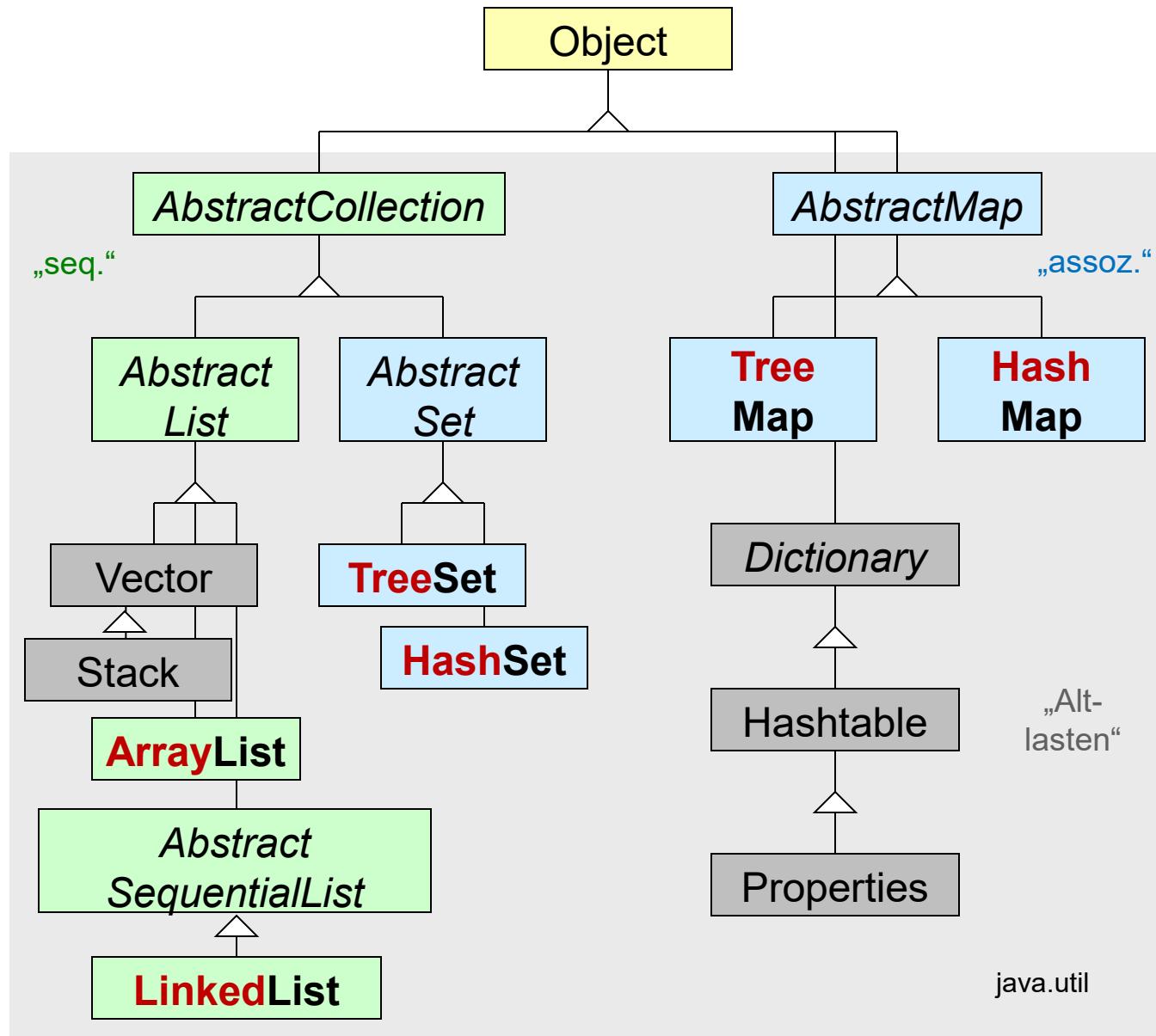
Alle hier aufgeführten
Schnittstellen zeigen
nur die wichtigsten
Methoden!

```
public interface Iterator {
    boolean hasNext();
    Object next(); "ForwardIt."
    void remove(); // optional
} // Iterator
```

```
public interface ListIterator
    extends Iterator {
    boolean hasPrevious();
    Object previous(); "Bidirect.It."
    int nextIndex();
    int previousIndex();
} // ListIterator
```

Aber kein "RandomAccessIterator"

Abstrakte und konkrete Behälterklassen



Algorithmen auf Behältern

Algorithmen auf Behältern sind in Form von Klassenmethoden realisiert

```
public class Collections {  
  
    public static Enumeration enumeration(Collection coll);  
  
    public static Object min(Collection coll); // both use ...  
    public static Object max(Collection coll); // ... compareTo()  
  
    // list is a sequence with linear ordering of elements  
    public static int binarySearch(List list, Object key);  
    public static void reverse(List list);  
    public static void sort (List list); // merge sort  
    public static void sort (List list, Comparator cmp);  
    public static void shuffle(List list); // random permut.  
  
    public static Coll. synchronizedCollection(Coll. coll);  
    public static Coll. unmodifiableCollection(Coll. coll);  
  
} // Collections
```

Algorithmen auf Feldern

Algorithmen auf Feldern sind ebenfalls in Form von Klassenmethoden realisiert

```
public class Arrays {    jeweils gleicher Wertdatentyp oder Object
    public static int fill (...[] a, ... val);
    public static int equals(...[] a1, ...[] a2);
    public static int binarySearch(...[] a, ... key);
    public static int sort(...[] a); // quick sort for value types,
                                    // merge sort for ref.  types
    // for multi-dimensional arrays, since 5.0
    public static int deepEquals(Object[] a1, Object[] a2);
} // Arrays
```

Benchmark

Sortieren von *int*-Feldern und *ArrayList*-Behältern mit *Integer*-Elementen, mit *quick sort* bzw. mit *merge sort*, jedenfalls in $O(n \log n)$, Laufzeiten in ms

Verfahren	$n = 100.000$	$n = 1.000.000$
<i>Arrays.sort(...)</i> <i>int</i> -Elemente	31	328
<i>Collections.sort(...)</i> <i>Integer</i> -Elem.	256	2.265

- JCF nutzt nun die neuen Möglichkeiten der generischen Programmierung
- Alle Schnittstellen u. Klassen sind generisch, mit Elementtyp parametrisiert

Beispiele

neu s. u.

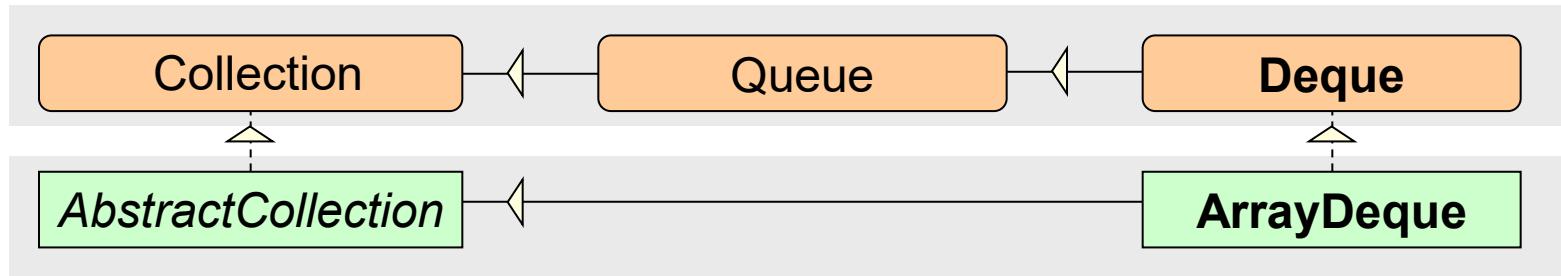
```
public interface Collection<T> extends Iterable<T> {...} // Iterable: new  
public abstract class AbstractCollection<T> implements Collection<T> {...}  
public class ArrayList<T> extends AbstractList<T>  
    implements List<T>, RandomAccess, Cloneable, Serializable {...}  
  
public abstract class AbstractMap<K, V> implements Map<K, V> {...}  
public class HashMap<K, V> extends AbstractMap<K, V>  
    implements Map<K, V>, Cloneable, Serializable {...}
```

- Neue Behälter für Warteschlangen (z. B. *Queue* und *PriorityQueue*)
- Neue Basisschnittstelle *java.lang.Iterable* für *Collection* zur Unterstützung der neuen *for*-Schleife (alle "alten" Behälterschnittstellen erweitern *Iterable*, s. o.)

```
public interface Iterable<T> { // T is element type of collection  
    java.util.Iterator<T> iterator(); // resulting Iterators next method ...  
} // Iterable // ... returns a T object
```

- Neue Algorithmen (z. B. *frequency*, *reverseOrder*)

- Neue Schnittstelle *Deque* (*double ended queue*), Implementierungen (z. B. *ArrayDeque*) erlauben schnelles Einfügen u. Löschen an beiden Enden



Wichtigste Methoden: *addFirst*, *removeFirst*, *addLast* und *removeLast*

- Neue Schnittstellen *NavigableSet* und *NavigableMap*, bieten Iteratoren in beiden Richtungen, z. B.

```
public interface NavigableSet<E> {  
    Iterator<E> iterator(); // it.next() -> next element  
    Iterator<E> descendingIterator(); // it.next() -> prev. element  
} // NavigableSet
```

Erlauben auch Suche nach Elementen, die am "besten" zu gesuchtem Wert passen (z. B. mit den Methoden *lower* und *greater*)

- Erweiterungen in bereits existierenden Klassen, damit sie zu den neuen Schnittstellen passen (z. B. ist *TreeSet* an *NavigableSet* angepasst)

- Erweiterung der Schnittstelle `java.lang.Iterable<T>` (bisher nur Methode `iterator`) um Methode `forEach` mit Standardimplementierung (*default meth.*)

```
default void forEach(Consumer<? super T> action);
```

- Erweiterung der Schnittstelle `jav.util.Collection<E>` (abgel. v. `Iterable<E>`)

```
default boolean removeIf(Predicate<? super E> filter);
```

- Erweiterung der Schnittstelle `jav.util.Map<K, V>` um viele weitere Methoden (z. B. auch `forEach`, weil `Map` `Iterable` nicht implementiert)
- Verbindung zwischen *Collections* und *Streams* mit Konvertierung in beiden Richtungen, z. B.:

```
Collection<Person> pc = Arrays.asList(new Person(...), ...);

// Collection -> Stream:
Stream<Person> ps = pc.stream();
Stream<Person> pps = pc.parallelStream();

Stream<Person> ads = ps.filter(p -> p.age >= 18); // adults only

// Stream -> Collection:
Collection<Person> adc = pc.stream() // using "fluent interfaces"
                           .filter(p -> p.age >= 18)
                           .collect(Collectors.toCollection(
                               ArrayList::new));
```

Java Generic Library (JGL) and Toolkit



- Umfangreiche Bibliothek analog zur STL mit 28 Behälter, 205 Algorithmen und 31 Iteratoren
- Altes Konzept: Generizität ersetzt durch Polymorphismus
- Seit Java 5.0: Verwendung von generischen Klassen wie i. d. STL
- <http://www.recursionsw.com/jgl-toolkit-overview/>, ca. \$ 100

Java Data Structures Library (JDSL)



- Bibliothek mit konventionellen Behältern (Sequenzen, Suchbäumen und Hashtabellen) sowie Algorithmen
- Außerdem **Graphen mit Algorithmen** (z. B. diverse Durchläufe, kürzeste Wege, kleinster aufspannender Baum)
- www.cs.brown.edu/cgc/jdsl, keine Kosten
- Buch: M. T. Goodrich and R. Tamassia: *Data Structures and Algorithms in Java (Second Edition)*, Wiley, 2001

Google Collections Library (GCL), neuer Name: Guava



- Erweiterung des Standard-JCF
- Neue Behälter (z. B. *Multimap*, *Multiset*, *BiMap*)
- Früher unter google-collections.googlecode.com, heute unter <https://github.com/google/guava>, keine Kosten

Software: Architektur & Design

Heinz Dobler

Version 21, 2025



fa·cade also **fa·çade**

- 1:** the front or face of a building;
- 2:** a false, superficial, or artificial appearance

Bildquelle: pixabxy.com
Textquelle: <https://www.merriam-webster.com/dictionary/facade>

Permission to copy without any fee all or part of these slides is granted provided that copies are not made or distributed for commercial advantage, the copyright notice, the title of the presentation as well as its date appear, and notice is given that "copying is by permission of Dr. Heinz Dobler". To copy otherwise, or to republish, requires a fee and/or the specific permission.

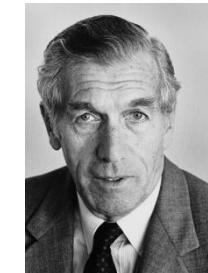
- Motivation
 - Meta-Schichtenarchitektur mit **dreischichtiger Konzepte Ebene**
 - **1. Ausdrucksweisen** (*idioms, coding patterns*)
 - **2. Entwurfsmuster** (*design patterns*)
 - Prinzipien für guten oo Entwurf
 - Einteilung und Raum der GoF-Entwurfsmuster
 - Übersicht über alle GoF-Entwurfsmuster
 - Wichtige GoF-Entwurfsmuster im Detail
 - **3. Softwarearchitektur und -muster**
 - Was ist Softwarearchitektur?
 - Verschiedene Sichten auf Softwarearchitektur
 - Architekturmuster (*architectural patterns*)
 - Literaturempfehlungen
- 
- Mit einigen Beispielen (vor allem in Java)
- 
- Nur eine kurze Einführung, mehr im SE Master

- Erste Idee für Muster in der (Gebäude-)Architektur
- Beispiel zu *Déjà Vu*
- Substitutionsprinzip von Liskov
- Weitere Prinzipien für guten oo Entwurf
- Behälter in Java 1.0 mit *Enumeration*
- *Java Collection Framework (JCF)* seit Java 1.2
- Entwurfsmuster in der MiniLib
- Drei-Schichten-Architektur (*three tier architecture*)
- Physikalische Verteilung bei Drei-Schichten-Architektur

- Softwareentwicklung ist (mittlerweile!) ein extrem **kommunikationsintensiver Prozess**:
 - Zwischen den (verschiedenen Arten von) "Entwickler*innen"
 - Zwischen "Entwickler*innen", Anwender*innen und sonstigen ...
- Rasante Weiterentwicklung bei Hardware und bei Software-technologien, z. B. bei Pgm.Sprachen, Bibliotheken u. Werkzeugen
- Wie steht es mit der **Weiterentwicklung der Kommunikation?**
 - Auf "unterster", technischer Ebene meist kein Problem
 - Auf "höherer", abstrakter Ebene bestehen oft Defizite
- Abraham H. **Maslov**, *1908 †1970
Kommunikationswissenschaftler, Begründer d. humanist. Psychologie und Erfinder der Bedürfnispyramide
- Paul **Watzlawick**, *1921 †2007
Psychotherapeut und Kommunikationswissenschaftler:



"Wer als Werkzeug nur einen **Hammer** hat, sieht in jedem Problem einen **Nagel**."



Grady Boochs düsteres Bild der Softwareentwicklung

mit Ivar Jacobson und James Rumbaugh: *The Three Amigos*

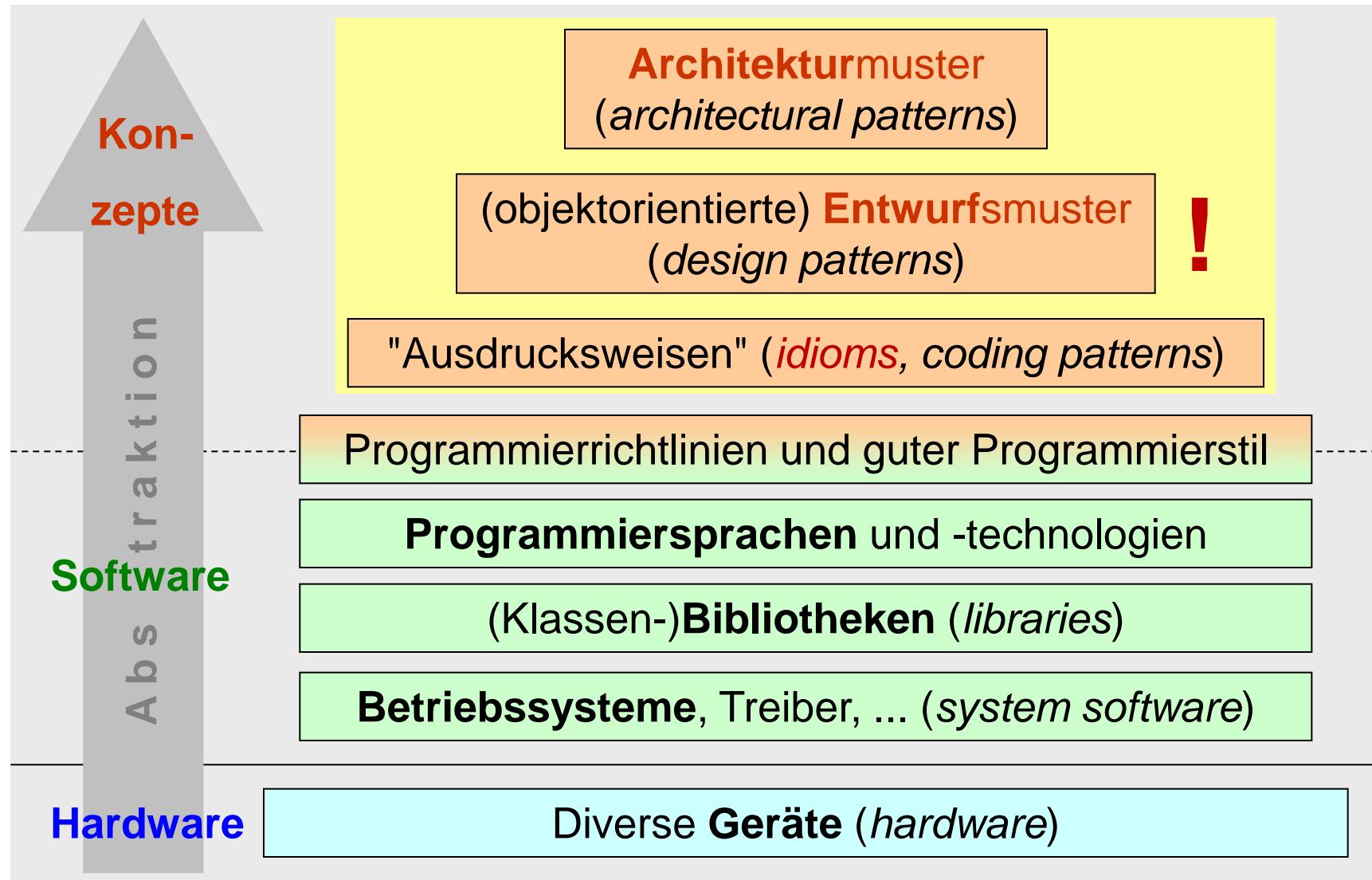
Grady Booch, Miterfinder der UML (*Unified Modeling Language*) und Vordenker von IBMs Rational-Sparte, zeichnet ein pessimistisches Bild der Softwarezukunft. Vor allem bei den Entwicklungsprozessen werde sich nicht viel ändern. "Mir wird übel, wenn ich mir anschauе, was immer noch in den Informatik-Studiengängen gelehrt wird", sagte er in einem Gespräch mit heise



So würden das *Patchen* von Systemen und das Entfernen von Viren einen viel zu breiten Raum einnehmen. "Softwareentwickler müssen vor allem lernen, im **Team zu arbeiten** und zu **kommunizieren**, sie müssen ihre **Ergebnisse** besser **präsentieren** können und sie müssen deutlich mehr betriebswirtschaftliches Basiswissen mitbringen", lautet sein Rat an die Hochschulen.

Einen überraschenden Aspekt stellt Boochs Ansicht dar, dass alle Ausbildungseinrichtungen Software-Lesekurse zur Pflicht machen sollten. "Es gibt keine einzige Ingenieursdisziplin, in der nicht die bedeutenden Werke früherer Meister genau unter die Lupe genommen werden. Auch alle angehenden Autoren müssen zunächst viel lesen. Nur die Schreiber von Programmen meinen, dass sie genial sind und nicht den **Code von anderen analysieren** müssen", war seine massive Kritik an der Ausbildung.

Quelle: <http://www.heise.de/developer/meldung/UML-Erfinder-Grady-Booch-zeichnet-duesteres-Bild-der-Softwareentwicklung-1020658.html>



Charakterisierung:

- *Idioms* (Ausdrucksweisen): **sehr einfache Muster**, bestehen meist nur aus einer Klasse
- Abstraktionsgrad: zwischen "herkömmlichen" Programmierrichtlinien (Hinweisen für guten Programmierstil) und "echten" Entwurfsmustern (s. u.) angesiedelt
- Anwendungsbereich: oft sehr spezifisch und nur **für eine bestimmte Programmiersprache** (häufig für C++, weil ...)

Drei typische Beispiele (weiter unten im Detail):

1. **Smart Pointer** in C++,
z. B. früher: `std::auto_ptr<T>`, `boost::scoped_ptr<T>`
oder eigene für assoz. STL-Behälter (`smart_ptr<T>`, s. u.)
seit C++11: `std::unique_ptr<T>` und `std::shared_ptr<T>`
2. **Property** in C++, Java und C#
3. **Enum** in Java (bis Java 1.5, seither `enum` als Datentyp i. d. Sprache)

Idiom-Beispiel 1: Smart Pointer für C++

Klassenschablone für `smart_ptr<T>`, z. B. f. assoz. Behälter der STL:

```
template <class T> // e.g., for STL assoc. containers:  
class smart_ptr { // operator< compares T values  
private:           // not addresses  
    T *p;  
public:  
    explicit smart_ptr(T *p = nullptr) : p(p) { }  
    ~smart_ptr() { /*if (this is owner of *p) delete p; */ }  
    T & operator*() const { return *p; }  
    T * operator->() const { return p; }  
    operator T*() const { return p; }  
    bool operator< (const smart_ptr &sp) const {  
        return (*p) < (*sp.p); // uses bool operator<(T, T)  
    } // operator<           // or bool T::operator<(T)  
}; // smart_ptr
```

`smart_ptr`-Objekt und STL-Behälter `set` mit Auspr. für `smart_ptr`:

```
smart_ptr<string> sp(new string("..."));  
cout << *sp;  
set<smart_ptr<string>> s; // order strings by values  
s.insert(sp);
```

Idiom-Beispiel 2: *Property* in C++

Generische Klasse *property*<T> für beliebige Eigenschaften:

```
template <class T>
class property {
    private:
        T data;
        void operator=(const property<T> &p);
    public:
        property(const T &d = T()) : data(d) { }
        T operator()() const { return data; }
        void operator()(const T &d) { data = d; }
}; // property
```

Klasse mit Eigenschaften:

```
class person {
    public:
        property<string> name;
        property<date> date_of_birth;
        person(string n, date d): name(n), ... { }
}; // Person
```

Verwendung:

```
person p(...);
p.name("Bjarne"); // set name
cout << p.name(); // get name
```

Idiom-Beispiel 2: *Property* in Java

Beispiel: Einfache Personenklasse mit nur zwei Eigenschaften

```
public class Person {  
  
    private String name;  
    private Date dateofBirth;  
  
    public Person(String name, Date dateofBirth) { ... }  
  
    public String getName() { return name; }  
    public void setName(String name) { this.name = name; }  
  
    public Date getDateofBirth() { return dateofBirth; }  
  
} // Person
```

Hinweis: Einfacher mit `@Getter/@Setter`
aus dem Projekt Lombok (projectlombok.org)

Java-Terminologie

- **JavaBeans** sind spezielle Klassen/Objekte, die spez. Zugriff erlauben (= einfaches Komponentenmodell)
- *Introspection* ist Erweiterung von *Reflection*, z. B. erlauben *Introspector* und *BeanInfo* Zugriff auch auf Ereignisse und Methoden
- Spezielle Werkzeuge, z. B. *BeanBox* oder *PropertyEditor*

Z. B. in JavaFX 8:
abstrakte generische.
Klasse `Property<T>`
im Paket `javafx.beans.property`
mit der Möglichkeit, auf
Änderungen zu reagieren,
und mit davon abgeleiteten
konkreten Klassen für
versch. Datentypen

Idiom-Beispiel 2: *Property* in C#

Eigenschaften (*properties*) sind Kombinationen aus (privaten) Datenkomponenten mit speziellen (öffentlichen) Zugriffsmethoden, z. B.:

```
class Person {  
    private String name;  
    private DateTime dateOfBirth;  
  
    public Person(String name, DateTime dob) { ... }  
  
    public String Name {  
        get {  
            return name;  
        } // get  
        set { // implicit parameter list: (String value)  
            name = value;  
        } // set  
    } // Name  
  
    public DateTime DateOfBirth {  
        get {  
            return dateOfBirth;  
        } // get  
    } // DateOfBirth  
}  
} // Person
```

```
// since C# 3.0 auto implemented prop.  
class Person {  
    public String Name {get; set;}  
    public DateTime DateOfBirth {get; }  
    ...  
} // Person
```

Eigenschaften werden vom C#-Compiler aber in *set-* und *get*-Methoden übersetzt, weil ...

Idiom-Beispiel 3: *Enum* in Java

Früher: Einfache (unsichere) Nachbildung mittels *int*-Konstanten, z. B.:

```
class TrafficLight {  
    public static final int RED      = 0;  
    public static final int YELLOW   = 1;  
    public static final int GREEN   = 2;  
} // TrafficLight
```

```
int t1;  
t1 = TrafficLight.RED;  
t1 = 17; // ???
```

Verbesserung mittels speziellem Datentyp (gen. *enum pattern*), z. B.:

```
class TrafficLight {  
    public static final TrafficLight RED = new TrafficLight(0);  
    ...  
    public final int value;  
    private TrafficLight(int value) {  
        this.value = value; // now this.value is fixed  
    } // TrafficLight  
    public String toString() {  
        return "" + value;  
    } // toString  
} // TrafficLight
```

Probleme: Hoher Schreibaufwand, keine Verwendung in *switch*-Anweisung

Heute (seit Java 1.5): **enum** TrafficLight { RED, YELLOW, GREEN; }

Exkurs: Erste Idee für Muster in der Architektur

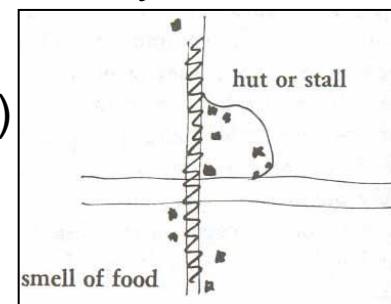
Christopher ALEXANDER

1936 in Österreich geborener, mit Eltern zuerst nach England
dann nach Amerika ausgewanderter Mathematiker und Architekt
ab 1963 Prof. an der Berkley Univ. of California, † 17. 3. 2022



Qu.: wikipedia.org,M. Mehaffy

- Mit Sara Ishikawa, Murray Silverstein u. anderen dreibändiges Werk, Trilogie (Oxford University Press, New York, 1977):
 - Vol. 1: *The Timeless Way of Building*
 - **Vol. 2: A Pattern Language**
 - Vol. 3: *The Oregon Experiment*
- **Idee der Muster (patterns):** "Each pattern describes a **problem** which occurs over and over again in our environment, and then describes the **core of the solution** to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way".
- In **Vol. 2** insgesamt **253 Muster**, davon
 - 94 f. Städte/Infrastr. (z. B. Würstelstand->)
 - 111 f. Gebäude (z. B. Dachgarten)
 - 48 f. Konstruktionen (z. B. Fundament)



Quelle: Gamma et al., "Design Patterns"

Prämissen

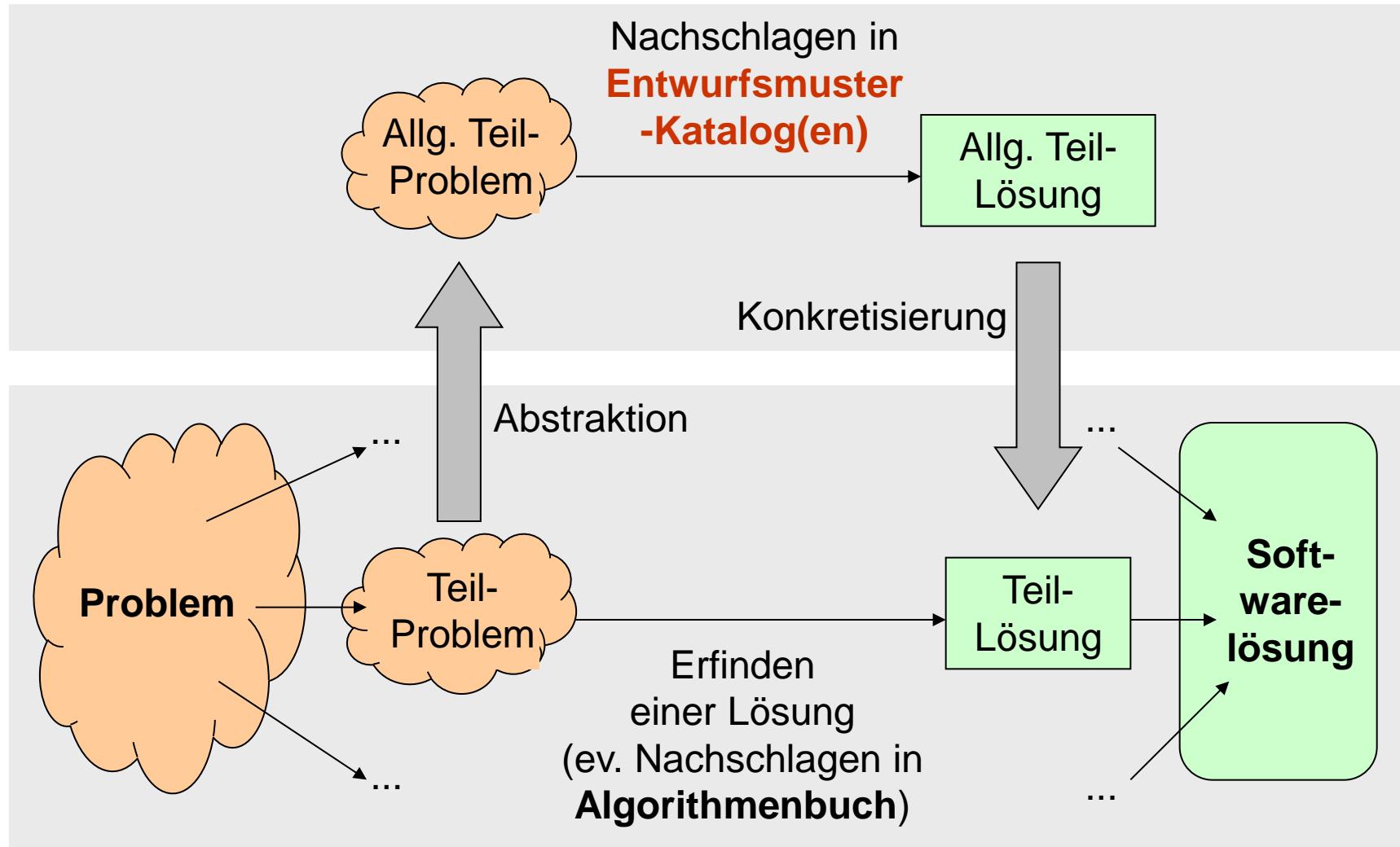
- OOP ist nicht so einfach (Vererbung, Polymorphismus, dyn. Bindung, ...)
- Herstellung **guter** oo Software ist schwierig
- Herstellung **wiederverwendbarer** oo Software ist noch schwieriger

Problem

- Anwendungen folgen oft gewissen Grundmustern
- Es kommt immer wieder zu Déjà-vu-Erlebnissen
- Erfahrene oo-Entwickler*innen treffen i. d. R. von vornherein relativ gute Designentscheidungen
- Über die Zeit wird Design verbessert u. "poliert"
- Gutes Design lernt man vor allem durch Erfahrung
- Zentrale Frage: Wie können gute Designs ("Erfahrungsschätzte") so **dokumentiert** und aufbereitet werden, dass auch das konzeptuelle Design und nicht nur die spezielle Lösung (Implementierung) wieder verwendet werden kann?

Richard Helm
zum Thema **Dokumentation**:
"In the software world, we tend to spend much effort
(1) describing descriptions of software (notations)
(2) descriptions of how to create software (methods)
(3) but far less effort
describing the actual software artefact itself".

Motivation für Musterkatalog (vgl. Algorithmenbuch)

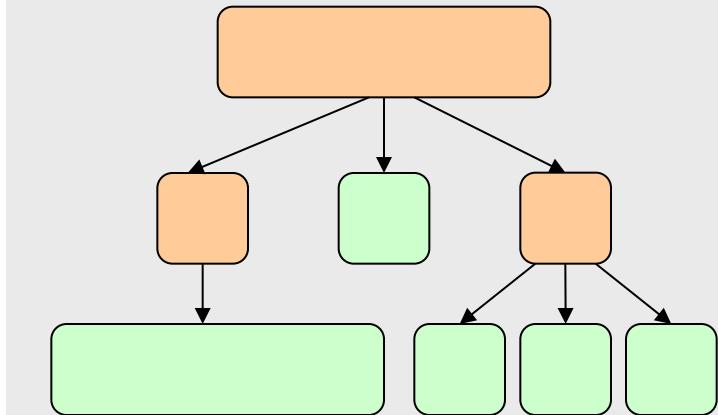


Exkurs: Beispiel zu *Déjà Vu*

In der Realität kommen sehr häufig **Hierarchien** vor, vier Beispiele:

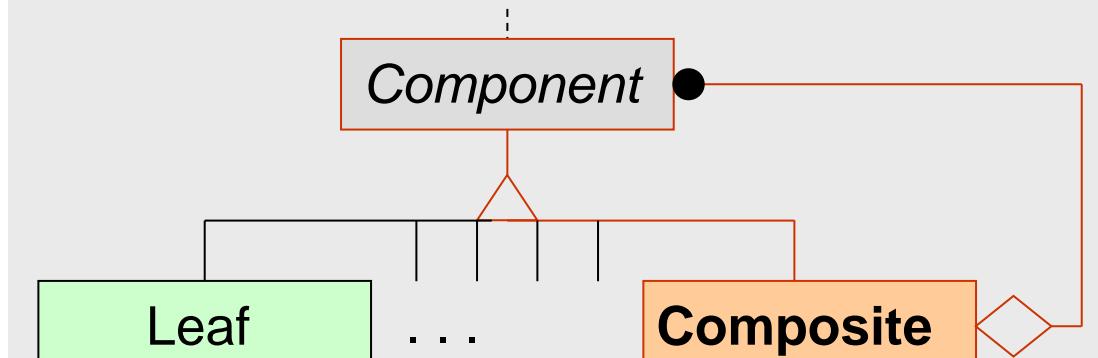
1. Hierarchische Aufbauorganisation
2. Hierarchie grafischer Objekte
3. Hierarchisches Dateiverzeichnis
4. Hierarchische Behälter

Spez. Objekthierarchie



Hierarchien können mit **"Mustern"** aus mehreren **zusammenhängenden Klassen** (in Form einer Klassenhierarchie) modelliert werden

Allgemeine Klassenhierarchie



Entwurfs- oder Designmuster (*design patterns*)

Entwurfsmuster beschreiben einfache, elegante und flexible Lösungen auf abstrakter Ebene (unabhängig von einer Programmiersprache)

Erich Gamma

früher UBILAB (ET++),
dann IBM OTI Lab
(Eclipse, JUnit), jetzt
Microsoft (VS Code)



Bildquelle: twitter.com

Wichtigste Bestandteile:

gemeinsam mit Richard Helm, Ralph Johnson und John Vlissides:
Gang of Four (GoF)

- **Name:** Ein oder zwei Wörter, Name kann später zur Beschreibung des Entwurfsproblems herangezogen werden, erhöht das Designvokabular und führt zu einem Design auf höherer Abstraktionsebene, wie immer: gute Namenswahl ist sehr schwierig
- **Problembeschreibung:** Wann kann das Muster angewendet werden? Oft mit einer Liste von Bedingungen, die erfüllt sein müssen
- **Lösungsbeschreibung:** Bestandteile des Musters, ihre Zusammenhänge, Verantwortlichkeiten und ihr Zusammenspiel
- **Konsequenzen:** Ergebnisse (Vor- und Nachteile) aus der Anwendung des Musters (meist kostet Zugewinn an Flexibilität auch etwas Speicher und/oder Laufzeit)

Alles unter Beachtung wichtiger **Entwurfsprinzipien** für oo Design

Beispiel: Model/View/Controller (MVC)

Lösung in Smalltalk 80 zu Realisierung grafischer Benutzeroberflächen,
Zusammenwirken von Objekten **dreier Klassen**:

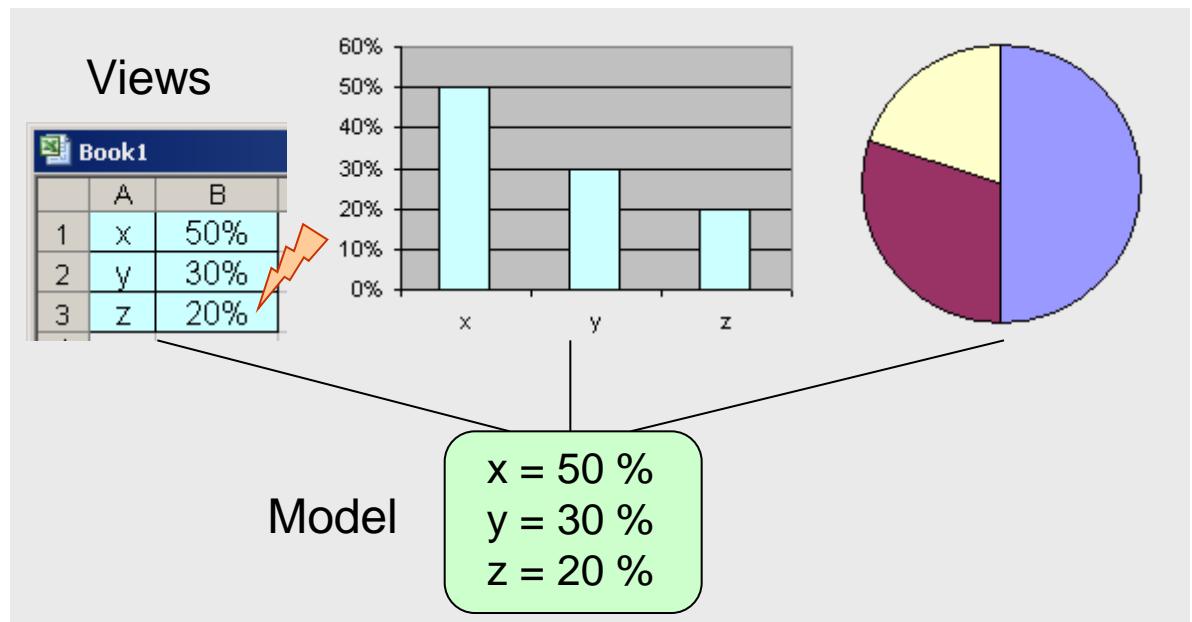
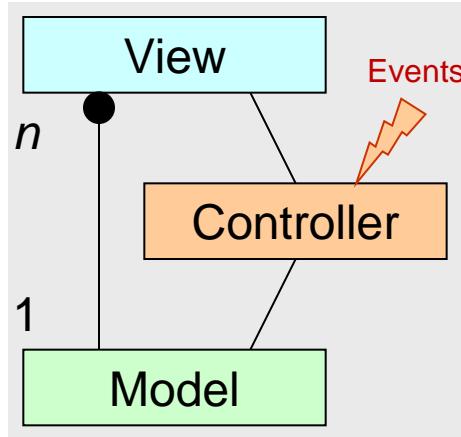
1. **Modell (model)**: Daten der Anwendung (*business object*)
2. **Ansicht (view)**: Darstellung der Daten auf dem Bildschirm
3. **Steuerung (controller)**: Interaktion mit Benutzer*in

Adele
Goldberg

Quelle: www.cs.umd.edu



Beispiel: Tabellenkalkulation

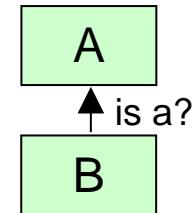


MVC entkoppelt (somit lose Bindung) Modell und Ansichten durch ein definiertes Protokoll:

- Jede Ansicht muss jeden Zustand des Modells darstellen können
- Wenn Benutzer*in Änderungen über Benutzeroberfläche verlangt, wird Modell geändert
- Wenn Modell geändert wurde, werden die Ansichten benachrichtigt u. können reagieren

Liskovsches Substitutionsprinzip (LSP, informell)

Objekte der Klasse *A* dürfen nur dann durch Objekte der Klasse *B* substituiert werden, wenn dabei auch alle **erwünschten Eigenschaften** d. Programms erhalten bleiben



Beispiel: Problematische oo Modellierung graf. Objekte (hier mit Java)

```

class Rectangle { // with two components ...
    private int width, height; // ... as properties with get+set
    public Rectangle(int width, int height) { ... }
    public int getwidth or Height() { return width or height; }
    public void setwidth or Height(int width or height) { ... }
} // Rectangle
  
```

```

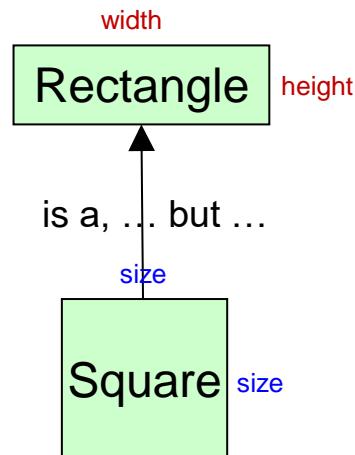
class Square extends Rectangle { // size is "implicit"
    // where (size == width) and (size == height)
    public Square(int size) { super(size, /*==*/ size); }
    public int getSize() { return getwidth(); /*or Height*/ }
    public void setSize(int size) {
        setwidth(size); /*and*/ setHeight(size);
    } // setSize
} // Square
  
```

```

void m(Rectangle r) {
    r.setwidth(17);
} // m
  
```

```

Square s = new Square(4);
m(s); // ok, s is a Rectangle
// ... but: what is s now ?
  
```



Prinzip 1: "Program to an Interface, not to an Implementation"

- Wichtig ist die Schnittstelle einer Klasse, nicht deren Implementierung (da sich diese häufiger ändert als die Schnittstelle)
- Variablen sollten möglichst nicht konkrete Klassen als Typ haben, sondern abstrakte Klassen oder noch besser Schnittstellen (bringt höhere Flexibilität durch bessere Austauschbarkeit)

Beispiel 1

```
Collection c = new AnyClassImplementingCollection(...);  
c.add(anyObject); // add is a Collection method
```

Irgendwie müssen konkrete Objekte (Instanzen konkreter Klassen) erzeugt werden: Objekterzeugung sollte möglichst indirekt erfolgen (mit Hilfe entsprechender Entwurfsmuster, z. B. *Factory Method*)

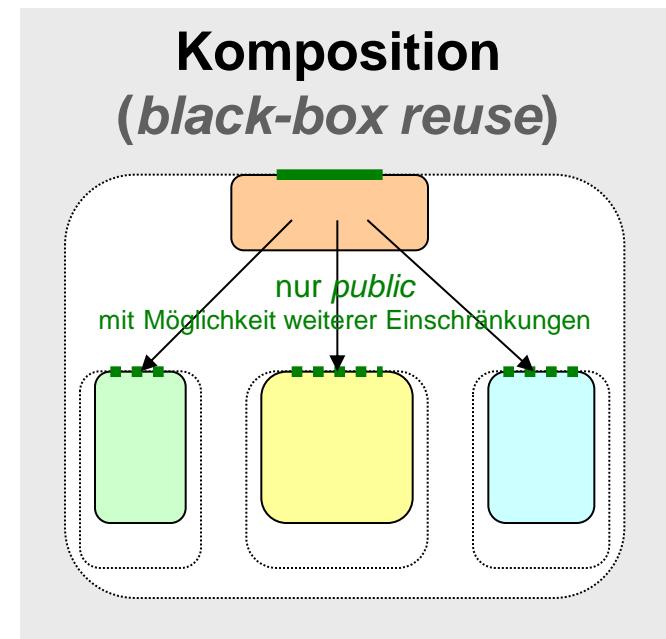
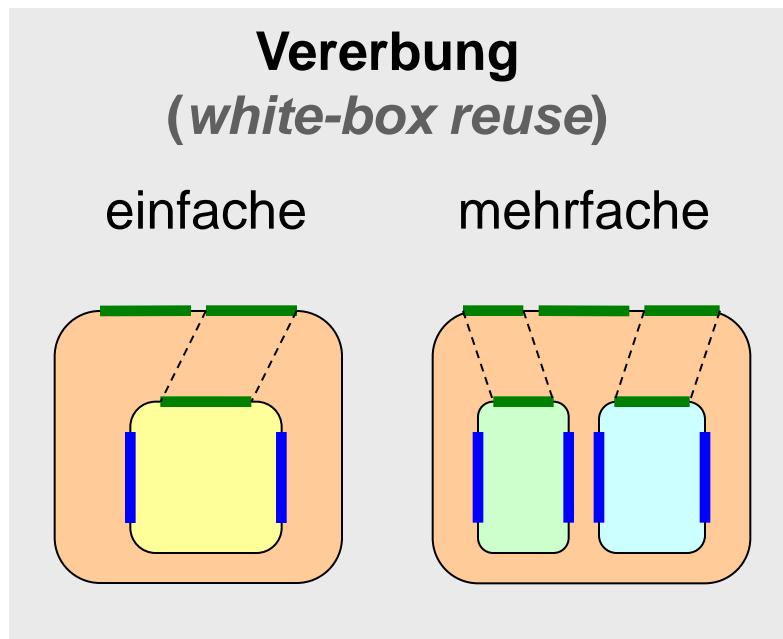
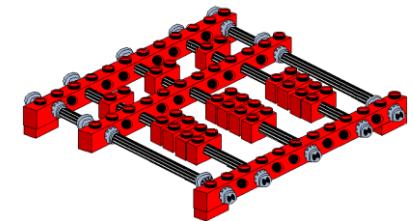
Beispiel 2

```
Iterator it = c.iterator(); // returns obj. of a class  
while (it.hasNext()) ... // implementing Iterator
```

oder: `for (Object o : c) ... // since Java 1.5`

Prinzip 2: "Favor [object] Composition over [class] Inheritance" (FCoI)

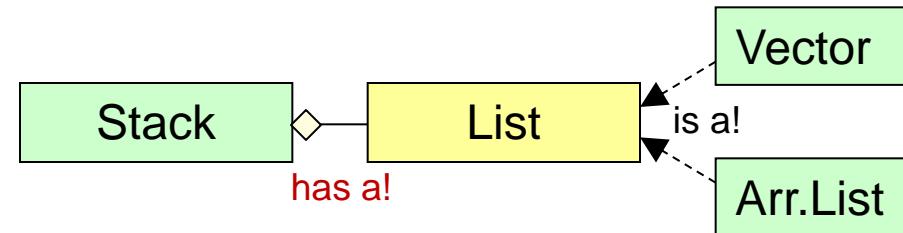
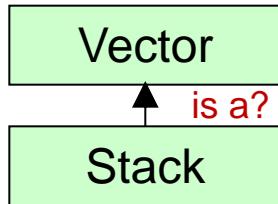
- Besser mächtigere Objekte aus kleineren Objekten bereits vorhandener Klassen zusammenbauen, als neue Klassen von vorhandenen Klassen ableiten
- Dazu müssen genügend "kleine" Klassen für die Erstellung flexibler Objektgraphen zur Verfügung stehen ("Lego-Prinzip")



Vererbung	Komposition
<p>Neue Funktionalität durch Vererbung von Eigenschaften an abgeleitete Klasse, haben i. d. R. Zugriff auf Internas der Basisklasse (white-box reuse)</p> <p>Vorteile:</p> <ul style="list-style-type: none"> ▪ Statisch definiert ▪ Ermöglicht statische Typprüfung ▪ Direkt durch oo Programmiersprachen unterstützt ▪ Wiederverwendung durch Ableitung weiterer Klassen möglich <p>Nachteile:</p> <ul style="list-style-type: none"> ▪ Keine Anpassungen zur Laufzeit mehr möglich ▪ Basisklasse gibt abgeleiteter Klasse Implementierung vor 	<p>Neue Funktionalität durch geschickten Zusammenbau von passenden Objekten bereits vorhandener "kleiner" Klassen (black-box reuse)</p> <p>Vorteile:</p> <ul style="list-style-type: none"> ▪ Nur wenige Implementierungs-abhängigkeiten ▪ Austauschbarkeit zur Laufzeit (dynamische Anpassung) ▪ Nutzung von Polymorphismus und dynamischer Bindung <p>Nachteile:</p> <ul style="list-style-type: none"> ▪ Explizite Objektkonstruktion muß zur Laufzeit erfolgen ▪ Erfordert sorgfältig entworfene Schnittstellen und hohe Granularität

Prinzipien für guten oo Entwurf (2.c)

Beispiel aus dem *Java Collection Framework (JCF)* in *java.util*



```
class Stack // publ. inh.  
extends Vector {  
    ...  
} // Stack
```

```
class Stack {  
    (prot.|priv.) List l;  
    ...  
} // Stack
```

besser

Beispiel aus der *Standard Template Library (STL)* für C++

```
template <class T, class Container>  
class stack {  
protected:  
    Container c;  
    ...  
} // stack
```

besser

```
...  
Container *c;  
...
```

Prinzip 3: "Delegate wherever possible"

- Mehrere Obj. "arbeiten zusammen": Empfänger leitet Anfrage weiter
- Delegation gibt Objektgruppen gleiche Mächtigkeit der Wiederverwendung wie Ableitung einer Klasse
- Gute Anwendung von Komposition im Gegensatz zu Vererbung

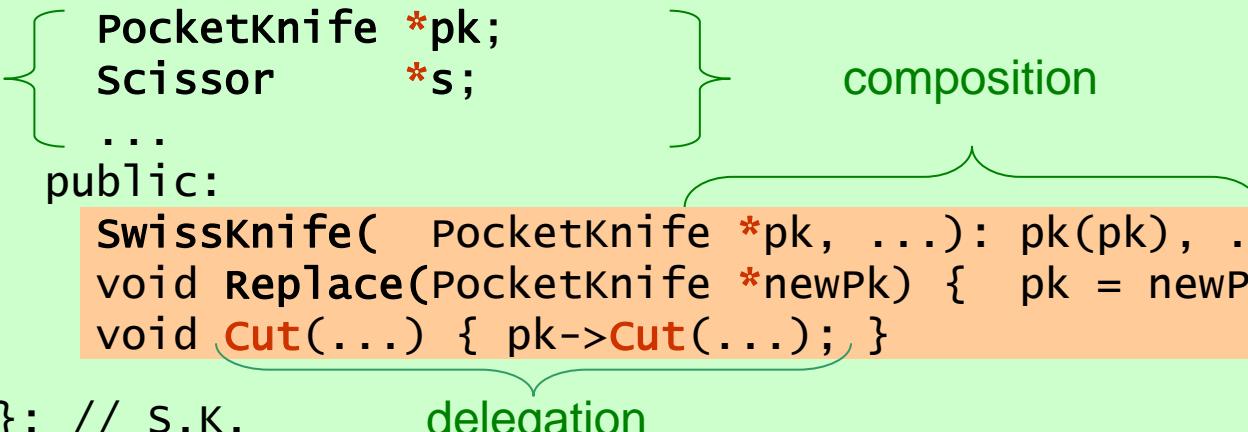
Vorteil: hohe Flexibilität und änderungsfreundlich (Änd. lokal mögl.)

Nachteil: schlechtere Laufzeiteffizienz

Beispiel für Komposition und Delegation: Schweizermesser ohne V.

```
class SwissKnife { // no base class any more
    protected:
        PocketKnife *pk;
        Scissor      *s;
        ...
    public:
        SwissKnife( PocketKnife *pk, ...): pk(pk), ... { }
        void Replace(PocketKnife *newPk) { pk = newPk; }
        void Cut(...) { pk->Cut(...); }
}; // S.K.
```

interface? composition delegation



Exkurs: Viele weitere Prinzipien für guten oo Entwurf



Hier angeordnet gemäß der Initiative *Clean Code* (siehe gleichnam. Buch von Robert C. Martin oder im Web, z. B. www.clean-code-developer.de oder cleancode.sourceforge.net):

- **Don't repeat yourself** (DRY): Vermeide Code-Verdopplung
 - **Keep it simple, stupid** (KISS): Mache Dinge so einfach wie möglich, aber nicht einfacher (A. Einstein)
 - **Single Level of Abstraction** (SLA): Alle Anweisungen in einer Methode sollen auf dem gleichen Abstraktionsniveau liegen (vgl. schrittweise Verfeinerung)
 - **Single Responsibility Principle** (SRP): Objekte einer Klasse sollen *nur eine* "Verantwortlichkeit" haben (bewirkt Lokalität von Änderungen), vgl. Aspekt-orientierte Programmierung (AOP)
 - **Separation of Concerns** (SoC): Trennung der "Belange" so, dass Objekte einer Klasse sich nur um einen "Belang" (oder einen Aspekt i.S.d. AOP) kümmern müssen ("Belange" sollten orthogonal sein)
 - **Interface Segregation Principle** (ISP) = Program to an Interface, not to an Implementation
 - **Dependency Inversion Principle**: Abhängigkeiten zwischen Objekten sollen so spät als möglich gesetzt werden, z. B. durch Konstruktorparameter, vgl. AOP: Depend. Injection (DI) / Inv. of Control (IoC)
 - **Principle of Least Astonishment**: Verwendung einer Klasse / eines Objekts soll zu keinen "Überraschungen" führen, z. B. in Form von Nebenwirkungen
 - **Open/Closed Principle**: Klassen sollen offen für Erweiterungen (neue Funktionalität) jedoch geschlossen gegenüber Modifikationen sein (Modifikationen nur an wenigen, "ungefährlichen" Stellen)
 - **Tell, don't ask**: Objekte sollen viel Funktionalität (zum Aufrufen) und wenig Daten (zur Abfrage) bieten
 - **Law of Demeter** (Don't talk to strangers): Abhängigkeiten zwischen Objekten sollen minimiert werden (innerhalb einer Methode sollten nur Methoden der eigenen Klasse, der Parameter, von assoziierten Klassen und von selbst erzeugten Objekten aufgerufen werden)
 - **You Ain't Gonna Need It** (YAGNI): Implementiere nur das, was gefordert und/oder nutzbringend ist
- SOLID-Prinz.: Single Respons. + Open Closed + Liskov Substit. + Interface Segreg. + Depend. Inversion**

Erzeugungsmuster (*creational patterns*), 5 Stk.

- Abstrahieren den Klassen- oder **Objekterzeugungsprozess** durch Vererbung für Klassen- oder Delegation für Objekterzeugung
- Erzeugungsmuster brauchen zwar Wissen über konkrete Klasse, aber nur lokal und nur an einer Stelle

Strukturmuster (*structural patterns*), 7 Stk.

- Beantworten Frage: Wie müssen Klassen oder Objekte zusammengebaut werden, um **mächtigere Strukturen (Statik)** zu bilden?
- Unterscheidung zwischen Klassen- und Objektstruktur
 - Darstellung d. Klassenstruktur durch Vererbung
 - Darstellung d. Objektstruktur durch Beschreibung der Implementierung

Verhaltensmuster (*behavioral patterns*), 11 Stk.

- Algorithmen und Verantwortlichkeiten spielen Hauptrolle
- Beschreiben **Dynamik** des Zusammenwirkens – Kommunikat.Muster
- Ziel ist möglichst lose Kopplung (keine direkte Verbindung)
- Unterscheidung zwischen Klassen- und Objektzusammenhängen
 - Klassenmuster durch Vererbung
 - Objektmuster durch Komposition

Übersicht: Erzeugungsmuster (5 Stk.)

Abstract Factory oder Kit

has some Schnittstelle zur Herstellung von Familien verwandter oder voneinander abhängiger Objekte ohne Angabe der konkreten Klasse

Builder

Trennung der Erzeugung eines komplexen Objekts von seiner Repräsentation ⇒ gleicher Erzeugungsprozess kann auch unterschiedliche Repräsentationen herstellen

Factory Method oder Virtual Constructor

Abstrakte Schnittstelle zur Objekterzeugung; erst abgeleitete Klassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist

Prototype

may be a Definition einer Art von Objekten über ein prototypisches Objekt;
Herstellung neuer Objekte durch kopieren (nicht durch konstruieren)

Singleton

Klasse mit max. einem Objekt und geregeltem Zugriff darauf

Übersicht: Strukturmuster (7 Stk.)

Adapter oder Wrapper

Anpassung d. Schnittstelle einer Klasse an Schnittstelle, die Klienten erwarten

Bridge

Entkopplung einer Abstraktion von einer Implementierung, beide können unabhängig voneinander verändert werden

Composite

Hierarchische Anordnung von Objekten, einzelne Objekte und Sammlungen von Objekten können gleich behandelt werden

Decorator oder Wrapper

Zusätzliche Aufgaben können dynamisch an Objekte angebunden werden

Facade

Einheitliche Schnittstelle für Menge von Schnittstellen eines Teilsystems; Schnittstelle auf höherer Ebene, vereinfacht Verwendung eines Teilsystems

Flyweight oder Shared Object

Mehrfachnutzung eines Objekts, um viele gleichartige Objekte zu modellieren, Unterscheidung zwischen internem u. externem Zustand: mehrfach genutzte Objekte haben nur int. Zustand, ext. muss von Klienten kommen

Proxy

Platzhalter für ein anderes Objekt, um den Zugriff darauf zu regeln

Übersicht: Verhaltensmuster (11 Stk., Teil 1)

Chain of Responsibility

Vermeidung direkter Kopplung eines Senders einer Anforderung an einen Empfänger; mehrere Objekte bekommen Möglichkeit, Anforderung zu behandeln; Reihung potentieller Empfänger, Anforderungen werden entlang der Reihe gesandt

Command

Anforderung wird als Objekt mit *Execute*-Methode modelliert; Möglichkeit der Parametrisierung, Speicherung und Protokollierung

Interpreter

Definition der syntakt. Darstellung einer Sprache (gem. einer Grammatik) und Möglichkeit zur Auswertung der Sätze; Darstellung einer Grammatik in Form einer Klassenhierarchie, Interpretierer arbeitet auf einer Instanz der Klassenhierarchie (Objektgraph)

Iterator

Möglichkeit alle Elemente einer Sammlung zu behandeln, ohne interne Darstellung (Datenstruktur) kennen zu müssen

Mediator

Definition der Interaktion einer Menge von Objekten; erlaubt lose Kopplung von Objekten (Objekte referenzieren sich nicht direkt); Zusammenarbeit kann unabhängig voneinander variiert werden

Übersicht: Verhaltensmuster (11 Stk., Teil 2)

Memento

Ohne Datenkapselung zu verletzen: Möglichkeit den internen Zustand eines Objekts mit *GetState* festzuhalten z. B. damit Objekt später wieder mit *SetState* in diesen Zustand gebracht werden kann

Observer oder Publisher/Subscriber

Definition einer $1:n$ -Beziehung zwischen Objekten; wenn ein Objekt Zustand ändert, werden n Objekte benachrichtigt und autom. angepasst (z. B. MVC)

State

Ermöglicht Objekten ihr Verhalten zu ändern, wenn sich ihr interner Zustand ändert; so als hätte sich Klasse des Objekts geändert: Zustand ist Objekt mit Methode *Handle* und kann zur Laufzeit ausgetauscht werden

Strategy

Familie von Algorithmen die austauschbar sind; Strategien können unabhängig von Klienten verschiedene Algorithmen verwenden

Template Method

Gerüst eines Algorithmus wird festgelegt: abstrakte Definition eines Algorithmus durch Schritte, jeder Schritt ist entweder Aufruf einer abstrakten Methode oder ein primitiver Schritt; Details werden erst in abgeleiteten Klassen spezifiziert

Visitor

Operation, die auf die Elemente einer Sammlung anzuwenden ist

"Raum" der Entwurfsmuster

		Zweck		
		Erzeugung	Struktur	Verhalten
Bereich	Klasse	Factory Method	Adapter	Interpreter Template Method
	Objekt	Abstract Factory	Adapter	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

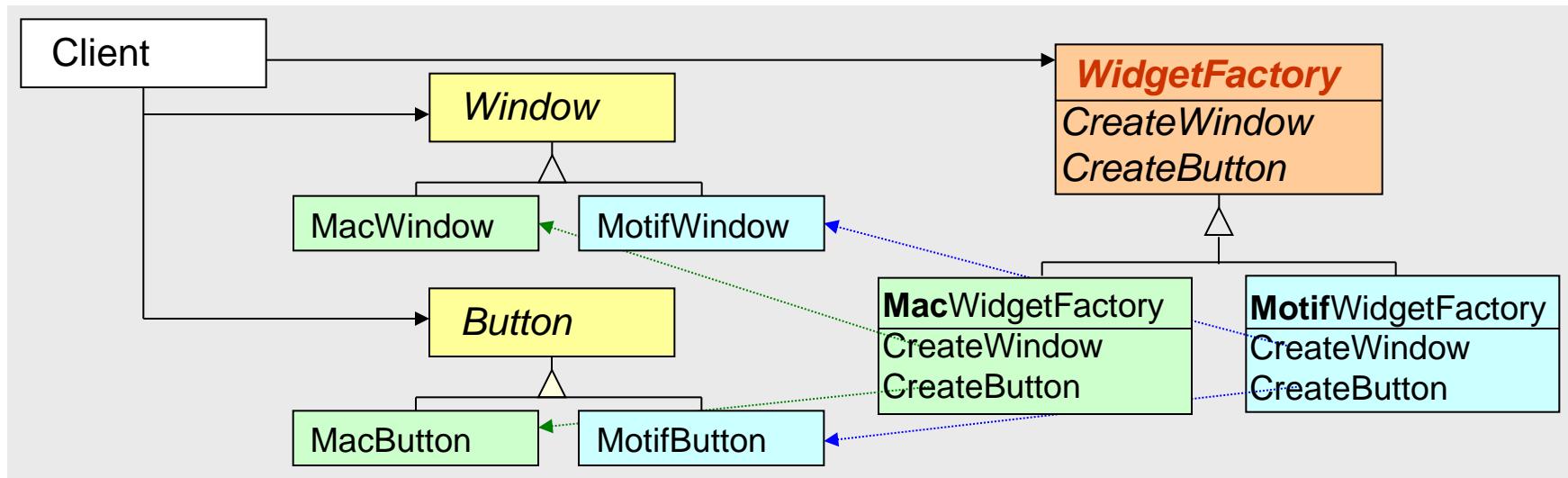
Erzeugungsmuster *Abstract Factory* oder *Kit* (1)

Zweck

Abstrakte Schnittstelle zur Erzeugung von Familien verwandter oder voneinander abhängiger Objekte ohne konkrete Klassen angeben zu müssen

Motivation

Eine Anwendung für unterschiedliche GUI-Standards (z. B. Mac und Motif)



Flexible Lösung durch Aufteilung auf abstrakte und konkrete Klassen:

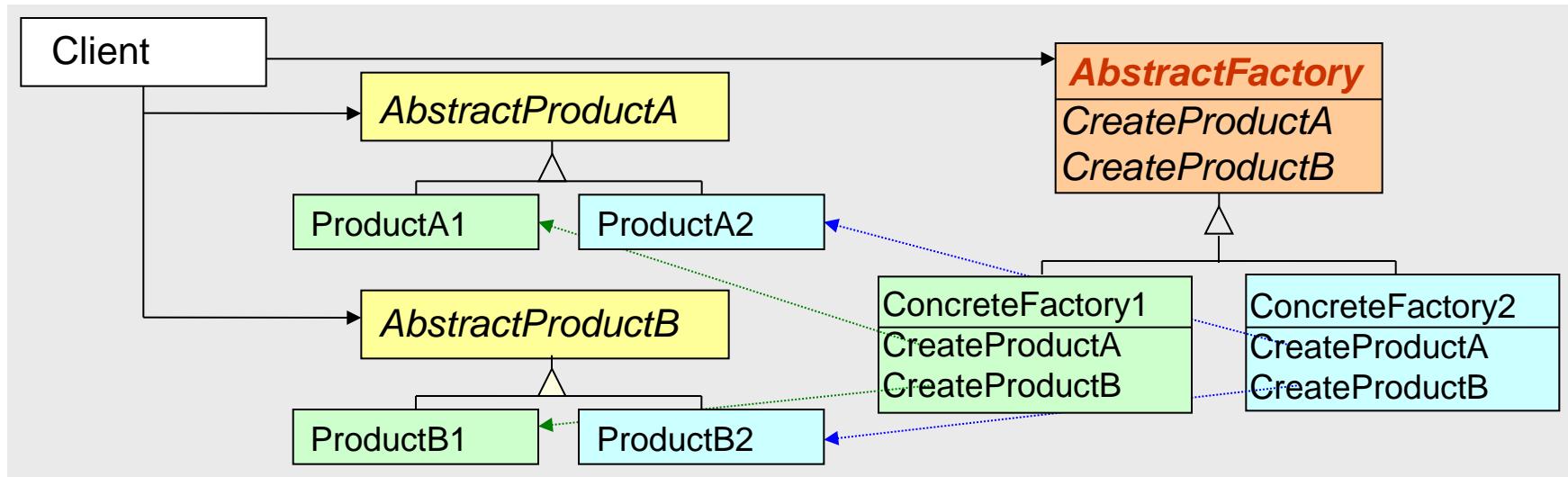
- Abstr. Klasse *WidgetFactory* hat zur Erzeug. v. GUI-Elementen je eine abstr. Methode
- Je eine abstrakte Klasse für jedes GUI-Element
- Je eine konkrete *Factory*-Klasse für jeden GUI-Standard
- Je eine konkrete Element-Klasse für jeden GUI-Standard

Erzeugungsmuster *Abstract Factory* oder *Kit* (2)

Anwendbarkeit: Immer wenn ...

- ein System unabhängig von einer bestimmten Familie von Objekten sein soll
- eine Konfiguration mit unterschiedlichen Produktfamilien gewünscht ist

Struktur



Konsequenzen

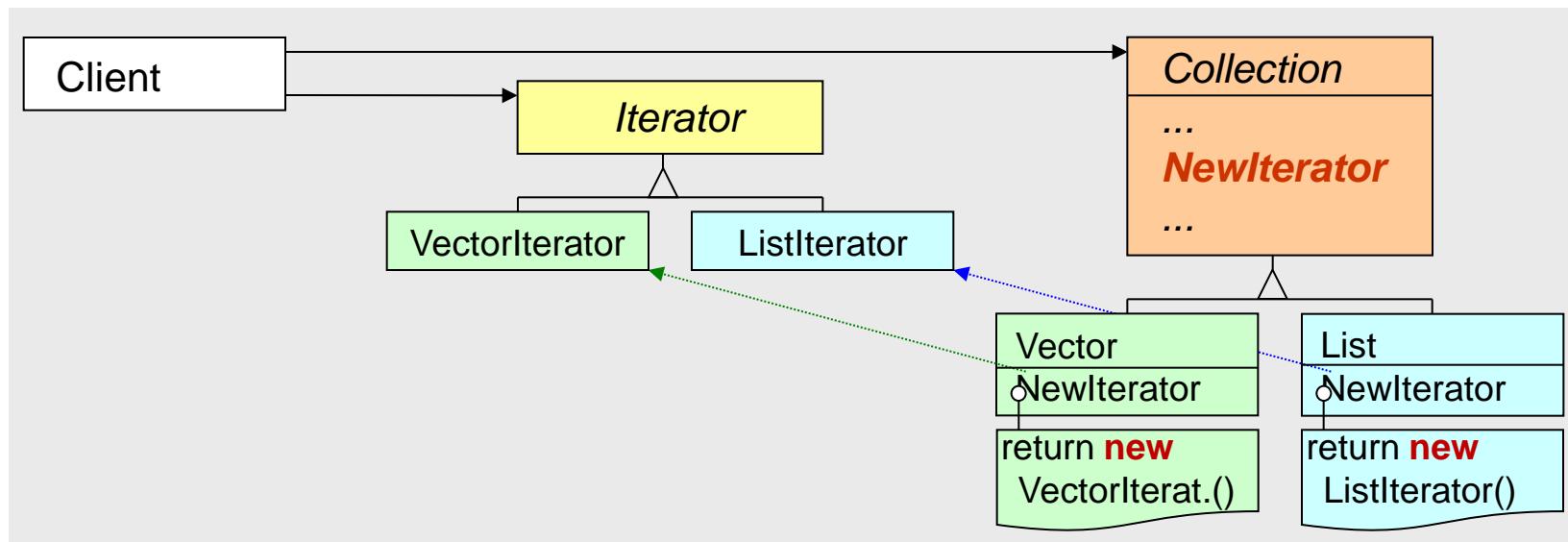
- Isolation konkreter Klassen von Anwendungen (*Client*)
- Produktfamilien können leicht ausgetauscht werden
- Förderung der Konsistenz
- Neue Produkte können nicht einfach hinzugefügt werden

Zweck

Abstrakte Schnittstelle für indirekte Objekterzeugung, so dass erst abgeleitete Klassen entscheiden, welche Objekte konkret erzeugt werden

Motivation

Anwendungen brauchen unterschiedliche Behälter, Iteratoren erlauben einheitliche Abarbeitung der Elemente, allerdings müssen konkrete Iteratoren über konkret verwendete Datenstrukturen Bescheid wissen



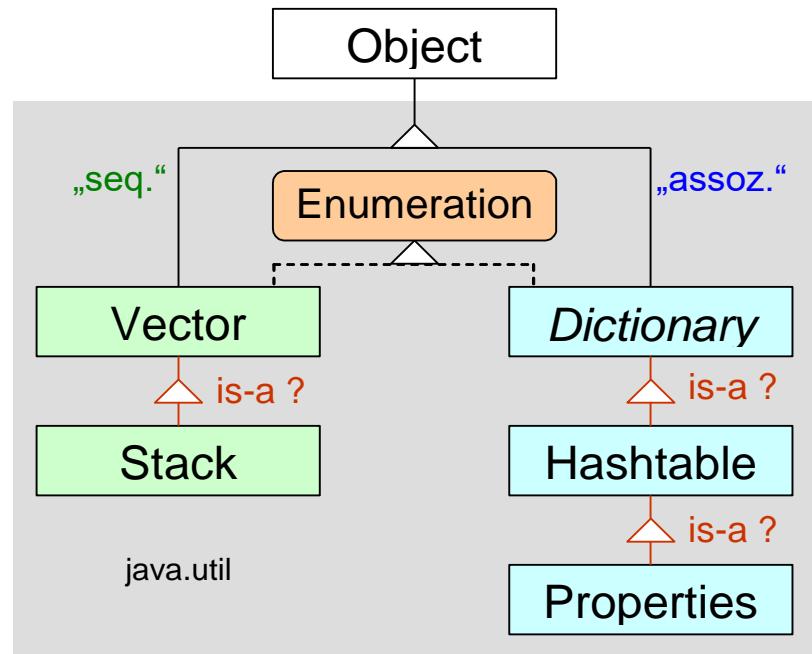
Factory-Methode (hier NewIterator) ist abstrakt, erst in abgeleiteten Klassen gibt es konkrete Methoden, die Objekte aus konkreten Klassen erzeugen

Exkurs: Behälter in Java 1.0 mit *Enumeration*

- Schon seit Java 1.0 gibt es Behälterklassen in der Java-Standardbibliothek
- Zwei "sequentielle" und zwei (konkrete) "assoziative" Behälterkl. im Paket *java.util*
- Unübliches Iterator-Konzept mit Schnittstelle *Enumeration*
- Schlechtes oo Design (*is-a?*)

Beispiel zu „seq.“ Behälter:

```
vector v = new vector();
v.add(...);
Enumeration e = v.elements();
while (e.hasMoreElements()) {
    Object o = e.nextElement();
    ... // do anything with o
} // while
```



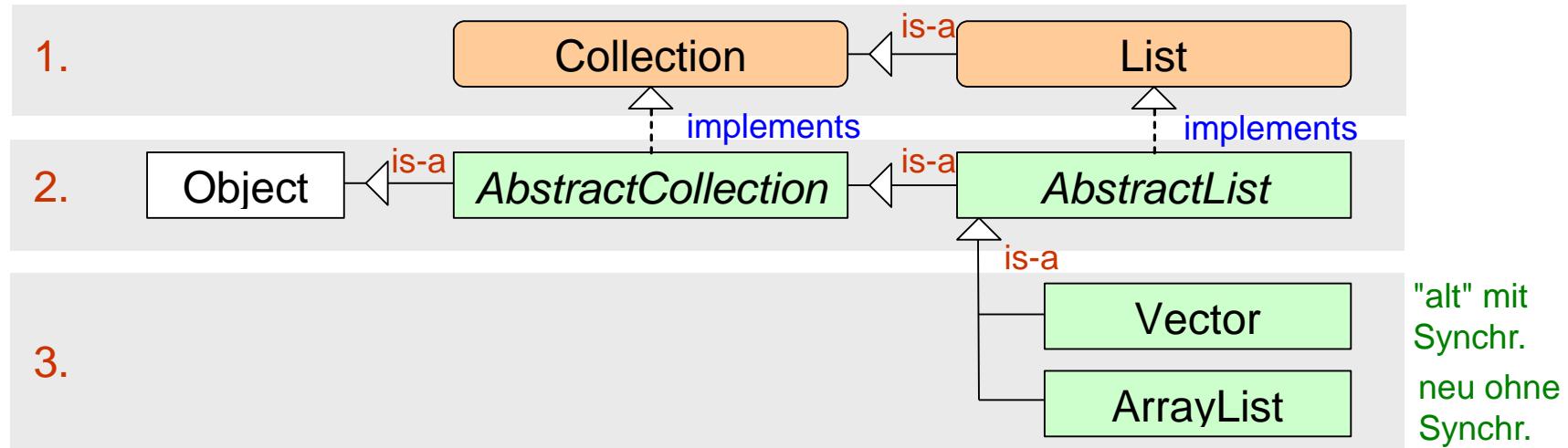
Beispiel zu „assoz.“ Behälter:

```
Dictionary d =
    new Hashtable();
d.put("one", "eins");
...
System.out.println(
    "german for one = " +
    d.get("one"));
```

Exkurs: Java Collection Framework (JCF) seit Java 1.2

- Mit Java 1.2 ("Java 2") umfangreiche Bibliothek mit 10 Schnittstellen und 13 Klassen für Behälter, Iteratoren (zusätzlich zum *Enumerator*-Konzept) und Algorithmen ⇒ *Java Collection Framework (JCF)*
- Dreischichtiger Aufbau (von sehr abstrakt bis konkret):
 1. Hierarchie von **Schnittstellen**
 2. Hierarchie **abstrakter Klassen**, die Schnittst. teilw. implement.
 3. Hierarchie **konkreter Klassen**, von abstrakten abgeleitet

Beispiele zum dreistufigen Aufbau: Ausschnitte aus dem JCF



Beispiel: *Factory Method* und *Iterator* im JCF (2)

```
public interface Collection {  
    int size();  
    boolean isEmpty();  
    boolean add(Object e);  
    boolean remove(Object e);  
    boolean contains(Object e);  
    Iterator iterator();  
} // Collection
```

```
public interface Iterator {  
    boolean hasNext();  
    Object next();  
    void remove();  
} // Iterator
```

Anwendung (seit Java 1.5 mit Generizität u. *foreach*-Schleife möglich):

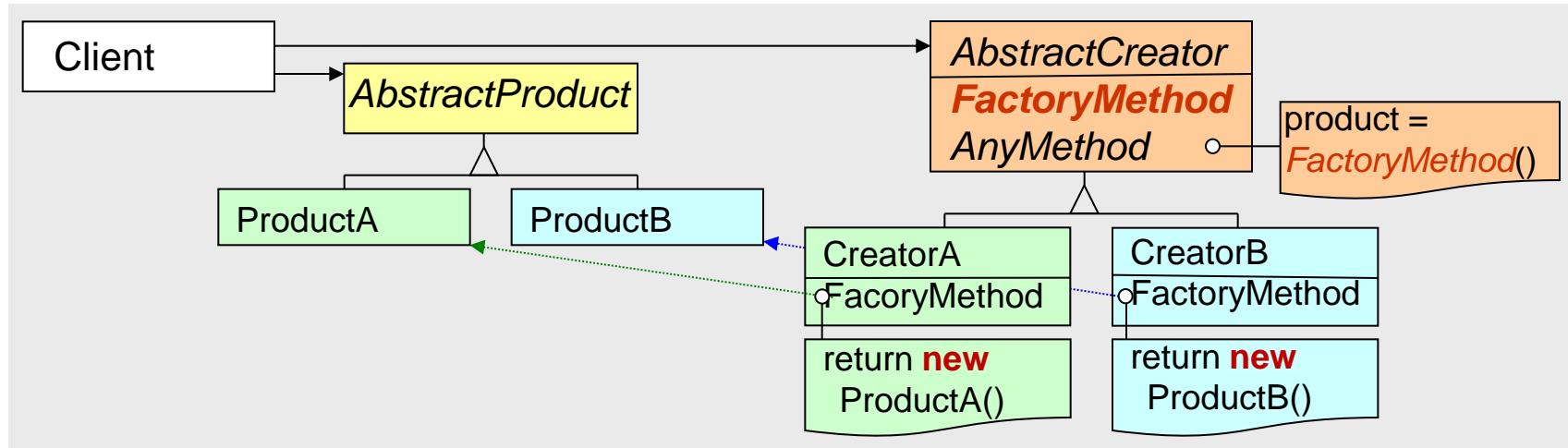
```
pub. interf. Iterable<T> {  
    Iterator<T> iterator();  
} // Iterable
```

```
Collection<String> c = new ArrayList<String>(); // program to ...  
c.add("any string"); // ... an interface not to an implementation  
  
...  
Iterator<String> it = c.iterator(); // alternatively ...  
while (it.hasNext()) { // ... using foreach:  
    String s = (String)it.next();  
    System.out.println(s);  
} // while  
for (String s : c) {  
    System.out.println(s);  
} // foreach
```

Erzeugungsm. *Factory Method* od. *Virt. Constr.* (2)

Anwendbarkeit: ist immer dann gegeben, wenn eine Klasse nicht vorhersehen kann, welche Objekte zur Laufzeit erzeugt werden müssen

Struktur



Konsequenzen

- Aufwendig: Für jede *ConcretProduct*-Klasse muss eigene *Creator*-Klasse mit eigener *FactoryMethod* implementiert werden
- Sehr flexibel: *FactoryMethod* kann schon Standardimplementierung haben, trotzdem Möglichkeit, Standardverhalten in abgel. Klasse zu überschreiben
- Verbindung "paralleler" Klassenhierarchien möglich

Beispiel: *Factory Method* bei Java Reflection

Java bietet reichhaltige Metainformation durch den Mechanismus *Reflection* (vor allem `java.lang.Class` und Paket `java.lang.reflect`)

Beispiel: (grün: "normale" Methoden, die Objekte liefern; rot: echte *Factory*-Methode)

```
java.io.Console con = System.console(); // since Java 1.6, wehere ...
con.printf("className > ");      // ... Console is a singleton, see below
String className = con.readLine();
Class<?> c = Class.forName(className);
con.printf(c.toString() + "\n");
Object o = c.newInstance(); // empty
con.printf("1. \\" + o.toString() + ...
if (c == String.class) { // needs no equals because of singleton
    Constructor sc = c.getConstructor(new Class[] {String.class});
    Object s = sc.newInstance(new Object[] {"Hello, world!"});
    con.printf("2. \\" + s.toString() + "\\n");
} // if
```

Ergebnis:

```
className > java.lang.String
class java.lang.String
1. ""
2. "Hello, world!"
```

Erzeugungsmuster *Prototype* (in Java)

Zweck

Definition einer Art von Objekten über ein prototypisches Objekt;
Herstellung neuer Objekte durch Kopieren (nicht mehr durch Konstruieren)

Java: Basisklasse *Object* bietet (geschützte) **clone**-Methode (liefert *shallow copy*)

Beispiel für eine Klasse deren Objekte dupliziert werden können

```
class Sheep extends object implements Cloneable {  
    private String name;  
    public Sheep(string name) { ... }  
    @Override // and protected -> public  
    public Sheep clone() { // using covariance  
        Object o = null;  
        try {  
            // delegation:  
            o = super.clone(); // object does the work  
        } catch (CloneNotSupportedException e) {  
            System.err.println("Sheep not cloneable");  
        } // catch  
        return (Sheep)o;  
    } // clone  
} // Sheep
```

Anwendung:

```
Sheep s = new Sheep("Dolly");  
...  
Sheep dolly = s.clone();  
...  
System.out.println(dolly);
```

Auch *deep copy* möglich:

- durch Erweiterung der Implementierung der *clone*-Methode oder
- mittels Serialisierung

Erzeugungsmuster *Singleton* (in C++)

Zweck

Klasse mit max. einem Objekt und geregeltem Zugriff darauf

```
class Singleton { // C++:  
  
private:  
    static Singleton *instance;  
public:  
    static Singleton *NewInstance() {  
        if (instance == nullptr)  
            instance = new Singleton();  
        return instance;  
    } // NewInstance  
  
private:  
    // Singleton data  
    Singleton() {  
        ...  
    } // singleton  
public:  
    // Singleton methods  
  
}; // Singleton  
  
Singleton *Singleton::instance = nullptr;
```

Verwendung:

```
singleton *s;  
  
s = new Singleton();  
...  
s = new Singleton();  
...  
s = Singleton::NewInstance();  
...  
s = Singleton::NewInstance();  
... // s points to the one and only
```

Problem:

```
...  
delete s;  
...
```

Erzeugungsmuster *Singleton* (in Java)

```
public class Singleton { // Java:  
  
    private static Singleton instance;  
  
    public static Singleton newInstance() {  
        if (instance == null)  
            instance = new Singleton();  
        return instance;  
    } // newInstance  
  
    private ... data;  
  
    private Singleton() {  
        ...  
    } // Singleton  
  
    // public object methods  
  
} // Singleton
```

Verwendung:

```
singleton s;  
...  
s = Singleton.newInstance();  
...  
s = Singleton.newInstance();  
// s is the one and only  
...
```

Weitere Möglichkeit ev. mit anonymer Klasse, wenn ... :

```
Object s = new CoI() {  
    ...  
}; // anonymous class
```

Achtung:

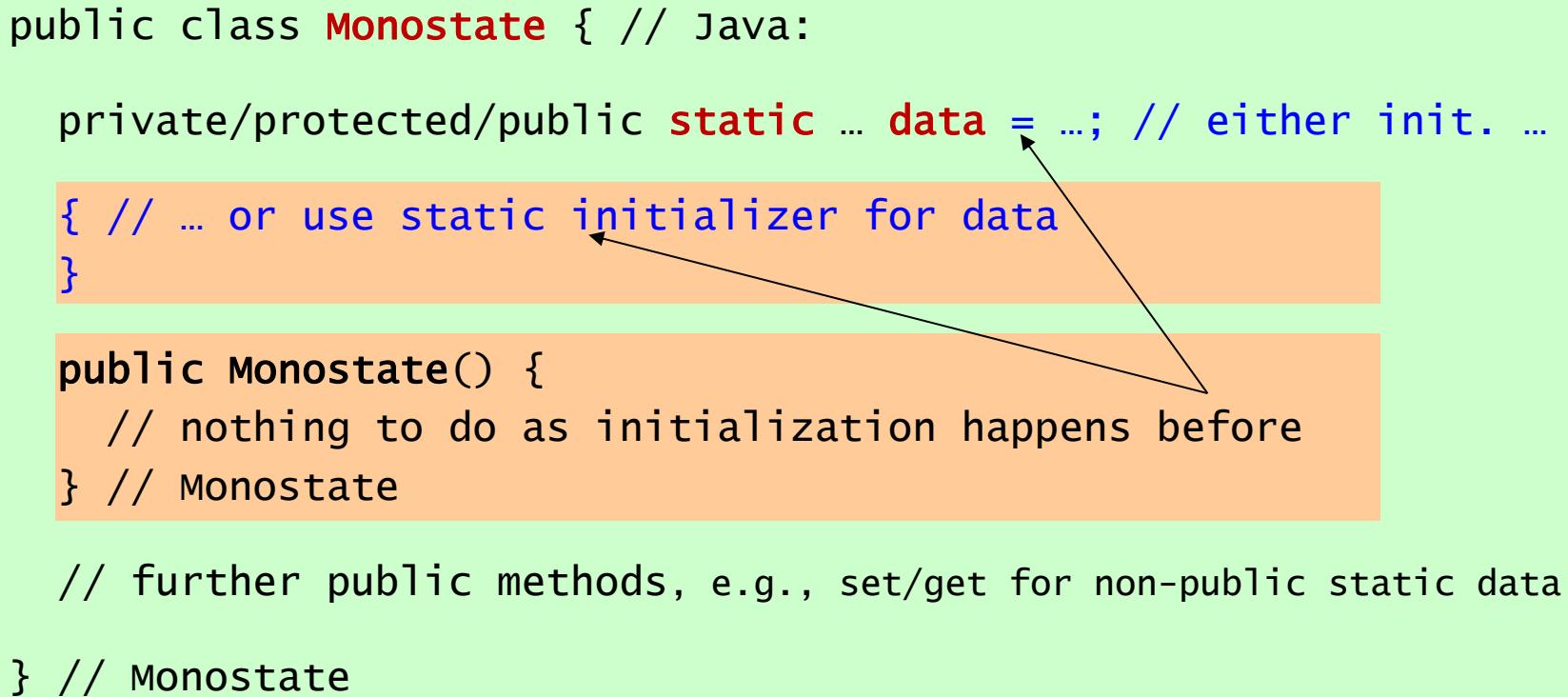
- *Singleton* ist globales Objekt und
- sollte *thread-safe* sein

Kritikpunkte am *Singleton*-Muster:

- unübliche Objekterzeugung: *Singleton.newInstance()* anstelle von *new S()*
- "Speicherleiche", da auch der *Garbage Collector* das eine *Singleton*-Objekt nicht freigehen kann, da es über *instance* weiterhin eine Referenz darauf gibt

Alternative: *Monostate* (mehrere Objekte, aber alle teilen sich einen Zustand)

```
public class Monostate { // Java:  
  
    private/protected/public static ... data = ...; // either init. ...  
    { // ... or use static initializer for data  
    }  
  
    public Monostate() {  
        // nothing to do as initialization happens before  
    } // Monostate  
  
    // further public methods, e.g., set/get for non-public static data  
}  
} // Monostate
```



The diagram illustrates the code structure for the Monostate pattern. It consists of several colored regions and arrows:

- A light green background covers the entire code area.
- A light orange rectangular box highlights the static variable declaration: `private/protected/public static ... data = ...;`
- A light orange rectangular box highlights the constructor: `public Monostate() {`
- A light orange rectangular box highlights the closing brace: `} // Monostate`
- A blue arrow points from the text "use static initializer for data" towards the static variable declaration.
- A blue arrow points from the text "or use static initializer for data" towards the constructor.

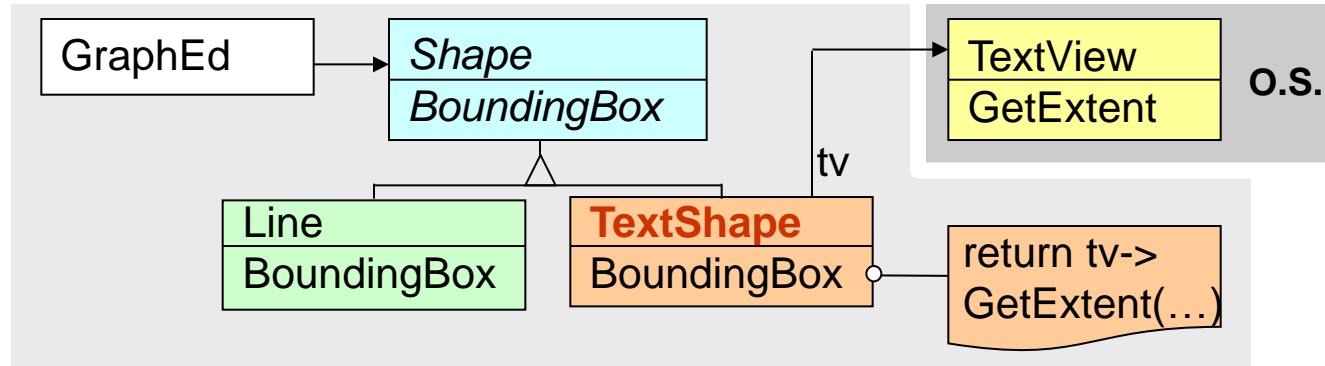
Strukturmuster Adapter oder Wrapper (1)

Zweck

- Konvertierung der Schnittstelle einer Klasse in solch eine, die Klienten erwarten
- Damit können Klassen/Objekte trotz inkompatibler Schnittstellen kooperieren

Motivation

Grafikeditor beherrscht bereits grafische Objekte; auch Text soll bearbeitet werden: vorhandene Klasse *TextView* soll an *Shape* angepasst werden



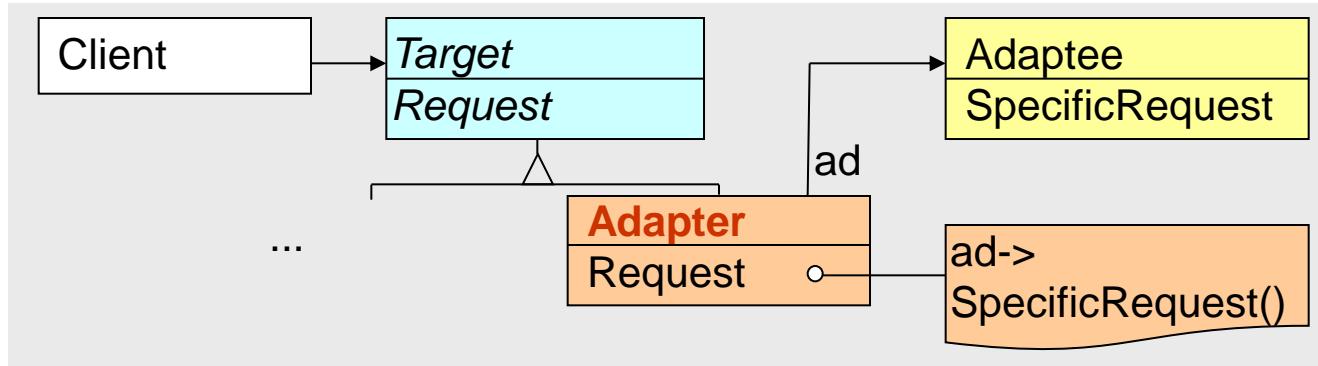
TextShape ist Objektadapter (*Wrapper*) für *TextView*

Anwendbarkeit: Immer dann wenn ...

- Existierende Klassen verwendet werden sollen, deren Schnittstellen nicht passen (Klassenadapter)
- Objekte existierender Klassen verwendet werden sollen, weil es unpraktisch wäre, durch Ableitung deren Schnittstellen anzupassen (Objektadapter)

Strukturmuster *Adapter* oder *Wrapper* (2)

Struktur **Objektadapter**

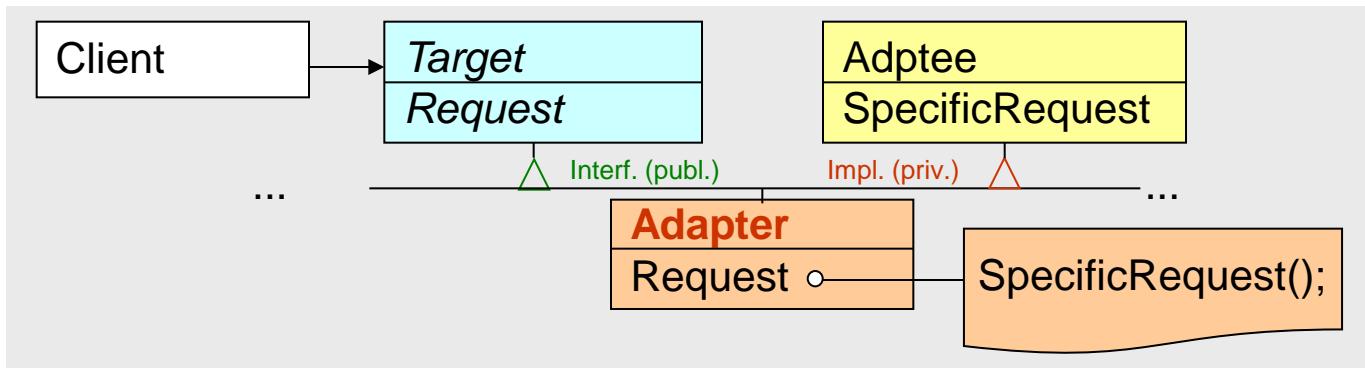


Konsequenzen für/aus Objektadapter

- Objektadapter kann mit Objekten einer Klasse und aller davon abgeleiteten Klassen zusammenarbeiten
- Verhalten der zu adaptierenden Klasse kann nur schwer verändert werden (nur durch Ableiten weiterer Klassen)

Strukturmuster *Adapter* oder *Wrapper* (3)

Struktur **Klassen**adapter



Konsequenzen für/aus Klassenadapter

- Adaptiert *nur eine* bestimmte Klasse; funktioniert nicht, wenn eine Klasse und alle davon abgeleiteten Klassen adaptiert werden sollen
- Im Klassenadapter kann das Verhalten der zu adaptierenden Klasse angepasst werden (z. B. durch Ableitung weiterer Adapterklassen)
- Nur ein zusätzliches Objekt, keine Indirektion über Zeiger/Referenz auf das Objekt der zu adaptierenden Klasse

Beispiele: Einfache Adapter bzw. Wrapper

Aus der MiniLib (C++-Version):

```
namespace m1 {  
    class string: public object {  
        protected:  
            std::string cppStr;  
        ...  
    }; // string  
}; // m1
```

Klassen-
oder
Objekt-
Adapter

Aus der Java-Standardbibliothek:

```
package java.lang;  
public final class Integer  
    extends Number  
    implements Comparable<Int.>, ... {  
        private final int value;  
        ...  
    } // Integer
```

Strukturmuster *Composite* (1)

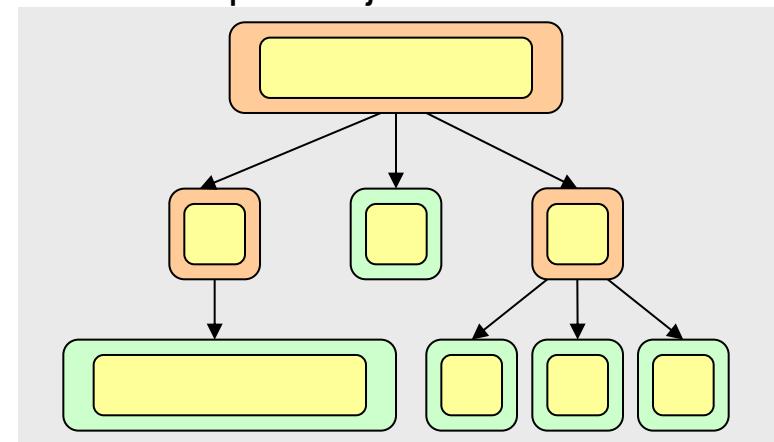
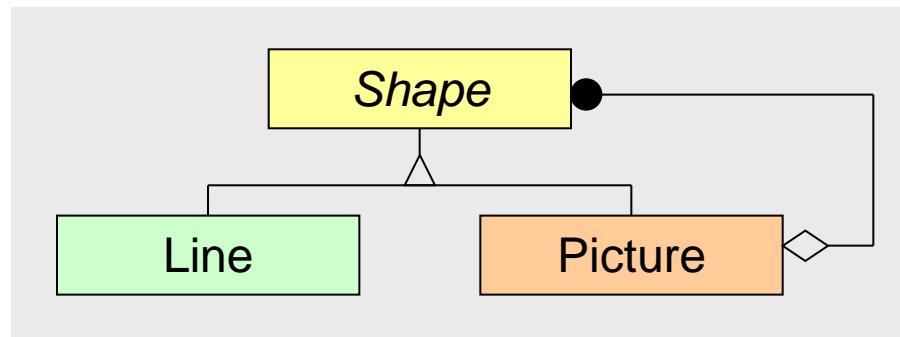
Zweck

- Hierarchische Anordnung von Objekten, Darstellung der Teil-Ganzes-Beziehung
- Klienten können einzelne Objekte und Sammlungen davon gleich behandeln

Motivation

Grafikeditoren, die einzelne grafische Objekte zu Bildern zusammenbauen und diese identisch behandeln wollen

Beispiel-Objekthierarchie:



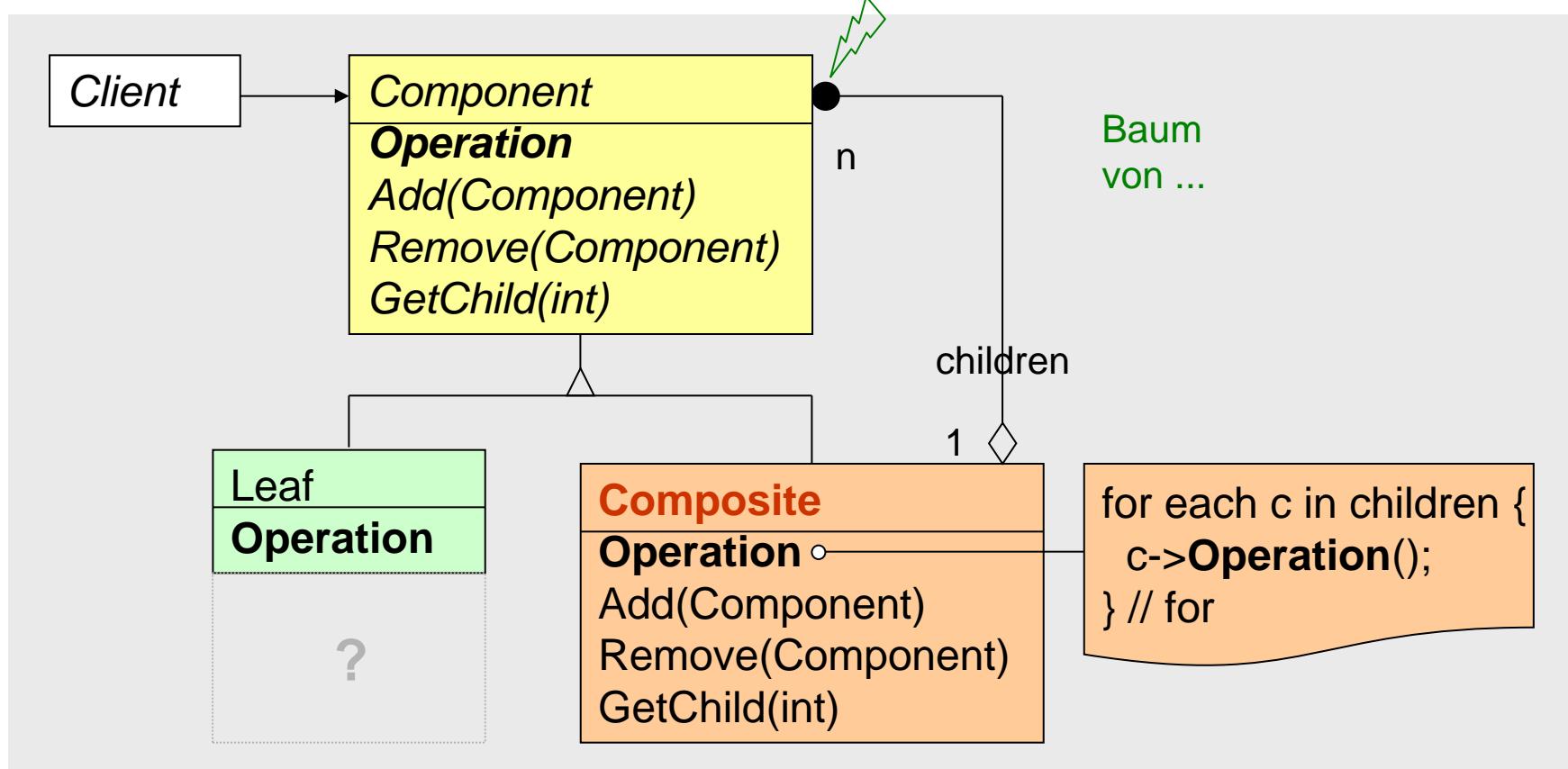
Idee der "rekursiven Aggregation"

Anwendbarkeit

- Darstellung von Teil-Ganzes-Beziehungen
- Klienten sollen Unterschiede zwischen einzelnen Objekten und Sammlungen von Objekten ignorieren können (Klienten können alle Bestandteile im *Composite* gleich behandeln)

Strukturmuster *Composite* (2)

Struktur



Konsequenzen

- Client wird einfach (durch uniforme Behandlung)
- Neue Komponenten können leicht dazu gefügt werden
- Beschränkungen bei den Komponenten können nur durch Laufzeitprüfungen (nicht statisch über das Typsystem) eingeführt werden

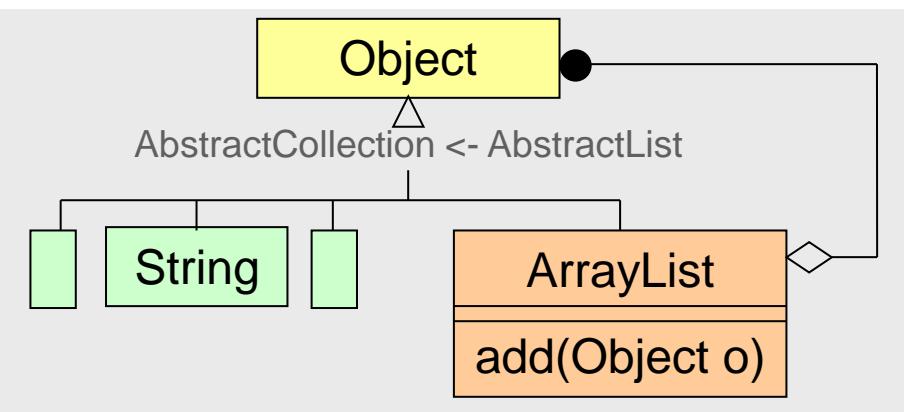
Implementierungsdetails

- Explizite Referenzen auf Elternknoten sind für Traversierung (*bottom-up*) und/oder für Ereignisbehandlung (über *Chain of Responsibility*) wichtig (üblicherweise in der Klasse *Component* angesiedelt)
- Dadurch wird Mehrfachnutzung von Komponenten (*component sharing*) aber erschwert
- Verwaltung der Kindknoten: Implementierung in der Klasse *Composite*, aber wo die Deklaration?
 - In der *Component*-Klasse: schafft Uniformität der Schnittstellen, ermöglicht Klienten "sinnlose" Operationen auf Blättern
 - In der *Composite*-Klasse: "sinnlose" Operationen werden zur Übersetzungszeit erkannt, aber Blätter und Sammlungen haben unterschiedliche Schnittstellen
- Weitere Fragen: Wie und wo werden Kindknoten verwaltet? (z. B. als Liste, aber wo wird Liste angesiedelt?) Werden die Kindknoten geordnet? Wer ist für das Freigeben verantwortlich?

Beispiele: *Composite* bei Behältern und GUI

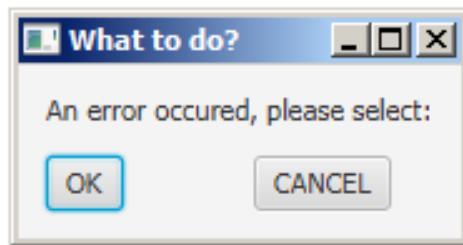
(Polymorphe) Behälterklassen:

- Behälter nehmen Objekte auf
- Behälter sind selbst Objekte
- Behälter können Elemente in Behältern (hoffentlich anderen;-) sein

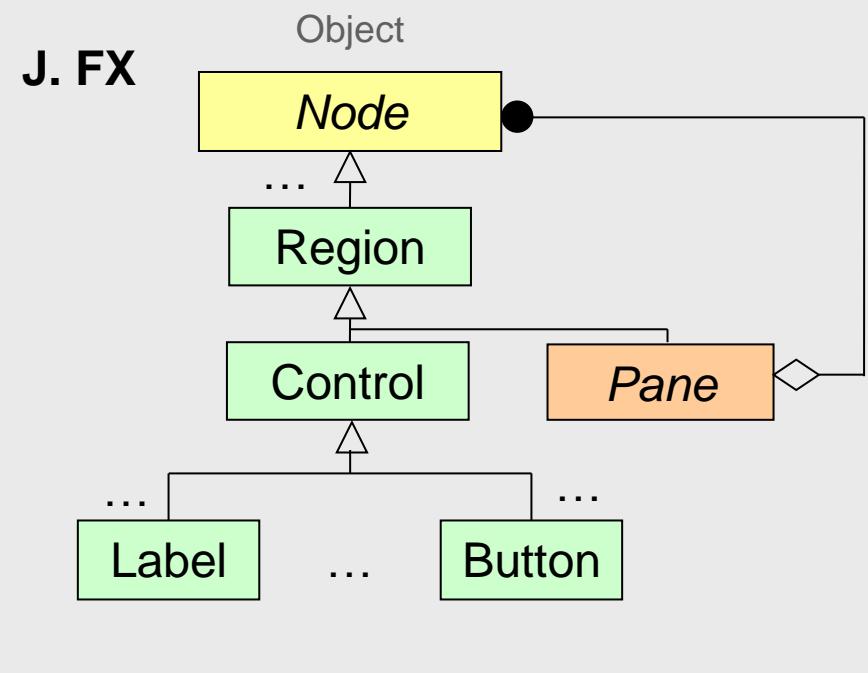
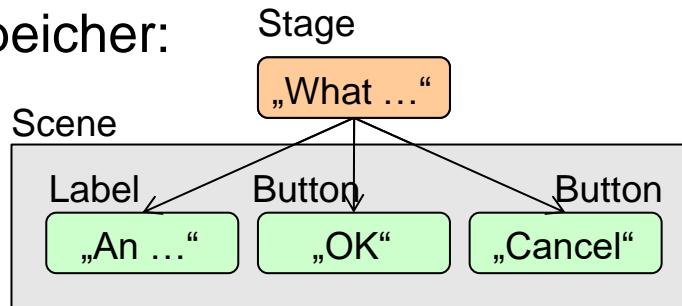


GUI-Steuerelemente:

Am
"Schirm":



Im Speicher:

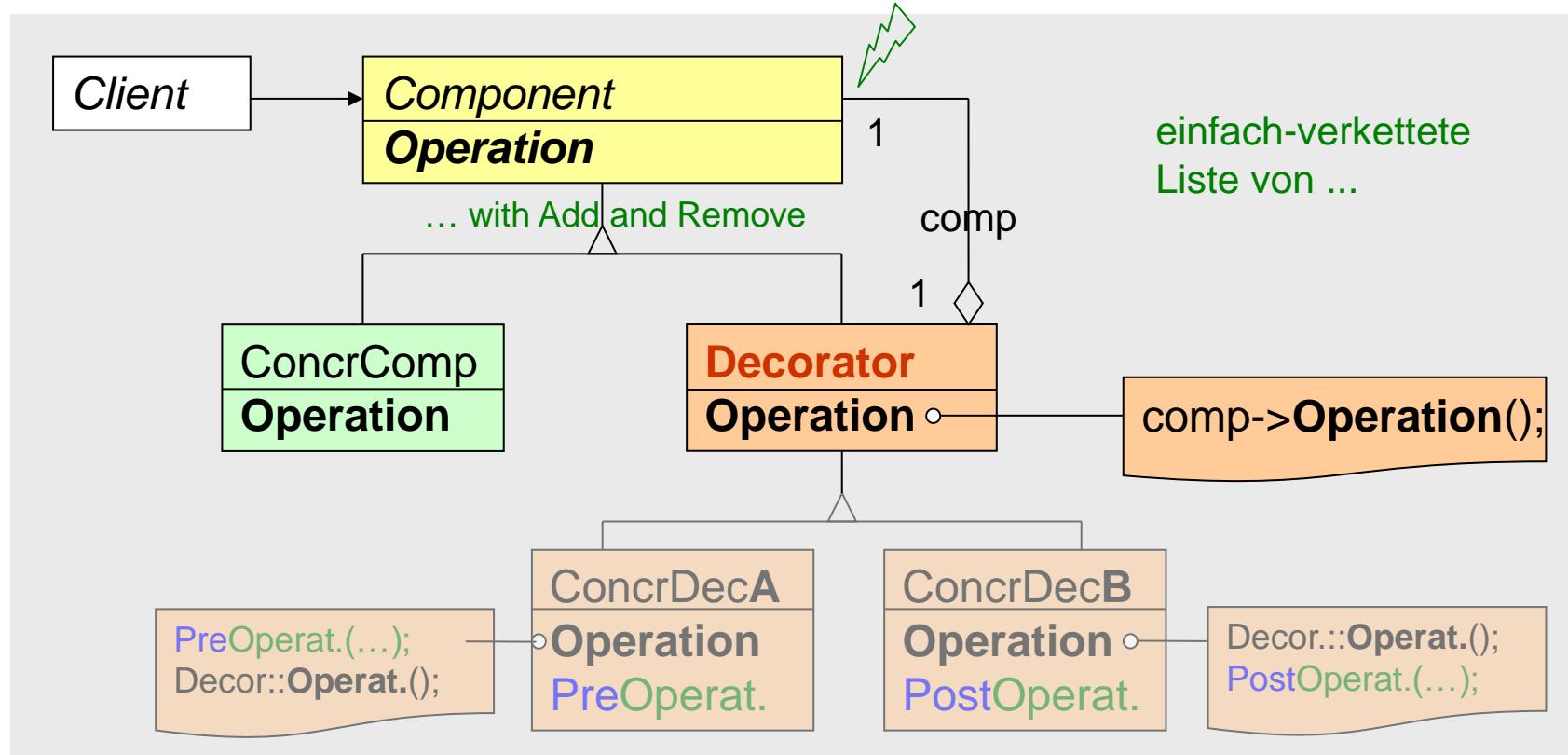


Strukturmuster *Decorator*

Zweck

Zusätzliche Aufgaben können dynamisch an Objekte angebunden werden

Struktur



Beispiel: *Decorator* in *java.io*

Beispiel: Konsolen-Eingabe in Java (im Paket *java.io*)

```
InputStream      is  = System.in;           // reads bytes
InputStreamReader isr = new InputStreamReader(is); // bytes -> chars
BufferedReader  br   = new BufferedReader(isr);    // chars -> String

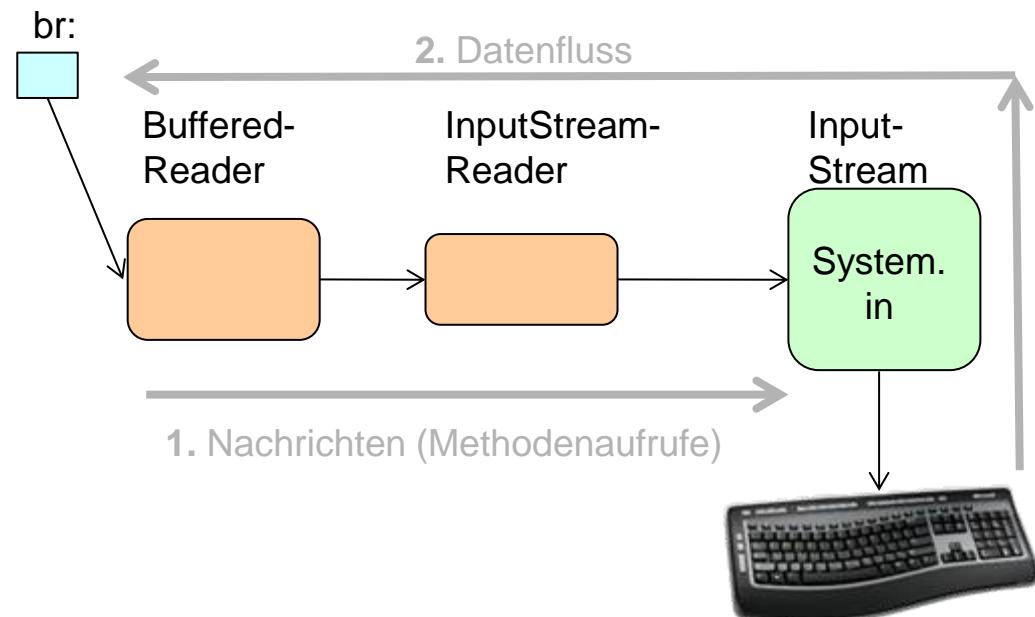
// shorter: br = new BufferedReader(new InputStreamReader(System.in));

String line;
```

```
try {
    line = br.readLine();
} catch (IOException e) {
    System.err.println(e);
} // catch
```

```
// alternatively, since 1.6:

Console con = System.console();
line = con.readLine();
```

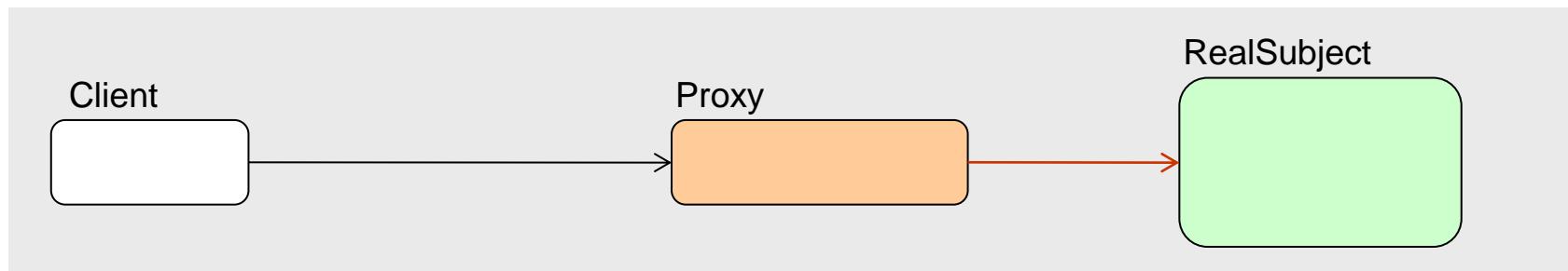
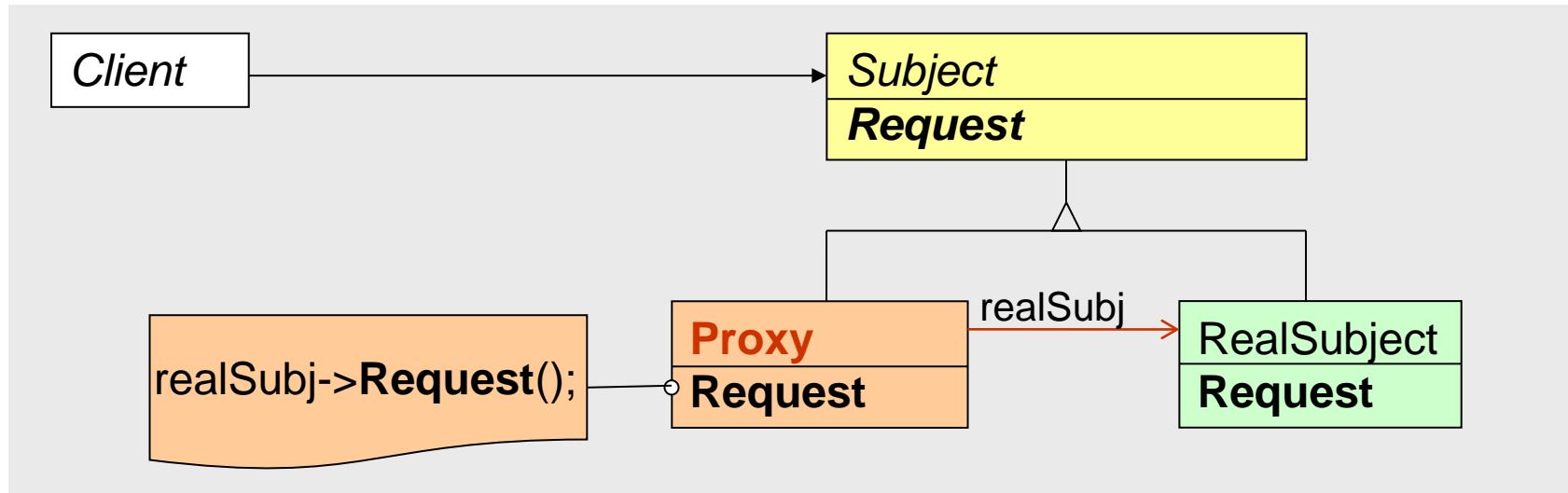


Strukturmuster Proxy

Zweck

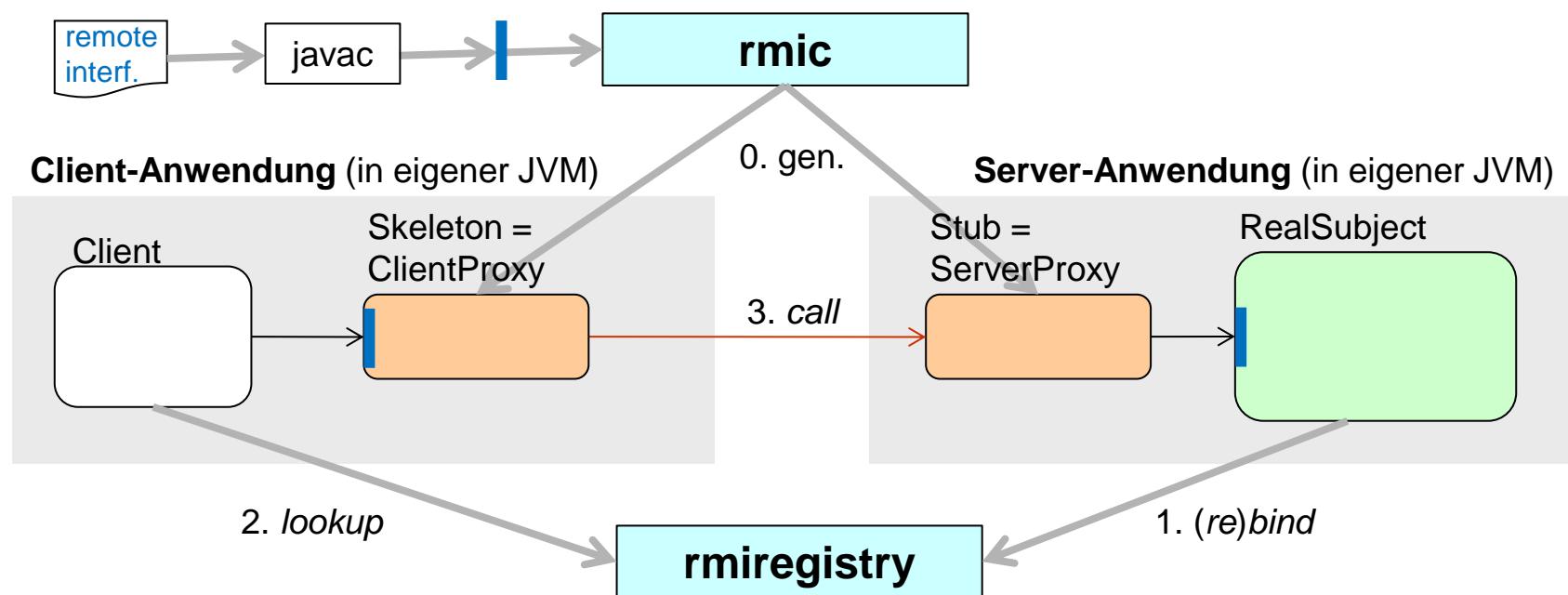
Platzhalter für ein anderes Objekt, um Zugriff darauf regeln zu können

Struktur



Beispiel: Proxy bei Java RMI

- Verteilte Anwendungen können in Java einfach mittels *Remote Method Invocation* (RMI) realisiert werden
- RMI-Compiler (*rmic*) erzeugt im Wesentlichen aus der *class*-Datei des *RealSubjects* (muss Schnittstelle *Remote* implementieren) zwei Proxy-Klassen (als *Skeleton* und *Stub* bezeichnet)
- Zur Laufzeit muss noch ein "Registrierungsdienst" (*rmiregistry*) vorhanden sein, der zum Registrieren und Abfragen verwendet werden kann



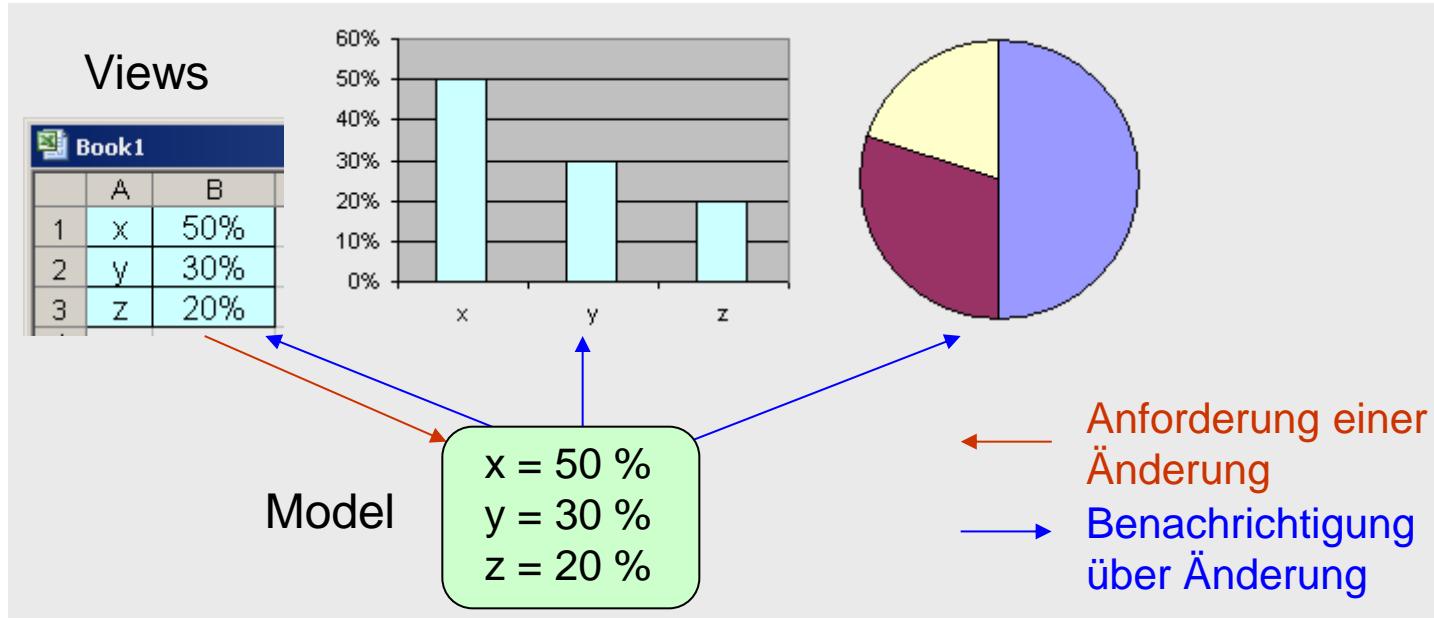
Verhaltensmuster Observer (1)

Zweck

Definiert eine $1:n$ -Beziehung zwischen Objekten: wenn sich Zustand eines Objekts ändert, werden n davon abhängige Objekte benachrichtigt

Motivation

Trennung eines Modells (der Daten) von ihrer Visualisierung (Ansicht), vgl. MVC

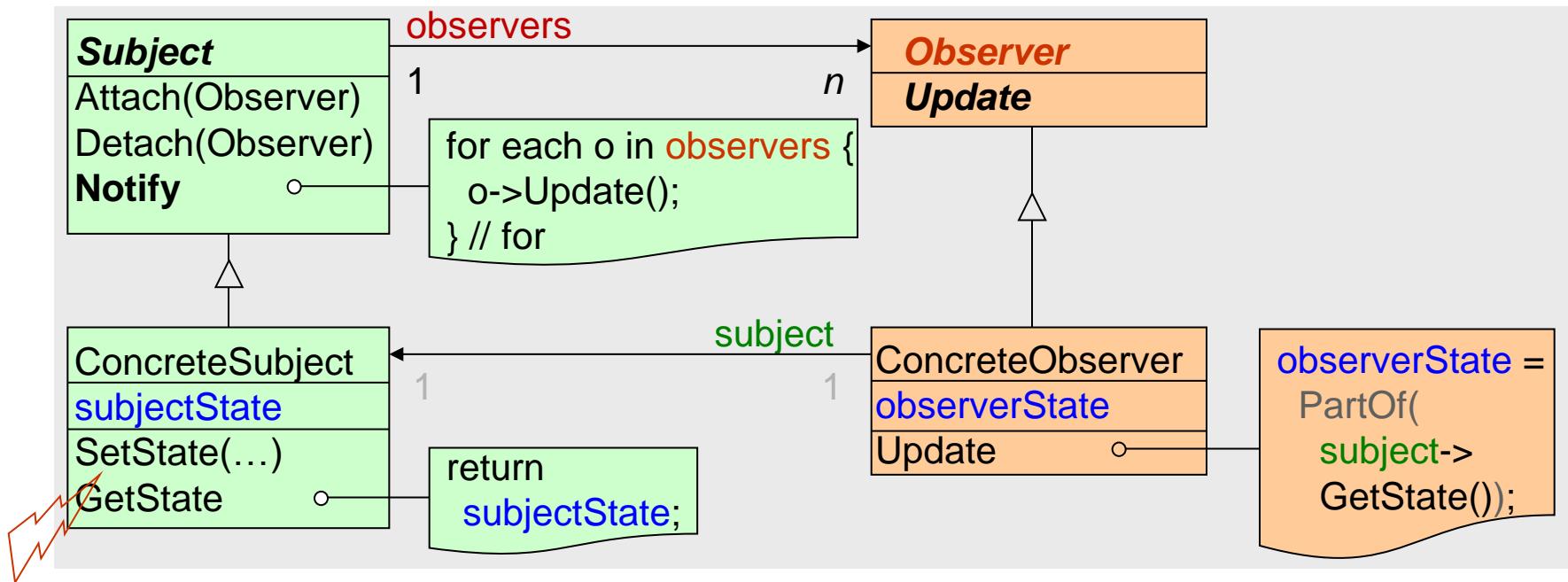


Anwendbarkeit

Wenn Änderungen an einem Objekt automatisch Änderungen an anderen Objekten nach sich ziehen sollen, Realisierung von schwacher Kopplung

Verhaltensmuster Observer (2)

Struktur

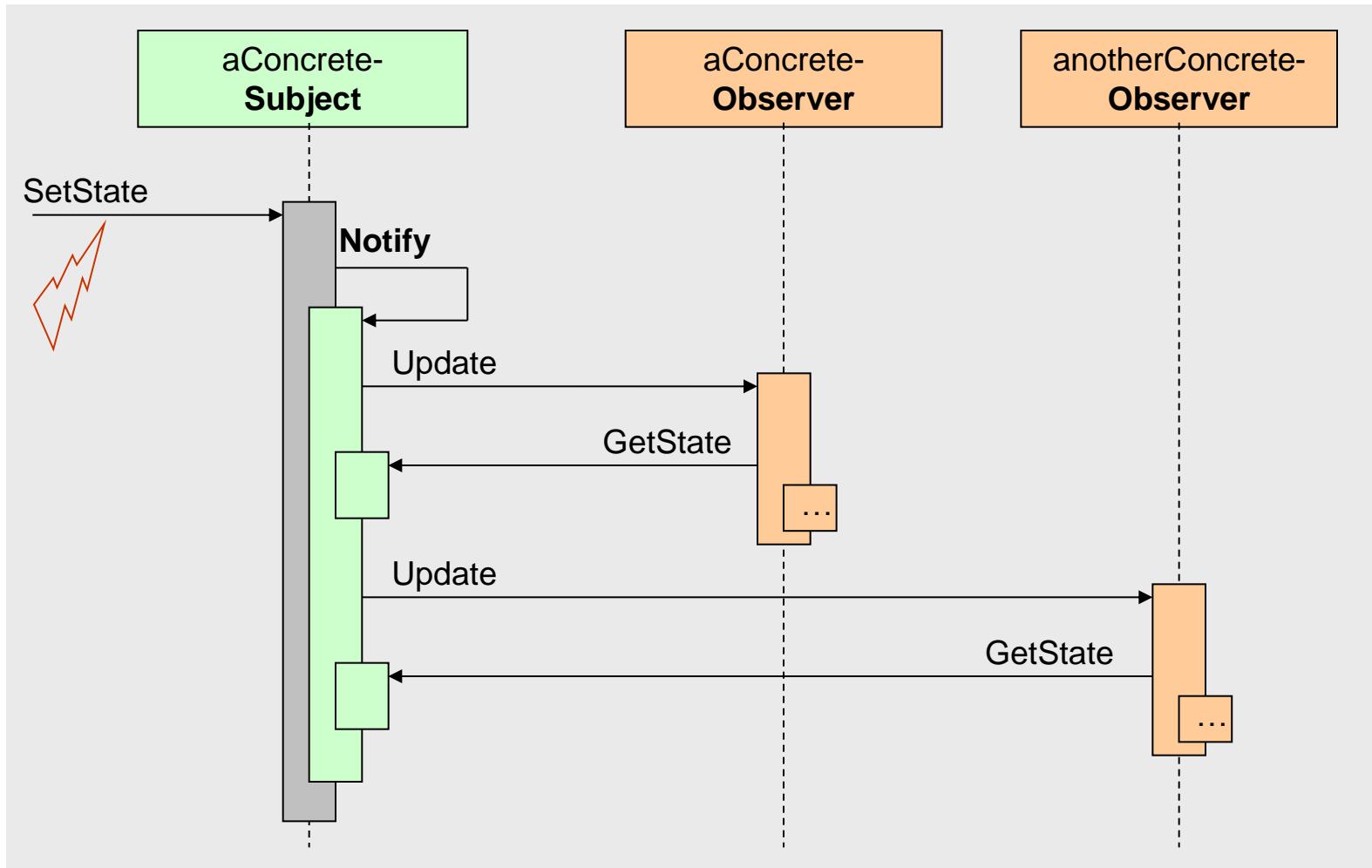


Konsequenzen

- Abstrakte (schwache) Kopplung
- *Broadcast-Kommunikation*: *Subject* informiert in *Notify alle* *Observer*
- (Minimale) Änderungen im *Subject* führen zu vielen *Update-Aufrufen*
- *Subject* muss in geeigneter Weise seine Zustandsänderung mitteilen

Verhaltensmuster *Observer* (3)

Verhalten zur Laufzeit (Dynamik) nach der Registrierung der Oberserver



Verhaltensmuster Observer (4)

Implementierungsdetails

- Wie werden die *Observer* von *Subjects* gespeichert?
Jedes *Subject* verwaltet seine *Observer* selbst oder eine "globale" Verwaltung der Relationen außerhalb des *Subjects* und der *Observer* (z. B. in einer *Map*)
- Wie wird *Update* ausgelöst?
 - Am Ende **jeder** Methode, die Zustand eines *Subjects* ändert, wird *Notify* aufgerufen:
ineffizient bei mehreren Änderungen hintereinander
 - Klienten teilen *Subject* mit, wann *Notify* aufzurufen ist:
effizient aber fehleranfällig und zusätzliche Verantwortlichkeit des Klienten
- Löschen des *Subjects* darf nicht zu *dangling references* in den *Observern* führen,
deshalb Meldung an *Observer* beim Löschen eines *Subjects*, diese müssen sich dann
vielleicht auch selbst beseitigen
- Art des ***Update*-Protokolls?**
Welche Information liefern *Subjects* den *Observern* bei Benachrichtigung?
Zwei Extrema:
 - ***Push-Modell***: *Subjects* verschicken Detailinformation, obwohl diese von vielen
Observern nicht gebraucht wird
(inflexibel, da *Subject* Annahmen über *Observer* trifft)
 - ***Pull-Modell***: *Subjects* verschicken Minimalinformation und *Observer* holen sich
jeweils, was sie noch zusätzlich brauchen
(ineffizient, da hoher Kommunikationsaufwand)

Beispiel 1: Observer in AWT und Swing

- Alte Ereignisbehandlung in Java 1.0 durch Ableiten und Überschreiben von Methoden in den jeweiligen GUI-Klassen (z. B. in *Button* die Methode *action*)
- Neue Ereignisbehandlung (seit Java 1.1): Ereignisse werden durch unterschiedl. *Event*-Objekte dargestellt, *Listener*-Objekte können bei GUI-Elementen registriert werden und so auf bestimmte Ereignisse reagieren (sowohl in AWT als auch in Swing)
- Paket *java.awt.event* hat Menge von *Listener*-Schnittstellen (z. B. *WindowListener*) mit leeren Implement. in *Adapter*-Klassen (z. B. *WindowAdapter*)

```
Button button = new Button("OK");           //   
// create separate ActionListener  Schnittstelle mit nur einer Methode  
ActionListener al = new ActionListener() { // anonymous class ...  
    public void actionPerformed(ActionEvent actionEvent) {  
        System.out.println("OK button pressed");  
    } // actionPerformed  
}; // anon. class ... implementing ActionListener  
  
// register first action listener  
button.addActionListener(al); // removeActionListener is allowed later on
```

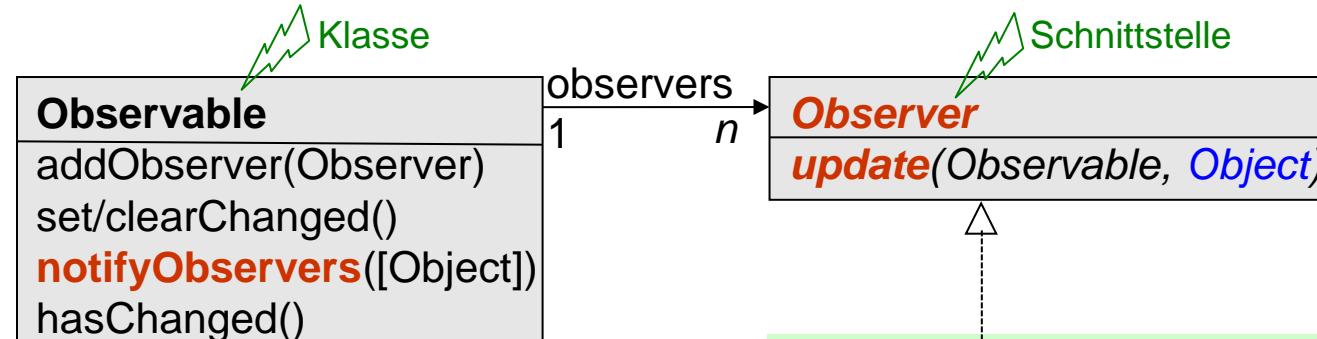
Beispiel 2: MVC mit AWT (vor Java 1.6 selbst gebaut)

```
class Model /*extends Anything*/ {  
    Data d;  
  
    Collection<ModelListener> ls =  
        new ArrayList<ModelListener>();  
  
    void addModelList(ModelList l) {  
        ls.add(l);  
    } // addListener  
  
    void ChangeData(...) {  
        d = ...;  
        notifyObservers();  
    } // ChangeData  
  
    void notifyObservers() {  
        ModelChangedEvent e =  
            new ModelChangedEvent(d);  
        for (ModelListener l: ls)  
            l.modelChanged(e); // push ...  
    } // notifyObservers  
  
} // Model
```

```
class ModelChangedEvent extends EventObject {  
    ModelChangedEvent(Object source) {  
        super(source);  
    } // ModelChangedEvent  
}  
  
interface ModelListener {  
    void modelChanged(ModelChangedEvent e);  
} // ModelListener
```

```
class View1 implements ModelList {  
  
    View1(Model m) {  
        m.addModelListener(this);  
    } // View1  
  
    void modelChanged(ModelCh.Event e) {  
        newData = e.getSource();  
    } // modelChanged  
}  
  
class View2 implements ModelList {  
    ... } // View2  
  
public class Application {  
    public static void main(...) {  
        Model m = new Model(...);  
        View1 v1 = new View1(m);  
        View2 v2 = new View2(m);  
        m.ChangeData(...);  
    } // main  
} // Application
```

Beispiel 3: Observer mit `java.util.*` (seit Java 1.6)



```
class Model  
    extends Observable {  
  
    Data d;  
  
    void ChangeData(...) {  
        d = ...;  
        setChanged();  
        notifyObservers([d]);  
        // now:  
        // hasChanged()==false  
        // because of update  
    } // ChangeData  
  
} // Model
```

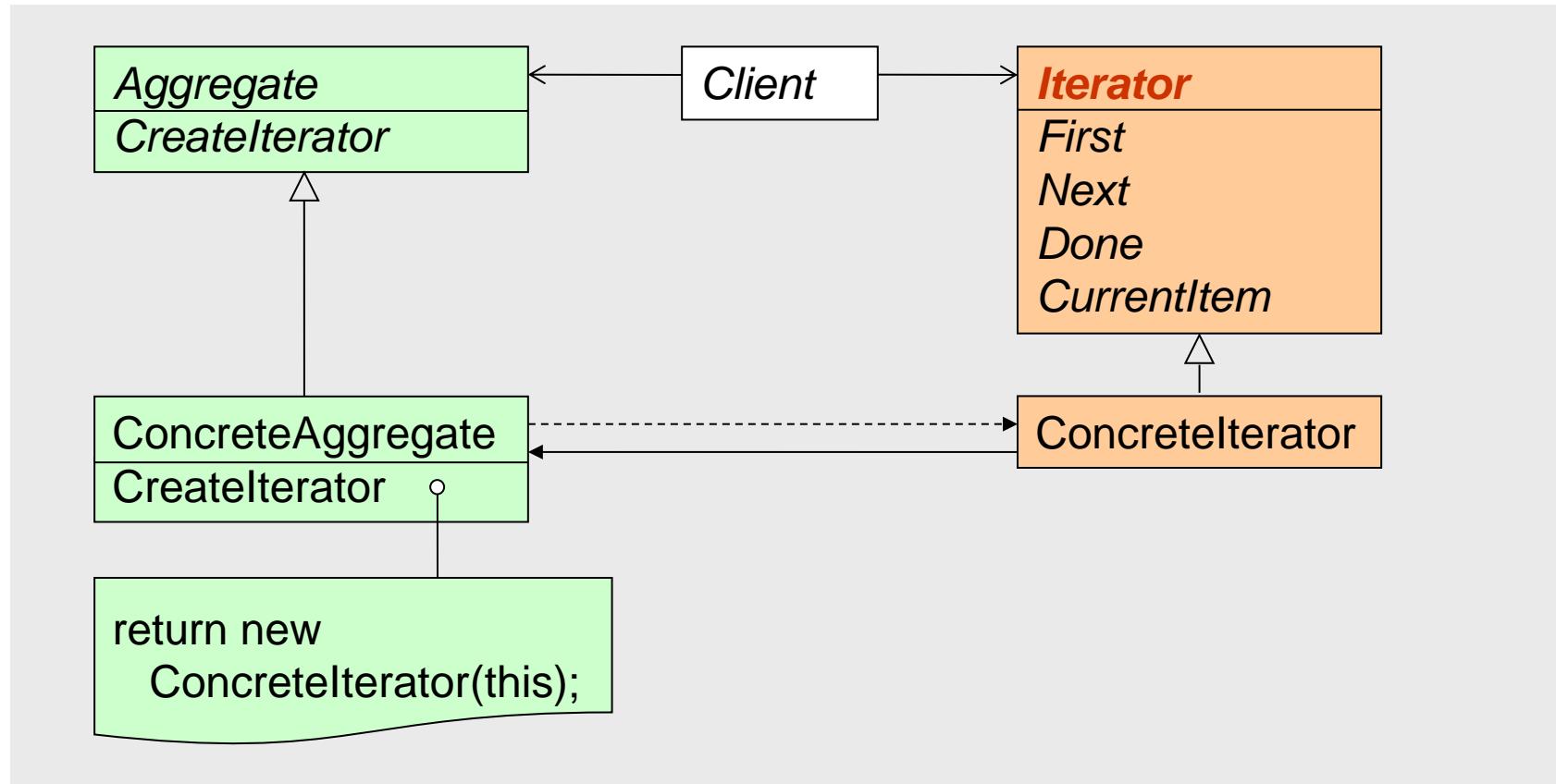
```
class View1 implements Observer {  
    View1(Model m) {  
        m.addObserver(this);  
    } // View1  
    void update(Observable obs,  
               Object arg) { ... }  
} // View1  
  
class View2 implements Observer { ... }  
  
public class Application {  
    public static void main(...) {  
        Model m = new Model(...);  
        View1 v1 = new View1(m);  
        View2 v2 = new View2(m);  
        m.ChangeData(...);  
    } // main  
} // Application
```

Verhaltensmuster *Iterator*

Zweck

Möglichkeit Elemente einer Sammlung zu behandeln, ohne interne Darstellung (Datenstruktur) kennen zu müssen

Struktur



Beispiel: Iterator im JCF

```
public interface Collection {  
    int size();  
    boolean isEmpty();  
    boolean add(Object e);  
    boolean remove(Object e);  
    boolean contains(Object e);  
    Iterator iterator();  
} // Collection
```

```
public interface Iterator {  
    boolean hasNext();  
    Object next();  
    void remove();  
} // Iterator
```

Anwendung (seit Java 1.5 mit Generizität u. *foreach*-Schleife möglich):

```
Collection<String> c = new ArrayList<String>();  
c.add("A...");  
...  
Iterator<String> it = c.iterator();  
while (it.hasNext()) {  
    String s = (String)it.next();  
    System.out.println(s);  
} // while
```

*// alternatively
// using foreach:
for (String s : c) {
 System.out.println(s);
} // foreach*

Verhaltensmuster *Strategy*

Zweck

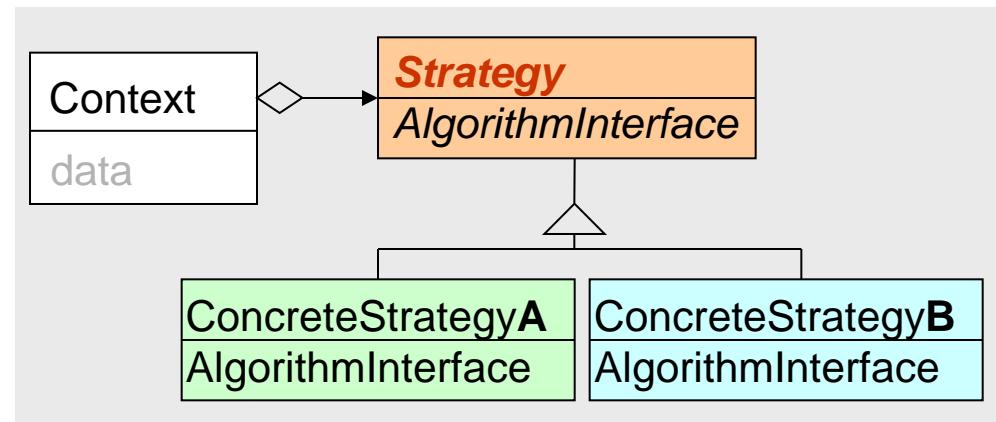
Definition einer Algorithmenfamilie mit Kapselung und Austauschbarkeit

Motivation

Viele Aufgaben, für die Algorithmen mit gleicher Schnittstelle u. Funktionalität aber unterschiedlichen Eigenschaften existieren

Anwendbarkeit

- Bei mehreren Klassen, die sich nur durch Verhalten unterscheiden
- Wenn untersch. Algorithmen-Varianten benötigt werden



Struktur

Konsequenzen

Gruppen verwandter Algorithmen können hierarchisch strukturiert werden; Alternative zur Vererbung, alle typischen Vorteile der Delegation, Klienten müssen sich bewusst sein, dass es unterschiedliche Strategien gibt

Implementierungsdetails

Strategy benötigt effizienten Zugriff auf Daten des *Context*-Objekts und ev. umgekehrt: z. B. Daten oder *Context*-Objekt als Parameter für *Strategy*

Verhaltensmuster *Template Method*

Zweck

Definition der Algorithmus-Grobstruktur, Details an abgel. Klasse delegieren

Motivation

Application-Framework mit *Application-* und *Window*-Klassen, *Application* definiert Algorithmus für Öffnen und Schließen von Fenstern, Lesen und Schreiben von Daten wird erst in abgeleiteter Klasse definiert

Anwendbarkeit

- Möglichst frühe Implement. invarianter Algorithmenteile
- Steuerung u./o. Erweiterungen in abgeleiteten Klassen

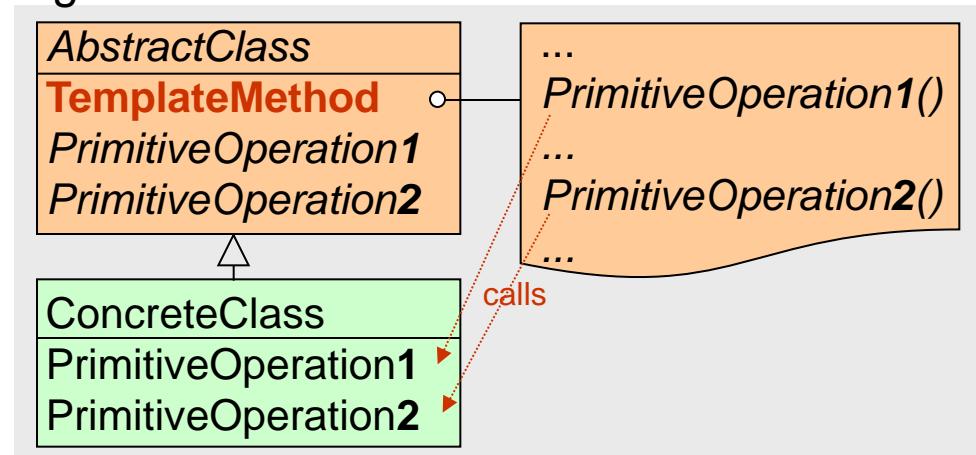
Struktur

Konsequenzen

Fundamentale Technik zur Erhöhung der Wiederverwendung; führt zu *invertierter Kontrollstruktur* ("Hollywood-Prinzip"): "Don't call us, we call you!"

Implementierungsdetails

Möglichst wenige (abstrakte) *Primitive*-Methoden, sollten *protected* sein;
Template-Methoden: nicht dynamisch binden (nicht *virtual* in C++),
wenn möglich vor Überschreiben schützen (*final* in Java)



Beispiel: *Template Method* im SAX-Parser

- Klasse `org.xml.sax.helpers.DefaultHandler` hat 17 "leere" Handler-Methoden
- Durch Ableiten einer Klasse und Überschreiben entspr. Methoden kann bei der Analyse eines XML-Dokuments auf entspr. Elemente zugegriffen werden

```
class SimpleHandler extends DefaultHandler {  
  
    public void startDocument()  
        throws SAXException {  
        System.out.println("start of document");  
    } // startDocument  
  
    public void startElement(String uri, String lName,  
                            String qName, Attributes attrs)  
        throws SAXException {  
        System.out.println(" element: " + qName);  
    } // startElement  
  
    public void endDocument()  
        throws SAXException{  
        System.out.println("end of document");  
    } // endDocument  
  
} // SimpleHandler  
  
...  
SAXParser sp = SAXParserFactory.newInstance().newSAXParser();  
sp.parse("Message.xml", new SimpleHandler());
```

Datei *Message.xml*:

```
<message>  
  <to>Anne</to>  
  <from>Pat</from>  
  <body>Hello...</body>  
</message>
```

Ergebnis:

```
start of document  
element: message  
element: to  
element: from  
element: body  
end of document
```

"Landkarte" der Entwurfsmuster

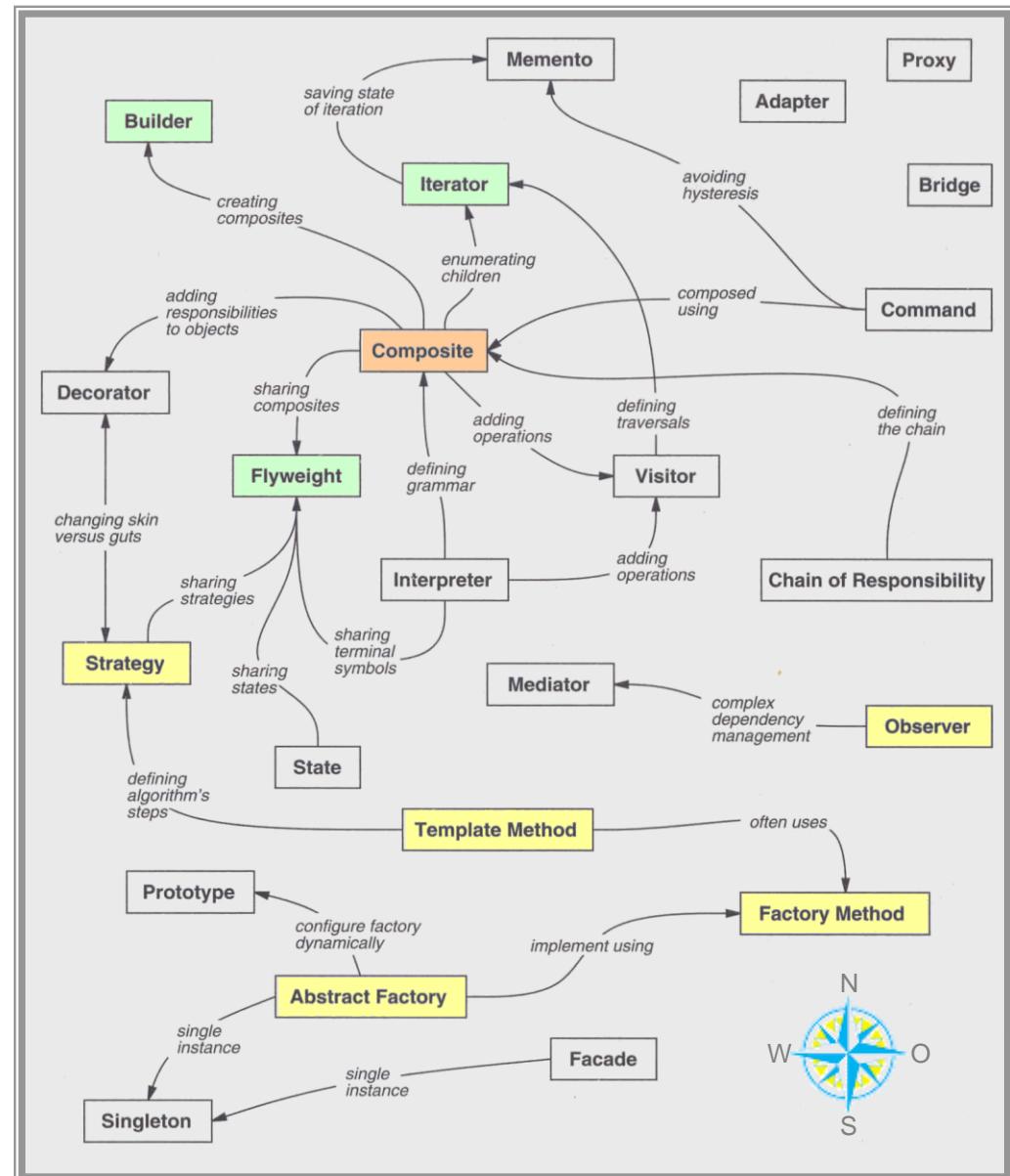
Entwurfsmuster sind nicht isoliert zu sehen

Oft bedingt/erlaubt ein Entwurfsmuster ein oder mehrere andere

Eines der zentralen Muster ist **Composite**:

- Comp. kann mit **Builder** erzeugt werden,
- Comp. verwendet **Iteratoren** zur Abarbeitung und
- Comp. kann über **Flyweight** Elemente mehrfach nutzen

Große Systeme enthalten immer mehrere Muster



Entnommen aus [Gamma 95]

Exkurs: Entwurfsmuster in der MiniLib

Jean Paul Buchwald (Daimler, Ulm) hat die Quelltexte der C++-Version der MiniLib mittels Mustererkennungswerkzeug (auf Basis der Erkennungsalgorithmen der TU Ilmenau) analysiert

Erkanntes Muster	Beteiligte Klassen (und deren Rolle)
<i>Factory Method</i>	In <i>Collection (Creator)</i> : <i>Iterator (Product)</i> , in <i>Vector (ConcreteCreator)</i> : <i>VectorIterator (Concr.Prod.)</i>
<i>Composite</i>	<i>Vector (Composite)</i> , <i>Object (Component)</i> , alle anderen Klassen (<i>Leaf</i>)
<i>Template Method</i>	In <i>Window</i> und <i>Application</i>
<i>Iterator</i>	<i>Collection</i> mit <i>Iterator</i> und <i>Vector</i> mit <i>VectorIterator</i>

Nicht erkannt: *Singleton* in *Application* (wegen unüblicher Implementierung)

Designprobleme und mögliche Lösungen mittels Entwurfsmustern:

1. Problem: Objekterzeugung durch explizite Angabe einer Klasse
Lösung durch indirekte Erzeugung: *Abstract Factory*, *Factory Method* oder *Prototype*
2. Problem: Abhängigkeit von bestimmten Operationen
Anforderungen nicht "hart codieren": *Chain of Responsibility* oder *Command*
3. Problem: Abhängigkeit von bestimmten Hard- u./o. Softwareplattformen
Unterstützung verschiedener Konfigurationen: *Abstract Factory* oder *Bridge*
4. Problem: Abhängigkeit von Objektrepräsentationen u./o. -implementierungen
Implementierungsdetails vor Klienten verbergen: *Abstract Factory*, *Bridge*, *Memento* oder *Proxy*
5. Problem: Große algorithmische Abhängigkeit
Algorithmen, die verändert oder verbessert werden könnten, sollen isoliert werden: *Builder*, *Iterator*, *Strategy*, *Template Method* oder *Visitor*
6. Problem: Erweiterung der Funktionalität durch Ableitung bereitet viel Aufwand und erfordert Verständnis der Basisklasse
Vermeidung durch Delegation: *Bridge*, *Chain of Responsibility*, *Composite*, *Decorator*, *Observer* oder *Strategy*

- Neben **Entwurfsmustern** (*design patterns*) der GoF viele weitere Entwurfsmuster, Vereinfachungen oder Erweiterung oder für bestimmte Einsatzgebiete, z. B. unternehmensweite Anwendungen von Martin Fowler, z. B. *Data Access/Transfer Object*, *Domain Model*, *MVC* und *Value Object*
- **Metamuster** (*meta patterns*) von Wolfgang Pree: beschreiben Muster in Entwurfsmustern, z. B. *hot spot*, *template method* und *hook method*
- **Analysemuster** (*analysis patterns*) von Martin Fowler: behandeln wiederkehrende Probleme und deren Lösung in der Analysephase
- **Organisationsmuster** (*organizational patterns*): beschreiben typische Strukturen (Muster) in der Aufbau- und Ablauforganisation von großen Unternehmen
- **Pädagogische Muster** (*pedagogical patterns*): beschreiben Muster, die sich in Lehre oder Training (z. B. für gutes Software-Design oder die Gestaltung der Mensch-Maschine-Schnittstelle) bewährt haben
- **Antimuster** (*anti patterns*) z. B. von Brown, Malveau u. McCromick: beschreiben typische Fehler und deren Vermeidung bzw. Möglichkeiten zur Beseitigung, z. B. *alien spiders* (alle Objekte kennen sich), *copy & paste programming*, *magic value* (z. B. 42) und *golden hammer* (= *silver bullet*)

- Jede Software hat eine Architektur!
Die Frage ist nur:
Welche Architektur hat sie und **wie gut** ist diese?
- Aber: **Was ist Softwarearchitektur überhaupt?**
- Beispiel für eine gute Definition
(von Philip Kruchten etwas gekürzt):
"Softwarearchitektur beschäftigt sich
 - **mit Abstraktion,**
 - **mit Zerlegung und Zusammenbau,**
 - **mit Stil und Ästhetik."**
- Software Engineering Institut (SEI) an der Carnegie-Mellon Universität (CMU) führt über 200 weitere Definitionen an, siehe www.sei.cmu.edu/architecture/start/glossary/community.cfm
- Gemeinsamkeit bei fast allen Definitionen:
Softwarearchitektur kann nur beschrieben/verstanden werden, wenn **verschiedene "Sichten"** auf die Software dargestellt werden



Bildquelle: pixabay.com (kostenlos)

gute
Behandl.
der
Theorie

1. Konzeptuelle Sicht

Wie "funktioniert" das System?
(Vision und Überblick,
z. B. mittels Use Cases)

2. Infrastruktursicht

In welcher H/S-Umgebung läuft
das System?
(Hard- und Softwarekomponenten,
Netzwerktopologie u. -protokolle,
sonst. Bestandteile d. Umgebung)

3. Implementierungssicht

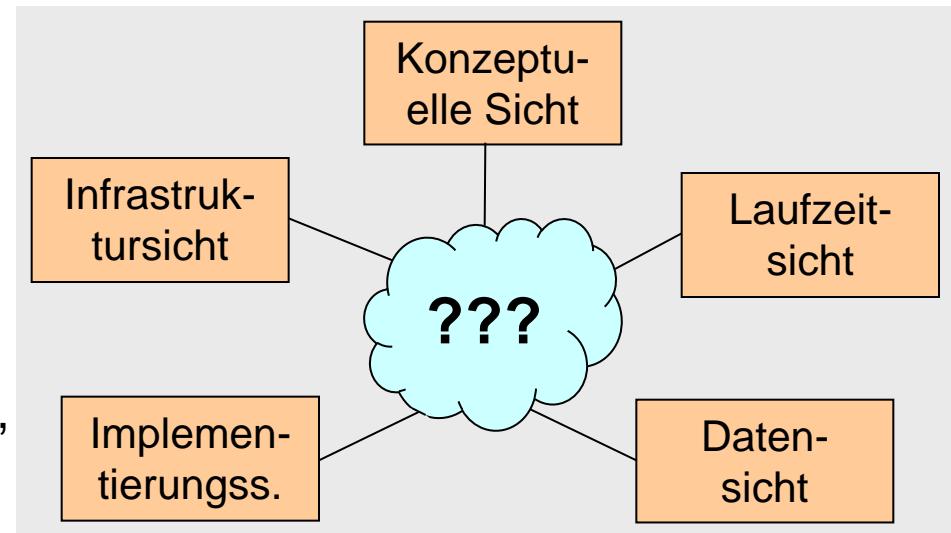
Wie ist das System intern aufgebaut?
(Bestandteile des Systems selbst:
Subsysteme, Komponenten, Klassen/Module, ... letzte Stufe ist der Quelltext)

4. Laufzeitsicht

Wie läuft das System ab?
(Welche Bestandteile, z. B. Objekte, gibt es zur Laufzeit und wie wirken sie in
zeitlicher Abfolge zusammen?)

5. Datensicht

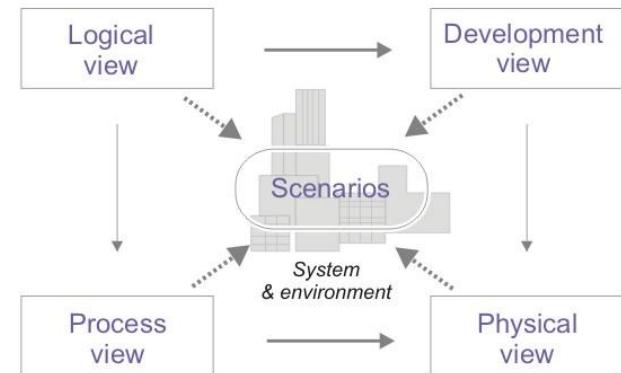
Welche Daten bearbeitet das System?
(Woher kommen die Daten, wohin gehen die Daten, welches Format haben
sie und welche Bedeutung?)



* In Anlehnung an G. Starke: *Effektive Softwarearchitekturen*, 6. Auflage, Hanser 2014

4+1-Sichtenmodell von Philipp Kruchten *

1. Logische Sicht betrachtet Funktionalität
2. Entwicklungssicht beschreibt das System aus Sicht der Entwickler*innen (Statik)
3. Prozesssicht beschreibt die Dynamik des Systems
4. Physische Sicht beschreibt die Verteilung der Komponenten
5. Szenarios skizzieren die wichtigsten Anwendungsfälle



Bildquelle:
https://de.wikipedia.org/wiki/4%2B1_Sichtenmodell

C4-Modell von Simon Brown

Basiert auf dem 4+1-Sichtenmodell, ist im Wesentlichen eine Notationstechnik zur Zerlegung und anschließenden Visualisierung eines Systems in Form von

1. *Context diagrams*: zeigen Umfang des Systems und Beziehungen zu anderen
2. *Container diagrams*: zeigen die miteinander verbundenen Container eines Systems
3. *Component diagrams*: zeigen die Komponenten eines jeden Containers
4. *Code diagrams*: zeigen Details der Implementierung

* Vgl. https://de.wikipedia.org/wiki/4%2B1_Sichtenmodell

Vgl. https://de.wikipedia.org/wiki/C4_model

Architekturmuster	Ausprägungen
Datenzentrierung (auch Client/Server)	Repository-Architektur Blackboard-Architektur
Datenflussorientierung	Batch/Sequential-Architektur Pipes&Filters-Architektur
Call & Return	Top-Down-Architektur Netzwerk-Architektur Schichten-Architektur (<i>layered a.</i>)
Virtuelle Maschine	Interpretierer-Architektur Regelbasierte Architektur
Unabhängige Komponenten	Ereignisgesteuerte Architektur

- Jedes Architekturmuster hat spezifische Ausprägungen
- In einem System können mehrere Architekturmuster vorkommen

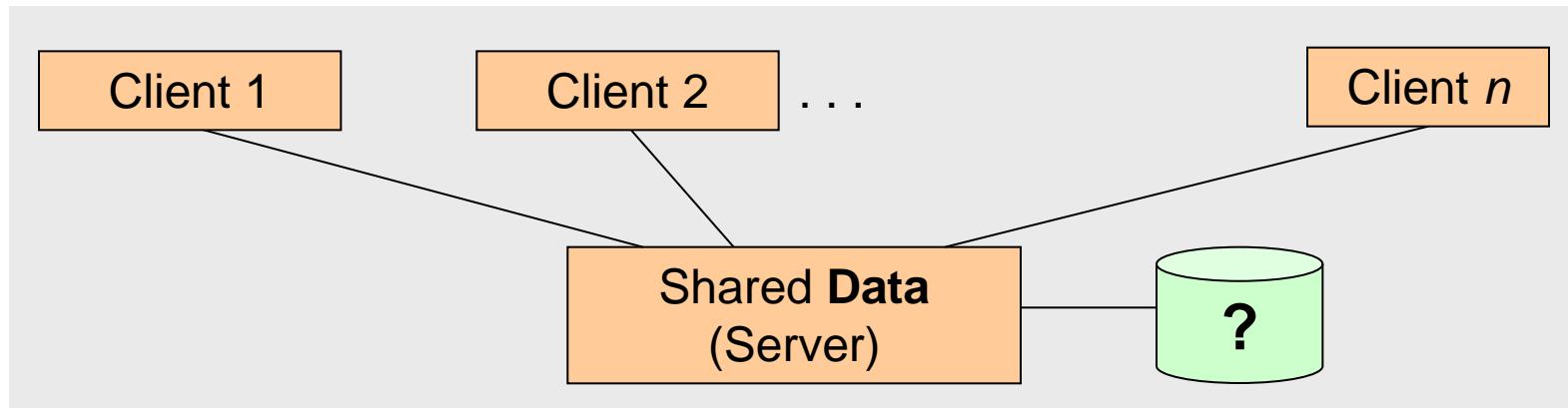
Oft auch als **Client/Server** bezeichnet

Gemeinsam genutzte **Daten** (in/auf einem *Server*) stehen im Zentrum
Benutzer (Klienten, *Clients*) greifen auf die Daten zu

Zwei Ausprägungen:

- Bei *Repository* sind Daten "passiv"
- Bei *Blackboard* sind Daten "aktiv", sie benachrichtigen interessierte Klienten bei Änderungen automatisch (vgl. *Observer*)

Vorteil: Klienten sind voneinander unabhängig



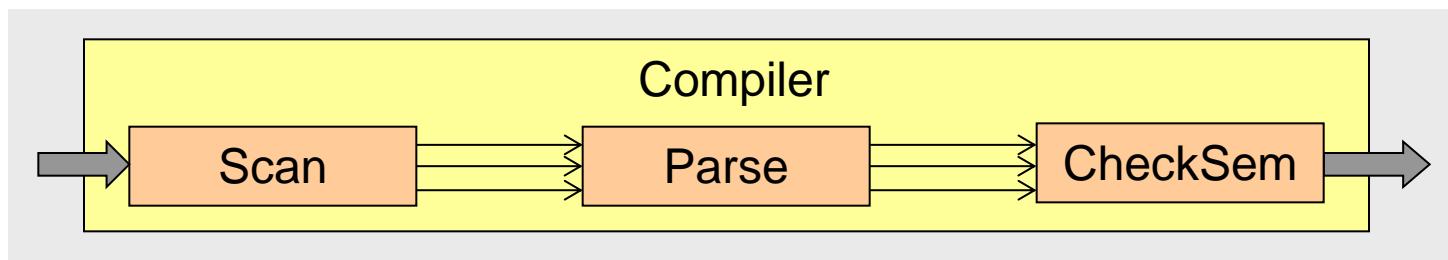
Transformation der Daten durch Verarbeitungsprozesse ist im Zentrum
Beschreibt die Abfolge der einzelnen Transformationen

Zwei Ausprägungen:

- Bei *Batch/Sequential* muss ein Schritt abgeschlossen sein, bevor der nächste beginnen kann, z. B. Kommando `C:\ > dir | sort | print`



- Bei *Pipes&Filters* werden Daten nicht "en block" sondern inkrementell transformiert, in kleineren Einheiten, (quasi) parallel, z. B:



Vorteile: Gute Wiederverwendbarkeit und Modifizierbarkeit

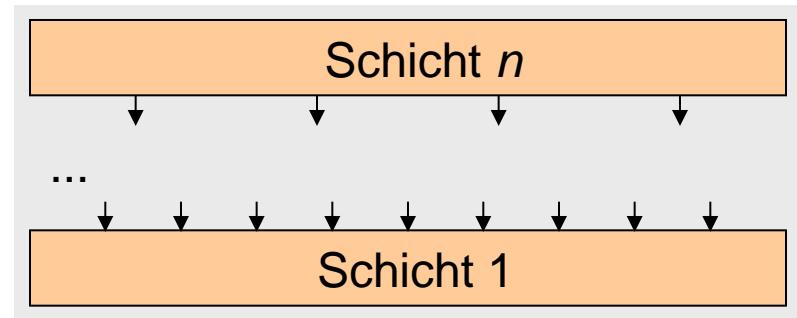
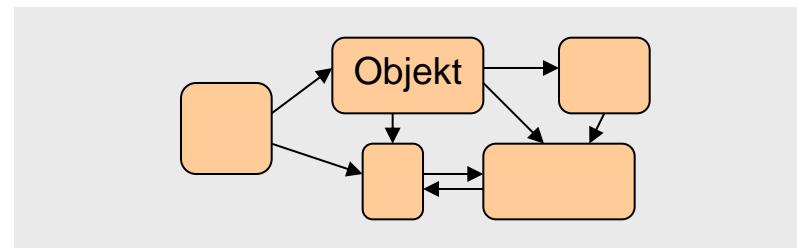
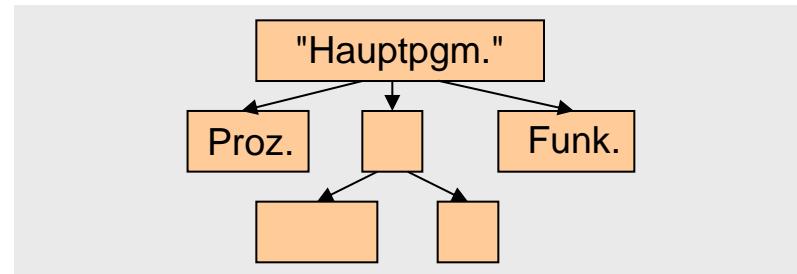
Kontrollfluss steht im Mittelpunkt

Drei Ausprägungen:

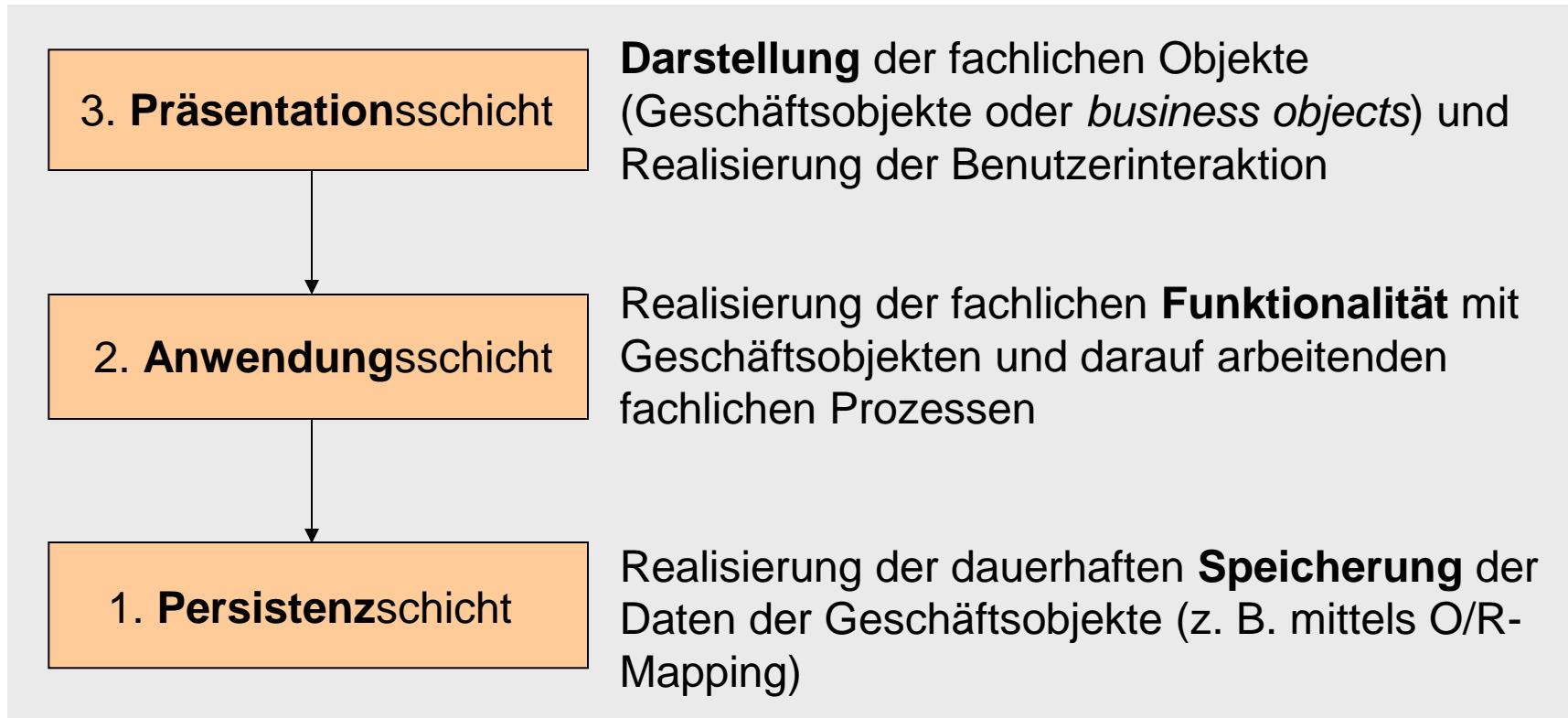
- *Top-Down-Architektur* ergibt sich bei konvent. Implementierungen (Aufruf v. Prozeduren/Funktionen)

- *Netzwerk-Architektur* ergibt sich bei oo Implementierungen (Akt. Objekte senden Nachrichten)

- *Schichten-Architektur*, wobei
 - Schicht ***n*** verwendet nur Funktionalität der Schicht ***n-1***
 - Schicht ist ein Teilsystem mit besonders loser Kopplung



Viele kommerzielle Systeme haben (mittlerweile) dreischichtigen Aufbau



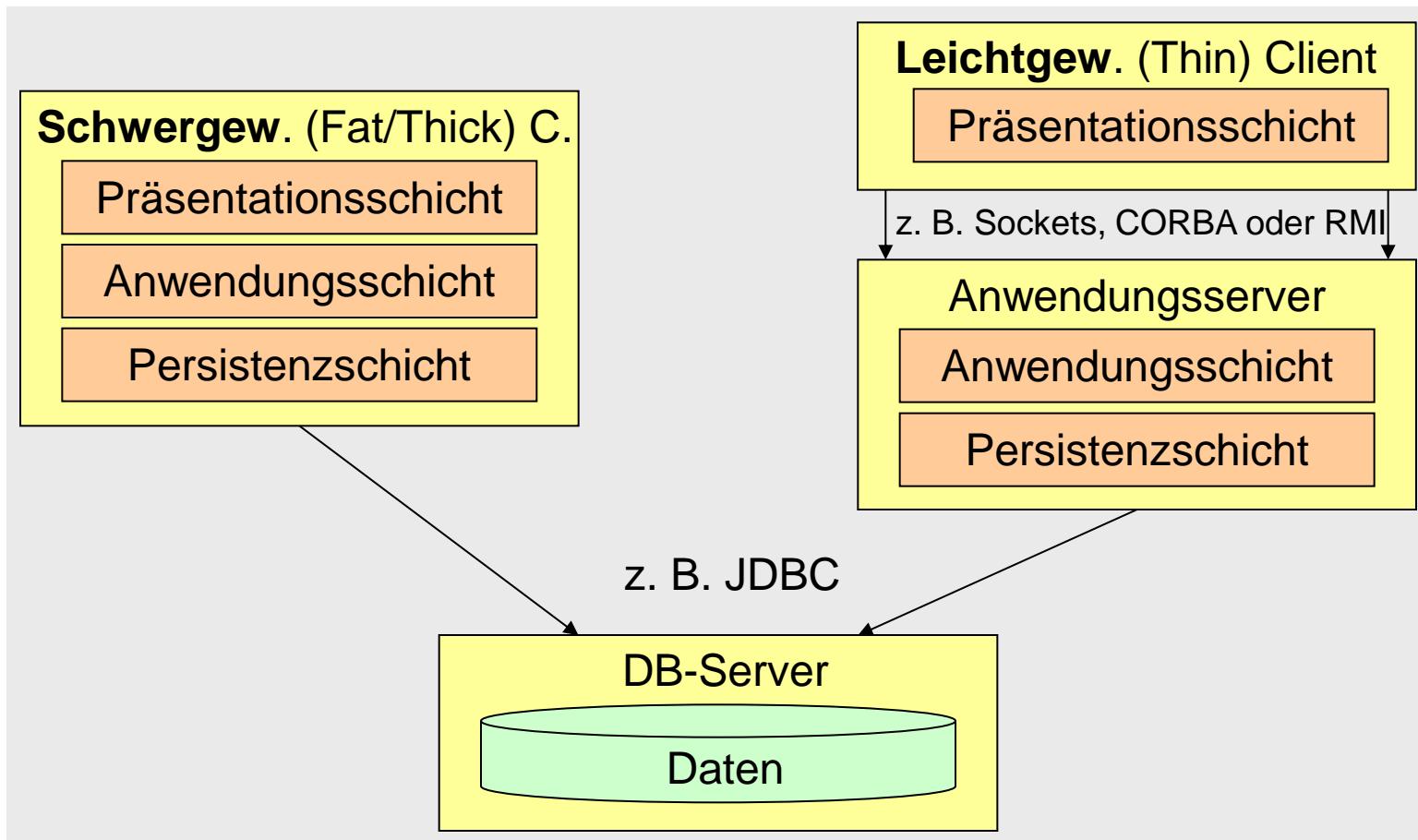
Bewertung

Gute Verständlichkeit, geringe Redundanzen, hohe Robustheit, gute Wiederverwendbarkeit ermöglicht außerdem bessere Projektorganisation

Exkurs: Physikalische Verteilung bei 3-S.A. (1)

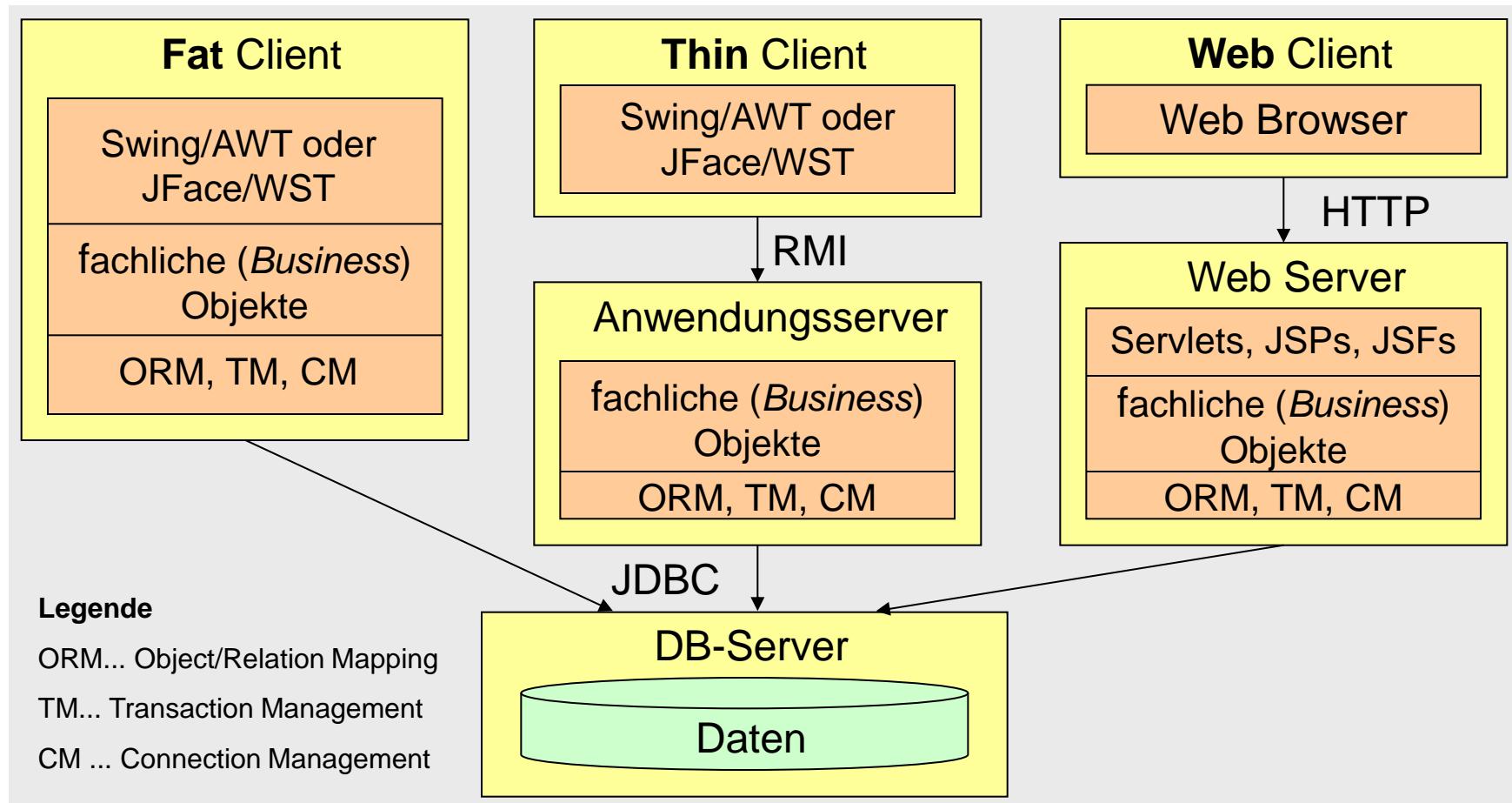
Moderne Softwaresysteme sind meist auf mehrere Rechner verteilt, die über Netzwerke kommunizieren (verteilte Systeme, *distributed systems*)

Unterscheidung nach der Art des Clients



Verteilungsszenarien lassen sich mit unterschiedlichen Technologien realisieren, daraus resultieren

Standardarchitekturen für verteilte Systeme (z. B. mit J2EE)

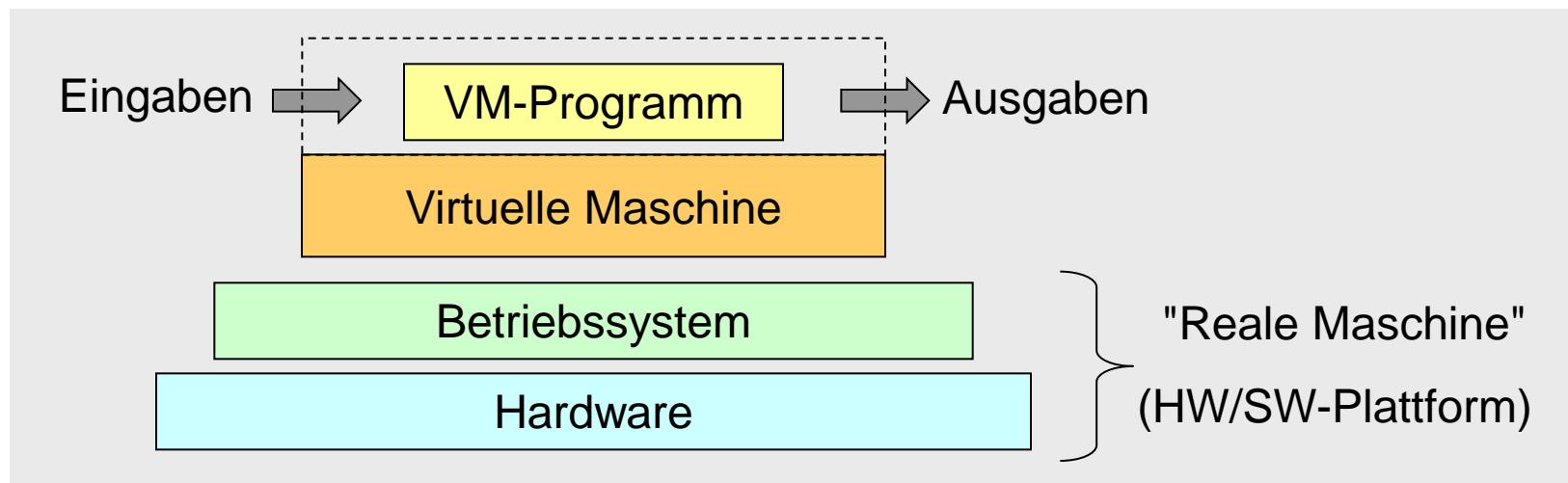


Virtuelle Maschine stellt **Funktionalität** zur Verfügung, die durch existierende Hard- und/oder Softwareplattform nicht geboten wird

Zwei Ausprägungen:

- *Interpretierer-Architekturen* interpretieren spezielle Programme (alte Idee, z. B. Smalltalk u. P-Code, durch Java wieder aktuell)
- *Regelbasierte Architekturen* interpretieren eine Menge von Regeln und können so z. B. "Expertensysteme" bilden (z. B. Prolog)

Vorteile: einfache Code-Erzeugung, hohe Portabilität



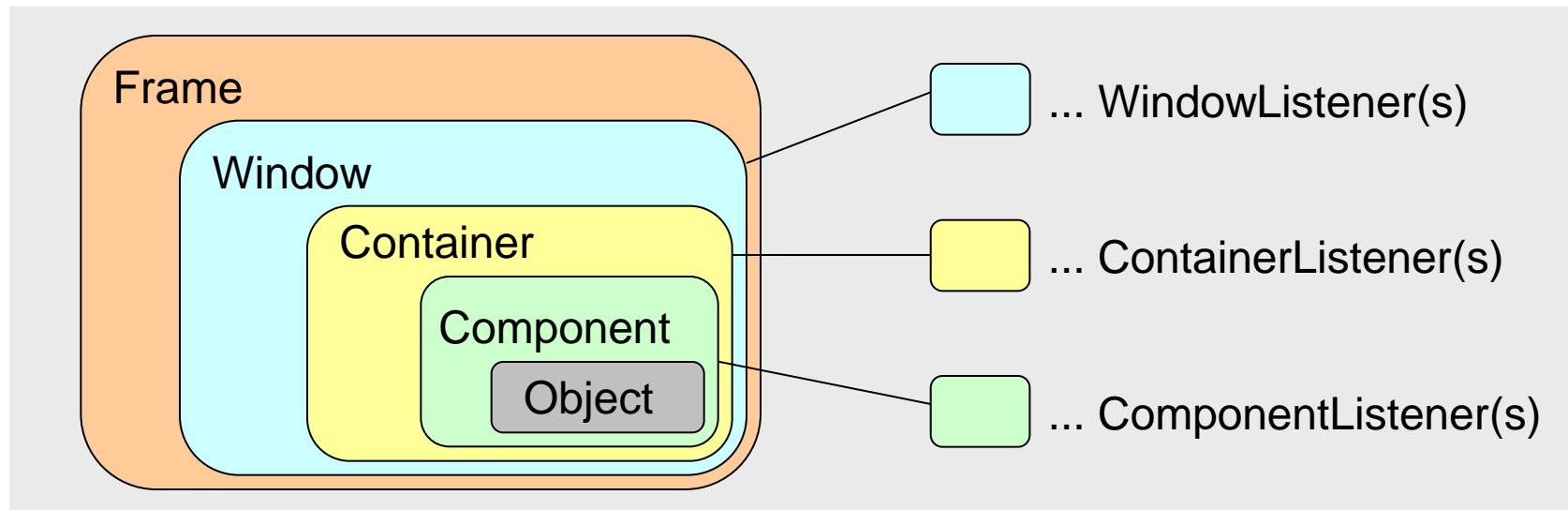
Lose miteinander gekoppelte, voneinander unabhängige Komponenten, die sich nicht direkt aufrufen

Verbindung wird üblicherweise nur über *Ereignisse* hergestellt (vgl. *Observer*):

- Komponenten registrieren sich bei anderen Komponenten
- Nur registrierte Komponenten werden über Ereignisse informiert

Vorteile: enge Schnittstelle (Ereignistyp), dynamisch veränderbar

Beispiel: Listener-Konzept zur Ereignisbehandlung im AWT von Java

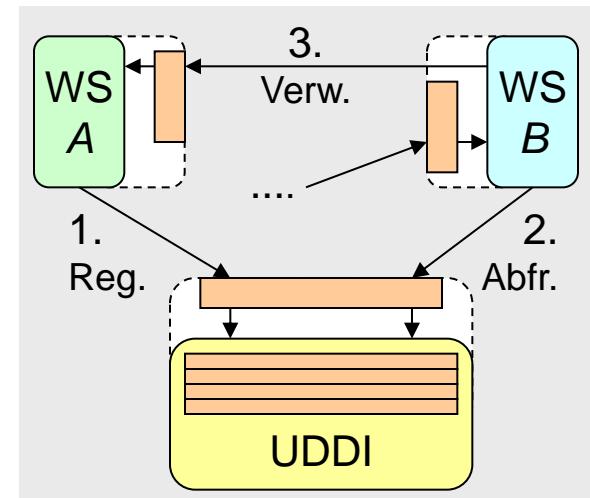


Serviceorientierte Architekturen (SOA)

- Service ist Funktionalität die Serviceanbieter (*service provider*) für Kunden (*service consumer*) zur Verfügung stellt
- Anbieter und Kunden sind beliebige Softwarekomponenten
- **Lose Kopplung** zwischen diesen Komponenten, Trennung von Schnittstelle und Implementierung
- Zwei Ausprägungen:
 - Zustandsloses (*stateless*) Service: Nachricht enthält alle Daten für Service; erhöht Skalierbarkeit und höhere Zuverlässigkeit
 - Zustandsbehaft. (*stateful*) Service: Daten werden zwischen Nachrichten im Service gespeichert; in vielen Fällen aus Effizienzgründen notwendig, z. B. für Sessions

Implementierung z. B. mittels **Web Services** (WS):

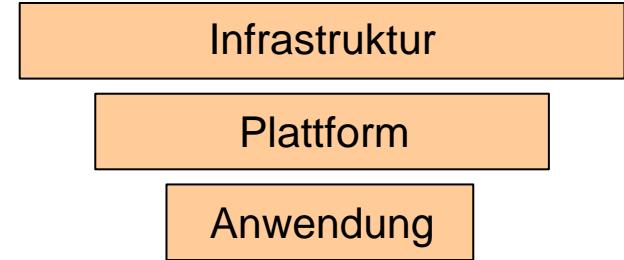
- Beschreibung der WS-Schnittstellen mittels *Web Service Description Language (WSDL)*
- Liste von Web Services (ihrer Schnittstellen) in Verzeichnisdienst (selbst ein WS), *Univ. Description, Discovery & Integration (UDDI)*
- Kommunikation zwischen WS mittels SOAP z. B. über HTTP ... weitere Ralisierungsmögl. (neben SOAP):
 - * XML-RPC und
 - * *Representational State Transfer (REST)*



... ist mehr als nur das Speichern irgendwelcher Daten i. d. "Wolke"

Typische technische Realisierungsformen:

- **Infrastruktur** (*infrastructure as a service, IaaS*),
z. B. Amazon Web Services (AWS),
darauf basiert z. B. auch Dropbox
- **Plattform** (*platform as a service, PaaS*): Anwendungen werden von den Entwickler*innen in die Cloud gestellt, z. B. mittels Microsoft Windows Azure, der Goolge App Engine oder AWS
- **Anwendung** (*software as a service, SaaS*): bestehende Anwendungen in der Cloud werden genutzt, z. B. Apple iCloud, Google Drive mit Google Docs oder MS Office 365



Organisatorische Ausprägungen (hinsichtl. Kreis der Benutzer*innen):

- **öffentliche** Cloud
 - **private** Cloud
 - **hybride** Cloud
- } macht aber keine Aussage darüber
wer die Cloud wo betreibt!

Standards für Architektur(beschreibung)

- Seit 1995 Arbeit an Standard "Empfohlene Praktiken für Architekturbeschreibung von Software-intensiven Systemen", 2000 als IEEE-Standard 1471 verabschiedet
- 2007 in erweiterter Version auch als ISO/IEC-Standard 42010 verabschiedet

Ziele

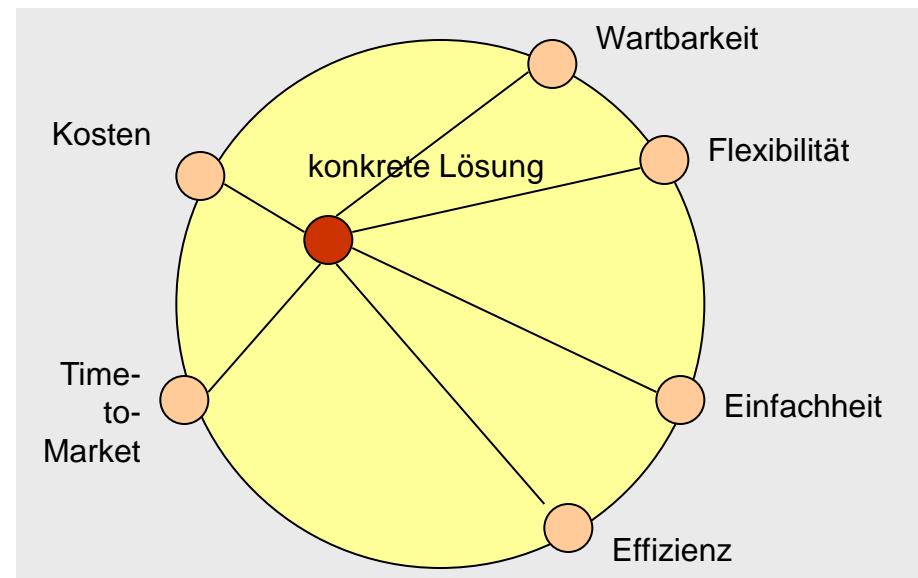
- Standardisierte und einheitliche (definierte) **Terminologie**
- Definierte **Anforderungen** an Inhalte (richtet sich an alle Beteiligten, engl. *stakeholder*, besteht aus mehreren Sichten, diese sind konsistent beschrieben, ...) und
- Offen für **Erweiterungen**

Problem

Softwarearchitekturen leben in einem **Spannungsfeld**, das keine "optimalen" Lösungen erlaubt

Gummiband-Diagramm

Konkrete Lösung (mit bestimmter Architektur) ist immer ein Kompromiss



Quelle: Gernot Starke, *Effiziente Softwarearchitekturen*

Literatur zu Entwurfsmustern

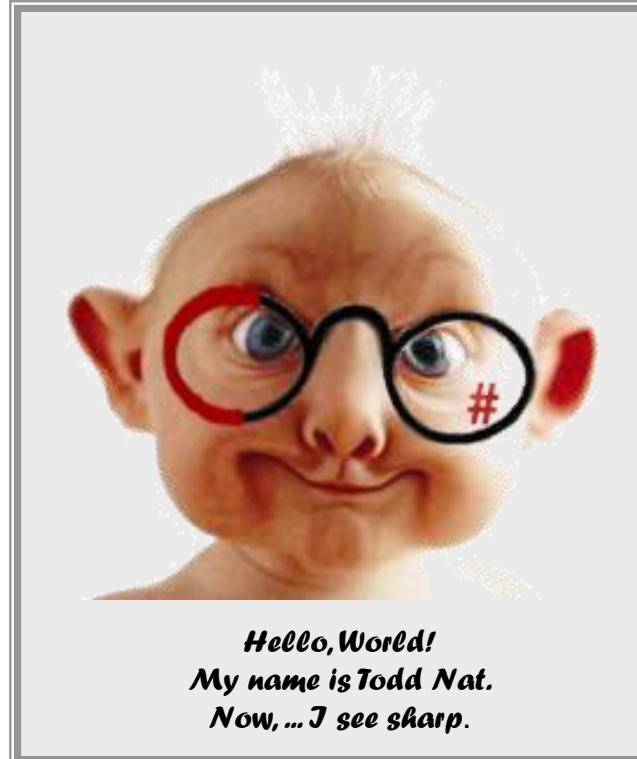
- Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel: *A Pattern Language*. Oxford University Press, 1977
- Erich Gamma, Richard Helm, Ralph Johnsons, and John Vlissides: *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995 GoF: Gang of Four
- Brandon Goldfedder: *The Joy of Patterns — Using Patterns for Enterprise Development*. Addison-Wesley, 2002
- Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal: *Pattern Oriented Software Architecture (POSA)*, Vol. 1: *A System of Patterns*. Wiley, 1996
- Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann: *POSA*, Vol. 2: *Patterns for Concurrent and Networked Objects*. Wiley, 2000
- James O. Coplien and Douglas C. Schmidt: *Pattern Languages of Program Design*. Addison-Wesley, 1995
- John Vlissides, James O. Coplien, and Norman L. Kerth: *Pattern Languages of Program Design 2*. Addison-Wesley, 1996
- Wolfgang Pree: *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1994

- **Gustav Pomberger** und Wolfgang Pree: *Software Engineering – Architektur-Design und Prozessorientierung*, 3. Auflage.
Hanser 2005
- **Gernot Starke**: *Effiziente Softwarearchitekturen – Ein praktischer Leitfaden*, 9. aktualisierte und erweiterte Auflage.
Hanser, 2020
- Jürgen Dunkel und Andreas Holitschke: *Softwarearchitektur für die Praxis*.
Springer, 2003
- **Martin Fowler** et al.: *Patterns of Enterprise Application Architecture*.
Addison-Wesley, 2003

.NET und C#

Heinz Dobler

Version 20, 2025



*Hello, World!
My name is Todd Nat.
Now, ... I see sharp.*

Übersicht

- **.NET**
 - Motivation und Konzepte
 - CLI, CIL, CLR, CTS, CLS und Typhierarchie
 - Assembly und Metadaten
- **C# (++ -> #)**
 - Geschichte und Charakteristik
 - Charakteristiken
 - Typhierarchie: Wert- und Referenzdatentypen
 - Konstanten, Variablen und Parameter(übergabe)
 - Operatoren und Anweisungen
 - Namensräume und Klassen
 - Eigenschaften und Indizierer (*properties and indexer*)
 - Schnittstellen
 - Weiterleitungen (*delegates*)
 - Ereignisse
 - Attribute
- **FCL:** Überblick über die *Framework Class Library* (FCL)

Motivation

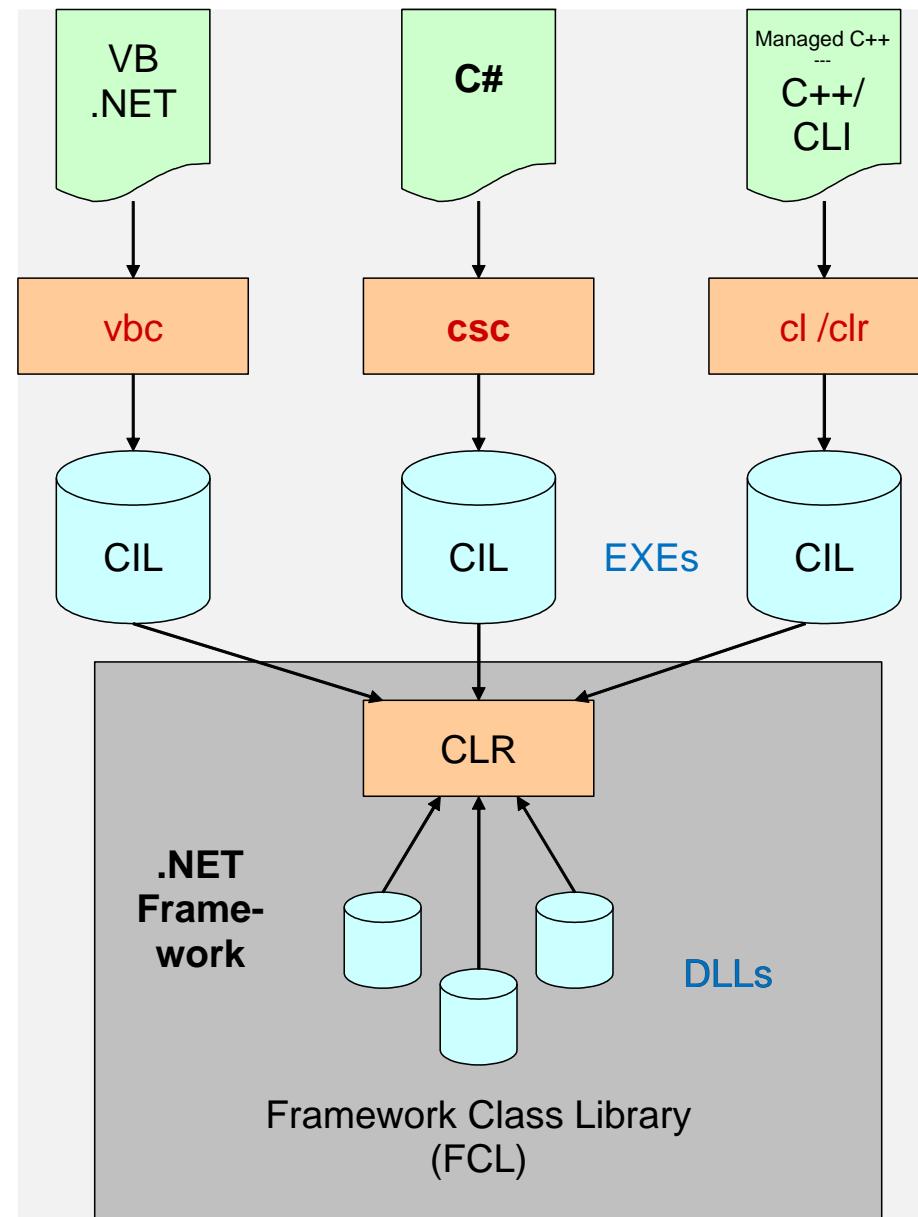
Quelltext

Compiler

"Compilat"

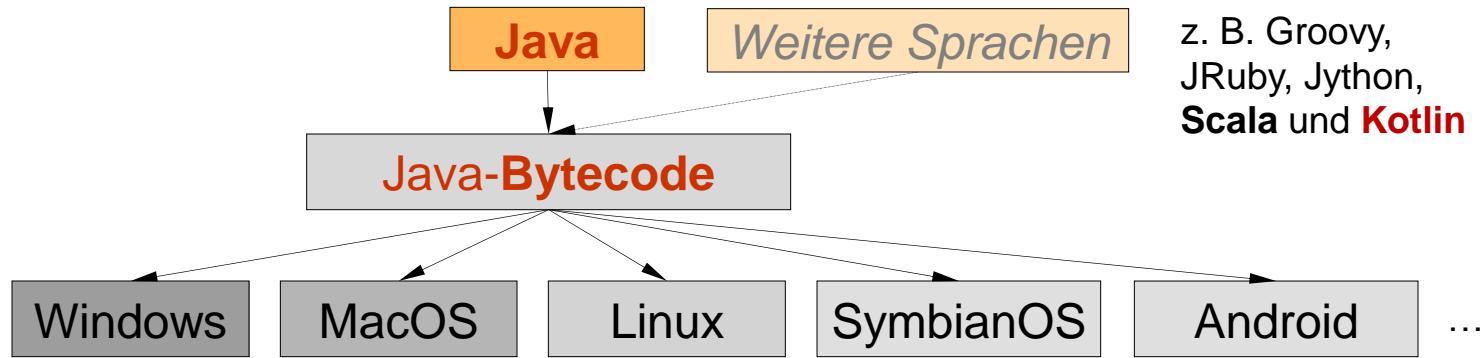
Ausführbares
Programm

Bibliothek(en)

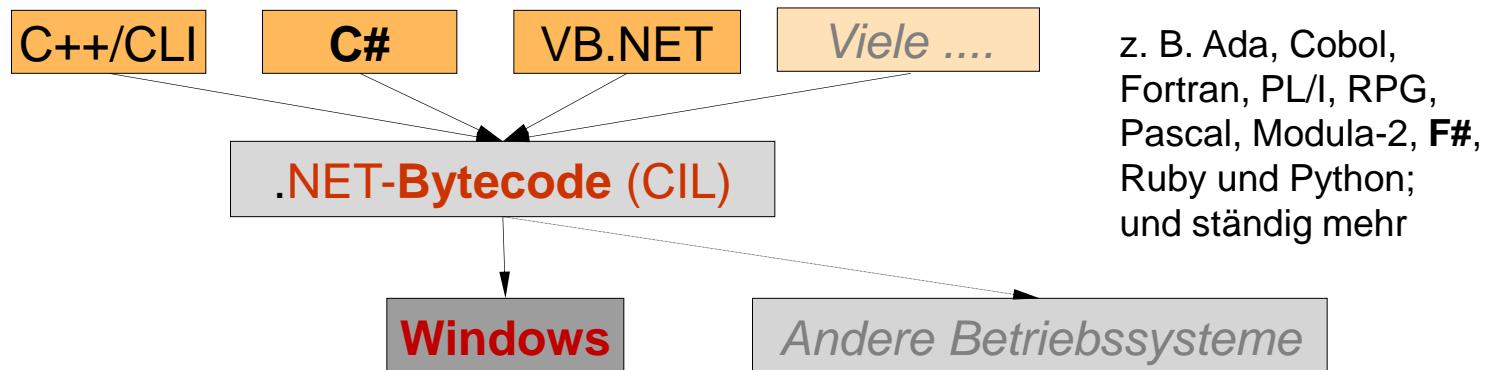


Exkurs: Vergleich von Java mit .NET

Java: Eine Sprache für alle Plattformen, *cross-platform development*



.NET: Viele Sprachen für eine Plattform, *cross-language development*



Wie in Java

- Standardbibliothek mit sehr großem Funktionsumfang (in V. 4.0 ca. 12.000 Klassen und andere Datentypen)
- Gemeinsame Laufzeitumgebung (vergleichbar mit einer virtuellen Maschine)

Anders als in Java

- Ursprünglich schon mehrere, jetzt **viele Programmiersprachen**
- Ursprünglich nur **eine Betriebssystem-Plattform** (MS Win. ab NT 4.0)
- Mittlerweile auch
 - * DotGNU (www.dotgnu.org), 2012 gestoppt
 - * Mono (www.mono-project.com)
- Seit 2016 **.NET Core** für Linux, Mac OS, Docker und auch f. Windows ohne Erweiterungen nur für Konsolenanwendungen (<https://docs.microsoft.com/en-us/dotnet/core>)
- Seit 2020 unter der Bezeichnung **.NET 5**

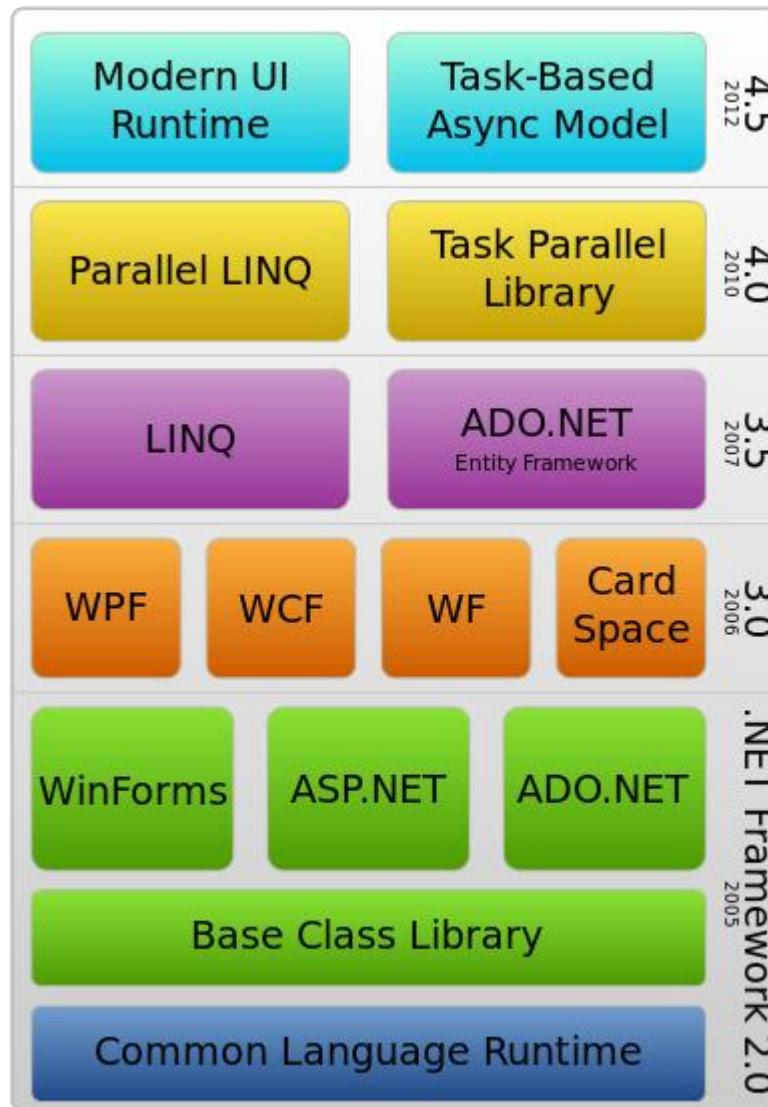
Neues Programmierparadigma

- Komponentenorientierte Programmierung (COP, auf Basis der OOP)

Verschiedene Versionen und Entwicklungsumgebungen

- *Microsoft .NET Framework SDK*
- *MS Visual Studio .NET* (Express Ed., Standard, Prof., Enterprise u. Architect)
- **V. 1.0 Jan. 2002**, V. 2.0 Nov. 2005, V. 3.0 Nov. 2006, V. 4 Apr. 2010, V. 4.5 Aug. 2012, V. 4.6 2015, V. 4.7 April 2017, V. 5 April 2019 und **aktuelle Version: Visual Studio 2022**

Exkurs: .NET Framework Stack



Quelle:
[https://en.wikipedia.org/
wiki/
File:DotNet.svg](https://en.wikipedia.org/wiki/File:DotNet.svg)

Common Language Infrastructure (CLI)

Internat. Standard für Sprach- und plattformunabhängige Entwicklungen

Common Intermediate Language (CIL)

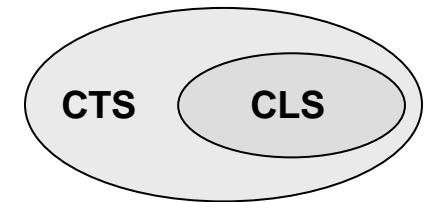
- Alle Compiler für alle .NET-Sprachen erzeugen anstelle von Objektcode für reale Prozessoren eine spezielle Zwischensprache
- CIL umfasst Programmcode und Typinformation (Metadaten)
- **CIL-Code wird nie interpretiert** sondern vor der Ausführung in Objektcode übersetzt *Just-In-Time (JIT)*, mit einem JIT-Compiler:
 1. **JIT** mit Optimierung (*Data Flow*), aufwendig,
 2. **EconoJIT** schnell aber nicht optimierend (bis .NET 2.0) oder
 3. **PreJIT** übersetzt schon bei der Installation (mit NGen)

Common Language Runtime (CLR)

- CIL wird von "virtueller Maschine" ausgeführt:
mit automatischer Speicherbereinigung (*Garbage Collection, GC*),
Ausnahmenbehandlung (*Exception Handling, EH*),
Sicherheitsmechanismen, Versionierung und Interoperabilität
- CLR besteht im wesentlichen aus
 1. Klassenlader (*Class Loader*),
 2. Code-Verifizierer (*Verifier*) und
 3. JIT-Compiler

Common Type System (CTS) 37 Seiten

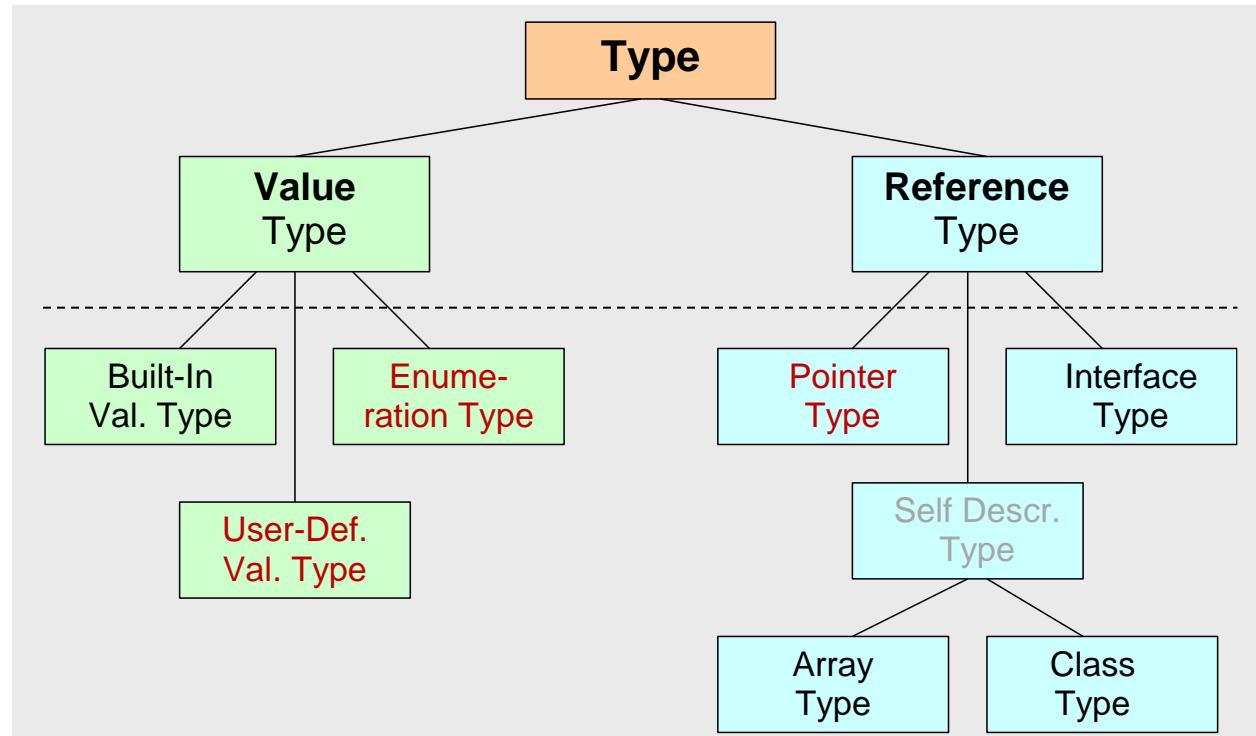
- System aller Typen für alle Sprachen (ermöglicht z. B. Vererbung über versch. Programmiersprachen)



Common Language Specification (CLS) 48 Regeln auf 3 Seiten

- Minimale Teilmenge des CTS, die von einer Programmiersprache unterstützt werden muss, damit sie mit anderen kombiniert werden kann ...

Typ-
hierarchie:



Assembly

- .NET-Anwendung besteht aus *Assemblies* (einzelne installierbare Einheiten, in Form von DLL- und EXE-Dateien)
- *Assemblies* erlauben einfache Installation (*Deployment*) in drei Formen (private, gemeinsame und öffentliche)
- *Assemblies* bestehen aus Manifest und ein oder mehreren Modulen (CIL-Code und Metadaten dafür)
- CLR führt Sicherheits- und Versionsprüfung auf Basis von *Assemblies* durch

Metadaten

- Information über die Datentypen in einem *Assembly*
- Fixer und sehr umfangreicher Bestandteil der CIL
- Verwendung für Übersetzung, Speicher- und Sicherheitsmanagement, in diversen Werkzeugen und Anwendungen
- Manipulation und Erzeugung von Metadaten möglich
- Zugriff über *Reflection*

C# -- Geschichte und Charakterisierung

Geschichte

Mitte 90'er	Microsoft J++: Java für Microsoft Windows, Windows-spezifische Java-Erweiterungen → erst Konflikt dann Rechtsstreit mit Sun: Microsoft verliert und gibt Java auf
1998	Erste Berichte über neue Programmiersprache, ursprünglicher Name COOL, sollte "Java-ähnlich" sein
Juni 2000	Spezifikation für die Sprache mit neuem Namen C# Anders Hejlsberg mit drei Kollegen
Juli 2000	Vorabversion des .NET Framework SDK inklusive C#-Compiler (fünf Entwickler*innen) und <i>Framework Class Library</i> (ca. 1000 Entwickler*innen)
Jan. 2002	Produktionsversion von Microsoft Visual Studio .NET inkl. Compiler für C# 1.0, VB.NET, C++.NET und Framew. V. 1.0
2003	.NET 1.1 mit Visual Studio .NET 2003 (in .NET implementiert) und C# 1.2
2005	.NET 2.0 mit Visual Studio 2005, neu in C# 2 z. B. Generizität, Iteratoren für <i>foreach</i>
2006	.NET 3.0 mit Visual Studio 2007, neu in C# 3 z. B. impliz. Typisierung (<i>var x = ...</i>), WPF, WCF, WF
2010	.NET 4.0 mit Visual Studio 2010, neu in C# 4 z. B. benannte und optionale Parameter
2012	.NET 4.5 mit Visual Studio 2012, alt z. B. wieder nur mehr ein Installationspaket dafür C #5
2015	.NET 4.6 mit Visual Studio 2015, mit .NET Core preview und C# 6
2017	.NET 4.7 mit Visual Studio 2017, viele kleine Erweiterungen und C# 7
2019	.NET 4.8 mit Visual Studio 2019, seit 30. April 2019



Quelle: Wikipedia

Charakterisierung (Zitat Hejlsberg)

"C# is a simple, modern, object-oriented, type-safe, versionable, compatible and flexible programming language derived from C and C++
... supporting component-based development."

Charakteristiken

- **Einfachheit:** viele Eigenschaften von C++ fehlen (z. B. keine Zeiger bei *managed types*), weniger Operatoren (z. B. `::` und `->` fehlen), einheitliches Typsystem (z. B. `int` von `Object` abgeleitet, somit skalare Werte \Leftarrow Objekte)
- **Modernität:** C# ist die bevorzugte Sprache für .NET, die einzige mit **allen** neuen Konzepten (und fast vollständiger Nutzung von CIL)
- **Objektorientierung:** Klassenkonzept basiert auf dem CTS der CLR, nur einfache Vererbung bei Klassen aber mehrfache bei Schnittstellen
- **Typsicherheit:** statische Typprüfung zur Übersetzungszeit, automatische Initialisierung von Datenkomponenten, keine unsicheren Typkonversionen, Bereichs- und Überlaufprüfungen (auch für arithm. Operationen auf `int`-Werten mögl.)
- **Versionierung:** Versionsnummern für Programmeinheiten sind möglich, auch mehrerer Versionen können gleichzeitig genutzt werden (vgl. *DLL hell*)
- **Kompatibilität:** Möglichkeit auf verschiedene APIs zuzugreifen und verschiedene Sprachen zu integrieren, auch alte COM-Komponenten und alte DLLs können weiterverwendet werden (*Interop*, z. B. mit *P/Invoke*)
- **Flexibilität:** Unterscheidung zwischen sicherem (*managed*) und unsicherem Code (*unmanaged*) Code, der beliebige Freiheiten hat (auch Zeiger mit Zeigerarithmetik)
- **Komponentenorientierung:** besondere Sprachkonstrukte, welche Herstellung v. Komponenten unterstützen, z. B. *Properties*, *Events* u. *Delegates*

Beispielprogramm: Sieve.cs

```
using System; // necessary even for namespace System
namespace Erathostenes {
    public class Sieve {
        [public] static (void|int) Main(string[] args) {
            const int SIZE = 1000;
            bool[] sieve = new bool[SIZE + 1];
            int col;
            for (int i = 1; i <= SIZE; i++)
                sieve[i] = true;
            Console.WriteLine(2); // first prime
            col = 1;
            for (int i = 3; i <= SIZE; i += 2) {
                if (sieve[i]) {
                    if (col == 10) {
                        Console.WriteLine(); col = 0; }
                    Console.Write(i + "\t"); // next prime
                    col++;
                    for (int j = 2 * i; j <= SIZE; j += i)
                        sieve[j] = false;
                } // if
            } // for
        } // Main
    } // Sieve
} // Erathostenes
```

1. Übersetzung

csc Sieve.cs

2. Ausführung ...

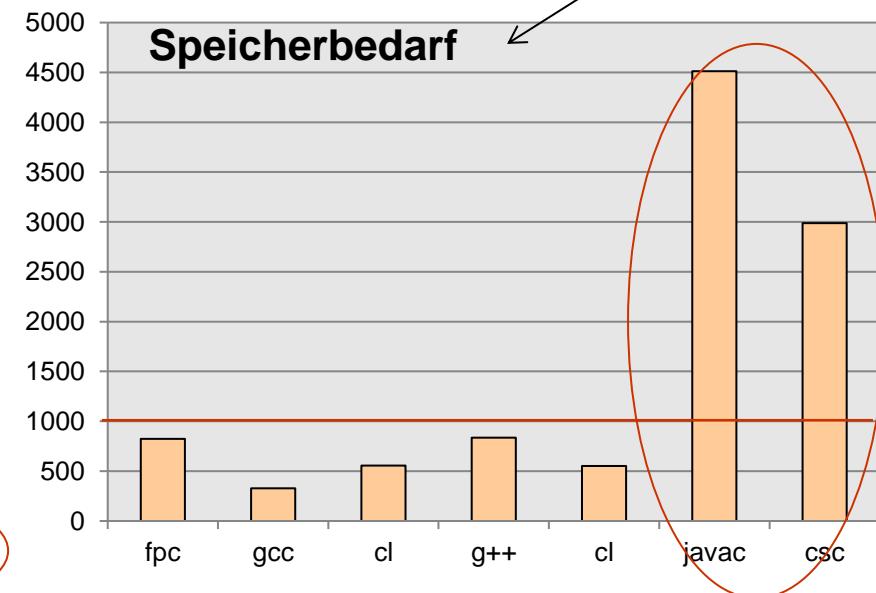
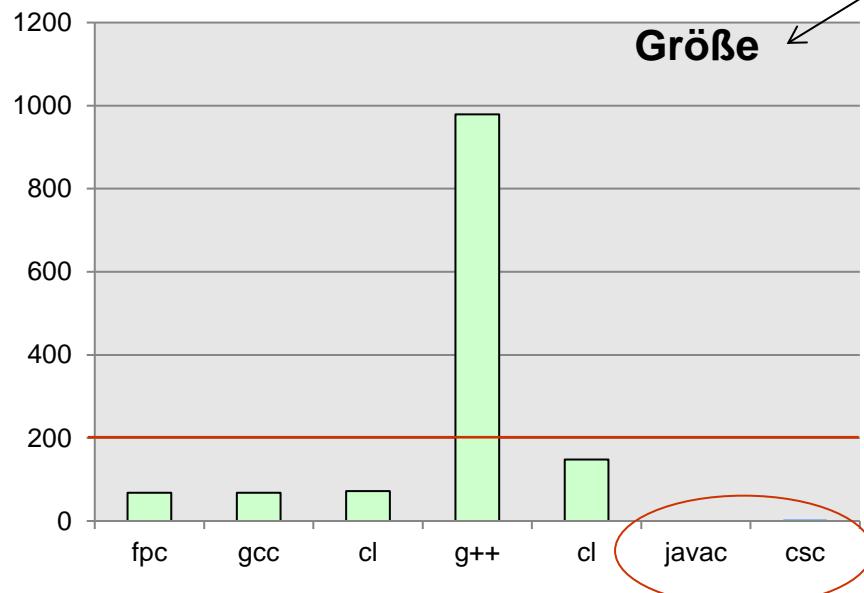
Sieve [.exe]

... liefert Ergebnis:

2 3 5 7 11 ...

Exkurs: *HelloWorld*, Exe-Größen und Speicherbedarf

Sprache	Werkzeug	Größe (ext. Speicher) ausführb. Programm [KB]	Speicherbedarf (RAM) (Prozess unter Windows) [KB]
Pascal	fpc	68	824
C	GNU gcc	68	328
	Microsoft cl	72	556
C++	GNU g++	979	836
	Microsoft cl	148	552
Java	javac	1	4.512
C#	csc	4	2.988



Schlüsselwörter (keywords)

abstract	add	alias	as	ascending
base	bool	break	by	byte
case	catch	char	checked	class
const	continue	decimal	default	delegate
descending	do	double	dynamic	else
enum	equals	event	explicit	extern
false	finally	fixed	float	for
foreach	from	get	global	goto
group	if	implicit	in	int
interface	internal	into	is	join
let	lock	long	namespace	new
null	object	on	operator	orderby
out	override	params	partial 2x	private
protected	public	readonly	ref	remove
return	sbyte	sealed	select	set
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	value	var
virtual	void	volatile	where 2x	while
yield				

Anzahl: C# 1.0: 77, kontextspezifische in 4.0: 25 , Summe = 102

▪ Freies Format

- Groß- und Kleinschreibung bei Namen signifikant
- Konvention zur Schreibweise von Namen: *use Camel Casing*

▪ Kommentare in vier Formen für drei Arten

1. C-Kommentare und XML-Dokumentationskommentare (s. u.)

```
/* any text */    /* ... */
```

2. C++-Kommentare

```
// any text
```

3. Dokumentationskommentare (mittels XML-Marken)

```
/// <tag>  
/// any text  
/// </tag>
```

vgl.

Beispiele f. vordefinierte Marken (tags) in Dokumentationskommentaren:

value: Eigenschaft (*property*)

summary: Datenkomponente oder Methode

param: Parameter einer Methode, z. B.

<param name="anyName"> description </param>

returns: Ergebnis einer Methode

Beispiel: XML-Dokumentation

```
/// <summary> This is the famous HelloWorld program in C#. </summary>
public class HelloWorld {
    /// <summary> And this is the famous Main method </summary>
    /// <param name="args"> command line parameters </param>
    public static void Main(string[] args) {
        System.Console.WriteLine("Hello, World!");
    } // Main
} // HelloWorld
```

csc /doc:HelloWorld.xml HelloWorld.cs

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>HelloWorld</name>
    </assembly>
    <members>
        <member name="T:HelloWorld">
            <summary>
                This is the famous HelloWorld program in C#.
            </summary>
        </member>
        <member name="M:HelloWorld.Main(System.String[])>">
            <summary>
                And this is the famous Main method
            </summary>
            <param name="args">
                command line parameters
            </param>
        </member>
    </members>
</doc>
```

Quelle:
H. Mössenböck

Einfügen:

```
<?xml-stylesheet
    href="doc.xsl"
    type="text/xsl"
?>
```

HelloWorld

This is the famous HelloWorld program in C#.

Main(String[]) method

And this is the famous Main method

Parameters

args

command line parameters

„Präprozessor“-Anweisungen

- C# erlaubt "Präprozessor"-Anweisungen (angelehnt an C und C++)
- Es gibt aber keinen eigenen Präprozessor,
Behandlung dieser Anweisungen durch den C#-Compiler
- Gültigkeitsbereich von "Makros" auf aktuelle Datei beschränkt

"Präprozessor"-Anweisungen, die unterstützt werden:

```
#define symbol // or via command line: csc /define:symbol ...
#undef symbol
```

```
#if symbol [operator symbol12]
```

```
...
#endif symbol [operator symbol12]
```

```
#else
```

```
...
#endif
```

```
#line number ["file"]
```

```
#region ... #endregion // e.g. for visual studio
```

z. B.: #if DEBUG

```
...
#endif
```

Weitere:

```
#warning
#error
#pragma
```

Nicht mehr unterstützt: textuelles Einbinden mittels ...

```
#include <...>
#include "..."
```

Exkurs: Wurzelklasse *System.Object*

```
namespace System {  
  
    public class Object { // concrete class  
        // no data components  
  
        public          Object();  
  
        public virt. String ToString();  
        public virt. bool Equals(Object o);  
        public virt. int  GetHashCode();  
  
        prot.         Object MemberwiseClone();  
  
        public        Type GetType();  
  
        prot. virt. void Finalize(); // same as ~Object() in C#  
  
        public static bool   Equals(0. x, 0. y); // works well with null  
        public static bool Ref.Equals(0. x, 0. y); // works with null and  
                                                // values are boxed  
    } // Object  
  
} // System
```

Beispiel:

```
Object o = new Object();  
Console.WriteLine(o.ToString());  
Console.WriteLine(o.GetHashCode());
```

Ergebnis:

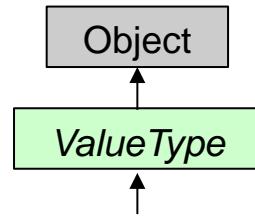
```
System.Object // no @xyz as in Java  
21083178
```

!

Wert-Datentypen

- Variablen eines Wert-Datentyps enthalten den Wert
- Wert-Semantik bei Zuweisung (Wert wird kopiert)

Klassen-
hierarchie:



Standard-Datentypen (vordefiniert)

- Ganzz. Datentypen mit u. ohne Vorzeichen: [s]byte, [u]short, [u]int, [u]long
- Fließkomma-Datent.: float, **double** und **decimal** (z. B. 1.0m) f. exakte Arithm.
- Sonstige: bool, char (Unicode, zwei Bytes)
- Synonyme (vordefinierte structs im Namensraum System):
[S]Byte, [U]Int16, [U]**Int32**, [U]Int64, Single, **Double**, Decimal, Boolean, Char

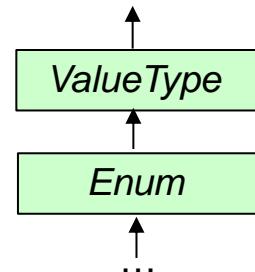
z. B.: int ≈ Int32

Verbunde (benutzerdefiniert)

Verbund mehrerer Werte bel. Typs für "leichtgewicht." Objekte am Laufzeitkeller

```

struct Point { // no inheritance with structs
    public int x, y;
    public Point(int x, int y) { this.x = x; ... }
} // Point
Point p1; p1.x = 0; p1.y = 0; // p1 has already memory
Point p2 = new Point(1, 2); // no memory allocation on heap!
  
```



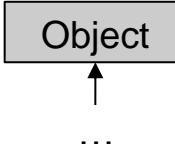
Aufzählungen (benutzerdefiniert)

```

enum Direction {North, East, West, South};
Direction d = Direction.North;
  
```

Referenz-Datentypen

- Variable eines Referenz-Datentyps enthält Referenz (vier oder acht Bytes) auf einen Wert, ein Feld oder auf ein ("schwergewichtiges") Objekt
- Referenz-Semantik bei Zuweisung (Referenz wird kopiert)
- Explizites Anlegen mit *new*, keine Dereferenzierung notwendig



Standard-Datentypen

- *object* bzw. *System.Object*: Basistyp für alle Wert- **und** Referenztypen
- *string* bzw. *System.String*: konstante Zeichenketten aus Unicode-Zeichen, auch alle Zeichenkettenliterale ("...") sind von diesem Typ

Besonderheiten von Zeichenketten (Klasse *String*, Schlüsselwort *string*)

- Vergleich auf Gleichheit (**==**) und Ungleichheit (**!=**) ist "direkt" möglich, auch mit *Equals* (liefert *bool*-Ergebnis) oder *CompareTo* (liefert *int*-Ergebnis)

```
String s1 = "...";
String s2 = new String(...);
if (s1 == s2) // same as s1.Equals(s2)
```

- Zugriff auf einzelne Zeichen mittels Indizierung, z. B. **s[0]**
- Verkettung (Konkatenation) mittels Operator **+** oder *Concat*-Methoden
- Veränderbare Zeichenketten über die Klasse *System.Text.StringBuilder*

Boxing & Unboxing

- *Boxing*: zur uniformen Behandlung von Wert- und Referenz-Datentypen, Wert-Datentypen werden automatisch in Referenzdatentypen umgewandelt

```
int i = 1;
Object o = i; // boxing "raw" int to a "boxed" Int32
```

- *Unboxing*: aus Objekt kann jederzeit durch explizite Typumwandlung (cast) Wert gewonnen werden

```
int j = (int)o; // unboxing "boxed" Int32 to "raw" int
```

- Möglich für alle (also auch für benutzerdefinierte!) Wertdatentypen (diese haben jeweils zwei Ausprägungen: *raw* und *boxed*)

Verwendung

Z. B. für die Verwaltung von Wert-Datentypen in Behältern der Namensräume *System.Collections* und *System.Collections.Generic* (seit .NET 2.0)

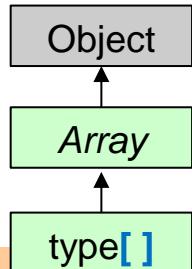
Beispiele

<pre>Stack s = new Stack(); s.Push(17); // w. boxing ... int top = (int)s.Pop(); ...</pre>	Werttyp (value type) <pre>Queue<char> q = new Queue<char>(); q.Enqueue('x'); // w.o. boxing ... char first = q.Dequeue(); ...</pre>
--	--

Felder (Referenz-Datentypen)

- Ein- u. mehrdimensionale Felder, Startindex in jeder Dim. ist 0
- Dynamisch aber mit fixer Größe angelegt (mittels `new type[size]`)
- Laufzeitprüfung bei Indizierung, ev. `IndexOutOfRangeException`

Klassen-
hierarchie:



Eindimensionale Felder

```
char[] digits = new char[10]; // not: char digits[] ...
digits[0] = '0';
```

Mehrdimensionale Felder

- Mehrdim. Felder können rechteckig (*rectangular*) oder flatternd (*jagged*) sein

```
// rectangular:
double[,] mat =
    new double[4, 3];
mat[i, j] = ...
```

Anm.: einfacher zu verwenden,
aber etwas langsamer beim Zugriff
auf die Elemente, dafür viel schneller
beim Anlegen und Freigeben

```
// jagged:
int[][] triangle =
    new int[4][/*must be empty*/];
for (int i = 0; i < 4; i++) {
    triangle[i] = new int[i+1];
} // for
m[i][j] = ...
```

- Alle Felder "kennen" ihre Größe (in jeder Dimension):

```
digits.Length == 10      // only one dimension
mat.Length == 12         // all elements == 4*3
mat.GetLength(0) == 4    // dimension as param.
```

Gleiches
Typproblem
wie in Java,
anstelle von
`ArrayTypeMismatchEx`.
in .Net aber

Konstanten

Konstanten können mittels Schlüsselwort *const* (vgl. C++) vereinbart werden

Variablen

Variablen müssen vor der ersten Verwendung initialisiert werden:
implizit mit Standardwert (bei Datenkomp.) oder explizit durch Init./Zuweisung

Parameterübergabe mit drei (!) Mechanismen

- **Eingangsparameter:** standardmäßig *by value* für Wert- und Referenzdatentypen

```
int Succ(/*in*/ int i) { return i + 1; } // Succ  
int j = Succ(/*in*/ 17);
```

- **Übergangsparameter:** durch explizite Anforderung ist *by reference* möglich
(Aktualparameter muß initialisiert sein)

```
void Inc(ref int i) { i = i + 1; } // Inc  
int j = 17;  
Inc(ref j);
```

- **Ausgangsparameter** (Aktualparameter muß nicht initialisiert sein)

```
void Concat(string a, string b, out string ab) {  
    ab = a + b;  
} // Concat  
String s; // not initialized  
Concat("abc", "xyz", out s);
```

Vorrang u. Assoziativität der Operatoren wie in C/C++ (inkl. `sizeof` und Adressoperator &)

Neue Operatoren

- `typeof` liefert Metainformation für einen Datentyp (in Form von *Type*-Objekten)

```
Type t = typeof(System.Int32);           // static
Type t = Type.GetType("System.Int32"); // dynamic
```

- `checked/unchecked`: explizite Angabe ob Laufzeitprüfungen (z. B. auf Überlauf bei arithmetischen Operationen) durchzuführen sind

```
c = checked(a * b); // checked expression ... may throw
[un]checked {        // [un]checked block, Overflow-
...; c = a * b; ...; // where unchecked Exception
} // [un]checked // is default
```

- `as`: für sichere Typkonversion in einer Klassenhierarchie nach unten (*downcast*)

```
A a; // base class
B b; // derived class, so class B : A { ... }
a = new ...;
b = a as B; // results in null if cast fails
```

- `is`: für Typtest (vgl. `instanceof` in Java)

```
if (a is B) ...
```

- `stackalloc`: zum Anlegen von Speicherplatz für Felder innerhalb eines Blocks am Laufzeitkeller (*stack*), nur in unsicherem Code, `unsafe { ... }` und ↴

```
*! int *a = stackalloc int[100]; // compile with /unsafe
*a = 17; Console.WriteLine(a[0]); // writes 17, as in C
```

Anweisungen

Anweisungen wie in C/C++ mit folgenden Änderungen/Erweiterungen:

- *switch*-Anweisung auch mit *String*-Ausdruck erlaubt, sonst restriktiver: am Ende jedes *case*-Blocks und des optionalen *default*-Zweigs muss eine *break*-, *return*-, *throw*- oder *goto*-Anweisung stehen
- neue *foreach*-Schleife (für alle Felder und Behälter, die *IEnumerable* impl.)

```
double[] a = new double[100];
for (int i = 0; i < a.Length; i++) { a[i] = 0.0; }
foreach (double e in a) { e = 0.0; }
```

- *goto*-Anweisung mit Anweisungsmarken
- *throw* mit *try/catch/finally* für Ausnahmenbehandlung (wie in Java)
- *lock*-Anweisung für Synchronisation bei Threads mit gemeinsam genutzten Ressourcen: kritischer Bereich mit *Monitor* (vgl. *synchronized* in Java)

```
lock (expr) ...; // expr must eval. to a reference
lock (expr) { ...; ...; ...; }
```

lock verwendet *Monitor.Enter(expr)* und *Monitor.Exit(expr)*

- *using*-Anweisung zur Deklaration von Ressourcen und automatischer Freigabe mit Methode *Dispose* aus *IDisposable* (vgl. *try with res.* in Java)

```
using (Font f = new Font("Arial", 12)) {
    // use f, on end of block f will be disposed
} // end of using: f.Dispose()
```

Namenräume (*Namespaces*)

- Namenräume dienen der logischen Gliederung unabhängig von der Verzeichnisstruktur

```
namespace N {  
    class X { ... }  
    interface Y { ... }  
} // N
```

- Namenräume sind offen, können in beliebigen Quelltext-Dateien erweitert werden
- Namenräume können beliebig geschachtelt werden (z. B. *System.Net.Sockets*)
- Der äußerste Namenraum ist namenlos und enthält alle andern Vereinbarungen von Datentypen und Namenräumen
- Zugriff auf Elemente eines Namensraums durch Qualifikation mit dem Namensraumnamen (z. B. *System.String*) oder durch *using*-Deklarat.
- Üblich aber nicht (wie in Java) zwingend erforderlich:
 - eine Datei/Klasse (Dateiname = Klassenname)
 - ein Verzeichnis/Namenraum (Namenraumname = Verzeichnisname)

Klassen (Classes)

- (Wie in C++ und Java): Klassen haben Datenkomponenten (auch *const* und *readonly*) und Methoden, beides auf Klassen- (mittels *static*) und/oder Objektebene
- Konstruktoren für Objekte und für Klassen- (mittels *static*)
- Ev. vorhandene Destruktoren (werden vom *Garbage Collector* aufgerufen, also nicht-deterministisch), sind aber nur eine "Abkürzung" für *Finalize*-Methode, also ...

```
class C {  
    static C() { ... } // class initializer  
    C() { ... } // object initializer  
  
    // either:  
    ~C() { ... }      // ->Finalize  
    // or:  
    protected override void Finalize() { ... }  
}  
// C
```

- Methoden **u. einige Operatoren** (z. B. arith. u. Vergleichsop.) können überladen werden
- Schlüsselwort *this* in Methoden für Empfängerobjekt
- Klassen können beliebig geschachtelt werden
- Spezielle Arten von Klassen:
 - abstrakte Klassen (Schlüsselwort *abstract*) und Erweiterungsmethoden, z. B.
static ... EM(**this** AnyClass o, ...);
 - Klassen von denen nicht weiter abgeleitet werden kann (Schlüsselwort *sealed*)
 - "statische" Klassen (haben nur Klassenkomponenten, keine Objekte möglich)
 - partielle Klassen (Schlüsselwort *partial*), die auf mehrere Dateien verteilt sind

Beispiel: Klassenteile und Erweiterungsmethoden

```
partial class A { // part 1 of A
    public int i = 0;
    public void PrintD() {
        System.Console.WriteLine(d);
    } // PrintD
} // A, part 1
```

```
partial class A { // part 2 of A
    public double d = 0.0;
    public void PrintI() {
        System.Console.WriteLine(i);
    } // PrintI
} // A, part 2
```

```
public static class T { // class must be static to host extension methods
    // extension methods: static methods within a static class and this param.

    static void PrintAll(this A a) { // praram a plays the role of this
        System.Console.WriteLine(" " + a.i + ", " + a.d);
    } // PrintAll

    static String ToString(this A a) { // valid, but good idea?
        return " " + a.i + ", " + a.d;
    } // PrintAll

    public static void Main() {
        A a = new A();
        a.PrintI(); // normal method
        a.PrintD(); // normal method
        a.PrintAll(); // stand. way to call an extension method
        PrintAll(a); // another way to call an extension method
        Console.WriteLine(a.ToString()); // ... calls object.ToString
    } // Main
} // T
```

Eigenschaften (*Properties*)

Eigenschaften (*properties*) sind Kombination aus speziellen (privaten) Datenkomponenten und (öffentlichen) Zugriffsmethoden, z. B.:

```
class Person {  
  
    private DateTime dateOfBirth;  
    private String lastName;  
    private String firstName;  
  
    public Person(DateTime dob, String fn, String ln) { ... }  
  
    public DateTime DateOfBirth {  
        get {  
            return dateOfBirth;  
        } // get  
    } // DateofBirth  
  
    public String LastName {  
        get {  
            return lastName;  
        } // get  
        set { // implicit parameter list: (String value)  
            lastName = value; // value is implicit String param.  
        } // set  
    } // LastName  
  
} // Person
```

// since C# 3.0: auto impl. properties

```
class Person {  
  
    public DateTime DateOfBirth {get; }  
    public String LastName {get; set;}  
    ...  
} // Person
```

Eigenschaften werden vom Compiler automatisch in *set*- und *get*-Methoden übersetzt, weil...

Indizierer (*Indexer*)

- Indizierer (*indexers*) sind ähnlich den Eigenschaften, erlauben aber die Behandlung eines Objekts so, als wäre es ein Feld (mit Indizierungsoperator)

```
class ProbabilityArray { // all elements in [0, 1]
    private double[] elements = new double[10];

    public double this[int index] {
        get {
            return elements[index];
        } // get
        set { // impl. param. list: (double value)
            if (value >= 0.0 && value <= 1.0) {
                elements[index] = value;
            } else ... // e. g., throw new InvalidValueException();
        } // set
    } // this

} // ProbabilityArray

ProbabilityArray pa = new ProbabilityArray();
pa[0] = 3.5;
```

- Mehrere Indizierer mit unterschiedlichen Schnittstellen pro Klasse (auch mit mehr als einem Parameter) sind möglich (Überladen von Indizierern)
- Indizierer werden vom Compiler automatisch in *set*- und *get*-Methoden mit zusätzlichem Index-Parameter übersetzt, weil ...

Vererbung (*Inheritance*)

- Nur einfache und öffentliche Vererbung auf Klassenebene (wie in Java)
- Basisklasse aller Klassen ist *object == System.Object*
- Abgel. Klassen können eine oder mehrere Schnittstellen implementieren

```
class A /*: System.Object */ {...} // A
class B: A {...} // B inherits from A
class C: I {...} // C inh. from Object and impl. I
class D: A, I {...} // D inh. from A      and impl. I
```

- Methoden werden standardmäßig statisch gebunden
- Dynamische Bindung ist mit Schlüsselwort *virtual* explizit anzufordern
- Überschreiben einer mit *virtual* dynamisch gebundenen Methode in der abgeleiteten Klasse ist nur mit Angabe von *override* erlaubt

```
class A { virtual void M() {...} } // A
class B: A { override void M() {...} } // B
```

- Abstrakte Methoden (*abstract* ohne *virtual*) müssen in der abgeleiteten Klassen überschrieben werden (*override* ist anzugeben)
- Schlüsselwort *base* zum Zugriff auf Komponenten der Basisklasse in der abgeleiteten Klasse

Schnittstellen (*Interfaces*)

- Schnittstellen (*interfaces*) sind wie in Java definiert
- Schnittstellen sind Referenzdatentypen, vergleichbar mit abstrakten Klassen, die nur eine Schnittstelle definieren, aber keine Implementierung angeben
- Schnittstellen dürfen nur Methoden-, Eigenschafts-, Indizierer- und Ereignis- (siehe unten) Deklarationen enthalten

```
public interface IDisposable { // for using statement
    public void Dispose();
} // IDisposable
```

Weitere Beispiele: *IComparable* und *ICloneable*, vgl. Java

- Vererbung ist auch auf Schnittstellenebene definiert, einfache und mehrfache Vererbung ist möglich

```
public interface IB: IA1, IA2, ... IAn {
    ...
} // IB
```

Weiterleitungen oder "Delegate" (*Delegates*)

- Erweiterung des Konzepts der Funktionszeiger aus C
- *Delegate* ist spez. Ref.-Datentyp, definiert **Schnittstelle für Elemente**, z. B.:

```
delegate void Filter(ref String s);
```

- *Delegate* ist ein "Behälter" für Elemente mit der definierten Schnittstelle (*Delegate* mit mehreren Elementen heißt *multicast delegate*)
- Elemente können Klassenmethoden und/oder Objektmethoden sein
- Erstes Element kann schon im Konstruktor, weitere dann mit Operator **+ =** hinzugefügt werden, z. B. (Details in Beispiel auf nächster Seite):

```
// either:  
Filter f = new Filter(Identity);  
  
// or:  
Filter f = null; f += new Filter(Identity);  
  
// additional  
f += new Filter(Reverse);
```

- Elemente können mittels Operator **- =** auch wieder entfernt werden
- Aufruf der Methoden (also der Elemente) in der Reihenfolge des Eintragens

```
String s = "A B C":  
f(ref s); // invoke 1. Identity and 2. Reverse
```

Beispiel zu Weiterleitungen (Delegates)

```
public class De1Ex { // delegate example  
  
    delegate void Filter(ref String s);  
  
    static void Identity(ref String s) {  
    } // Identity  
  
    static void Reverse(ref String s) {  
        StringBuilder sb =  
            new StringBuilder();  
        foreach (char ch in s)  
            sb.Insert(0, ch);  
        s = sb.ToString();  
    } // Reverse  
  
    void StripBlanks(ref String s) {  
        StringBuilder sb =  
            new StringBuilder();  
        foreach (char ch in s)  
            if (ch != ' ')  
                sb.Append(ch);  
        s = sb.ToString();  
    } // StripBlanks  
  
} // De1Ex
```

Verwendung:

```
Filter f =  
    new Filter(Identity);  
  
f +=  
    new Filter(Reverse);  
  
De1Ex d = new De1Ex();  
f +=  
    new Filter(d.stripBlanks);  
  
String s = "A B C";  
f(ref s);  
Console.WriteLine(s); // => CBA
```

Ereignisse (*Events*) (1)

- Ereignisse (*events*) sind spezielle Datenkomponenten eines *delegate*-Typs, die aber nur innerhalb ihrer Klasse "ausgelöst" werden können
- Ermöglicht z. B. einfache Realisierung des Entwurfsmusters *Observer*

Beispiel (vgl. MVC in Java)

```
delegate void Notifier(Data d);  
  
class Model {  
    Data d;  
    event Notifier n = null;  
    void ChangeData(...) {  
        d = ...;  
        n(d); // call update methods  
    } // ChangeData  
} // Model
```

```
class View1 {  
    View1(Model m) {  
        m.n +=  
            new Notifier(Update);  
    } // View1  
    void Update(Data d) { ... }  
} // View1  
  
class View2 { ... } // View2  
  
class Application {  
    static void Main(...) {  
        Model m = new Model(...);  
        View1 v1 = new View1(m);  
        View2 v2 = new View2(m);  
        m.ChangeData(...);  
    } // Main  
} // Application
```

Ereignisse (*Events*) (2)

Ereignisse (*events*) werden für die Realisierung des Ereignisbehandlungs-Mechanismus auf Basis von Weiterleitungen (*delegates*) verwendet

Beispiel

```
public class ESArgs :  
    EventArgs {  
  
    public int data;  
  
    public ESArgs(int data) {  
        this.data = data;  
    } // ESArgs  
} // ESArgs
```

```
EventSource es =  
    new EventSource();  
es.eh +=  
    new ESEventHandler(...);  
es.Data(100);
```

```
delegate void ESEventHandler(  
    Object s, ESArgs e);  
  
class EventSource {  
  
    public event ESEventHandler eh = null;  
  
    private int data;  
    public int Data {  
        get { ... } // get  
        set {  
            if (data != value) {  
                data = value;  
                ESArgs esa = new ESArgs(data);  
                eh(this, esa); // invoke  
            } // if  
        } // set  
    } // Data  
} // EventSource
```

Ereignisse (*Events*) (3)

(Properties: set- und get-Zugriffsmethoden für Datenkomponenten)
Accessors: add- und remove-Zugriffsmethoden für Ereignisbehandler

Beispiel

```
class EventSource {  
  
    private event ESEventHandler eh = null;  
  
    public event ESEventHandler Eh {  
        add {  
            eh += value;  
        } // add  
        remove {  
            eh -= value;  
        } // remove  
    } // ESEventHandler  
}  
// EventSource
```

```
EventSource es = new EventSource();  
es.Eh += new ESEventHandler(...);  
es.Data(100);
```

Attribute (*Attributes*)

- Sprache hat fixen Satz von Beschreibungsmechanismen (z. B. Modifizierer wie *ref* und *out*, Angaben über Sichtbarkeit *public* und *private*, ...)
- Attribute (*attributes*) sind **benutzerdefinierbare** Sprachkonstrukte zur weiteren Beschreibung (*decoration*) anderer Sprachelemente (z. B. Datentypen, Komponenten, Parameter, ...)
- Attribute haben keinen Einfluss auf "Standardverhalten" (Semantik) sondern ermöglichen zusätzliche Angaben und damit Funktionalität, z. B.:

```
[Serializable] // is an interface in Java  
public class Person { ... }
```

- Attribute werden mit speziellen Klassen implementiert, z. B.:

```
class SerializableAttribute: Attribute { ... }  
[System.SerializableAttribute]  
public class Person { ... }
```

- Attribute können auch mit Parametern versorgt werden, z. B.:

```
[Obsolete("use class New instead", IsError = true)]  
public class Old { ... }
```

- Attribute werden in Form von Metainformation in CIL repräsentiert und können mittels *Reflection* abgefragt werden

- Klasse `System.Console` stellt einfachen Mechanismen für Ein/Ausgabe von/auf die Konsole zur Verfügung
- Kapselt `stdin`, `stdout` und `stderr` in Form von drei Eigenschaften (*properties*) mit `get`-Methoden: `Console.In`, `Console.Out`, `Console.Err`

Eingabe

- `int Console.Read()`: liefert Zeichencode des nächsten Eingabezeichens in der aktuellen Eingabezeile
- `String Console.ReadLine()`: liefert die gesamte nächste Eingabezeile

Beispiel: Einlesen einer ganzen Zahl

```
int n = Int32.Parse(Console.ReadLine());
```

Ausgabe

- `void Console.Write(...)`: Ausgabe beliebiger Werte
- `void Console.WriteLine(...)`: w.o. aber in eigener Zeile

Beispiel: Ausgabe von mehreren Werten in einem Aufruf

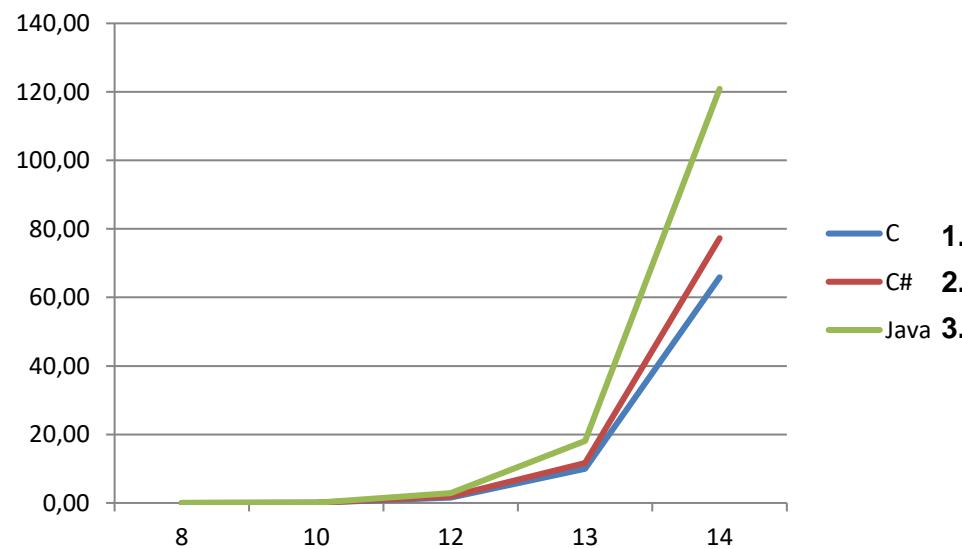
```
int    n;
double s;
...
// 0 1
Console.WriteLine( "{0} results in {1} seconds", n, s);
Console.WriteLine($"'{n}' results in '{s}' seconds"); // interpolated
```

2002: Benchmarks (rein prozedural)

Vergleich von C, C# und Java

n-Damen-Problem, Zeiten in Sekunden auf Pentium Pro mit 200 MHz

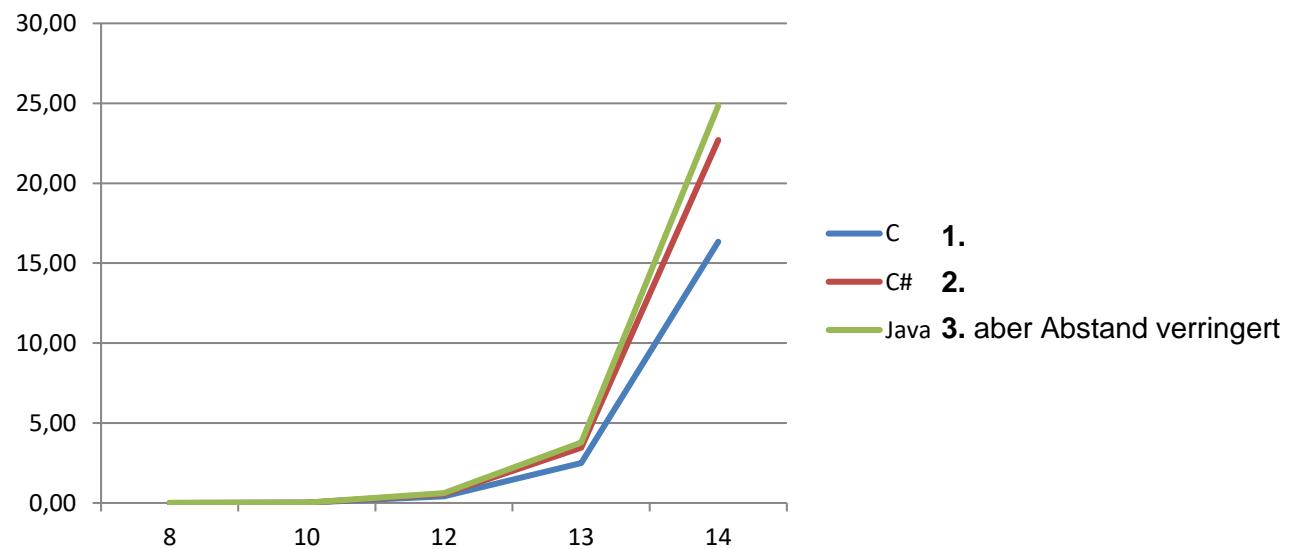
$n =$	8	10	12	13	14
Lösungen	92	724	14.200	73.712	365.596
C	0,00	0,05	1.61	9,99	65,88
C#	0,00	0,06	1,91	11,65	77,28
Java	0,00	0,09	2,87	18,06	120,89



Vergleich von C, C# und Java

n-Damen-Problem, Zeiten in Sekunden auf Pentium 4 mit 2 GHz

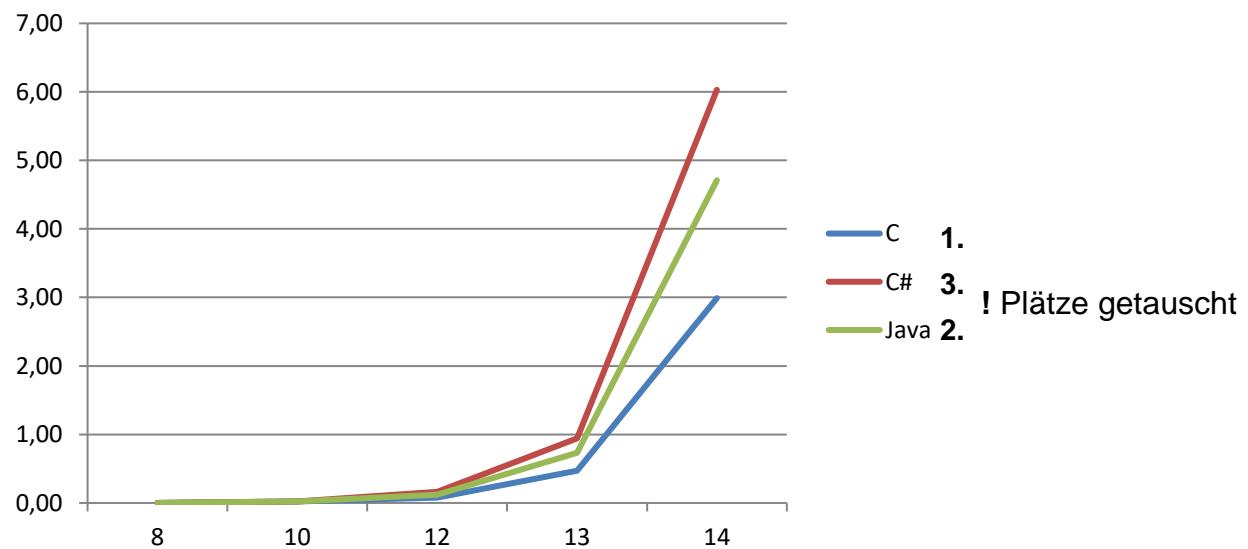
$n =$	8	10	12	13	14
Lösungen	92	724	14.200	73.712	365.596
C	0,00	0,02	0,41	2,50	16,34
C#	0,00	0,02	0,56	3,45	22,70
Java	0,00	0,02	0,61	3,78	24,85



Vergleich von C, C# und Java

n-Damen-Problem, Zeiten in Sekunden auf Intel Xeon E3-1271 mit 3,6 GHz

$n =$	8	10	12	13	14
Lösungen	92	724	14.200	73.712	365.596
C (gcc V. 5.1 mit -O3)	0,00	0,02	0,08	0,47	2,99
C# (csc V. 6 mit /o+)	0,00	0,02	0,16	0,94	6,03
Java (javac V. 1.8)	0,00	0,02	0,12	0,73	4,71

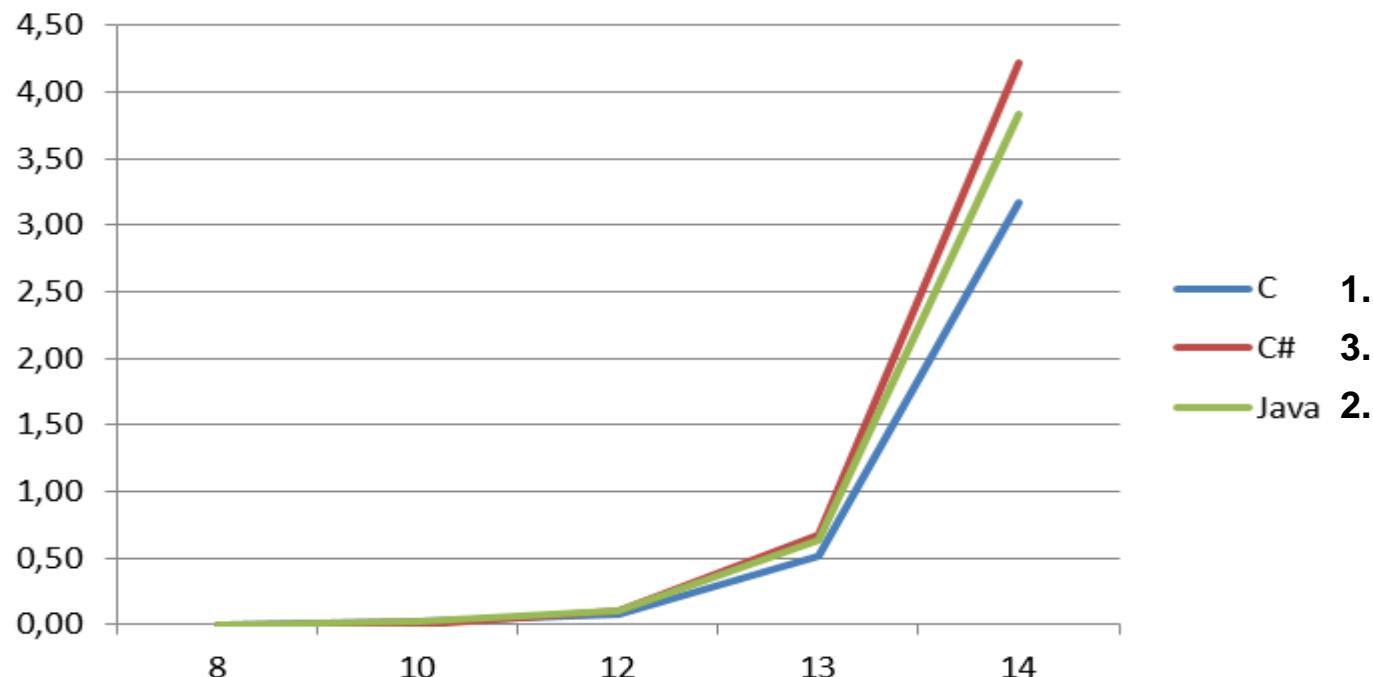


2019: Benchmarks (rein prozedural)

Vergleich von C, Java und C#

n-Damen-Problem, Zeiten in Sekunden auf Intel Xeon E3-1271 mit 3,6 GHz

$n =$	8	10	12	13	14
Lösungen	92	724	14.200	73.712	365.596
C (V. 8.3.0: gcc -O3)	0,00	0,02	0,08	0,52	3,17
Java (V. 11: javac)	0,00	0,02	0,10	0,63	3,83
C# (V. 7: csc -optimize)	0,00	0,02	0,11	0,67	4,22



- Funktionalität der CLR wird durch eine Menge von Klassen realisiert und zur Verfügung gestellt
- Fast alle dieser Klassen können von fast allen .NET-Sprachen verwendet werden, C# bietet alle Möglichkeiten
- Umfang in V. 4.0: ca. 12.000 Klassen, aufgeteilt auf mehrere *Assemblies* (DLLs)

Kernfunktionalität: Namenraum System

Klassen, Schnittstellen und Attribute, auf denen alle anderen Klassen aufbauen, z. B.:

- Klasse *Object* (Basis aller Datentypen) und Basisklasse für Metainformation (*Type*)
- Klasse *Exception* (Basisklasse aller Ausnahmenklassen),
- Klasse *Convert* mit Methoden für alle Konvertierungsmethoden (in Form von über 300 Klassenmethoden),
- Klassen für Datum und Uhrzeit (*DateTime*), für Zeitspannen (*TimeSpan*) und Zeitzonen (*TimeZone*)
- Klassen für mathematische Operationen (*Math*) und Zufallszahlen (*Random*)
- Viele Schnittstellen zur Definition von Eigenschaften (*ICloneable*, *IComparable*, ...) und Attribute (*SerializableAttribute*, *ObsoleteAttribute*, ...)

Zeichenketten und Text

Zeichenketten und Text sowie deren Bearbeitung wird durch Klassen in mehreren Namensräumen unterstützt:

- Namensraum *System* (z. B. Klasse *String*)
- Namensraum *System.Text* (z. B. *StringBuilder*, *Encoder* und *Decoder* unter Verwendung benutzerdefinierter bzw. vordefinierter *Encodings* wie *ASCII*, *Unicode*, ...)
- Namensraum *System.Text.RegularExpressions* (z. B. Klasse *Regex*)

Behälter (*.NET collections*)

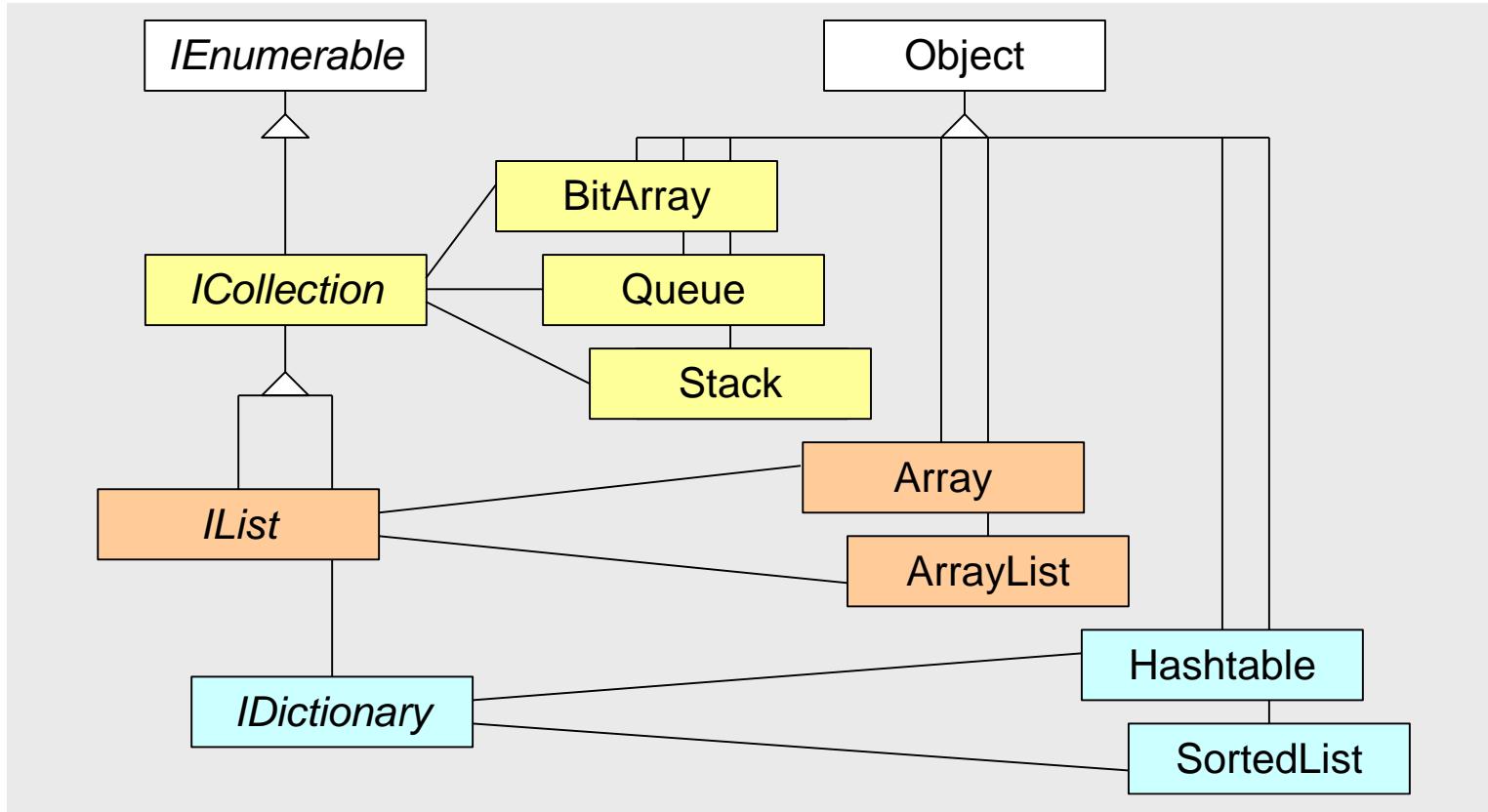
- Namensräume *System.Collections[.Generic]* und *System.Collections.Specialized* bieten Behälterklassen wie *Array* und *BitArray*, *ArrayList*, *Hashtable*, *Queue* und *Stack*
- Iterator-Mechanismus über Schnittstellen *IEnumerable* und *IEnumerator*, Beispiel für typische Verwendung:

```
collection c = new AnyCollection();

// 1. with iterator
IEnumerator ie = c.GetEnumerator();
while (ie.MoveNext()) {
    T e = (T)ie.Current; // now do anything with e
} // while

// 2. with foreach
foreach (T e in c) {
    /* do anything with e */
} // foreach
```

Exkurs: Behälter (1)



Datentypen für Iterator-Mechanismus:

```
interface IEnumerable {  
    IEnumerator Getenumerator();  
} // IEnumerable
```

```
interface IEnumerator {  
    bool MoveNext();  
    Object Current {get;};  
    void Reset();  
} // IEnumerator
```

Exkurs: Behälter (2)

Schnittstelle für ALLE Behälter:

```
interface ICollection: IEnumerable {
    int Count {get;};
    bool IsSynchronized {get;};
    Object SyncRoot {get;};
    void CopyTo(Array a, int startIndex);
} // ICollection
```

Dürftig:
kein *Add*,
Remove,
Contains!

Schnittstelle für alle
sequentiellen Behälter:

```
interface IList: ICollection {
    bool IsFixedSize {get;};
    int Add(Object o);
    void Insert(Object o,
                int atIndex);
    void Remove(Object o);
    void RemoveAt(int index);
    bool Contains(Object o);
    int IndexOf(Object o);
    object this[int index] {
        get; set; };
    void Clear();
} // IList
```

Schnittstelle für alle
assoziativen Behälter:

```
interface IDictionary: ICollection {
    int Add(Object key,
            Object value);

    void Remove(Object key);

    bool Contains(Object key);

    object this[Object key] {
        get; set; };
    void Clear();
    IDictionaryEnumerator Get Enumerator();
} // IDictionary
```

.NET-Klassenbibliothek (3)

- Ein/Ausgabe (*System.IO*, ...)
- Netzwerk (*System.Net*, *System.Net.Sockets*)
- Nebenläufigkeit (*System.Threading*, *System.Timers*)
- Sicherheit (*System.Security*, ...)
- Metainformation (*System.Reflection* und *System.Reflection.Emit*)
- Serialisierung (*System.Runtime.Serialization*, ...)
- Verteilung (*System.Runtime.Remoting*, ...)
- Web Services (*System.Web.Services*)
- Zugriff auf relationale Datenbanken (*System.Data*, ...)
- XML-Bearbeitung (*System.Xml*, ...)
- Grafik (*System.Drawing*)
- GUI-Programmierung (*System.Windows.Forms*)
- Web-Programmierung (*System.Web*)
- Konfiguration (*System.Configuration*)
- Komponentenorientierte Programmierung (*System.EnterpriseServices*)
- Diagnose und Debugging (*System.Diagnostics*)
- ... und Vieles mehr

.NET Framework 3.0 und C# 3.0

- Mit Windows Vista (Ende 2006) ist .NET Framework 3.0 veröffentlicht worden, baut auf 2.0 auf und bringt neu: *Windows Workflow Foundation (WWF)*, *Windows Communication Foundation (WCF)* und *Windows Presentation Foundation (WPF)*
- Mit MS Visual Studio 2008 wird C# 3.0 veröffentlicht, neue Eigenschaften:

- *Language Integrated Query (LINQ)*, z. B.:

```
IEnumerable<String> result =  
    from s in students where s.Studies("SE") select s.LastName;
```

- Lambda-Ausdrücke, z. B.:

```
delegate double Function(double x);  
double Apply(Function f, double x) { return f(x); }  
double d = Apply(x => 2.0 * x + x, 3.14);
```

- Erweiterungsmethoden, z. B.:

```
static void newMethod(this AnyClass t, ...) { ... }  
AnyClass o; o.newMethod(...);
```

- Objektinitialisierer, z. B.:

```
Person p = new Person() { name = "Donald", dateOfBirth = 19310501 };
```

- Anonyme Typen, z. B.:

```
Object o = new { Name = "Daisy", DateOfBirth = 19370213 };
```

Mit Visual Studio 2010 (seit April 2010) gibt es sowohl eine neue Version des Frameworks als auch der Sprache C#

Beispiele neuer Eigenschaften in C#

- Dynamische Objekte (zur Laufzeit typisiert, Typ *dynamic*)
- Benannte und optionale Parameter bei Methoden
- Asynchr. Aufrufe von Weiterleitungen (*asynchr. delegate invocation*)
- Parallel Behandlung von Ausnahmen
- Parallel Ausführung von LINQ-Abfragen
- Bei Generizität: Ko- und Kontravarianz, "faules" Nachladen (*lazy loading*)

Beispiele neuer Elemente im Framework

- Nebenläufige Behälterklassen (*concurrent coll. cl.*) Viel mehr dazu im 5. Semester
- Klasse *Monitor* zur Synchronisation

Buchempfehlungen

- Wolfgang Beer, Dietrich Birngruber, Hanspeter Mössenböck u. Albrecht Wöss:

*Die .NET-Technologie –
Grundlagen und Anwendungsprogrammierung, 2. Auflage
dpunkt.verlag, 2006*

- Hanspeter Mössenböck:

*Kompaktkurs C# 7.0,
aktualisierte und erweiterte Auflage
dpunkt.verlag, 2019*



- Joseph Albahari:

*C# 12 in a Nutshell,
O'Reilly, 2023*

- Mark Michaelis:

*Essential C# 12.0
Addison Wesley, 2023*

